UNIVERSIDADE FEDERAL DE GOIÁS / INSTITUTO DE INFORMÁTICA

Aprendizado por Reforço para Desenvolvimento de Jogos

Uma Abordagem prática com Unity ML-Agents

Lucas Brandão Rodrigues

UFG
UNIVERSIDADE FEDERAL DE GOIÁS

UNIVERSIDADE FEDERAL DE GOIÁS (UFG)
INSTITUTO DE INFORMÁTICA (INF)

LUCAS BRANDÃO RODRIGUES

Aprendizagem por Reforço para Desenvolvimento de Jogos

Uma Abordagem prática com a Unity ML-Agents

Goiânia
2025

UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

TERMO DE CIÊNCIA E DE AUTORIZAÇÃO PARA DISPONIBILIZAR VERSÕES ELETRÔNICAS DE TRABALHO DE CONCLUSÃO DE CURSO DE GRADUAÇÃO NO REPOSITÓRIO INSTITUCIONAL DA UFG

Na qualidade de titular dos direitos de autor, autorizo a Universidade Federal de Goiás (UFG) a disponibilizar, gratuitamente, por meio do Repositório Institucional (RI/UFG), regulamentado pela Resolução CEPEC no 1240/2014, sem ressarcimento dos direitos autorais, de acordo com a Lei no 9.610/98, o documento conforme permissões assinaladas abaixo, para fins de leitura, impressão e/ou download, a título de divulgação da produção científica brasileira, a partir desta data.

O conteúdo dos Trabalhos de Conclusão dos Cursos de Graduação disponibilizado no RI/UFG é de responsabilidade exclusiva dos autores. Ao encaminhar(em) o produto final, o(s) autor(a)(es)(as) e o(a) orientador(a) firmam o compromisso de que o trabalho não contém nenhuma violação de quaisquer direitos autorais ou outro direito de terceiros.

1. Identificação do Trabalho de Conclusão de Curso de Graduação (TCCG)

Nome(s) completo(s) do(a)(s) autor(a)(es)(as): LUCAS BRANDÃO RODRIGUES

Título do trabalho: Aprendizado por Reforço para Desenvolvimento de Jogos

Uma Abordagem prática com a Unity ML-Agents

2. Informações de acesso ao documento (este campo deve ser preenchido pelo orientador) Concorda com a liberação total do documento [X] SIM [ ] NÃO[1]

[1] Neste caso o documento será embargado por até um ano a partir da data de defesa. Após esse período, a possível disponibilização ocorrerá apenas mediante: a) consulta ao(à)(s) autor(a)(es)(as) e ao(à) orientador(a); b) novo Termo de Ciência e de Autorização (TECA) assinado e inserido no arquivo do TCCG. O documento não será disponibilizado durante o período de embargo.

Casos de embargo:

- Solicitação de registro de patente;

- Submissão de artigo em revista científica;

- Publicação como capítulo de livro.

Obs.: Este termo deve ser assinado no SEI pelo orientador e pelo autor.

Documento assinado eletronicamente por **Lucas Brandão Rodrigues**, **Discente**, em 12/01/2025, às 14:44, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Fernando Marques Federson**, **Professor do Magistério Superior**, em 15/01/2025, às 16:22, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020 .

A autenticidade deste documento pode ser conferida no site https://sei.ufg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **5089793** e o código CRC **E079367E**.

Referência: Processo nº 23070.001593/2025-60 SEI nº 5089793

LUCAS BRANDÃO RODRIGUES

Aprendizagem por Reforço para Desenvolvimento de Jogos

Uma Abordagem prática com a Unity ML-Agents

Relatório final de Trabalho de Conclusão de Curso, apresentado à Universidade Federal de Goiás, como parte das exigências para a obtenção do título de Bacharel em Inteligência Artificial.

Orientador: Prof. Dr. Fernando Marques Federson

Goiânia

2025

Ficha de identificação da obra elaborada pelo autor, através do
Programa de Geração Automática do Sistema de Bibliotecas da UFG.

RODRIGUES, LUCAS BRANDÃO
Aprendizagem por Reforço para Desenvolvimento de Jogos [manuscrito] : Uma Abordagem prática com a Unity ML-Agents / LUCAS BRANDÃO RODRIGUES. - 2025.
217 f.

Orientador: Prof. Dr. Fernando Marques Federson.
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Goiás, Instituto de Informática (INF), Inteligência Artificial, Goiânia, 2025.

1. inteligência artificial. 2. aprendizado por reforço. 3. jogos. I. Federson, Fernando Marques , orient. II. Título.

CDU 004

LUCAS BRANDÃO RODRIGUES

Aprendizagem por Reforço para Desenvolvimento de Jogos

Uma abordagem prática com a Unity ML-Agents

Relatório final de Trabalho de Conclusão de Curso, apresentado à Universidade Federal de Goiás, como parte das exigências para a obtenção do título de Bacharel em Inteligência Artificial.

Data da Aprovação: 17 de dezembro de 2024.

FERNANDO MARQUES FEDERSON

Prof. Dr. Fernando Marques Federson
Orientador (INF-UFG)

Aldo André Díaz Salazar

Prof. Dr. Aldo André Díaz Salazar
Coordenador de TCC do BIA (INF-UFG)

Anderson da Silva Soares

Prof. Dr. Anderson da Silva Soares
Coordenador do BIA (INF-UFG)

Prof. Dr. Leonardo Antônio Alves
(INF-UFG)

LUCAS BRANDÃO RODRIGUES

Aprendizagem por Reforço para Desenvolvimento de Jogos

Uma Abordagem prática com a Unity ML-Agents

RESUMO

Este Relatório de Conclusão de Curso tem como objetivo reunir os resultados da minha jornada para me tornar um especialista em **Aprendizado por Reforço (Games)**. Uma ilustração e sua narrativa descrevem os períodos de trabalho. Os Apêndices contêm os Termos de Aceite de Entrega e os resultados obtidos durante cada período de trabalho.

Palavras-chave: inteligência artificial, modelos grandes de linguagem, geração automática de datasets.

ABSTRACT

This Course Completion Report aims to bring together the results of my journey to become an expert in **Reinforcement Learning (Games)**. An illustration and its narrative describe the work periods. The Appendices contain the Delivery Acceptance Terms and the results obtained during each work period.

Keywords: artificial intelligence, large language models, automatic dataset generation.

Goiânia

2025

Minha Jornada

Lucas Brandão Rodrigues

Especialista em: Reinforcement Learning Aplicado a Games

MINHA JORNADA

Nome: **Lucas Brandão Rodrigues**

Especialidade: **Aprendizado por Reforço (Games)**

Objetivo deste documento

Durante o processo da disciplina Residência em IA[1], foram gerados diversos resultados na construção da minha especialização. A cada semana, um conjunto de resultados foi formalizado por um Termo de Aceite de Entrega e avaliado por uma banca, considerando o planejado e o realizado para o período. Este documento tem como objetivo descrever esses resultados obtidos, fazendo referência aos Termos de Aceite de Entrega e seus documentos associados.

Minha Jornada

Durante a minha primeira "Jornada" no Bacharelado em Inteligência Artificial (BIA), tive contato com muitas áreas de aplicação de IA. Em especial, trabalhei com **NLP** em alguns projetos acadêmicos e depois com **Gamificação**. Tive a oportunidade de estudar outras áreas de aplicação como **Visão Computacional, Processamento de Áudio e Voz, além de Aprendizado Supervisionado, Não Supervisionado e por Reforço.** Após vivenciar o BIA em toda a sua "plenitude" (hehe), meus interesses foram aos poucos se revelando. Em especial, descobri que não preciso deixar de lado minha paixão por jogos (games), podendo explorar aspectos de Gamificação e IA aplicada a jogos nos projetos que participo, graças ao professor Leonardo.

Minha segunda Jornada se passou durante a Residência em IA e começou nas **Semanas 1 e 2**, quando me dediquei à definição do tema de pesquisa e ao planejamento inicial das atividades. Explorei materiais como os livros **AI for Games**, de Ian Millington, e **Artificial Intelligence and Games**, de Georgios N. Yannakakis e Julian Togelius, e me informei sobre tópicos de conferências como o SBGames e o CSCE2024. Por fim, decidi pelo tema **Aprendizado por Reforço Aplicado a Games**. A busca por tópicos de interesse e o planejamento inicial do processo da Residência podem ser encontrados no **Apêndice 1**.

A partir da definição do tema, iniciei, nas **Semanas 3 e 4**, uma Revisão Bibliográfica para explorar a história, evolução e o estado da arte em Aprendizado por Reforço (RL) aplicado a jogos. Durante esse período, levantei 53 artigos científicos, utilizando a

[1] Dez semanas, entre setembro de 2024 e dezembro de 2024.

metodologia de Screening, que foram reduzidos a 25 após uma triagem baseada em relevância e aplicabilidade. Para organizar essas informações, produzi um **catálogo detalhado** com resumos de conceitos, algoritmos e ferramentas abordados nos artigos mais importantes. Por fim, escolhi 10 artigos que considerei mais relevantes desses 25 para estudar mais a fundo e produzir resumos. O **Apêndice 2** apresenta a relação completa de artigos levantados, assim como o catálogo e os resumos.

Nas **Semanas 5 e 6**, aprofundei-me na pesquisa de frameworks populares e ferramentas que possibilitam a integração de RL com engines de jogos. Entre os frameworks analisados, estavam OpenAI Gym e PyTorch RL. Também explorei ferramentas específicas para engines como Unity ML-Agents e Unreal Engine AirSim. O **Apêndice 3** reúne o **catálogo técnico** que detalha como cada ferramenta pode ser utilizada no desenvolvimento de IA para jogos. Após estudar os frameworks mencionados, **escolhi o Unity ML-Agents como framework principal para o projeto.**

As **Semanas 7, 8 e 9** foram marcadas pela criação de um **material didático** abrangente sobre Unity ML-Agents. No **Apêndice 4**, são abordados desde os primeiros passos com o framework até conceitos fundamentais de RL, enquanto no **Apêndice 5** são explorados tópicos mais avançados como Imitation Learning e Curriculum Learning, complementando com exemplos práticos aplicados no Unity ML-Agents. Ao final desse período, o material alcançou mais de 100 páginas, tornando-se um guia abrangente e prático para futuros estudos e aplicações.

Na **Semana 10**, dediquei-me à implementação prática do Aprendizado por Reforço em jogos. Utilizei um clone do **Flappy Bird** como caso de estudo, desenvolvendo cenários de Imitation Learning e integração com o Unity ML-Agents. Para compartilhar os resultados, criei um repositório no GitHub[2] contendo todos os artefatos e implementações desenvolvidos ao longo da jornada.

Todas as entregas realizadas durante a Residência, destacando todo a Jornada que percorri, podem ser encontradas no **Apêndice 6**. Ao final dessa experiência, percebo que cada etapa foi essencial para consolidar meu conhecimento em Aprendizado por Reforço Aplicado a Games. A Jornada reforçou meu interesse pela integração de Inteligência Artificial com jogos e abriu caminhos para mais projetos no futuro. Sempre fui apaixonado por Games, e a perspectiva de conseguir unir esses dois campos me traz muita alegria. Espero que os materiais construídos durante esse trabalho sirvam de ajuda e inspiração para outros jovens como eu, que têm um sonho e precisam apenas de um “empurrãozinho” para começarem a voar.

[2] https://github.com/lucasbrandao4770/Reinforcement-Learning-with-Unity-ML-Agents

APÊNDICE 1

Termo de Aceite de Entrega

Objetivo deste documento

Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião ("gate") de aprovação: 19 de set. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

Lucas Brandão e Rafaela Mota

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

Durante esse primeiro stage, foram realizadas as seguintes atividades:

- Pesquisa inicial CSCE : Pesquisa de tópicos de interesse em conferências do CSCE2024;
- Stage 1 - 190924 : Pesquisa de tópicos de interesse com base nos seguintes meios:
  - Livro 1 - AI for Games , 2nd edition, do autor Ian Millington;
  - Livro 2 - Artificial Intelligence and Games , dos autores Georgios N. Yannakakis e Julian Togelius;
  - Congresso 1 - SBGames .

Ao final do stage, decidi como foco da minha pesquisa o tema: **"Aprendizado por Reforço Aplicado a Games".**

Planejamento: [descrever o que pretende fazer para realizar a próxima ENTREGA]

- **Realizar uma pesquisa sobre Aprendizado por Reforço em geral,** focando em uma visão breve da história e evolução do campo.
- **Levantar artigos, livros e estudos que tratem da aplicação de Aprendizado por Reforço em Games,** buscando entender a história e evolução do tema, além de aplicações atuais e estado da arte da área.
- **Produzir resumos dos artigos e tópicos pesquisados,** destacando as principais contribuições e avanços na área.
- **Pesquisar e levantar ferramentas e frameworks** usados para integração de IA com engines de jogos, focando nos modelos de Aprendizado por Reforço.
- **Elaborar uma lista de ferramentas identificadas,** com uma breve explicação de como são utilizadas no contexto dos temas de pesquisa.

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

A pesquisa por tópicos foi realizada em conjunto com a estudante Rafaela Mota. Em alguns dos documentos, pode ser notado que utilizamos cores para marcar e dar ênfase a certos tópicos. As cores usadas para indicar interesse particular foram:

- Lucas
- Rafaela

ACEITE DA ENTREGA:

CEDRIC LUIZ DE CARVALHO: Go!

Termo de Aceite de Entrega

Objetivo deste documento

Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião ("gate") de aprovação: 26 de set. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

Lucas Brandão e Rafaela Mota

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

Durante esse Stage (Stage 2 - 260924), foram realizadas as seguintes atividades:

- **Revisão Bibliográfica - RL :** Realização de uma busca por livros, artigos e publicações científicas sobre Aprendizado por Reforço, com foco na história e evolução do campo. Para isso, foi utilizada uma abordagem de revisão bibliográfica do tipo Screening, conforme apresentado pelo professor Federson em sala de aula.
- **Cronograma Residência em IA :** Com base no feedback do coach career e orientação do professor Federson, foi elaborado um planejamento geral para o processo da Residência em IA. **O planejamento servirá apenas como guia** para a execução das atividades, e está **sujeito a adaptações e mudanças** conforme o desenvolvimento do projeto.
- **Fundamentals of Artificial Intelligence (2020) :** Leitura do sumário e identificação dos capítulos relevantes para minha pesquisa.

Planejamento: [descrever o que pretende fazer para realizar a próxima ENTREGA]

Objetivos Principais:

- **Levantar artigos, livros e estudos que tratem da aplicação de Aprendizado por Reforço em Games,** através de uma revisão bibliográfica, buscando entender a história e evolução do tema, além de aplicações atuais e estado da arte da área.
- **Identificar e catalogar, de forma superficial, frameworks de aprendizado por reforço populares** (ex.: OpenAI Gym, TensorFlow Agents (TF-Agents), Unity ML-Agents, PyTorch RL).

Objetivos Adicionais:

- **Produzir resumos detalhados dos artigos e tópicos pesquisados,** destacando contribuições e avanços importantes.
- **Pesquisar e levantar ferramentas e frameworks usados para integração de IA com engines de jogos,** focando nos modelos de Aprendizado por Reforço.
- **Elaborar uma lista de ferramentas identificadas,** com uma breve explicação de como são utilizadas no contexto dos temas de pesquisa.

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

Após a realização do primeiro Stage/Gate, recebi feedback indicando que o planejamento inicial continha mais atividades do que o esperado para uma semana e fiz ajustes para tornar o planejamento mais explícito e realista em relação ao tempo disponível. Em especial, os Objetivos Adicionais são "opcionais" e podem conter atividades das semanas seguintes.

ACEITE DA ENTREGA:

CEDRIC LUIZ DE CARVALHO: Go!

Pesquisa Inicial

Durante a minha “Jornada” no Bacharelado em Inteligência Artificial (BIA), tive contato com muitas áreas de aplicação de IA. Em especial, trabalhei com **NLP** em alguns projetos do CEIA, e agora estou trabalhando com **Gamificação**. Tive a oportunidade de estudar outras áreas de aplicação como **Visão Computacional, Processamento de Áudio e Voz, além de Apredizado Supervisionado, Não Supervisionado e por Reforço.**

Após vivenciar o BIA em toda a sua “plenitude” (hehe), meus tópicos de interesse foram aos poucos se revelando. Em especial, descobri que não preciso deixar de lado minha paixão por jogos, podendo explorar aspectos de gamificação e IA aplicada a jogos nos projetos que participo, graças ao professor Leonardo.

Foi com isso em mente que cheguei no oitavo período, para a tão “temida” Residência em IA. Durante a pesquisa inicial de tópicos de interesse, proposta durante a segunda semana de aulas, tive dificuldade para encontrar tópicos que se encaixassem nos meus interesses dentro dos links propostos pelo professor Federson. Insatisfeito com minha pesquisa inicial, procurei outras conferências realizadas pelo CSCE, onde consegui encontrar mais tópicos que me interessaram.

A pesquisa de tópicos foi feita em conjunto com minha colega Rafaela Mota, pois começamos a Residência com interesses parecidos. Após conversarmos com o professor Federson sobre nossa visão da Residência em IA, percebemos que estávamos pensando muito em campo de aplicação, com uma visão muito ampla de “IA para Jogos”. Foi sugerido conferirmos alguns livros no assunto para tentarmos encontrar algo que “brilhasse os olhos” (rsrs), que nós gostaríamos de nos especializar e trabalhar no assunto, para nos tornarmos especialistas do tópico.

Tópicos de Interesse

Conseguimos acesso aos livros e começamos nossa pesquisa dos tópicos de interesse. Também checamos o congresso de games “SBGames”. Ao final da nossa pesquisa, selecionamos os seguintes tópicos de interesse:

26th International Conference on Artificial Intelligence

Topics of interest - ICAI'24

- **Natural language processing**
- Software tools for AI
- **Reinforcement learning**
- Aspects of natural language processing

20th International Conference on Data Science

Topics of interest - ICDATA'24

- **Large Language Models**
- **Data and knowledge representation**
- **Simulation and Modeling**
- Recommendation Systems

8th International Conference on Applied Cognitive Computing

Topics of interest - ACC'24

- Improving Cognition in machine learning systems
- **Reinforced learning**
- Dynamical learning systems
- **Information and Knowledge retrieval and searching algorithms**
- **Natural Language Processing**

FCS'24 - The 20th Int'l Conf on Foundations of Computer Science

Topics of interest - FCS

- **Game Theory and Methods**
- **Graph Algorithms**
- Randomness in Computing

Livro 1: AI For Games

Livro 1 - AI for Games

- Movement
- Pathfinding
- **Decision Trees**
- **Reinforcement Learning**
- **Artificial Neural Networks**
- **Game Theory**
- **Designing Game AI**
- Real Time Strategy
- Ecosystem Design

Livro 2: Artificial Intelligence and Games

Livro 2 - Artificial Intelligence and Games

- **Player Experience and Behavioral Data Analytics**
- Finite State Machines
- Behavior Trees
- Evolutionary Algorithms
- **Supervised Learning**
    - **Artificial Neural Networks**
    - **Support Vector Machines**
    - **Decision Tree Learning**
- **Reinforcement Learning**
- Neuroevolution
- **Game Design and AI Design Considerations**
    - **Characteristics of AI Algorithm Design**
- **Generating Content**

Livro 3: Fundamentals of Artificial Intelligence

Fundamentals of Artificial Intelligence (2020)

11) Adversarial Search and Game Theory

13) Machine Learning

Congresso 1: SBGames

Congresso 1 - SBGames 2024

- **Design de jogo**
- **Design de níveis em jogos**
- Narrativas de jogos
- **Tecnologias em jogos**
- Computer Games
- **AI for Games**
- **Game Development**
- **Game Design**
- Game Graphics
- **Algorithmic Game Theory**
- Immersive Games
- **Game Technology**
- **Game-based Learning**
- **Gamification**
- **Game Industry**
- **Experiência de jogo e estudos do jogador**
- **Aprendizagem baseada em jogos**
- **Avaliação de estratégias de gamificação**
- **Gamificação na educação**

Conclusão

Após essa busca inicial, algumas áreas se destacaram como interesses em potencial. Decidimos como foco da nossa pesquisa:

- Lucas: **“Aprendizado por Reforço Aplicado a Games”**
- Rafaela: **“Tradução Simultânea Usando Modelos de Linguagem Natural e Processamento de Áudio e Voz”**

Planejamento

As entregas da Residência em IA foram planejadas e guiadas por **3 fases principais:**

- **Fase 1 - História, fundamentos, ferramentas (Semanas 01~04):** Foco na pesquisa bibliográfica e na compreensão dos principais conceitos de aprendizado por reforço. Nessa fase, são estudadas a história, as bases teóricas e os diferentes tipos de algoritmos de reforço. Além disso, é realizada a escolha dos frameworks e ferramentas mais adequados para as fases seguintes.
- **Fase 2 - Frameworks (Semanas 05~06):** Exemplos práticos com os frameworks selecionados, incluindo o desenvolvimento de exemplos didáticos e a criação de artefatos que consolidem o domínio técnico sobre as ferramentas. Esta fase visa especializar o uso de ferramentas específicas, preparando a base para a aplicação prática na fase seguinte.
- **Fase 3 - Trilha "Reforço para Games" (Semanas 07~10):** Aplicação prática dos conhecimentos adquiridos, com o objetivo de criar artefatos concretos. Nesta fase, a teoria é aplicada "mão na massa", com exemplos práticos que podem incluir:
    - **Ideia 1**: Implementar aprendizado por reforço em um jogo inicialmente sem IA, demonstrar o funcionamento do agente inteligente e comparar o antes e o depois.
    - **Ideia 2**: Analisar o código de um jogo que já utiliza aprendizado por reforço, explicando as escolhas dos algoritmos e como eles influenciam o comportamento do agente.

APÊNDICE 2

Termo de Aceite de Entrega

Objetivo deste documento

Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião ("gate") de aprovação: 3 de out. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

Lucas Brandão

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

Durante esse Stage (Stage 3 - 031024), foram realizadas as seguintes atividades:

- **Levantamento de artigos e estudos que tratam da aplicação de Aprendizado por Reforço em Games,** através de uma revisão bibliográfica do tipo Screening, buscando entender a história e evolução do tema, além de aplicações atuais e estado da arte da área. **Foram levantados 53 artigos** durante esse processo.
  - Revisão Bibliográfica - Reinforcement Learning
  - Revisão Bibliográfica - RL for Games
  - Revisão Bibliográfica - AI for Games
- **Produção de um catálogo dos artigos levantados,** com destaque para as informações relevantes de cada artigo (ano de publicação, número de páginas, abstract, keywords), com o objetivo de facilitar referências e consultas futuras. Durante a produção do catálogo, **o número de artigos foi reduzido para 25,** levando em consideração a disponibilidade, relevância e aplicabilidade dos artigos para o projeto.
  - Catálogo de Artigos
- **Produção de resumos detalhados de três artigos relevantes em cada tópico pesquisado,** destacando os principais conceitos, algoritmos, ferramentas e técnicas abordados.
  - Resumos

Planejamento: [descrever o que pretende fazer para realizar a próxima ENTREGA]

Objetivos:

- **Catalogar frameworks de aprendizado por reforço populares** (ex.: OpenAI Gym, TensorFlow Agents (TF-Agents), Unity ML-Agents, PyTorch RL).
- **Pesquisar e levantar ferramentas e frameworks usados para integração de IA com engines de jogos,** focando nos modelos de Aprendizado por Reforço.
- **Elaborar uma lista de ferramentas identificadas,** com uma breve explicação de como são utilizadas no contexto dos temas de pesquisa.

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

ACEITE DA ENTREGA:

CEDRIC LUIZ DE CARVALHO: Go!

Pesquisa Bibliográfica

O foco da pesquisa foi levantamento de artigos e estudos que tratam da aplicação de Aprendizado por Reforço para games, nos tópicos de **Aprendizado por Reforço (RL), Aprendizado por Reforço aplicado a Games, e Inteligência Artificial aplicada a Games.** Para isso, foi realizada uma revisão bibliográfica do tipo Screening, tendo sido levantados **53 artigos** durante esse processo.

Os materiais encontrados foram organizados em um **catálogo** que destaca os principais conceitos, algoritmos, ferramentas e técnicas abordados, com o objetivo de **facilitar referências e consultas futuras.** Durante a produção do catálogo, **o número de artigos foi reduzido para 25,** levando em consideração a disponibilidade, relevância e aplicabilidade dos artigos para o projeto.

Foram selecionados 10 artigos para realização de uma análise mais aprofundada, com pelo menos 3 artigos de cada tópico**.** Para cada artigo, foi feito um estudo sistemático, com divisão das informações relevantes em **“Overview”, “Conceitos e Algoritmos”, “Ferramentas e Técnicas” e “Conclusões”.**

Tópicos de Pesquisa

Revisão Bibliográfica (Screening):

- Revisão Bibliográfica - Reinforcement Learning
- Revisão Bibliográfica - RL for Games
- Revisão Bibliográfica - AI for Games

Plataformas utilizadas:

- IEEE Xplore
- ACM Digital Library
- ScienceDirect
- Google Scholar

Reinforcement Learning

String de busca: ("Reinforcement Learning" OR "RL" OR "Reinforcement Learning Theory") AND ("history" OR "evolution" OR "development" OR "progress") AND ("review" OR "comprehensive review" OR "literature review" OR "systematic review" OR "survey" OR "state-of-the-art review" OR "overview")

From Classical to Quantum: A Review of Recent Progress in Reinforcement Learning
https://ieeexplore.ieee.org/document/9456218

Transfer Learning in Deep Reinforcement Learning: A Survey
https://ieeexplore.ieee.org/document/10172347

Review of Deep Reinforcement Learning
https://ieeexplore.ieee.org/document/10020015

Multiobjective Reinforcement Learning: A Comprehensive Overview
https://ieeexplore.ieee.org/document/6918520

Reinforcement Learning Methods for Computation Offloading: A Systematic Review
https://dl.acm.org/doi/10.1145/3603703

Causal Question Answering with Reinforcement Learning
https://dl.acm.org/doi/10.1145/3589334.3645610

Explainable Reinforcement Learning: A Survey and Comparative Review
https://dl.acm.org/doi/10.1145/3616864

Redefining Counterfactual Explanations for Reinforcement Learning: Overview, Challenges and Opportunities
https://dl.acm.org/doi/10.1145/3648472

Deep Reinforcement Learning Verification: A Survey
https://dl.acm.org/doi/10.1145/3596444

Reinforcement Learning in Natural Language Processing: A Survey
https://dl.acm.org/doi/10.1145/3639479.3639496

A Systematic Literature Review of Reinforcement Learning-based Knowledge Graph Research
https://www.sciencedirect.com/science/article/pii/S0957417423023825

Comprehensive survey on reinforcement learning-based task offloading techniques in aerial edge computing
https://www.sciencedirect.com/science/article/abs/pii/S254266052400283X

Deep Reinforcement Learning: An Overview
https://arxiv.org/abs/1701.07274

Reinforcement Learning – Overview of recent progress and implications for process control
https://www.sciencedirect.com/science/article/abs/pii/S0098135419300754

Deep Reinforcement Learning: A Brief Survey
https://ieeexplore.ieee.org/abstract/document/8103164

Reinforcement Learning: A Survey
https://www.jair.org/index.php/jair/article/view/10166

A Review of Safe Reinforcement Learning: Methods, Theory and Applications
https://arxiv.org/abs/2205.10330

RL for Games

String de busca: ("Reinforcement Learning" OR "RL") AND ("games" OR "video games" OR "game AI" OR "game development" OR "game design" OR "gaming") AND ("applications" OR "implementation" OR "survey" OR "review" OR "overview" OR "state-of-the-art" OR "framework" OR "techniques")

A Survey of Deep Reinforcement Learning in Video Games
https://arxiv.org/abs/1912.10944

A Survey of Reinforcement Learning Toolkits for Gaming: Applications, Challenges and Trends
https://link.springer.com/chapter/10.1007/978-3-031-18461-1_11

OpenSpiel: A Framework for Reinforcement Learning in Games
https://arxiv.org/abs/1908.09453

Deep Reinforcement Learning based Video Games: A Review
https://ieeexplore.ieee.org/abstract/document/9781752

Reinforcement Learning in Games
https://link.springer.com/chapter/10.1007/978-3-642-27645-3_17

A Deep Reinforcement Learning Agent for General Video Game AI Framework Games
https://ieeexplore.ieee.org/document/9844524

Implementation of Reinforcement Learning in 2D Based Games Using Open AI Gym
https://ieeexplore.ieee.org/document/10017810

Distilling Reinforcement Learning Tricks for Video Games
https://ieeexplore.ieee.org/document/9618997

A Survey of Deep Reinforcement Learning in Game Playing
https://ieeexplore.ieee.org/document/10575819

Evolutionary game theory combined with reinforcement learning synthesis - A comprehensive survey
https://ieeexplore.ieee.org/document/10587977

Analysis of Artificial Intelligence Applied in Video Games
https://ieeexplore.ieee.org/document/9361299

Immersive Game Worlds: Using Deep Reinforcement Learning for Lifelike Non-Player Characters
https://ieeexplore.ieee.org/document/10559740

The Implementation of Reinforcement Learning Algorithm for AI Bot in Fighting Video Game
https://ieeexplore.ieee.org/document/9567749

Towards safe and sustainable reinforcement learning for real-time strategy games
https://www.sciencedirect.com/science/article/pii/S0020025524008946

A framework for designing Reinforcement Learning agents with Dynamic Difficulty Adjustment in single-player action video games
https://www.sciencedirect.com/science/article/abs/pii/S1875952124000545

AI for Games

String de busca: ("Artificial Intelligence" OR "AI" OR "Game AI" OR "AI techniques") AND ("games" OR "video games" OR "game development" OR "game design" OR "gaming" OR "game AI") AND ("applications" OR "implementation" OR "survey" OR "review" OR "overview" OR "state-of-the-art" OR "framework" OR "techniques" OR "methods")

Current AI in games : a review
https://eprints.qut.edu.au/45741/

Review of the Use of AI Techniques in Serious Games: Decision Making and Machine Learning
https://ieeexplore.ieee.org/abstract/document/7366548

Artificial Intelligence for Adaptive Computer Games
https://cdn.aaai.org/FLAIRS/2007/FLAIRS07-007.pdf

Recent Research on AI in Games
https://ieeexplore.ieee.org/abstract/document/9148327

Artificial Intelligence in Video Games: Towards a Unified Framework
https://onlinelibrary.wiley.com/doi/full/10.1155/2015/271296

Game AI: artificial intelligence for 3D path finding
https://ieeexplore.ieee.org/document/1414592

The Ethics of AI in Games
https://ieeexplore.ieee.org/document/10125072

The Working Principle of Artificial Intelligence in Video Games
https://ieeexplore.ieee.org/document/10223491

General general game AI
https://ieeexplore.ieee.org/document/7860385

Large Language Models and Video Games: A Preliminary Scoping Review
https://dl.acm.org/doi/10.1145/3640794.3665582

Game AI techniques from algorithmic approach to machine learning
https://dl.acm.org/doi/10.1145/3277644.3277792

Game AI hyperparameter tuning in rinascimento

https://dl.acm.org/doi/10.1145/3319619.3326842

Rethinking dynamic difficulty adjustment for video game design
https://www.sciencedirect.com/science/article/abs/pii/S1875952124000314

Game-changing intelligence: Unveiling the societal impact of artificial intelligence in game software
https://www.sciencedirect.com/science/article/abs/pii/S1875952124002301

Review and analysis of research on Video Games and Artificial Intelligence: a look back and a step forward
https://www.sciencedirect.com/science/article/pii/S1877050922007761

Using Machine Learning to Predict Game Outcomes Based on Player-Champion Experience in League of Legends
https://dl.acm.org/doi/10.1145/3472538.3472579

Catálogo de Artigos

Catálogo de Artigos

Para cada artigo, foram destacados:

- Link para o PDF do artigo
- Link original para a página onde encontrei o artigo
- Título
- Ano de Publicação
- Número de páginas
- Abstract
- Keywords

Reinforcement Learning

Deep Reinforcement Learning: A Brief Survey

Deep Reinforcement Learning - A Brief Survey.pdf

Link: https://ieeexplore.ieee.org/abstract/document/8103164

Year: 2017

Pages: 13

Keywords: Artificial intelligence, Signal processing algorithms, Visualization, Machine learning, Learning (artificial intelligence), Neural networks.

Abstract: Deep reinforcement learning (DRL) is poised to revolutionize the field of artificial intelligence (AI) and represents a step toward building autonomous systems with a higher-level understanding of the visual world. Currently, deep learning is enabling reinforcement learning (RL) to scale to problems that were previously intractable, such as learning to play video games directly from pixels. DRL algorithms are also applied to robotics, allowing control policies for robots to be learned directly from camera inputs in the real world. In this survey, we begin with an introduction to the general field of RL, then progress to the main streams of value-based and policy-based methods. Our survey will cover central algorithms in deep RL, including the deep Q-network (DQN), trust region policy optimization (TRPO), and asynchronous advantage actor critic. In parallel, we highlight the unique advantages of deep neural networks, focusing on visual understanding via RL. To conclude, we describe several current areas of research within the field.

Deep Reinforcement Learning: An Overview

Deep Reinforcement Learning - An Overview.pdf

Link: https://arxiv.org/abs/1701.07274

Year: 2018

Pages: 85

Abstract: We give an overview of recent exciting achievements of deep reinforcement learning (RL). We discuss six core elements, six important mechanisms, and twelve applications. We start with background of machine learning, deep learning and reinforcement learning. Next we discuss core RL elements, including value function, in particular, Deep Q-Network (DQN), policy, reward, model, planning, and exploration. After that, we discuss important mechanisms for RL, including attention and memory, unsupervised learning, transfer learning, multi-agent RL, hierarchical RL, and learning to learn. Then we discuss various applications of RL, including games, in particular, AlphaGo, robotics, natural language processing, including dialogue systems, machine translation, and text generation, computer vision, neural architecture design, business management, finance, healthcare, Industry 4.0, smart grid, intelligent transportation systems, and computer systems. We mention topics not reviewed yet, and list a collection of RL resources. After presenting a brief summary, we close with discussions.

A Review of Safe Reinforcement Learning: Methods, Theory and Applications

A Review of Safe Reinforcement Learning - Methods, Theory and Applications.pdf

Link: https://arxiv.org/abs/2205.10330

Year: 2024

Pages: 97

Abstract: Reinforcement Learning (RL) has achieved tremendous success in many complex decision-making tasks. However, safety concerns are raised during deploying RL in real-world applications, leading to a growing demand for safe RL algorithms, such as in autonomous driving and robotics scenarios. While safe control has a long history, the study of safe RL algorithms is still in the early stages. To establish a good foundation for future safe RL research, in this paper, we provide a review of safe RL from the perspectives of methods, theories, and applications. Firstly, we review the progress of safe RL from five dimensions and come up with five crucial problems for safe RL being deployed in real-world applications, coined as "2H3W". Secondly, we analyze the algorithm and theory progress from the perspectives of answering the "2H3W" problems. Particularly, the sample complexity of safe RL algorithms is reviewed and discussed, followed by an introduction to the applications and benchmarks of safe RL algorithms. Finally, we open the discussion of the challenging problems in safe RL, hoping to inspire future research on this thread. To advance the study of safe RL algorithms, we release an open-sourced repository containing the implementations of major safe RL algorithms at the link.

Reinforcement Learning: A Survey

Reinforcement Learning - A Survey.pdf

Link: https://www.jair.org/index.php/jair/article/view/10166

Year: 1996

Pages: 49

Abstract: This paper surveys the field of reinforcement learning from a computer-science perspective. It is written to be accessible to researchers familiar with machine learning. Both the historical basis of the field and a broad selection of current work are summarized. Reinforcement learning is the problem faced by an agent that learns behavior through trial-and-error interactions with a dynamic environment. The work described here has a resemblance to work in psychology, but differs considerably in the details and in the use of the word ``reinforcement." The paper discusses central issues of reinforcement learning, including trading off exploration and exploitation, establishing the foundations of the field via Markov decision theory, learning from delayed reinforcement, constructing empirical models to accelerate learning, making use of generalization and hierarchy, and coping with hidden state. It concludes with a survey of some implemented systems and an assessment of the practical utility of current methods for reinforcement learning.

Multiobjective Reinforcement Learning: A Comprehensive Overview

Multiobjective Reinforcement Learning - A Comprehensive Overview.pdf

Link: https://ieeexplore.ieee.org/document/6918520

Year: 2015

Pages: 14

Keywords: Markov decision process (MDP), Multiobjective reinforcement learning (MORL), Pareto front, Reinforcement learning (RL), Sequential decision-making.

Abstract: Reinforcement learning (RL) is a powerful paradigm for sequential decision-making under uncertainties, and most RL algorithms aim to maximize some numerical value which represents only one long-term objective. However, multiple long-term objectives are exhibited in many real-world decision and control systems, so recently there has been growing interest in solving multiobjective reinforcement learning (MORL) problems where there are multiple conflicting objectives. The aim of this paper is to present a comprehensive overview of MORL. The basic architecture, research topics, and naïve solutions of MORL are introduced at first. Then, several representative MORL approaches and some important directions of recent research are comprehensively reviewed. The relationships between MORL and other related research are also discussed, which include multiobjective optimization, hierarchical RL, and multiagent RL. Moreover, research challenges and open problems of MORL techniques are suggested.

Explainable Reinforcement Learning: A Survey and Comparative Review

Explainable Reinforcement Learning - A Survey and Comparative Review.pdf

Link: https://dl.acm.org/doi/10.1145/3616864

Year: 2024

Pages: 36

Keywords: Reinforcement learning, Explainable reinforcement learning, Interpretability, Explainability.

Abstract: Explainable reinforcement learning (XRL) is an emerging subfield of explainable machine learning that has attracted considerable attention in recent years. The goal of XRL is to elucidate the decision-making process of reinforcement learning (RL) agents in sequential decision-making settings. Equipped with this information, practitioners can better understand important questions about RL agents (especially those deployed in the real world), such as what the agents will do and why. Despite increased interest, there exists a gap in the literature for organizing the plethora of papers—especially in a way that centers the sequential decision-making nature of the problem. In this survey, we propose a novel taxonomy for organizing the XRL literature that prioritizes the RL setting. We propose three high-level categories: feature importance, learning process and Markov decision process, and policy-level. We overview techniques according to this taxonomy, highlighting challenges and opportunities for future work. We conclude by using these gaps to motivate and outline a roadmap for future work.

Deep Reinforcement Learning Verification: A Survey

Deep Reinforcement Learning Verification - A Survey.pdf

Link: https://dl.acm.org/doi/10.1145/3596444

Year: 2023

Pages: 31

Keywords: Deep reinforcement learning, Neural network verification, Trustworthy reinforcement learning.

Abstract: Deep reinforcement learning (DRL) has proven capable of superhuman performance on many complex tasks. To achieve this success, DRL algorithms train a decision-making agent to select the actions that maximize some long-term performance measure. In many consequential real-world domains, however, optimal performance is not enough to justify an algorithm's use—for example, sometimes a system's robustness, stability, or safety must be rigorously ensured. Thus, methods for verifying DRL systems have emerged. These algorithms can guarantee a system's properties over an infinite set of inputs, but the task is not trivial. DRL relies on deep neural networks (DNNs). DNNs are often referred to as "black boxes" because examining their respective structures does not elucidate their decision-making processes. Moreover, the sequential nature of the problems DRL is used to solve promotes significant scalability challenges. Finally, because DRL environments are often stochastic, verification methods must account for probabilistic behavior. To address these complications, a new subfield has emerged. In this survey, we establish the foundations of DRL and DRL verification, define a taxonomy for DRL verification methods, describe approaches for dealing with stochasticity, characterize considerations related to writing specifications, enumerate common testing tasks/environments, and detail opportunities for future research.

Causal Question Answering with Reinforcement Learning

Causal Question Answering with Reinforcement Learning.pdf

Link: https://dl.acm.org/doi/10.1145/3589334.3645610

Year: 2024

Pages: 12

Keywords: Causal networks, Causal reasoning and diagnostics, Question answering, Causality graphs, Reinforcement learning.

Abstract: Causal questions inquire about causal relationships between different events or phenomena. They are important for a variety of use cases, including virtual assistants and search engines. However, many current approaches to causal question answering cannot provide explanations or evidence for their answers. Hence, in this paper, we aim to answer causal questions with a causality graph, a large-scale dataset of causal relations between noun phrases along with the relations' provenance data. Inspired by recent, successful applications of reinforcement learning to knowledge graph tasks, such as link prediction and fact-checking, we explore the application of reinforcement learning on a causality graph for causal question answering. We introduce an Actor-Critic-based agent which learns to search through the graph to answer causal questions. We bootstrap the agent with a supervised learning procedure to deal with large action spaces and sparse rewards. Our evaluation shows that the agent successfully prunes the search space to answer binary causal questions by visiting less than 30 nodes per question compared to over 3,000 nodes by a naive breadth-first search. Our ablation study indicates that our supervised learning strategy provides a strong foundation upon which our reinforcement learning agent improves. The paths returned by our agent explain the mechanisms by which a cause produces an effect. Moreover, for each edge on a path, our causality graph provides its original source allowing for easy verification of paths.

Redefining Counterfactual Explanations for Reinforcement Learning: Overview, Challenges and Opportunities

Redefining Counterfactual Explanations for Reinforcement Learning - Overview, Challe…

Link: https://dl.acm.org/doi/10.1145/3648472

Year: 2024

Pages: 33

Keywords: Reinforcement learning, Explainability, Interpretability, Counterfactual explanations.

Abstract: While AI algorithms have shown remarkable success in various fields, their lack of transparency hinders their application to real-life tasks. Although explanations targeted at non-experts are necessary for user trust and human-AI collaboration, the majority of explanation methods for AI are focused on developers and expert users. Counterfactual explanations are local explanations that offer users advice on what can be changed in the input for the output of the black-box model to change. Counterfactuals are user-friendly and provide actionable advice for achieving the desired output from the AI system. While extensively researched in supervised learning, there are few methods applying them to reinforcement learning (RL). In this work, we explore the reasons for the underrepresentation of a powerful explanation method in RL. We start by reviewing the current work in counterfactual explanations in supervised learning. Additionally, we explore the differences between counterfactual explanations in supervised learning and RL and identify the main challenges that prevent the adoption of methods from supervised in reinforcement learning. Finally, we redefine counterfactuals for RL and propose research directions for implementing counterfactuals in RL.

Reinforcement Learning – Overview of Recent Progress and Implications for Process Control

Reinforcement Learning – Overview of recent progress and implications for process co…

Link: https://www.sciencedirect.com/science/article/abs/pii/S0098135419300754

Year: 13

Pages: 2019

Keywords: Reinforcement learning, Mathematical programming, Model predictive control, Process control, Strategic/operational decision-making.

Abstract: This paper provides an introduction to Reinforcement Learning (RL) technology, summarizes recent developments in this area, and discusses their potential implications for the field of process control, and more generally, of operational decision-making. The paper begins with an introduction to RL that allows an agent to learn, through trial and error, the best way to accomplish a task. We then highlight new developments in RL that have led to the recent wave of applications and media interest. A comparison of the key features of RL and mathematical programming based methods (e.g., model predictive control) is then presented to clarify their similarities and differences. This is followed by an assessment of several ways that RL technology can potentially be used in process control and operational decision applications. A final section summarizes our conclusions and lists directions for future RL research that may improve its relevance for the process systems engineering field.

A Systematic Literature Review of Reinforcement Learning-based Knowledge Graph Research

A Systematic Literature Review of Reinforcement Learning-based Knowledge Graph R…

Link: https://www.sciencedirect.com/science/article/pii/S0957417423023825

Year: 2024

Pages: 17

Keywords: Knowledge graphs, Reinforcement learning, Systematic literature review, Markov decision processes.

Abstract: Knowledge graphs (KGs) model entities or concepts and their relations in a structural manner. The incompleteness has turned out to be the main challenge that hinders the application of KGs. Recently, reinforcement learning (RL) has been recognized as an effective method to deal with such a challenge, which models research tasks into a sequence decision problem without labels. Although an increasing number of studies investigate and analyze KGs using RL, there lacks a systematic literature review that comprehensively and quantitatively analyzes the landscape of RL-based KG research (RL-KG for short). As a result, researchers may have encountered difficulties in appropriately adopting RL techniques in KG research, even reinventing the wheels. In this paper, we follow the Systematic Literature Review (SLR) methodology to survey, screen, and investigate papers of RL-KG. Specifically, we identify 109 highly related papers from 1542, and systematically investigate them with regard to the following five research questions: (1) to what extent RL-KG have been investigated; (2) what application domains have been covered; (3) what RL techniques have been mainly considered; (4) whether there is a connection between the influence and reproducibility of these papers; (5) what specialized datasets, evaluation metrics, and publication venues have been applied. Through an in-depth analysis of the review results, we systematically and comprehensively identify some significant phenomena and analyze the reasons and difficulties of these phenomena. Based on such analysis, we tentatively propose promising future research topics to promote the RL-KG.

RL for Games

A Survey of Deep Reinforcement Learning in Video Games

A Survey of Deep Reinforcement Learning in Video Games.pdf

Link: https://arxiv.org/abs/1912.10944

Year: 2019

Pages: 13

Keywords: Reinforcement learning, Deep learning, Deep reinforcement learning, Game AI, Video games.

Abstract: Deep reinforcement learning (DRL) has made great achievements since proposed. Generally, DRL agents receive high-dimensional inputs at each step, and make actions according to deep-neural-network-based policies. This learning mechanism updates the policy to maximize the return with an end-to-end method. In this paper, we survey the progress of DRL methods, including value-based, policy gradient, and model-based algorithms, and compare their main techniques and properties. Besides, DRL plays an important role in game artificial intelligence (AI). We also take a review of the achievements of DRL in various video games, including classical Arcade games, first-person perspective games and multi-agent real-time strategy games, from 2D to 3D, and from single-agent to multi-agent. A large number of video game AIs with DRL have achieved super-human performance, while there are still some challenges in this domain. Therefore, we also discuss some key points when applying DRL methods to this field, including exploration-exploitation, sample efficiency, generalization and transfer, multi-agent learning, imperfect information, and delayed spare rewards, as well as some research directions.

OpenSpiel: A Framework for Reinforcement Learning in Games

OpenSpiel - A Framework for Reinforcement Learning in Games.pdf

Link: https://arxiv.org/abs/1908.09453

Year: 2020

Pages: 27

Abstract: OpenSpiel is a collection of environments and algorithms for research in general reinforcement learning and search/planning in games. OpenSpiel supports n-player (single- and multi- agent) zero-sum, cooperative and general-sum, one-shot and sequential, strictly turn-taking and simultaneous-move, perfect and imperfect information games, as well as traditional multiagent environments such as (partially- and fully-observable) grid worlds and social dilemmas. OpenSpiel also includes tools to analyze learning dynamics and other common evaluation metrics. This document serves both as an overview of the code base and an introduction to the terminology, core concepts, and algorithms across the fields of reinforcement learning, computational game theory, and search.

Analysis of Artificial Intelligence Applied in Video Games

Analysis of Artificial Intelligence Applied in Video Games.pdf

Link: https://ieeexplore.ieee.org/document/9361299

Year: 4

Pages: 2020

Keywords: Deep reinforcement learning, Game AI, Videogames, Board games, Sample efficiency, Delayed & sparse rewards.

Abstract: Deep reinforcement learning is growing faster than ever before in the artificial intelligence world. As its applications and accomplishments proliferate, the challenges it faces will also increase in number. In this paper, the author focuses on deep reinforcement learning being applied to the field of games and video games. S some of the achievements in various types of games, from 2D perfect information environment (board games) to 3D imperfect information environment (first person perspective games) and the various challenges that DRL faces when being applied to the field will be analyzed. Challenges such as sample efficiency, exploration & exploitation trade-off, and delayed & sparse rewards will be discussed. In addition, some solutions are suggested. The solutions might also be applied to the various game examples in order to improve their performance.

Towards Safe and Sustainable Reinforcement Learning for Real-Time Strategy Games

Towards safe and sustainable reinforcement learning for real-time strategy games.pdf

Link: https://www.sciencedirect.com/science/article/pii/S0020025524008946

Year: 2024

Pages: 25

Keywords: Reinforcement learning, Markov decision processes, Neural networks, State-space models, Model-based reinforcement learning, Risk-aware RL, Mission-critical safety.

Abstract: Combining Deep Neural Networks with Reinforcement Learning, known as Deep Reinforcement Learning (DRL), is revolutionizing fields like medicine, industry, and gaming. DRL has achieved groundbreaking results, particularly in complex Real-Time Strategy (RTS) games such as StarCraft II and Dota 2, serving as benchmarks for testing RL algorithms' robustness and safety.

Despite these successes, DRL algorithms face challenges, including high computational costs and a lack of safety-aware approaches. Training these algorithms requires extensive computational resources, leading to a significant divide between algorithms developed on supercomputers and those feasible on standard hardware. This also raises sustainability concerns due to increased CO2 emissions. Additionally, most RL algorithms are risk-neutral, limiting their deployment in safety-critical systems.

We present a novel model-based DRL approach, the Safe Observations Rewards Actions Costs Learning Ensemble (S-ORACLE), to address these challenges. S-ORACLE balances robust safety awareness with minimized risk and computational efficiency. Empirical validation across complex game environments—Deep RTS, ELF: MiniRTS, MicroRTS, Deep Warehouse, and StarCraft II—demonstrates that S-ORACLE outperforms state-of-the-art methods by significantly improving safety performance, reducing computational costs, and lowering environmental impact, while maintaining high efficiency and adaptability in training.

AI for Games

Current AI in Games: A Review

Current AI in games - a review.pdf
Link: https://eprints.qut.edu.au/45741/
Year: 2002
Pages: 19
Keywords: Computer games, Artificial intelligence.
Abstract: As the graphics race subsides and gamers grow weary of predictable and deterministic game characters, game developers must put aside their “old faithful” finite state machines and look to more advanced techniques that give the users the gaming experience they crave. The next industry breakthrough will be with characters that behave realistically and that can learn and adapt, rather than more polygons, higher resolution textures and more frames-per-second. This paper explores the various artificial intelligence techniques that are currently being used by game developers, as well as techniques that are new to the industry. The techniques covered in this paper are finite state machines, scripting, agents, flocking, fuzzy logic and fuzzy state machines decision trees, neural networks, genetic algorithms and extensible AI. This paper introduces each of these technique, explains how they can be applied to games and how commercial games are currently making use of them. Finally, the effectiveness of these techniques and their future role in the industry are evaluated.

Artificial Intelligence for Adaptive Computer Games

Artificial Intelligence for Adaptive Computer Games.pdf
Link: https://cdn.aaai.org/FLAIRS/2007/FLAIRS07-007.pdf
Year: 2007
Pages: 8
Abstract: Computer games are an increasingly popular application for Artificial Intelligence (AI) research, and conversely, AI is an increasingly popular selling point for commercial games. Although games are typically associated with entertainment, there are many "serious" applications of gaming, including military, corporate, and advertising applications. There are also so-called "humane" gaming applications for medical training, educational games, and games that reflect social consciousness or advocate for a cause. Game AI is the effort of going beyond scripted interactions, however complex, into the arena of truly interactive systems that are responsive, adaptive, and intelligent. Such systems learn about the player(s) during gameplay, adapt their own behaviors beyond the pre-programmed set provided by the game author, and interactively develop and provide a richer experience to the player(s).
The long-term goal of our research is to develop artificial intelligence techniques that can have a significant impact on the game industry. In this paper, we present a list of challenges and research opportunities in developing techniques that can be used by computer game developers. We discuss three Case-Based Reasoning (CBR) (Aamodt & Plaza, 1994; Kolodner, 1993) approaches to achieve adaptability in games: automatic behavior adaptation for believable characters, drama management and user modeling for interactive stories, and strategic behavior planning for real-time strategy games.

Recent Research on AI in Games

Recent Research on AI in Games.pdf

Link: https://ieeexplore.ieee.org/abstract/document/9148327

Year: 2020

Pages: 6

Keywords: Game, Artificial intelligence (AI), Game AI.

Abstract: Games tend to have the properties of vast state space and high complexity, making them excellent benchmarks for evaluating various techniques, including AI ones. Techniques utilized in games capable of making them more attractive, immersive, smarter etc. can all be considered to be certain forms of game AI. Considering there are few reviews on the more recent work in the game AI field from the perspective of essential applications, in this paper, we make a systematic review of typical research from 2018 on three application fields of game AI: believable agents in non-player characters research, game level generation in procedural content generation, and player profiling in player modeling. We also provide a timeline of game AI history to give the readers a clearer picture of the game AI field. Moreover, general game AI and hybrid intelligence for games are discussed.

Artificial Intelligence in Video Games: Towards a Unified Framework

Artificial Intelligence in Video Games - Towards a Unified Framework.pdf

Link: https://onlinelibrary.wiley.com/doi/full/10.1155/2015/271296

Year: 2015

Pages: 30

Abstract: With modern video games frequently featuring sophisticated and realistic environments, the need for smart and comprehensive agents that understand the various aspects of complex environments is pressing. Since video game AI is often specifically designed for each game, video game AI tools currently focus on allowing video game developers to quickly and efficiently create specific AI. One issue with this approach is that it does not efficiently exploit the numerous similarities that exist between video games not only of the same genre, but of different genres too, resulting in a difficulty to handle the many aspects of a complex environment independently for each video game. Inspired by the human ability to detect analogies between games and apply similar behavior on a conceptual level, this paper suggests an approach based on the use of a unified conceptual framework to enable the development of conceptual AI which relies on conceptual views and actions to define basic yet reasonable and robust behavior. The approach is illustrated using two video games, Raven and StarCraft: Brood War.

Game AI: Artificial Intelligence for 3D Path Finding

Game AI - artificial intelligence for 3D path finding.pdf

Link: https://ieeexplore.ieee.org/document/1414592

Year: 2004

Pages: 4

Keywords: Artificial intelligence, Intelligent systems, Smart cameras, Application software, Scalability, Programming profession, Engines, Intelligent vehicles.

Abstract: The role of artificial intelligence (AI) in games is gaining importance and often affects the success or failure of a game. In this paper, we investigate the use of AI in game development. Research is done on how AI can be applied in games, and the advantages it brings along. As the fields of AI in game development are too wide to be covered, the focus of this project is placed on certain areas. Two programs are implemented through this project - (i) an intelligent camera system, and (ii) path-finding in a 3D application.

General general game AI

General general game AI.pdf

Link: https://ieeexplore.ieee.org/document/7860385

Year: 2016

Pages: 8

Keywords: Games, Artificial intelligence, Benchmark testing, Conferences, Software, Computational intelligence.

Abstract: Arguably the grand goal of artificial intelligence research is to produce machines with general intelligence: the capacity to solve multiple problems, not just one. Artificial intelligence (AI) has investigated the general intelligence capacity of machines within the domain of games more than any other domain given the ideal properties of games for that purpose: controlled yet interesting and computationally hard problems. This line of research, however, has so far focused solely on one specific way of which intelligence can be applied to games: playing them. In this paper, we build on the general game-playing paradigm and expand it to cater for all core AI tasks within a game design process. That includes general player experience and behavior modeling, general non-player character behavior, general AI-assisted tools, general level generation and complete game generation. The new scope for general general game AI beyond game-playing broadens the applicability and capacity of AI algorithms and our understanding of intelligence as tested in a creative domain that interweaves problem solving, art, and engineering.

Game AI Techniques from Algorithmic Approach to Machine Learning

Game AI techniques from algorithmic approach to machine learning.pdf

Link: https://dl.acm.org/doi/10.1145/3277644.3277792

Year: 2018

Pages: 491 (Slides)

Abstract: Since 2004, I have been developing game AI for many titles in AAA titles:

- Chrome Hounds (Xbox360®)
- Demon's Souls (PS3®)
- Armored Core V (Xbox360®/PS3®)
- Final Fantasy XIV: A Realm Reborn
- Final Fantasy XV

Review and Analysis of Research on Video Games and Artificial Intelligence: A Look Back and a Step Forward

Review and analysis of research on Video Games and Artificial Intelligence - a look ba…

Link: https://www.sciencedirect.com/science/article/pii/S1877050922007761

Year: 2022

Pages: 9

Keywords: Computer games, AI, Games, Bibliometric analysis, Virtual reality (VR).

Abstract: This article shows the intimate relationship between Artificial Intelligence (AI) and video games research in 13 categories of analysis based on a bibliometric survey carried out in the Scopus database. We first briefly reviewed the relation between video games and AI. Then, we introduced the methodology of literature collection, presented and discussed the query, as well the flow of data treatment in the applications and plugins used. Since the article is concerned with a historical point of view of the relationship between digital games and AI the results were many and, therefore, we focused on the top 10 of each ranking, and discussed these results separately. Finally, we discuss the limitations of our review, proposing future research directions for scholars.

Resumos

Resumos

Artigos selecionados:

- Deep Reinforcement Learning: A Brief Survey
- Reinforcement Learning – Overview of Recent Progress and Implications for Process Control
- Multiobjective Reinforcement Learning: A Comprehensive Overview
- A Survey of Deep Reinforcement Learning in Video Games
- OpenSpiel: A Framework for Reinforcement Learning in Games
- Analysis of Artificial Intelligence Applied in Video Games
- Current AI in Games: A Review
- Artificial Intelligence for Adaptive Computer Games
- Recent Research on AI in Games
- Review and Analysis of Research on Video Games and Artificial Intelligence: A Look Back and a Step Forward

Deep Reinforcement Learning: A Brief Survey

Deep Reinforcement Learning - A Brief Survey.pdf

Overview: O artigo aborda como o Aprendizado por Reforço Profundo (DRL) tem revolucionado o campo da Inteligência Artificial, permitindo que o aprendizado por reforço (RL) escale para tarefas de alta complexidade, como jogos de vídeo e controle de robôs. O DRL combina aprendizado profundo (DL) com RL, possibilitando que agentes aprendam a partir de entradas complexas, como imagens, e tomem decisões baseadas em recompensas. A pesquisa foca em algoritmos chave do DRL, como:

- **Deep Q-Network (DQN)**
- **Trust Region Policy Optimization (TRPO)**
- **Asynchronous Advantage Actor-Critic (A3C)**

Esses algoritmos aproveitam redes neurais profundas para aprender representações eficientes de dados de alta dimensão, resolvendo problemas complexos diretamente de entradas visuais e ambientes simulados.

Conceitos e Algoritmos Principais:

1. **Deep Q-Network (DQN)**:
    - **Objetivo**: Ensinar agentes a jogar jogos de Atari a partir de entradas visuais (imagens).
    - **Mecanismo**: Usa uma rede neural convolucional (CNN) combinada com Q-learning para estimar valores de ação a partir de estados visuais.
    - **Inovações**:
        - **Experience Replay**: Amostra experiências passadas para quebrar correlações temporais e estabilizar o aprendizado.
        - **Redes Alvo (Target Networks)**: Mantém uma rede com pesos fixos para estabilizar as estimativas de Q-valores.
    - **Aplicações**: Jogar jogos com entradas visuais e tarefas reais com feedback visual, como navegação autônoma.
2. **Trust Region Policy Optimization (TRPO)**:
    - **Objetivo**: Otimizar políticas em ambientes de controle contínuo, como robôs.
    - **Mecanismo**: Restringe atualizações de políticas dentro de uma região segura, garantindo melhorias progressivas sem mudanças drásticas.
    - **Vantagens**: Oferece maior estabilidade no aprendizado, usando funções de perda substitutas e regiões de confiança, sendo ideal para tarefas de controle contínuo.
    - **Aplicações**: Problemas de controle de movimento, como manipulação robótica.
3. **Asynchronous Advantage Actor-Critic (A3C)**:

- **Objetivo**: Atingir alto desempenho em ambientes de aprendizado distribuído ou single-machine.
- **Mecanismo**: Utiliza múltiplos agentes que interagem com seus próprios ambientes de maneira assíncrona, atualizando uma política global compartilhada.
- **Vantagens**: Grande eficiência de dados e exploração simultânea de estratégias diferentes, aumentando a diversidade de experiências.
- **Aplicações**: Tarefas de navegação robótica, aprendizado multiagente e outras situações onde a exploração paralela é benéfica.

Ferramentas e Técnicas:

- **DQN**: Pode ser adaptado para tarefas visuais onde o agente precisa aprender a partir de imagens, como jogos de vídeo, permitindo que o modelo se aperfeiçoe diretamente a partir de frames de jogos ou simulações.
- **TRPO**: Excelente para problemas de controle contínuo onde transições suaves entre ações são essenciais. É útil para cenários onde agentes precisam de movimentos naturais e não podem realizar saltos bruscos nas suas ações.
- **A3C**: Ideal para ambientes de múltiplos agentes, onde várias simulações podem ocorrer ao mesmo tempo, o que acelera o aprendizado e aumenta a eficiência ao explorar diferentes caminhos e estratégias.

Conclusão: O artigo enfatiza como o DRL está resolvendo problemas de escalabilidade e eficiência de amostragem ao combinar aprendizado profundo com aprendizado por reforço. Com algoritmos como DQN, TRPO e A3C, é possível treinar agentes para lidar com ambientes de alta dimensionalidade e tomar decisões complexas com base em dados visuais ou simulações. Essas técnicas são essenciais para resolver desafios modernos em IA, como controle robótico, jogos e navegação autônoma.

Reinforcement Learning – Overview of Recent Progress and Implications for Process Control

Reinforcement Learning – Overview of recent progress and implications for process co…

Overview: O artigo oferece uma visão abrangente sobre os avanços recentes no Aprendizado por Reforço (RL) e suas potenciais implicações para o controle de processos industriais e tomada de decisão operacional. O RL permite que agentes aprendam através de tentativa e erro, ajustando ações com base em recompensas. Este campo tem evoluído rapidamente, com aplicações em áreas como controle de robôs, otimização de processos e tomada de decisões complexas. O artigo também compara o RL com métodos baseados em programação matemática, como o controle preditivo de modelo (MPC), destacando suas diferenças e potenciais integrações.

Conceitos e Algoritmos Principais:

1. **Model-Free vs. Model-Based RL**:
    - **RL Baseado em Modelo (Model-Based RL)**: Estima o modelo de transição do sistema e utiliza esse modelo para controle ou para estimar funções de valor e políticas.
    - **RL Sem Modelo (Model-Free RL)**: Aprende diretamente com dados através de interações com o ambiente, utilizando recompensas. Algoritmos sem modelo são mais simples de aplicar em sistemas complexos, mas podem exigir mais interações para convergirem.
2. **Value-Based (VB) e Policy-Gradient (PG)**:
    - **Value-Based (VB)**: Utiliza funções de valor (Q-Learning) para determinar a melhor ação em um estado específico.
    - **Policy-Gradient (PG)**: Ajusta diretamente a política de ações com base em gradientes de recompensa acumulada. Tem sido mais eficiente em problemas de controle contínuo.
    - **Ator-Crítico (Actor-Critic)**: Combina características dos métodos VB e PG, resultando em aprendizado mais rápido e estável.
3. **Exploração vs. Exploração**:
    - Um desafio central no RL é equilibrar exploração de novas ações e a exploração de ações já conhecidas que geram boas recompensas. Métodos como aprendizado off-policy permitem que agentes explorem mais eficientemente estados menos explorados para descobrir melhores políticas de longo prazo.
4. **Deep Reinforcement Learning (DRL)**:
    - **Redes Neurais Profundas (DNN)**: O uso de redes neurais profundas para aproximar funções de valor e políticas tem sido um grande avanço no campo.

Essas redes podem processar grandes volumes de dados e extrair características úteis automaticamente.
    - **Paralelização**: Técnicas de paralelização, como o algoritmo A3C, aumentam a velocidade do aprendizado em sistemas de controle distribuído, permitindo que vários agentes aprendam simultaneamente em diferentes ambientes simulados.

Ferramentas e Técnicas:

- **DNN em RL**: Essencial para aproximar funções de valor e políticas em problemas de alta dimensão, como jogos complexos ou controle de robôs. As DNNs eliminam a necessidade de engenharia manual de características, permitindo que o agente aprenda representações diretamente dos dados.
- **A3C (Asynchronous Advantage Actor-Critic)**: Um dos algoritmos de RL mais eficientes para tarefas de controle contínuo. A vantagem do A3C é que ele permite que múltiplos agentes explorem simultaneamente em ambientes distribuídos, melhorando a estabilidade do aprendizado.
- **Integração com MPC**: Métodos baseados em RL podem ser integrados ao MPC para otimizar controle de processos em tempo real, usando o aprendizado de longo prazo do RL para melhorar as predições de curto prazo do MPC.

Conclusão: O Aprendizado por Reforço tem potencial para transformar a área de controle de processos, permitindo a otimização contínua de sistemas em ambientes dinâmicos e incertos. Embora os métodos baseados em RL e MPC apresentem vantagens distintas, sua combinação oferece novas possibilidades para o controle de sistemas complexos, integrando aprendizado com dados e otimização preditiva. A capacidade de RL de aprender diretamente com dados de processos sem a necessidade de um modelo explícito o torna uma ferramenta poderosa para sistemas onde a modelagem exata é difícil.

Multiobjective Reinforcement Learning: A Comprehensive Overview

Multiobjective Reinforcement Learning - A Comprehensive Overview.pdf

Overview: O artigo apresenta uma visão abrangente sobre o Aprendizado por Reforço Multiobjetivo (MORL), que lida com problemas de tomada de decisão sequencial envolvendo múltiplos objetivos conflitantes. O MORL combina técnicas de Aprendizado por Reforço (RL) e otimização multiobjetivo (MOO) para encontrar soluções que equilibrem esses objetivos. O artigo explora as arquiteturas básicas, tópicos de pesquisa, e revisa diversas abordagens representativas para MORL, além de discutir os desafios e problemas abertos.

Conceitos e Algoritmos Principais:

1. **Modelos de Processo de Decisão de Markov (MDP)**:
    - O MORL usa MDPs, onde um agente toma decisões sequenciais com base em estados, ações, recompensas e probabilidades de transição. Cada objetivo no MORL tem sua própria função de recompensa, resultando em um vetor de recompensas, em vez de uma única recompensa escalar.
2. **Métodos Baseados em Soma Ponderada (Weighted Sum Approach)**:
    - Um dos métodos mais simples para MORL, em que uma soma ponderada das funções de valor para cada objetivo é usada para selecionar ações. No entanto, este método pode não capturar bem as soluções em regiões côncavas do Pareto front.
3. **W-Learning**:
    - Uma abordagem que calcula um valor W para cada objetivo e seleciona a ação que otimiza o valor W máximo. O W-learning evita que apenas um objetivo domine os outros, permitindo que todos os objetivos tenham uma chance justa de serem otimizados.
4. **Hierarquia de Análise de Processo (AHP)**:
    - Utilizado para expressar as preferências entre os objetivos quando há poucas informações quantitativas. Este método usa regras qualitativas e transforma essas preferências em uma função objetiva composta.
5. **Ranking Approach**:
    - Estabelece uma ordem de importância entre os objetivos, colocando restrições em alguns objetivos enquanto otimiza outros. Isso é particularmente útil em casos onde há uma clara hierarquia de objetivos, como em sistemas de controle que precisam otimizar o desempenho e evitar riscos ao mesmo tempo.
6. **Geometric Approach**:
    - Baseado em técnicas geométricas, este método tenta encontrar uma solução ótima para uma função sintética que satisfaça condições geométricas

específicas. Este método é útil em ambientes onde há múltiplas direções ou objetivos conflitantes que precisam ser equilibrados.

7. **Convex Hull Approach**:
    - Abordagem que utiliza o conceito de "convex hull" para encontrar políticas ótimas para todas as combinações lineares de preferências entre os objetivos. Esse método garante que o agente aprenda múltiplas políticas que se aproximam da Pareto front.

Ferramentas e Técnicas:

- **Aproximação de Função de Valor**: Técnicas de aproximação são essenciais em MORL, já que os espaços de estados e ações tendem a ser muito grandes ou contínuos. Algoritmos de função de valor precisam ser adaptados para otimizar múltiplos objetivos simultaneamente.
- **Uso de Preferências Dinâmicas**: Alguns algoritmos de MORL incorporam preferências que mudam com o tempo, permitindo que os agentes ajustem suas políticas conforme os objetivos mudam.
- **Otimização de Fronteira de Pareto**: Muitos algoritmos MORL focam em encontrar ou aproximar a Fronteira de Pareto, um conjunto de soluções ótimas que representam diferentes trade-offs entre os objetivos conflitantes.

Conclusão: O MORL oferece um framework robusto para resolver problemas de tomada de decisão sequencial com múltiplos objetivos conflitantes, sendo cada vez mais relevante em aplicações do mundo real. As abordagens discutidas no artigo fornecem uma visão ampla de como resolver esses problemas, cada uma com vantagens e desvantagens, dependendo do cenário e das preferências do usuário. Desafios futuros incluem a melhoria das técnicas de aproximação de função de valor e a generalização para grandes espaços de estado, além da aplicação prática em sistemas multiagentes e problemas de controle.

A Survey of Deep Reinforcement Learning in Video Games

A Survey of Deep Reinforcement Learning in Video Games.pdf

Overview: O artigo revisa os principais avanços no uso do Aprendizado por Reforço Profundo (Deep Reinforcement Learning - DRL) em jogos de vídeo. O DRL combina redes neurais profundas (DL) com aprendizado por reforço (RL), o que permitiu a criação de agentes que tomam decisões com base em entradas de alta dimensionalidade, como imagens de jogos, aprendendo de forma end-to-end. O artigo aborda as principais técnicas e algoritmos, discute as plataformas de pesquisa em IA de jogos e revisa os avanços em jogos de 2D a 3D, desde agentes de um único jogador até ambientes de múltiplos agentes.

Conceitos e Algoritmos Principais:

1. **Deep Q-Network (DQN)**:
    - **Objetivo**: Usado para jogos como Atari, onde o agente recebe pixels como entrada e estima funções de valor para tomar decisões.
    - **Técnicas principais**: Replay de experiências e redes alvo para estabilizar o aprendizado. Variações como Double DQN e Dueling DQN melhoram a precisão e eficiência do aprendizado.
2. **Policy Gradient Methods**:
    - **Ator-Crítico (Actor-Critic)**: Estrutura popular que combina gradientes de política com funções de valor para melhorar a política diretamente.
    - **A3C (Asynchronous Advantage Actor-Critic)**: Aprendizado assíncrono que permite que múltiplos agentes explorem simultaneamente, acelerando o aprendizado em ambientes distribuídos.
    - **PPO (Proximal Policy Optimization)**: Algoritmo que otimiza políticas de maneira mais estável e eficiente, usando gradiente estocástico com clipping.
3. **Model-Based DRL**:
    - **TreeQN** e **ATreeC**: Métodos que combinam planejamento em árvore com DRL, gerando melhorias em ambientes de ações discretas.
    - **MuZero**: Combina busca em árvore com aprendizado de modelo para resolver problemas com recompensas esparsas e atrasadas, como em jogos complexos.

Ferramentas e Técnicas:

- **DRL com Redes Neurais Convolucionais (CNN)**: Essencial para capturar informações de estados visuais de jogos, especialmente em ambientes de primeira pessoa ou jogos 3D.
- **Algoritmos Baseados em Política**: Úteis para otimizar diretamente as ações do agente em jogos com ações contínuas, como simuladores de corrida e controle de personagens.

- **Plataformas de Pesquisa**: OpenAI Gym, ViZDoom, e StarCraft II são algumas das plataformas mencionadas como cruciais para testar e avaliar a IA em jogos.

Conclusão: O artigo destaca que o DRL tem demonstrado desempenhos acima do nível humano em diversos jogos, como Atari e StarCraft. No entanto, ainda existem desafios como eficiência amostral, generalização entre jogos diferentes, e aprendizado multiagente, que continuam sendo áreas de pesquisa ativa. A aplicação de DRL em jogos abre caminho para o desenvolvimento de IA mais avançada, com potencial de resolver problemas complexos em ambientes de tomada de decisão em tempo real.

OpenSpiel: A Framework for Reinforcement Learning in Games

OpenSpiel - A Framework for Reinforcement Learning in Games.pdf

Overview: O **OpenSpiel** é um framework desenvolvido pelo DeepMind para pesquisa em Aprendizado por Reforço (RL) e planejamento em jogos. Ele oferece suporte a uma ampla variedade de jogos, como jogos de soma-zero, jogos cooperativos, jogos de movimento simultâneo e jogos com informação perfeita e imperfeita. Além disso, permite estudar o aprendizado multiagente, oferecendo ferramentas para análise da dinâmica de aprendizado e avaliação de métricas de desempenho. O framework inclui implementações de mais de 20 jogos e algoritmos para RL e planejamento.

Conceitos e Algoritmos Principais:

1. **Extensive-Form Games (Jogos em Forma Extensiva)**:
    - **Jogos de soma-zero e soma-constante**: Permitem a modelagem de jogos onde os ganhos e perdas dos jogadores se equilibram.
    - **Jogos com Informação Perfeita e Imperfeita**: Modelam cenários onde os jogadores possuem (ou não) conhecimento total das ações anteriores.
2. **Algoritmos de Pesquisa e Otimização**:
    - **Minimax e Alpha-Beta Search**: Usados em jogos de soma-zero com informação perfeita, como xadrez.
    - **Monte Carlo Tree Search (MCTS)**: Realiza buscas probabilísticas em árvores de jogo, essencial para lidar com jogos de informação imperfeita.
    - **Counterfactual Regret Minimization (CFR)**: Algoritmo focado em minimizar o arrependimento contrafactual, muito usado em jogos como pôquer.
3. **Algoritmos Tradicionais de RL**:
    - **Q-learning e Value Iteration**: Algoritmos de aprendizado para ambientes de jogador único, estendíveis para configurações multiagente.
    - **Deep Q-Networks (DQN)**: Usado para aproximar funções de valor em jogos com entradas visuais complexas.
    - **Advantage Actor-Critic (A2C)**: Um método baseado em gradientes de política, que equilibra a exploração e a exploração em ambientes multiagente.
4. **Aprendizado em Jogos Parcialmente Observáveis**:
    - **Best Response**: Um algoritmo para calcular a melhor resposta dada a política dos outros jogadores.
    - **Fictitious Play (FP)**: Procedimento clássico que calcula políticas através da iteração de melhores respostas.
    - **Neural Fictitious Self-Play (NFSP)**: Algoritmo que usa redes neurais para aprender políticas em jogos com informação imperfeita.

Ferramentas e Técnicas:

- **Visualização de Árvore de Jogos**: Utilizando ferramentas como o Graphviz, é possível visualizar a estrutura de decisão em jogos extensivos, facilitando a análise de decisões.
- **Alpha-Rank**: Algoritmo de classificação de agentes em jogos multiplayer, usando conceitos de teoria evolucionária de jogos para ranquear os agentes de forma eficiente, mesmo em cenários complexos como o jogo "Pedra, Papel e Tesoura".

Conclusão:

O OpenSpiel oferece uma plataforma robusta para pesquisa em RL e jogos, suportando uma ampla gama de algoritmos e tipos de jogos. Ele permite não apenas estudar o desempenho de agentes em jogos de um único jogador, mas também em ambientes multiagente, com foco em jogos de informação imperfeita e planejamento estratégico. As ferramentas de visualização e análise permitem avaliar a dinâmica de aprendizado e a evolução das políticas dos agentes, tornando o OpenSpiel uma ferramenta poderosa para estudos em RL e jogos.

Analysis of Artificial Intelligence Applied in Video Games

Analysis of Artificial Intelligence Applied in Video Games.pdf

Overview: O artigo discute o crescimento do Aprendizado por Reforço Profundo (Deep Reinforcement Learning - DRL) no campo da Inteligência Artificial, especialmente quando aplicado a jogos de vídeo. O foco é analisar como o DRL foi aplicado a diferentes tipos de jogos, desde ambientes 2D de informação perfeita, como jogos de tabuleiro, até jogos de perspectiva em primeira pessoa com informação imperfeita, como Doom. Além disso, o artigo discute os desafios enfrentados pelo DRL em jogos, como eficiência amostral, exploração versus exploração e recompensas esparsas e atrasadas, e sugere soluções para esses problemas.

Conceitos e Algoritmos Principais:

1. **AlphaGo e AlphaGo Zero**:
    - **Objetivo**: AlphaGo foi pioneiro ao vencer o jogo Go usando uma combinação de busca em árvore de Monte Carlo (MCTS) e aprendizado supervisionado, enquanto o AlphaGo Zero usou aprendizado por reforço sem supervisão jogando contra si mesmo.
    - **Inovações**: AlphaGo Zero melhorou significativamente ao usar uma rede neural residual para prever a distribuição de movimentos e a probabilidade de vitória. A autossupervisão de AlphaGo Zero permitiu que ele superasse seus predecessores que dependiam de dados humanos.
2. **Deep Blue**:
    - **Objetivo**: Um exemplo mais antigo de IA em jogos, o Deep Blue foi projetado para jogar xadrez e venceu o campeão mundial Garry Kasparov em 1997. Embora utilizasse força bruta e o algoritmo minimax, sua vitória foi um marco na IA aplicada a jogos de tabuleiro.
    - **Componentes**: Incluía uma função de avaliação para a colocação de peças e outras heurísticas de xadrez, além de uma busca alfa-beta para otimizar movimentos.
3. **MuZero**:
    - **Objetivo**: O sucessor do AlphaZero, MuZero, foi projetado para jogar jogos como Atari e Go sem precisar de um modelo explícito do ambiente. Ele usa MCTS com redes neurais para otimizar suas políticas e valores de recompensa.
    - **Inovação**: Ao contrário de seus predecessores, MuZero aprende diretamente com as interações com o ambiente, sem a necessidade de modelar explicitamente as transições de estado.
4. **Doom e VizDoom**:

- **Objetivo**: Jogos de perspectiva em primeira pessoa, como Doom, introduzem o desafio de lidar com informações imperfeitas. Usando o engine VizDoom, o agente navega no ambiente e combate inimigos usando uma rede Q-Recorrente (Deep Recurrent Q-Network - DRQN).
- **Técnicas**: O uso de LSTMs (Long Short-Term Memory) para armazenar estados anteriores melhora o desempenho do agente em jogos com visão limitada, como Doom, onde o agente não tem uma visão completa do mapa.

Ferramentas e Técnicas:

- **Monte Carlo Tree Search (MCTS)**: Usado em AlphaGo e MuZero para busca eficiente em jogos de grande espaço de estados, especialmente jogos de tabuleiro como Go.
- **Deep Recurrent Q-Network (DRQN)**: Implementado em jogos como Doom para lidar com entradas de informação imperfeita, onde o agente deve lembrar estados anteriores para tomar decisões melhores.
- **Curiosity-Driven Exploration**: Técnica que incentiva o agente a explorar estados desconhecidos, dando recompensas intrínsecas ao prever estados futuros incorretamente, o que é útil em ambientes com recompensas esparsas e atrasadas.

Conclusão: O artigo ressalta que o DRL tem alcançado grandes marcos em jogos, como vencer campeões humanos em Go e superar jogadores em Doom. No entanto, ele ainda enfrenta desafios como a necessidade de grandes volumes de dados de treinamento e a dificuldade em equilibrar exploração e exploração. Soluções como exploração impulsionada pela curiosidade e maior eficiência amostral estão sendo desenvolvidas para mitigar esses problemas. O uso de IA em jogos continua a ser um campo promissor para melhorar a capacidade dos agentes de lidar com ambientes complexos e de grande dimensão.

Current AI in Games: A Review

Current AI in games - a review.pdf

Overview: O artigo faz uma análise das técnicas de Inteligência Artificial (IA) utilizadas em jogos de vídeo, destacando as mais comumente usadas e aquelas emergentes que podem moldar o futuro da IA em jogos. O texto explora como as técnicas tradicionais como máquinas de estados finitos (FSMs), scripting e agentes têm sido amplamente adotadas devido à sua simplicidade e eficiência, enquanto métodos mais complexos, como redes neurais, algoritmos genéticos e lógica difusa, estão sendo introduzidos para oferecer mais profundidade e imprevisibilidade ao comportamento dos personagens em jogos.

Conceitos e Algoritmos Principais:

1. **Finite State Machines (FSMs)**:
    - **Objetivo**: As FSMs são amplamente utilizadas em jogos para modelar o comportamento de objetos e personagens, dividindo as ações em estados e transições pré-definidas.
    - **Aplicações**: Comumente usadas em jogos como *Age of Empires* e *Half-Life* para gerenciar o comportamento dos inimigos e suas transições entre estados como "patrulhando", "atacando", ou "fugindo".
    - **Limitações**: Embora eficientes, FSMs podem ser difíceis de escalar e manter, especialmente em jogos complexos com muitos estados e transições possíveis.
2. **Scripting**:
    - **Objetivo**: Scripting permite que designers de jogos criem comportamentos de IA de forma simples, sem precisar de grandes habilidades de programação. Jogos como *Unreal* e *Baldur's Gate* utilizam scripts para controlar eventos e comportamento de personagens não-jogadores (NPCs).
    - **Aplicações**: Usado para criar diálogos, comportamento de combate e eventos em jogos de aventura e estratégia.
    - **Limitações**: Assim como FSMs, scripts são determinísticos e exigem que os desenvolvedores antecipem todas as situações possíveis, o que pode limitar a flexibilidade da IA.
3. **Agentes Inteligentes**:
    - **Objetivo**: Agentes em jogos percebem o ambiente e agem de acordo com seus objetivos. Eles podem ser programados para reagir ao ambiente de forma mais autônoma, integrando diversas habilidades, como aprendizado, comportamento emocional e inferência.
    - **Aplicações**: Em jogos de estratégia, agentes são responsáveis pela gestão de unidades e recursos, como em *Empire Earth*, onde gerentes controlam aspectos específicos do jogador controlado pela IA.

4. **Lógica Difusa (Fuzzy Logic)**:
    - **Objetivo**: A lógica difusa permite que a IA tome decisões com base em dados imprecisos ou incompletos, oferecendo transições suaves entre estados de comportamento.
    - **Aplicações**: Usada em jogos como *SWAT 2* e *Call to Power* para ajustar o comportamento dos NPCs com base em suas emoções e status de saúde.
    - **Vantagens**: Permite mais flexibilidade e respostas mais realistas comparado a sistemas determinísticos como FSMs.
5. **Flocking**:
    - **Objetivo**: Simula comportamentos naturais de grupos, como bandos de aves ou cardumes de peixes, através de interações simples entre indivíduos.
    - **Aplicações**: Usada em jogos como *Half-Life* e *Unreal* para simular o comportamento em grupo de inimigos ou criaturas.
6. **Árvores de Decisão (Decision Trees)**:
    - **Objetivo**: Estruturas usadas para tomada de decisão baseada em atributos de um ambiente de jogo. As árvores de decisão são especialmente úteis para classificar situações e prever ações.
    - **Aplicações**: Implementadas em jogos como *Black & White*, onde criaturas aprendem a partir das ações do jogador e constroem árvores de decisão para prever o que fazer em situações futuras.
7. **Redes Neurais Artificiais (NNs)**:
    - **Objetivo**: Simulam o aprendizado humano, permitindo que a IA aprenda e adapte seu comportamento ao longo do tempo.
    - **Aplicações**: Usadas em jogos como *Black & White* e *Creatures*, onde a IA aprende e modifica seu comportamento com base em experiências passadas.
    - **Desafios**: Redes neurais exigem muitos recursos computacionais e são difíceis de ajustar corretamente, o que limita seu uso em jogos comerciais.
8. **Algoritmos Genéticos (GAs)**:
    - **Objetivo**: Algoritmos genéticos são usados para otimização e aprendizado em jogos, simulando a evolução natural para encontrar soluções para problemas complexos.
    - **Aplicações**: Jogos como *Creatures* e *Cloak, Dagger & DNA* fazem uso de GAs para evoluir o comportamento dos personagens com base em regras genéticas.
    - **Limitações**: São intensivos em recursos e podem ser lentos, sendo mais apropriados para otimização fora do tempo real.

Conclusão: Embora técnicas como FSMs e scripting sejam amplamente usadas devido à sua simplicidade e eficiência, jogadores estão buscando IA mais imprevisível e adaptável. Técnicas emergentes, como lógica difusa, redes neurais e algoritmos genéticos, oferecem novos horizontes para tornar os personagens mais realistas e envolventes. No entanto,

essas técnicas ainda enfrentam desafios, como a alta demanda por recursos e dificuldades de implementação, mas à medida que o poder de processamento cresce, essas soluções deverão se tornar mais viáveis e comuns nos jogos do futuro.

Artificial Intelligence for Adaptive Computer Games

Artificial Intelligence for Adaptive Computer Games.pdf

Overview: Este artigo explora como a Inteligência Artificial (IA) pode ser aplicada para criar jogos de computador adaptativos, onde o comportamento dos personagens e o enredo se ajustam dinamicamente ao jogador. A ideia central é ir além das interações pré-programadas e fornecer um sistema verdadeiramente interativo, que aprende sobre o jogador durante o jogo, adaptando seu comportamento e criando uma experiência mais envolvente. O artigo discute abordagens para atingir essa adaptabilidade em jogos utilizando três principais técnicas de raciocínio baseado em casos (Case-Based Reasoning - CBR): adaptação automática de comportamento para personagens críveis, gerenciamento de drama e modelagem de usuário para histórias interativas, e planejamento de comportamento estratégico para jogos de estratégia em tempo real.

Conceitos e Algoritmos Principais:

1. **Adaptação de Comportamento para Personagens Críveis**:
    - **Objetivo**: Criar personagens que se comportem de maneira crível, adaptando seus comportamentos de acordo com novas situações que surgem no jogo.
    - **Mecanismo**: Utiliza um sistema de monitoramento que analisa os comportamentos dos personagens em tempo real, identifica comportamentos que precisam ser modificados e aplica modificações automaticamente.
    - **Exemplo**: Um personagem que joga "Pega-Pega" no *Unreal Tournament* adapta suas estratégias de perseguição com base no desempenho atual, ajustando-se a falhas no comportamento original.
2. **Gerenciamento de Drama e Modelagem de Usuário em Histórias Interativas**:
    - **Objetivo**: Gerenciar o enredo de histórias interativas para que o jogador tenha a sensação de controle sobre os eventos, enquanto mantém uma estrutura narrativa coesa.
    - **Mecanismo**: O Drama Manager (DM) influencia a progressão do enredo de acordo com o modelo de usuário gerado, que prevê quais elementos da história o jogador provavelmente preferirá com base em ações passadas de jogadores semelhantes.
    - **Exemplo**: No jogo *Anchorhead*, o sistema ajusta a narrativa e as ações dos personagens não-jogadores (NPCs) de acordo com o comportamento do jogador, melhorando a imersão.
3. **Planejamento Estratégico em Jogos de Estratégia em Tempo Real (RTS)**:
    - **Objetivo**: Reduzir o esforço de desenvolvimento necessário para criar comportamentos estratégicos em jogos de RTS.

- **Mecanismo**: Um sistema baseado em planejamento por casos extrai comportamentos de demonstrações de especialistas, reutilizando e adaptando esses comportamentos durante o jogo.
- **Exemplo**: Em *Wargus* (clone de *Warcraft II*), o sistema aprende com as demonstrações de especialistas e adapta as estratégias para diferentes mapas e unidades, permitindo uma IA mais eficiente e adaptativa.

Ferramentas e Técnicas:

- **Raciocínio Baseado em Casos (CBR)**: A técnica central utilizada em todos os exemplos apresentados, onde o sistema armazena e reutiliza casos anteriores para adaptar comportamentos e decisões em novos contextos.
- **Gerenciamento de Drama**: Componente utilizado para guiar a narrativa com base no comportamento do jogador, ajustando a história para manter a coerência e o interesse.
- **Modificação de Comportamento em Tempo Real**: Uma ferramenta que permite que personagens alterem seu comportamento durante a execução do jogo, com base no feedback do ambiente e nas falhas detectadas em suas ações.

Conclusão: O artigo demonstra que a aplicação de técnicas de IA, como o raciocínio baseado em casos, pode aumentar a adaptabilidade e a imersão em jogos de computador. Ao permitir que personagens e enredos se ajustem dinamicamente ao jogador, é possível criar experiências mais ricas e personalizadas. No futuro, o objetivo é simplificar ainda mais o processo de desenvolvimento de jogos com IA, facilitando a implementação dessas técnicas para desenvolvedores que não são especialistas em IA, tornando os jogos mais adaptativos e envolventes.

Recent Research on AI in Games

Recent Research on AI in Games.pdf

Overview: O artigo revisa a pesquisa recente sobre o uso de Inteligência Artificial (IA) em jogos, focando em três áreas principais de aplicação: personagens não-jogadores críveis (NPCs), geração de níveis procedurais (PLG) e modelagem da experiência do jogador. Ele também explora o conceito de inteligência híbrida e discute a aplicação de IA geral em jogos, oferecendo uma visão geral dos avanços entre 2018 e o presente.

Conceitos e Algoritmos Principais:

1. **IA para NPCs Críveis**:
    - **Objetivo**: Criar NPCs que possam interagir de forma mais humana e diversificada, aumentando a imersão dos jogadores.
    - **Técnicas Usadas**: Aprendizado por Reforço (RL) tem sido amplamente utilizado para criar NPCs críveis, como no caso de um framework para NPCs no jogo *Street Fighter IV*, que se aproximam do comportamento humano em mais de 60% dos casos em testes de Turing. Além disso, técnicas como Aprendizado por Imitação e Processamento de Linguagem Natural (NLP) têm sido aplicadas para melhorar a interatividade narrativa.
2. **Geração de Níveis Procedurais (PLG)**:
    - **Objetivo**: Gerar automaticamente níveis de jogos que variam em dificuldade e jogabilidade, muitas vezes de forma pseudo-aleatória ou baseada em certas mecânicas de jogo.
    - **Aplicações**: Jogos como *Super Mario Bros.* e *Angry Birds* foram utilizados como benchmarks para testar algoritmos de geração de níveis. Técnicas como Redes Adversariais Gerativas (GANs) e algoritmos evolutivos têm sido usadas para criar níveis que desafiam os jogadores e atendem a restrições específicas de design.
3. **Modelagem da Experiência do Jogador**:
    - **Objetivo**: Avaliar e ajustar o comportamento do jogo com base na experiência do jogador, melhorando o engajamento e ajustando a dificuldade.
    - **Técnicas Usadas**: Modelagem usando vídeos de gameplay e aprendizado profundo (CNNs) para prever as preferências dos jogadores sem a necessidade de acessar os logs dos jogos. Sinais psicofisiológicos, como batimentos cardíacos e respostas eletrodérmicas, também têm sido utilizados para captar o estado emocional dos jogadores.

Ferramentas e Técnicas:

- **Aprendizado por Reforço Federado (Federated Reinforcement Learning)**: Técnica usada para personalizar o comportamento dos NPCs de forma mais eficiente, permitindo aprendizado distribuído em diversos ambientes.
- **Geradores Procedurais Assistidos por IA**: Usados para colaborar com designers humanos na criação de níveis de jogos, uma forma de inteligência híbrida que integra a criatividade humana com a capacidade de otimização da IA.

Conclusão: O artigo destaca que a IA em jogos evoluiu significativamente nos últimos anos, com avanços em áreas como NPCs críveis, geração procedural de níveis e modelagem da experiência do jogador. A introdução de técnicas como aprendizado por reforço, aprendizado por imitação e processamento de linguagem natural está transformando a maneira como os jogos são projetados e como os jogadores interagem com eles. A pesquisa em IA geral para jogos e a aplicação de inteligência híbrida continuam a ser áreas promissoras, com potencial para revolucionar o design e a jogabilidade dos jogos.

Review and Analysis of Research on Video Games and Artificial Intelligence: A Look Back and a Step Forward

Review and analysis of research on Video Games and Artificial Intelligence - a look ba…

Overview: O artigo realiza uma análise bibliométrica abrangente da interseção entre videogames e Inteligência Artificial (IA), cobrindo mais de 50 anos de pesquisa, desde 1971 até 2021. A pesquisa se concentra nas contribuições acadêmicas mais influentes, nas principais fontes de publicação e nos autores mais prolíficos, com base em uma análise de publicações indexadas no Scopus. O estudo revela as principais tendências de pesquisa, as áreas mais citadas e os desafios futuros para a aplicação de IA em videogames.

Conceitos e Algoritmos Principais:

1. **Histórico de IA em Jogos**:
   - Desde os anos 1950, quando Claude Shannon propôs um algoritmo para jogar xadrez, jogos têm sido utilizados como ambientes de teste para IA. Exemplos como o *Pac-Man* (1981), que implementou heurísticas de IA para os fantasmas, e o *AlphaStar* (2019), que superou 99,8% dos jogadores humanos em *StarCraft II*, são marcos na evolução da IA em jogos.
   - A IA em jogos começou a ser reconhecida como crítica não apenas pela jogabilidade, mas também pela análise de desempenho, estratégia e decisão em ambientes controlados.
2. **Principais Trabalhos e Contribuições Científicas**:
   - **Deep Q-Networks (DQN)**: O artigo de 2015 de Mnih et al., que introduziu o uso de redes neurais profundas em RL para controlar agentes em jogos Atari, foi o mais citado, com 8905 citações. Ele foi o primeiro modelo de aprendizado profundo capaz de igualar o desempenho de jogadores humanos em jogos complexos.
   - **Kinect (2013)**: Outra contribuição de grande impacto foi a pesquisa sobre o Kinect, que trouxe avanços em visão computacional e reconhecimento de movimento para jogos interativos, acumulando 939 citações.
   - **AlphaGo e AlphaStar**: Também se destacam pesquisas como o *AlphaGo* de 2018, que superou o melhor jogador humano de Go, e o *AlphaStar*, que atingiu níveis de Grandmaster em *StarCraft II*.
3. **Bibliometria e Análise de Publicações**:
   - O estudo identificou um total de 2604 publicações no campo, com um crescimento acentuado no número de artigos a partir de 2004, especialmente em conferências e journals de IA e ciência da computação. O *Lecture Notes in Computer Science* é a principal fonte de publicações, seguido pelo *AAAI Workshop* e o *IEEE Conference on Computational Intelligence and Games*.

- Entre os autores mais influentes, destacam-se Vadim Bulitko (Universidade de Alberta) e Julian Togelius (New York University), ambos com produções contínuas e altamente citadas.

Ferramentas e Técnicas:

- **Monte Carlo Tree Search (MCTS)**: Utilizado em jogos como Go e StarCraft II, esse método de busca em árvore tornou-se uma técnica amplamente aplicada em jogos com alta complexidade estratégica.
- **Redes Neurais Convolucionais (CNNs)**: Combinadas com RL, as CNNs são usadas para entender o ambiente visual em jogos, como demonstrado no *Deep Q-Network*.
- **Redes Adversárias Generativas (GANs)**: Aplicadas à geração procedural de conteúdo, as GANs são cada vez mais usadas para criar níveis de jogo e comportamentos de NPCs que se adaptam ao jogador.

Conclusão: O estudo destaca a importância dos videogames como plataforma de teste para algoritmos de IA, onde a IA não só aprimora a jogabilidade, mas também evolui através de desafios complexos impostos por esses ambientes. Ao longo das últimas décadas, a pesquisa em IA para jogos tem crescido exponencialmente, e novas áreas como geração procedural e IA geral oferecem um vasto campo de exploração. No entanto, desafios como eficiência amostral, adaptação a ambientes complexos e interação multiagente ainda permanecem e continuam sendo tópicos de interesse para pesquisa futura.

APÊNDICE 3

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

Termo de Aceite de Entrega

Objetivo deste documento

Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião ("gate") de aprovação: 17 de out. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

LUCAS BRANDÃO RODRIGUES

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

Durante esse Stage (Stage 5 - 171024), foram realizadas as seguintes atividades:

- **Pesquisa de frameworks populares de aprendizado por reforço (Reinforcement Learning - RL):** OpenAI Gym, TensorFlow Agents (TF-Agents) e PyTorch RL.
- **Levantamento de ferramentas e frameworks utilizados para a integração de RL com engines de jogos:** Unity ML-Agents, Unreal Engine AirSim e Godot Engine + TensorFlow/PyTorch Integration.
- **Produção de um catálogo detalhando os frameworks e ferramentas identificadas,** incluindo uma breve explicação de como são utilizadas no contexto do desenvolvimento de IA para games:
  - Catálogo de Ferramentas e Frameworks

Planejamento: [descrever o que pretende fazer para realizar a próxima ENTREGA]

Objetivos:

- **Escolher dois frameworks/ferramentas de aprendizado por reforço** para me especializar.
- **Configurar os ambientes de desenvolvimento** para cada framework escolhido.
- **Desenvolver artefatos didáticos,** como scripts comentados, guias passo a passo, e pequenos projetos de exemplo que demonstrem a **aplicação dos modelos de aprendizado por reforço.**

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

ACEITE DA ENTREGA:

CEDRIC LUIZ DE CARVALHO: Go!

Instituto de Informática (INF) - UFG
Alameda Palmeiras, Quadra D, Câmpus Samambaia
Fone: (62) 3521-1181 / Fax: (62) 3521-1182

Frameworks

Durante o processo da Residência em IA, realizei uma pesquisa detalhada sobre frameworks utilizados no campo de **Reinforcement Learning (RL)**, com foco em **OpenAI Gym**, **TensorFlow Agents (TF-Agents)** e **PyTorch RL**. Cada framework foi analisado em profundidade, abordando aspectos como "Descrição Geral", "Principais Funcionalidades", "Relevância para IA em Games", "Vantagens" e "Limitações". **Essas análises foram organizadas em um catálogo** para servir como referência ao longo do projeto, facilitando a escolha e comparação entre frameworks de RL para futuros experimentos.

Além disso, foi conduzida uma pesquisa sobre ferramentas e frameworks utilizados para a **integração de IA com engines de jogos**, com foco em modelos de aprendizado por reforço aplicados diretamente a games. Foram analisadas tecnologias que permitem a implementação de agentes de RL em ambientes interativos, como os desenvolvidos no **Unity** e **Unreal Engine**, incluindo o uso do **Unity ML-Agents**, a integração de **TensorFlow com Unity** e o uso de **PyTorch com OpenAI Gym** em simulações personalizadas. Essas ferramentas também foram incluídas no catálogo, com explicações sobre suas funcionalidades e como são aplicadas no contexto de desenvolvimento de IA para games.

Catálogo de Frameworks

Catálogo de Ferramentas e Frameworks

- **Reinforcement Learning (RL):** Inclui OpenAI Gym, TensorFlow Agents e PyTorch RL, detalhando suas principais características e aplicabilidade.
- **RL para Games:** Envolve a análise de Unity ML-Agents, TensorFlow + Unity e PyTorch + OpenAI Gym, focando na integração de IA com engines de jogos.

Reinforcement Learning (RL)

OpenAI Gym (Gymnasium)

Link para Documentação Oficial: OpenAI Gym Documentation

Descrição Geral: OpenAI Gym é uma plataforma amplamente utilizada para o desenvolvimento e avaliação de algoritmos de Aprendizado por Reforço. Ela fornece um conjunto padronizado de ambientes de simulação que permitem aos pesquisadores e desenvolvedores testar diferentes algoritmos de RL de forma consistente e comparável.

Principais Funcionalidades: O OpenAI Gym oferece uma ampla variedade de ambientes, desde simples problemas clássicos de controle (como o CartPole e o MountainCar) até simulações mais complexas. Ele é altamente compatível com frameworks como TensorFlow e PyTorch, e pode ser facilmente integrado a diferentes algoritmos de aprendizado por reforço. O Gym também é flexível, permitindo que os usuários criem seus próprios ambientes personalizados.

Relevância para IA em Games: Embora o OpenAI Gym não seja diretamente voltado para desenvolvimento de jogos, ele é uma ferramenta poderosa para testar algoritmos de RL que podem ser aplicados em ambientes de jogos. Suas simulações de controle e navegação fornecem uma base sólida para a construção de agentes inteligentes que podem ser adaptados para interações mais complexas dentro de jogos.

Vantagens:

- Grande variedade de ambientes pré-definidos, o que facilita testes comparáveis.
- Integração fácil com bibliotecas populares como TensorFlow e PyTorch.
- Possibilidade de criar ambientes personalizados.
- Amplamente utilizado na pesquisa, com uma comunidade forte e bem documentada.

Limitações:

- Não é especificamente voltado para a criação de ambientes de jogos complexos ou com gráficos avançados.
- Não possui suporte nativo para engines de jogos como Unity, exigindo integração externa.
- Enfoca mais a simulação e controle do que a aplicação direta em jogos com múltiplas variáveis de complexidade.

TensorFlow Agents (TF-Agents)

Link para Documentação Oficial: TF-Agents Documentation

Descrição Geral: O TF-Agents é uma biblioteca de código aberto desenvolvida pelo Google para a criação e implementação de algoritmos de Aprendizado por Reforço usando a plataforma TensorFlow. Ele é projetado para oferecer uma base flexível e modular que permite o desenvolvimento de agentes de aprendizado por reforço tanto em ambientes simples quanto complexos, além de oferecer suporte para ambientes personalizados.

Principais Funcionalidades: TF-Agents oferece uma arquitetura modular, facilitando a criação de agentes RL ao fornecer blocos prontos para configurar algoritmos populares como DQN, PPO, A3C e muitos outros. Ele se integra facilmente ao TensorFlow, aproveitando as capacidades de deep learning para a criação de redes neurais complexas. A biblioteca também oferece suporte a simulações em tempo real e a ambientes externos, incluindo o OpenAI Gym.

Relevância para IA em Games: O TF-Agents é altamente relevante para IA em games, pois permite a criação de agentes complexos que podem aprender com grandes quantidades de dados e interações. Sua integração com TensorFlow torna possível o uso de redes neurais profundas para otimizar o comportamento de agentes em ambientes complexos, como jogos. Ele oferece a flexibilidade necessária para treinar IA em cenários de jogo mais avançados.

Vantagens:

- Total integração com TensorFlow, aproveitando as capacidades de deep learning.
- Suporte para algoritmos de RL avançados e personalizáveis.
- Arquitetura modular que facilita a criação e experimentação com diferentes agentes.
- Compatível com outros frameworks e ambientes, como o OpenAI Gym.

Limitações:

- A curva de aprendizado pode ser íngreme para quem não tem familiaridade com TensorFlow.
- Menos intuitivo para iniciantes, especialmente para aqueles que buscam simplicidade na implementação de RL.
- A configuração inicial do ambiente pode ser mais complexa em comparação com outras ferramentas mais especializadas em jogos.

PyTorch RL

Link para Documentação Oficial: PyTorch RL Documentation

Descrição Geral: PyTorch RL (Reinforcement Learning) refere-se ao uso da biblioteca PyTorch para implementar algoritmos de aprendizado por reforço. Embora não exista uma biblioteca única chamada “PyTorch RL”, a comunidade PyTorch oferece diversos pacotes e implementações de algoritmos de RL baseados em PyTorch, como Stable Baselines3, RLlib e outras ferramentas. PyTorch é amplamente utilizado devido à sua flexibilidade e facilidade de uso, permitindo a criação de agentes de aprendizado por reforço personalizados e de ponta.

Principais Funcionalidades:

- Suporte a uma vasta gama de algoritmos de aprendizado por reforço, como DQN, PPO, A3C, entre outros.
- Fácil integração com deep learning, utilizando a poderosa API de redes neurais de PyTorch.
- Flexibilidade na criação e personalização de redes neurais e políticas de agentes.
- Suporte ativo da comunidade, com implementações contínuas e repositórios atualizados.
- Compatível com diversos ambientes de simulação, incluindo OpenAI Gym e ambientes personalizados.

Relevância para IA em Games: PyTorch RL é altamente relevante para IA em games, pois oferece grande flexibilidade para o desenvolvimento de agentes de aprendizado por reforço que podem ser aplicados em cenários de jogos complexos. A facilidade de uso e customização do PyTorch permite que desenvolvedores adaptem algoritmos de RL para atender às necessidades específicas de jogos, seja otimizando a inteligência de NPCs (personagens não jogáveis) ou implementando agentes autônomos que interagem com os jogadores em tempo real.

Vantagens:

- Grande flexibilidade para criar agentes e redes neurais customizadas.
- Forte suporte para deep learning, facilitando a criação de agentes com capacidades avançadas de aprendizado.
- Comunidade ativa e vasto ecossistema de ferramentas e bibliotecas que suportam RL em PyTorch.
- Integração fácil com ambientes de simulação e ferramentas de desenvolvimento de IA.
- Suporte a múltiplos dispositivos (CPU/GPU) para treinos mais rápidos.

Limitações:

- Exige maior conhecimento técnico para implementar e ajustar algoritmos de RL, especialmente quando comparado a frameworks mais simplificados.
- Não possui uma biblioteca centralizada, o que pode aumentar o esforço de integração e desenvolvimento.
- A implementação de agentes RL complexos pode ser mais demorada devido à necessidade de customizações profundas.

RL for Games

Unity ML-Agents

Link para Documentação Oficial: Unity ML-Agents Documentation

Descrição Geral: O Unity ML-Agents é um kit de ferramentas desenvolvido pela Unity Technologies que permite integrar algoritmos de Aprendizado por Reforço e outras técnicas de machine learning diretamente em jogos criados com Unity. Ele facilita a criação de agentes inteligentes que podem aprender comportamentos complexos em ambientes simulados, utilizando o Unity como plataforma para treinar, testar e visualizar o aprendizado dos agentes.

Principais Funcionalidades:

- Integração direta com o Unity, permitindo simulações avançadas em ambientes tridimensionais e interativos.
- Suporte para múltiplos algoritmos de Aprendizado por Reforço (ex.: PPO, SAC) e frameworks como TensorFlow e PyTorch.
- Ferramentas para visualização em tempo real do treinamento dos agentes, permitindo ajustes dinâmicos durante o desenvolvimento.
- Simulação de múltiplos cenários simultaneamente, acelerando o tempo de treinamento.
- Suporte a treinamento distribuído em clusters ou múltiplos dispositivos para maior eficiência.

Relevância para IA em Games: O Unity ML-Agents é extremamente relevante para IA em jogos, pois oferece uma plataforma robusta para criar e treinar agentes diretamente em ambientes de jogos criados no Unity. Ele permite a criação de jogos inteligentes, onde os agentes podem adaptar-se ao comportamento dos jogadores, aprender com suas ações e melhorar a experiência do jogo em tempo real. A capacidade de simular e treinar agentes em ambientes tridimensionais faz dele uma ferramenta de destaque para o desenvolvimento de IA em games.

Vantagens:

- Integração total com a plataforma Unity, facilitando o uso em jogos já em desenvolvimento.
- Suporte para simulações complexas com ambientes ricos em interações e feedback visual imediato.
- Facilita a visualização e ajuste do comportamento dos agentes durante o treinamento.

- Compatível com TensorFlow e PyTorch, aproveitando técnicas avançadas de aprendizado profundo.
- Projetado especificamente para aplicações em jogos e simulações interativas.

Limitações:

- Requer conhecimentos de Unity, o que pode ser uma barreira para desenvolvedores que não estão familiarizados com o motor.
- A complexidade dos ambientes 3D pode aumentar significativamente o tempo de treinamento.
- A configuração inicial pode ser mais trabalhosa em comparação com frameworks mais simples de RL.
- O uso em ambientes fora do Unity é limitado.

Unreal Engine AirSim

Link para Documentação Oficial: AirSim Documentation

Descrição Geral: Unreal Engine, uma das engines de jogos mais avançadas, pode ser integrada ao aprendizado por reforço utilizando o framework AirSim. AirSim é uma plataforma de simulação de veículos autônomos desenvolvida pela Microsoft que permite treinar agentes em ambientes realistas criados com Unreal Engine. Embora não tenha uma integração direta como o Unity ML-Agents, o uso de Unreal junto com AirSim oferece um ambiente poderoso para simulações complexas e realistas.

Principais Funcionalidades:

- Simulação de ambientes realistas com suporte para gráficos avançados do Unreal Engine.
- Suporte para treinamentos complexos de aprendizado por reforço com foco em veículos autônomos, mas que pode ser adaptado para outras áreas de IA em jogos.
- Compatível com algoritmos de aprendizado por reforço e ferramentas como TensorFlow e PyTorch.
- Capacidade de realizar simulações de ambientes internos e externos com alta fidelidade.

Relevância para IA em Games: Unreal Engine, quando integrado com AirSim, oferece um ambiente robusto para a criação de agentes inteligentes em jogos realistas. Embora inicialmente voltado para veículos autônomos, suas funcionalidades podem ser adaptadas para o treinamento de agentes em outros tipos de jogos, especialmente aqueles que exigem simulações físicas realistas, como simuladores de voo ou de condução.

Vantagens:

- Gráficos e simulações de alta qualidade, com grande potencial para jogos hiper-realistas.
- Suporte para redes neurais profundas e algoritmos avançados de aprendizado por reforço.
- Flexibilidade para simular ambientes dinâmicos e realistas.

Limitações:

- A integração com aprendizado por reforço requer mais customização e não é tão direta quanto Unity ML-Agents.
- Focado principalmente em simulações de veículos autônomos, o que pode exigir ajustes para outros tipos de jogos.
- A curva de aprendizado para Unreal Engine pode ser íngreme para iniciantes.

Godot Engine + TensorFlow/PyTorch Integration

Link para Documentação Oficial: Godot Documentation

Descrição Geral: Embora o Godot Engine não tenha uma integração nativa para aprendizado por reforço como o Unity ML-Agents, ele pode ser usado junto com bibliotecas de aprendizado profundo como TensorFlow e PyTorch. A flexibilidade do Godot permite que desenvolvedores criem simulações personalizadas e treinem agentes inteligentes utilizando algoritmos de aprendizado por reforço implementados em frameworks externos. Essa integração exige a comunicação entre o ambiente de simulação do Godot e os modelos de RL, normalmente através de APIs ou sockets.

Principais Funcionalidades:

- Godot oferece uma plataforma leve e de código aberto para criar jogos e simulações em 2D e 3D, com grande flexibilidade na criação de ambientes customizados.
- Suporte para integração externa com bibliotecas de aprendizado por reforço, como TensorFlow e PyTorch, via API, WebSocket ou outros meios de comunicação.
- Ferramentas robustas para criar ambientes de simulação detalhados, que podem ser usados para treinar agentes em diferentes cenários de jogos.
- Suporte a scripting com GDScript, VisualScript, e integração com C#, facilitando o desenvolvimento de comportamentos customizados.

Relevância para IA em Games: O Godot Engine, com sua flexibilidade, oferece uma alternativa de código aberto para a criação de jogos e simulações, especialmente em projetos onde o desenvolvedor quer mais controle sobre a arquitetura. Ao integrar bibliotecas de RL como TensorFlow ou PyTorch, ele pode ser utilizado para desenvolver e treinar agentes inteligentes que interagem diretamente com o ambiente de simulação do jogo.

Vantagens:

- Plataforma de código aberto, sem taxas de licença, o que facilita o uso em projetos acadêmicos e comerciais.
- Suporte completo para ambientes 2D e 3D, permitindo a criação de simulações dinâmicas e interativas.
- Flexibilidade para integrar qualquer biblioteca de RL através de interfaces personalizadas.
- Comunidade ativa e suporte contínuo para novos desenvolvimentos e funcionalidades.

Limitações:

- Não possui integração nativa para aprendizado por reforço, exigindo maior esforço para implementar a comunicação entre o jogo e os modelos de RL.
- Curva de aprendizado elevada para configurar a comunicação com frameworks externos de aprendizado profundo.
- Ferramentas de simulação podem não ser tão avançadas quanto as oferecidas pelo Unity ou Unreal para jogos complexos em 3D.

APÊNDICE 4

Termo de Aceite de Entrega

Objetivo deste documento

Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião ("gate") de aprovação: 14 de nov. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

Lucas Brandão

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

Durante esse Stage (Stage 8 - 141124), foram realizadas as seguintes atividades:

Criação de um material didático Unity ML-Agents :

- **Seção "First Steps":**
  - Instruções para criação de um projeto Unity com cenas de exemplo do tutorial Getting Started Guide for ML-Agents.
  - Instruções para instalação do pacote ML-Agents dentro do projeto Unity.
- **Seção "Basics":**
  - Explicação dos princípios de Reinforcement Learning, com base no tutorial Background: Machine Learning.
  - Introdução ao PyTorch e TensorBoard, com base no tutorial Background: PyTorch.
- **Seção "Scene: Basic":**
  - Minhas observações sobre um dos exemplos de Environment da Unity chamado "Basic", seguindo o tutorial How to use Machine Learning AI in Unity! (ML-Agents) .

Planejamento: [descrever o que pretende fazer para realizar a próxima ENTREGA]

Objetivos:

- **Incorporar no meu material didático o conteúdo dos tutoriais:**
  - **ML-Agents Toolkit Overview**, que traz um overview das ferramentas disponíveis no pacote ML-Agents.
  - **Training ML-Agents**, que ensina como usar ML-Agents para realizar treinamentos de modelos.
  - **Background: Unity**, que ensina os princípios da Unity Engine.
  - **ML-Agents HummingBird**, que ensina como usar ML-Agents em um environment mais complexo.
- **Estudar o código-fonte do clone do Flappy Bird** para compreender sua estrutura e mecânicas de jogo e **adaptar o ambiente de jogo** para ser compatível com o treinamento de agentes de aprendizado por reforço.

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

ACEITE DA ENTREGA:

CEDRIC LUIZ DE CARVALHO: Go!

Termo de Aceite de Entrega

Objetivo deste documento

Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião ("gate") de aprovação: 28 de out. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

Lucas Brandão

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

Durante esse Stage (Stage 9 - 281124), foram realizadas as seguintes atividades:

Foram adicionadas 80 páginas no material didático Unity ML-Agents:

- **Seção "Basics" (10 páginas):** Explicação dos conceitos usados na Unity Engine, com base no tutorial Background: Unity. Explicação sobre o que é Imitation Learning e Curriculum Learning.
- **Seção "ML-Agents" (58 páginas):** Instruções sobre o uso do pacote ML-Agents, com base em minhas observações e nos tutoriais Getting Started Guide, ML-Agents Toolkit Overview, Training ML-Agents, Designing a Learning Environment e Designing Agents.
- **Seção "Scene: Basic" (22 páginas):** Minhas observações sobre um dos exemplos de Environment da Unity chamado "Basic", seguindo o tutorial How to use Machine Learning AI in Unity! (ML-Agents).

Planejamento: [descrever o que pretende fazer para realizar a próxima ENTREGA]

Objetivos:

- **Seção "Imitation Learning":** Minha implementação de um cenário demonstrando o uso de Imitation Learning, com base no tutorial Teach your AI! Imitation Learning with Unity ML-Agents! .
- **Seção "Flappy Bird":** Minha implementação de RL para um clone do jogo Flappy Bird (disponível em um repositório no github com tutorial de implementação), com base no tutorial AI Learns to play Flappy Bird! .
- **Revisar e consolidar todo o material gerado ao longo das semanas,** incluindo os materiais didáticos, artefatos práticos e relatórios (catálogos, cronograma, etc).
- **Refletir sobre o processo de pesquisa e aprendizado,** identificando áreas de melhoria e possíveis próximos passos para aprofundar o estudo do tema.

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

ACEITE DA ENTREGA:

CEDRIC LUIZ DE CARVALHO: Go!

Material Didático - Seções Teóricas

Durante as últimas semanas da Residência em IA, criei um material didático que aborda conceitos teóricos e práticos de Aprendizado por Reforço e Imitation Learning, com foco na aplicação prática utilizando Unity. O Unity ML-Agents é uma das opções mais populares e bem documentadas para integrar aprendizado por reforço em jogos. A engine Unity é amplamente utilizada na indústria de desenvolvimento de jogos, e o ML-Agents Toolkit oferece um conjunto robusto de recursos para criar agentes inteligentes em ambientes 3D interativos. Além disso, existem muitas informações disponíveis, incluindo tutoriais oficiais [1, 2, 3], vídeos explicativos [1, 2, 3], e exemplos práticos [1, 2, 3, 4] que facilitam o aprendizado e a aplicação rápida.

O material criado inclui tutoriais passo a passo, exemplos comentados e observações pessoais sobre as principais dificuldades e soluções encontradas. As seções cobrem desde a configuração do ambiente de desenvolvimento e a criação de agentes simples até a implementação de cenários mais complexos, como o controle de agentes em jogos e a análise de desempenho. Esse material serve como um guia de apoio para futuros estudos e projetos relacionados à área de IA aplicada a jogos.

A **parte teórica** do material didático desenvolvido consiste nas seguintes seções:

Seção “First Steps”:

- Instruções para instalação e configuração do ambiente de desenvolvimento.
- Instruções para treinamento de modelos usando um virtual environment.
- Instruções para criação de um projeto Unity com cenas de exemplo do tutorial Getting Started Guide for ML-Agents.
- Instruções para instalação do pacote ML-Agents dentro do projeto Unity.

Seção “Basics”:

- Explicação dos princípios de Reinforcement Learning, com base no tutorial Background: Machine Learning.
- Introdução ao PyTorch e TensorBoard, com base no tutorial Background: PyTorch.
- Informações sobre Imitation Learning e Curriculum Learning.
- Explicação dos conceitos usados na Unity Engine, com base no tutorial Background: Unity.

Seção “ML-Agents”: Instruções sobre o uso do pacote ML-Agents, com base em minhas observações e nos tutoriais Getting Started Guide, ML-Agents Toolkit Overview, Training ML-Agents, Designing a Learning Environment e Designing Agents.**Subseções:**

- **Getting Started Guide**: Introdução rápida ao uso do ML-Agents.
- **Toolkit Overview**: Visão geral do aprendizado por reforço, aprendizado por imitação e cenários de treinamento disponíveis no ML-Agents Toolkit.
- **Training ML-Agents**: Opções de treinamento disponíveis e parâmetros ajustáveis no toolkit.
- **Designing a Learning Environment**: Introdução prática ao design de ambientes de aprendizado.
- **Example Environments**: Visão geral de ambientes de exemplo mais complexos fornecidos pelo Unity.
- **Agents**: Uso da classe Agent dentro do toolkit.
- **Glossary**: Informações detalhadas sobre conceitos abordados nas outras seções.

First Steps

Installation and Environment

-> For installation, check out this video and the official documentation.

-> Alternatively, you can follow these steps:

- **Create a folder for your project:**
  - e.g.: *D:\Workspace\Unity\ML-Agents*
- **Open Command Prompt (or your terminal of choice) and move to your workspace folder:**
  - *cd D:\Workspace\Unity\ML-Agents*;
- **Create a new environment:**
  - *python -m venv venv*
- **Activate your environment:**
  - *venv\Scripts\activate*
- **Update pip:**
  - *python -m pip install --upgrade pip*
- **Clone the ML-Agents Toolkit Repository:**
  - *git clone --branch release_22 https://github.com/Unity-Technologies/ml-agents.git*
- **Install PyTorch:**
  - *pip3 install torch~=2.2.1 --index-url https://download.pytorch.org/whl/cu121*
- **Install ml-agents and ml-agents-envs from cloned repository:**
  - *cd D:\Workspace\Unity\ML-Agents\ml-agents*
  - *python -m pip install ./ml-agents-envs*
  - *python -m pip install ./ml-agents*

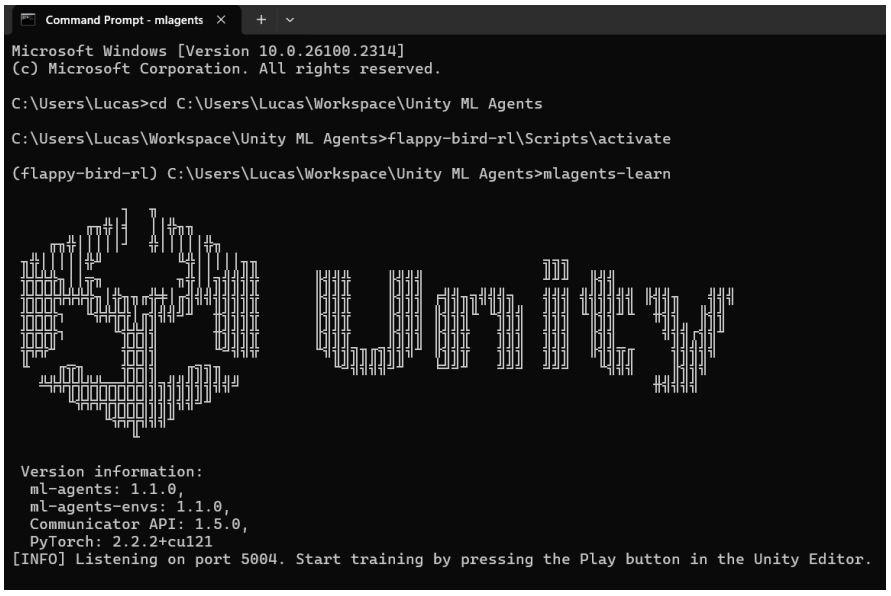
-> Check if everything is set up correctly by running the command “mlagents-learn --help”.

Training Models

-> After setting up your environment, for training models you need to:

- Open Command Prompt (or your terminal of choice);
- Move to your workspace folder (e.g.: *cd D:\Workspace\Unity\ML-Agents*);
- Activate your environment (e.g.: *venv\Scripts\activate*);
- Run the command "*mlagents-learn*".

If you did everything right, you should see the following screen:



Creating a new Unity Project

-> For creating a new Unity Project from one of the example scenes:

- Choose one of the examples from this tutorial;
- In the Projects tab on Unity Hub, add a new project from disk;
- Search for the folder location of the installed mlagents package (e.g: *D:\Workspace\Unity\ML-Agents\ml-agents*);
- Select the folder *ml-agents\Project* for the new Unity Project;
    - Alternatively, you can also change the name *Project* to what you want (e.g.: *AI For Games*)
- If necessary, install the corresponding Unity Editor version (2023.2.13f1 at the time of this tutorial);
- Inside the Unity Project, go to *Assets\ML-Agents\Examples*;
- Choose one of the example scenes and open the scene.

Editor

-> For VSCode:

- Open VSCode and and install the “Unity” extension.

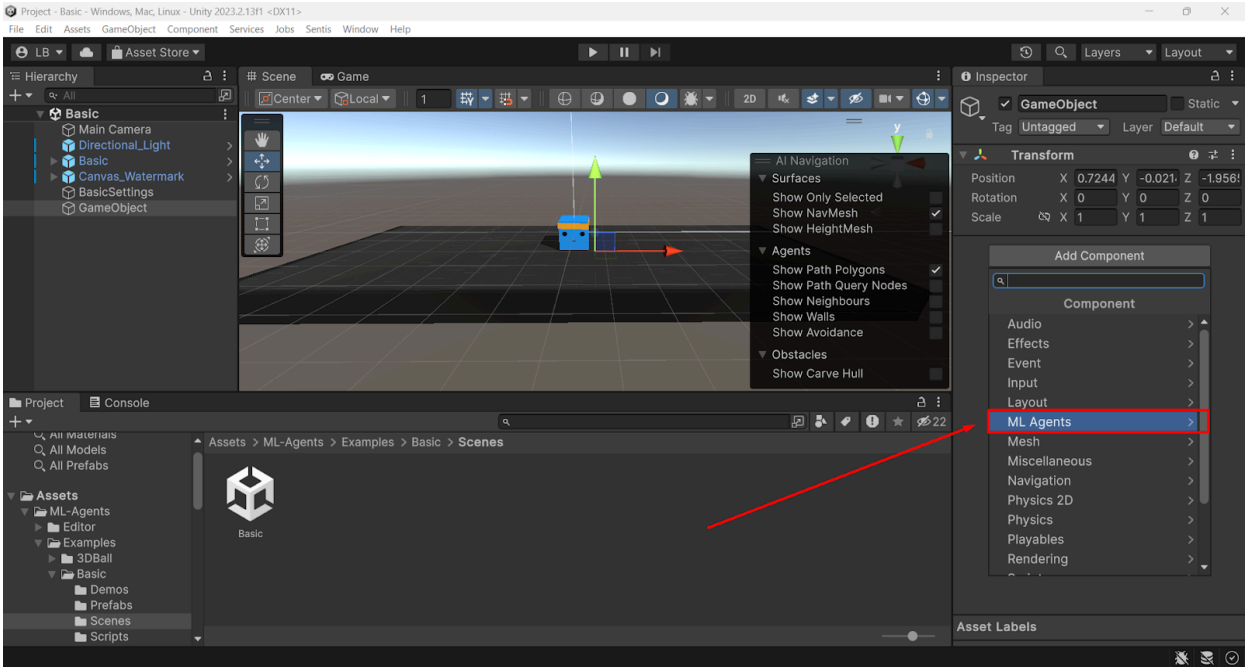
-> For Visual Studio:

- When installing the editor, remember to check the box “Game Development with Unity”
- Open your Unity Project and go to the Package Manager Window by navigating to *Window -> Package Manager* in the menu;
- Navigate to the “*Visual Studio Editor*” Package and install it.

ML Agents Package in Unity Project

-> For installing the package inside the Unity Project:

- Open your Unity Project and go to the Package Manager Window by navigating to *Window -> Package Manager* in the menu;
- Select the Unity Registry option;
- Navigate to the ML-Agents Package and install it.
- To verify if everything is working correctly, create an empty GameObject and check if there is a new component called “ML Agents”, as can be seen in the picture below:



Basics

This section will touch on the basics of RL[3] and PyTorch[4], as well as the basics of Unity[5]. It is meant to be used for consulting and reviewing the concepts.

[3] https://unity-technologies.github.io/ml-agents/Background-Machine-Learning/
[4] https://unity-technologies.github.io/ml-agents/Background-PyTorch/
[5] https://unity-technologies.github.io/ml-agents/Background-Unity/

Reinforcement Learning

The reinforcement learning cycle.

Reinforcement learning is a type of sequential decision-making process where an agent learns to achieve objectives by interacting with an environment. For example, a firefighting robot **perceives its environment through sensors** (e.g., cameras, heat detectors), **processes this information**, and **performs actions** (e.g., moving, turning a water hose) **to achieve its goal** — neutralizing a fire.

The core objective in RL is to learn a **policy**, which is a mapping from **observations** (sensor inputs) to **actions**. Observations reflect what the agent perceives, while actions represent changes the agent makes in the environment (e.g., adjusting position or activating equipment).

The last remaining piece of the reinforcement learning task is the **reward signal**, which guides the learning process by assigning positive or negative values based on task performance. For example, the robot may receive a large positive reward for putting out a fire and a small negative reward for delays. These rewards are often **sparse**, appearing only at critical moments, which makes learning good policies challenging, especially in complex environments.

Training in RL involves many trials where the agent explores different scenarios and iteratively improves its policy. It also requires careful **attribute selection** (choosing the most useful observations) and **model selection** (defining the policy structure and its parameters). Training is typically iterative, requiring adjustments to both attributes and models to optimize performance.

Imitation Learning

Imitation learning is a technique where an agent learns by mimicking demonstrations provided by an expert rather than relying solely on trial-and-error exploration. The expert demonstrations define the desired behaviors, making the agent's learning process faster and more stable.

Instead of learning exclusively from rewards, imitation learning focuses on minimizing the difference between the agent's actions and those in the demonstrations. This is particularly useful in complex environments where creating a meaningful reward function or allowing an agent to explore freely could lead to suboptimal behavior or inefficiency.

Curriculum Learning

Curriculum learning involves structuring the training process by gradually increasing the difficulty of the tasks presented to the agent. Similar to how humans learn, the agent begins with simpler tasks and progresses to more complex challenges as its abilities improve.

For instance, in the firefighting robot example, the robot might first learn to navigate an empty room, then move to a room with obstacles, and eventually handle environments with active fires. This approach helps the agent build foundational skills before tackling more complex scenarios, resulting in faster and more effective learning.

By breaking down tasks and adjusting their complexity over time, curriculum learning can significantly improve the efficiency and stability of the training process.

PyTorch

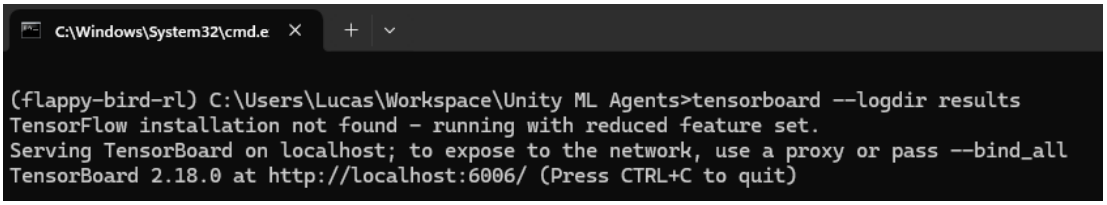
PyTorch is an open source library for performing computations using data flow graphs, the underlying representation of deep learning models. It facilitates training and inference on CPUs and GPUs in a desktop, server, or mobile device. Within the ML-Agents Toolkit, when you train the behavior of an agent, the output is a model (.onnx) file that you can then associate with an Agent. Unless you implement a new algorithm, the use of PyTorch is mostly abstracted away and behind the scenes.

TensorBoard

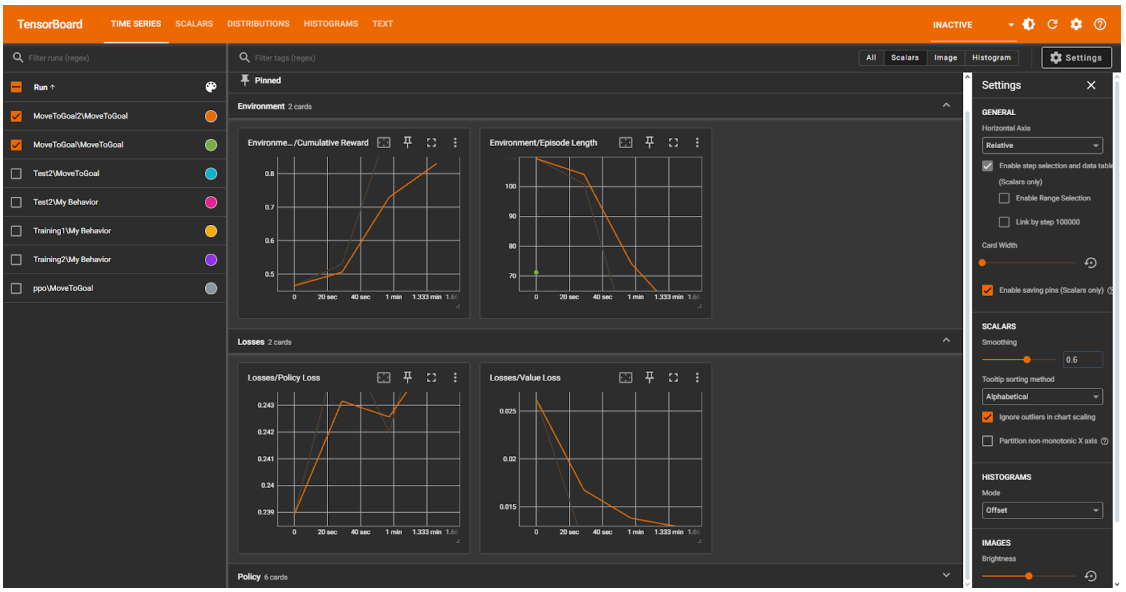
One component of training models with PyTorch is setting the values of certain model attributes (called *hyperparameters*). Finding the right values of these hyperparameters can require a few iterations. Consequently, ML Agents leverages a visualization tool called TensorBoard. It allows the visualization of certain agent attributes (e.g. reward) throughout training which can be helpful in both building intuitions for the different hyperparameters and setting the optimal values for your Unity environment. The Unity Team provide more details on setting the hyperparameters in the Training ML-Agents page.

To use TensorBoard for visualization, you can do the following:

- Open Command Prompt (or your terminal of choice);
- Move to your workspace folder (e.g.: *cd D:\Workspace\Unity\ML-Agents*);
- Activate your environment (e.g.: *venv\Scripts\activate*);
- Run the command "*tensorboard --logdir results*".



You will get this message stating that TensorBoard is running at http://localhost:6006/. Open your browser and type in that address. There you can see information about your runs, including Cumula tive Reward, Episode Length, Loss, etc.:


TensorBoard
TIME SERIES
SCALARS
DISTRIBUTIONS
HISTOGRAMS
TEXT
INACTIVE
Filter runs (regex)
Filter tags (regex)
All
Scalars
Image
Histogram
Settings
Run
MoveToGoal2\MoveToGoal
MoveToGoal\MoveToGoal
Test2\MoveToGoal
Test2\My Behavior
Training1\My Behavior
Training2\My Behavior
ppo\MoveToGoal
Pinned
Environment 2 cards
Environme.../Cumulative Reward
Environment/Episode Length
Losses 2 cards
Losses/Policy Loss
Losses/Value Loss
Policy 6 cards
GENERAL
Horizontal Axis
Relative
Enable step selection and data table
(Scalars only)
Enable Range Selection
Link by step 100000
Card Width
Enable saving pins (Scalars only)
SCALARS
Smoothing
0.6
Tooltip sorting method
Alphabetical
Ignore outliers in chart scaling
Partition non-monotonic X axis
HISTOGRAMS
Mode
Offset
IMAGES
Brightness

Unity

Editor

(A) The Toolbar provides access to your Unity Account and Unity Cloud Services. It also contains controls for Play mode; Undo history; Unity Search; a layer visibility menu; and the Editor layout menu.
(B) The Hierarchy window is a hierarchical text representation of every GameObject in the Scene. Each item in the Scene has an entry in the hierarchy, so the two windows are inherently linked. The hierarchy reveals the structure of how GameObjects attach to each other.
(C) The Game view simulates what your final rendered game will look like through your Scene Cameras. When you click the Play button, the simulation begins.
(D) The Scene view allows you to visually navigate and edit your Scene. The Scene view can display a 3D or 2D perspective, depending on the type of Project you are working on.
(E) Overlays contain the basic tools for manipulating the Scene view and the GameObjects within it. You can also add custom Overlays to improve your workflow.
(F) The Inspector window allows you to view and edit all the properties of the currently selected GameObject. Because different types of GameObjects have different sets of properties, the layout and contents of the Inspector window change each time you select a different GameObject.
(G) The Project window displays your library of Assets that are available to use in your Project. When you import Assets into your Project, they appear here.
(H) The status bar provides notifications about various Unity processes, and quick access to related tools and settings.

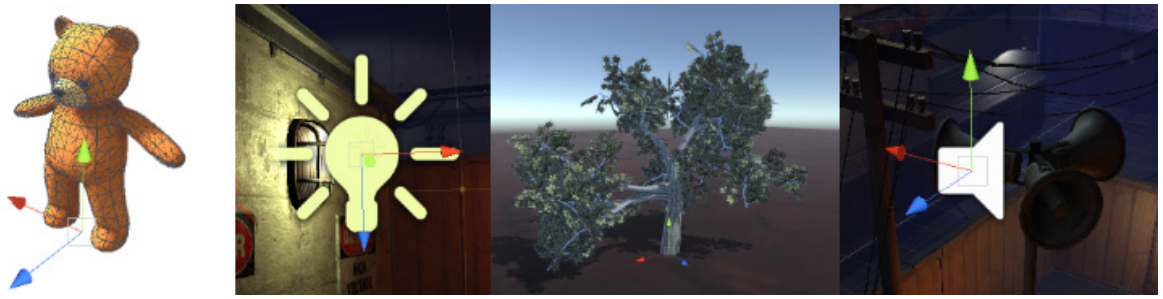
Scene

A Scene contains the environments and menus of your game. Think of each unique Scene file as a **unique level**. In each Scene, you place your environments, obstacles, and decorations, essentially designing and building your game in pieces.

Scenes are where you work with content in Unity. They are assets that contain all or part of a game or application. For example, you might build a simple game in a single scene, while for a more complex game, you might use one scene per level, each with its own environments, characters, obstacles, decorations, and UI. You can create any number of scenes in a project.

GameObject

GameObjects are the fundamental objects in Unity scenes, representing **characters, props, scenery, cameras, and more**. They act as containers for components, which define their functionality. Without components, a GameObject is empty and performs no actions.



Four different types of GameObject: an animated character, a light, a tree, and an audio source

To give a GameObject properties, you attach components such as a Light, Mesh Renderer, or Collider. Unity provides many built-in components, and you can create custom ones using the Unity Scripting API. For example, a light is created by adding a Light component, while a solid cube object has a Mesh Filter and Mesh Renderer component, to draw the surface of the cube, and a Box Collider component to represent the object's solid volume in terms of physics.

Every GameObject includes a Transform component, which defines its position, rotation, and scale. This component cannot be removed. Additional components can be added via the editor or scripts, and Unity also provides pre-made GameObjects like cameras and shapes in the **GameObject > 3D Object** menu.

GameObjects serve as the building blocks of Unity scenes, connecting functionality and appearance through components. They also offer methods for managing components, interacting with other GameObjects, and controlling their behavior in the scene.

RigidBody

A component that allows a GameObject to be affected by simulated gravity and other forces. The Rigidbody component can be used to apply a Rigidbody to your GameObject. A Rigidbody provides a physics-based way to control the movement and position of a GameObject. Instead of the Transform properties, you can use simulated physics forces and torque to move the GameObject, and let the physics engine calculate the results.

Camera

Just as cameras are used in films to display the story to the audience, Cameras in Unity are used to **display the game world to the player**.

Cameras display the game world to the player by capturing the 3D scene and flattening it for a 2D screen. Every Unity scene requires at least one camera, though you can have multiple cameras to create unique effects, render specific screen areas, or control rendering order.

Cameras can be customized, scripted, or parented to achieve various effects. For example, a static camera might suit a puzzle game, while a first-person shooter camera could be parented to the player character at eye level. Racing games often use cameras that follow the player's vehicle.

To create a camera, you attach a Camera component to a GameObject. This flexibility allows for endless customization, enabling you to tailor how the game world is presented to the player.

Scripting

Scripting in Unity involves writing code to define your project's functionality beyond the capabilities of the Unity Editor UI. By using Unity's public APIs, you can achieve finer control and greater customization over your project. Unity scripting primarily uses **C#**, a managed, object-oriented programming language that runs on the .NET platform.

Unity's scripting environment encompasses:

- **Editor APIs**: Customize and extend Unity's authoring tools to enhance workflows.
- **Engine APIs**: Define runtime functionalities such as graphics, physics, character behavior, and user input responses.

C# scripts (files with a.cs file extension) are assets in your project, stored in the Assets folder and saved as part of the asset database. Scripts can be created via the **Assets > Create > Scripting submenu** and are opened for editing in an External Script Editor. Usually this will be one of the supported IDEs for Unity development.

You can also create your own regular C# types and logic to use in your game, as long as the code you write is compatible with the active .NET profile. Scripts gain Unity-specific functionality when inheriting from built-in types:

- **MonoBehaviour**: Makes the script attachable to a GameObject as a component, enabling it to control GameObject behavior.
- **UnityEngine.Object**: Makes custom types assignable to fields in the Inspector.

Unity's extensibility allows developers to combine custom logic and Unity APIs to define both the development process and the gameplay experience.

Physics

Unity helps you simulate physics in your Project to ensure that the objects correctly accelerate and respond to collisions, gravity, and various other forces. Unity provides different physics engine implementations which you can use according to your Project needs: 3D, 2D, object-oriented, or data-oriented. For more information on Unity Engine's Physics system, check here.

Ordering of Event Functions

Event functions are a set of built-in events that your MonoBehaviour scripts can optionally subscribe to by implementing the appropriate methods, often referred to as callbacks. The callbacks correspond to events in core Unity subsystems like physics, rendering, and user input, or to stages of the script's own lifecycle such as its creation, activation, frame-dependent and frame-independent updates, and destruction. When the event occurs, Unity invokes the associated callback on your script, giving you the opportunity to implement logic in response to the event.

To the extent that Unity raises these events and calls the associated MonoBehaviour callbacks in a predetermined order, the order is documented here. It's important to understand the execution order so you don't try to use one callback to do work which depends on another callback that hasn't been invoked yet. However, bear in mind that some callbacks are for events, such as those triggered by user inputs, which can occur at any time while your game is running.

Prefabs

Unity's Prefab system enables you to create and store a **GameObject** with its components, properties, and child objects as a reusable **Asset**. Prefabs serve as templates to instantiate consistent and reusable GameObjects across scenes or within the same scene. The Prefab system ensures efficiency and consistency while allowing flexibility for customization and runtime use.

When you need to reuse a GameObject, such as an NPC, prop, or environment element, converting it to a Prefab is more efficient than copying and pasting. Changes made to the Prefab Asset **automatically update all its instances**, ensuring consistency across your project. Prefabs can also be nested within other Prefabs, allowing for complex object hierarchies with multi-level editing.

Prefab instances can have unique overrides, letting you customize specific instances without affecting others. You can also create variants — Prefabs with grouped overrides — to manage meaningful variations.

Prefabs are essential for runtime instantiation of GameObjects that don't exist at scene startup, such as power-ups, effects, projectiles, or NPCs.

Common Uses:

- **Environmental Assets**: Repeated elements like trees or rocks.
- **NPCs**: Reused across scenes with variations in behavior or sound via overrides.
- **Projectiles**: Example: cannonballs fired from a pirate's cannon.
- **Player Character**: Placed as a Prefab in starting positions across different levels.

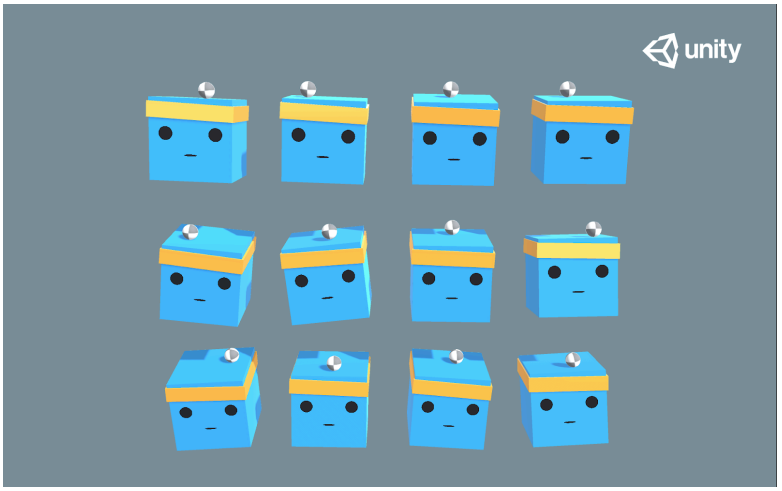
ML-Agents

This section will explain the usage of ML Agents:

- For a quick introduction to ML-Agents, check out the **Getting Started Guide**[6] section.
- For a broad overview of reinforcement learning, imitation learning and all the training scenarios, methods and options within the ML-Agents Toolkit, check out the **Toolkit Overview**[7] section.
- For more information on the various training options available, check out the **Training ML-Agents**[8] section.
- For a "Hello World" introduction to creating and design your own Learning Environment, check out the **Learning Environment**[9] section.
- For an overview on the more complex example environments that are provided in this toolkit, check out the **Example Environments**[10] page from Unity.
- For an overview of the usage of the Agent class, check out the **Agents**[11] section.
- Finally, for more information about concepts aborded on the project sections, check out the **Glossary** section.

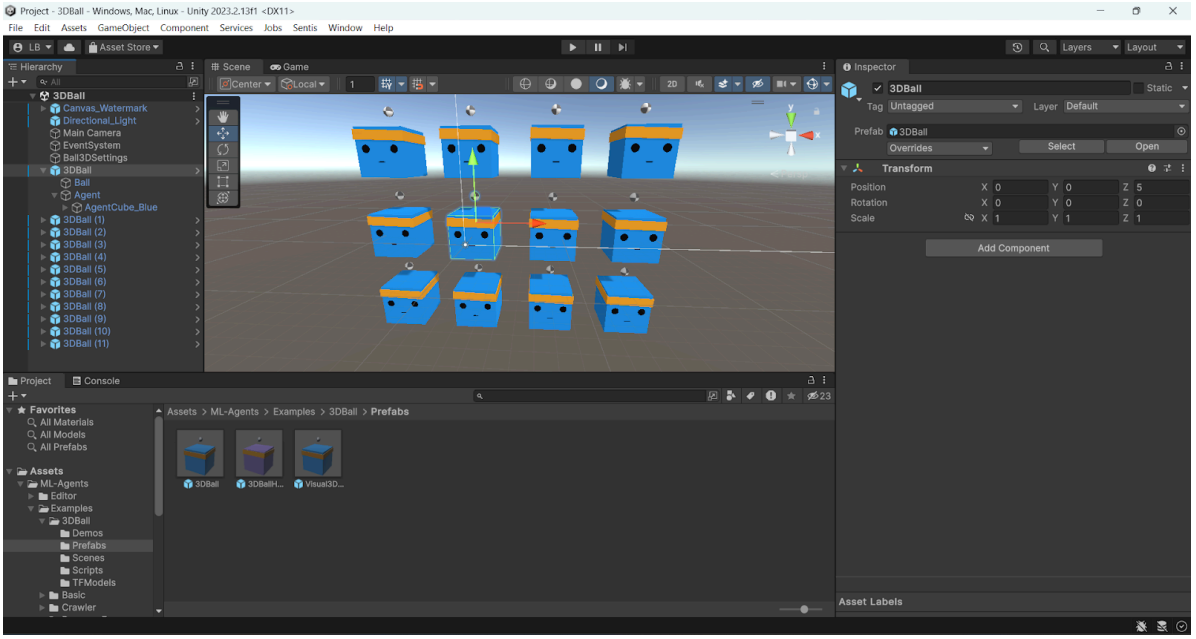
[6] https://unity-technologies.github.io/ml-agents/Getting-Started/
[7] https://unity-technologies.github.io/ml-agents/ML-Agents-Overview/
[8] https://unity-technologies.github.io/ml-agents/Training-ML-Agents/
[9] https://unity-technologies.github.io/ml-agents/Learning-Environment-Design/
[10] https://unity-technologies.github.io/ml-agents/Learning-Environment-Examples/
[11] https://unity-technologies.github.io/ml-agents/Learning-Environment-Design-Agents/

Getting Started Guide



For this section, we'll use the 3D Balance Ball environment, one of the example environments from Unity, which contains a number of agent cubes and balls (which are all copies of each other). Each agent cube tries to keep its ball from falling by rotating either horizontally or vertically. In this environment, an agent cube is an Agent that receives a reward for every step that it balances the ball. An agent is also penalized with a negative reward for dropping the ball. The goal of the training process is to have the agents learn to balance the ball on their head.

Understanding a Unity Environment



Agent

A Unity environment consists of a scene where Agent objects interact with other entities. Agents are autonomous actors that observe their environment, make decisions, and take actions to achieve specific objectives. In Unity, all elements in a scene are GameObjects, which act as containers for components like behaviors, graphics, and physics. You can inspect a GameObject's components using the Inspector window in the Unity Editor.

In the 3D Balance Ball example, the environment includes multiple agent cubes (12 in this case), each acting independently but sharing the same Behavior. This setup allows parallel training, where all agents contribute to learning simultaneously.

An Agent observes its environment and takes actions based on those observations. Key properties of an Agent include:

- **Behavior Parameters:** Define how the Agent makes decisions.
- **Max Step:** Specifies the maximum number of simulation steps before the Agent's episode ends. In 3D Balance Ball, this is set to 5000 steps.

Vector Observation Space

Agents collect observations to understand their environment and make decisions. In **3D Balance Ball**, the **vector observation space** has a size of 8, representing:

- The **x** and **z** rotation of the agent cube.
- The **x**, **y**, and **z** position and velocity of the ball relative to the cube.

Actions

Agents execute decisions through **actions**, which can be:

- **Continuous Actions**: Floating-point numbers that vary continuously.
- **Discrete Actions**: Actions selected from a predefined set of discrete choices.

In **3D Balance Ball**, the Agent uses continuous actions with a **space size of 2**, controlling the **x** and **z** rotations of the agent cube to keep the ball balanced.

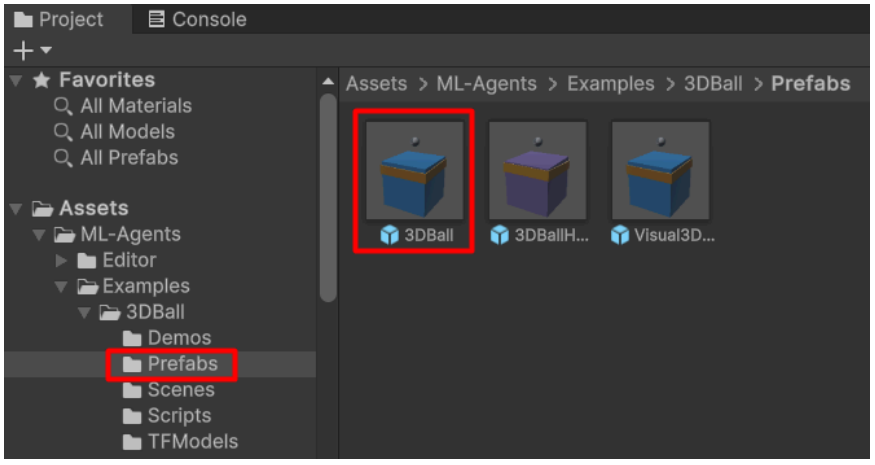
Running a pre-trained model

The ML-Agents Toolkit includes pre-trained models (.onnx files) that are executed inside Unity using Sentis. To use the pre-trained model for the 3D Ball example:

1. Locate the Agent Prefab:

- In the **Project** window, navigate to “Assets/ML-Agents/Examples/3DBall/Prefabs”.
- Expand **3DBall** and select the **Agent prefab**. Its properties will appear in the **Inspector** window.

Since the platforms in the 3DBall scene use the 3DBall prefab, updating the prefab automatically updates all 12 platforms:

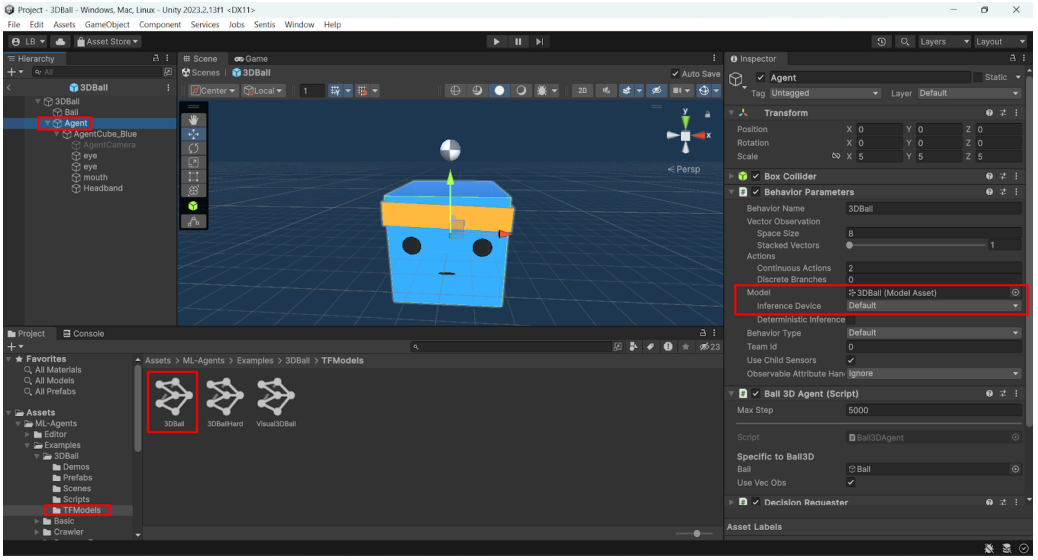


2. Assign the Pre-Trained Model:

- Drag the 3DBall Model file from “Assets/ML-Agents/Examples/3DBall/TFModels” into the **Model** field under the **Behavior Parameters (Script)** component in the **Inspector** window.
- Verify that all Agents in the **Hierarchy** window now use **3DBall** as their model in the **Behavior Parameters**.

3. Set the Inference Device:

- In the **Behavior Parameters**, set the **Inference Device** to **Default**.



4. Run the Scene:

- Click **Play** in the Unity Editor to see the platforms balance the balls using the pre-trained model.

Training a new model with Reinforcement Learning

For custom environments, you will need to train agents from scratch to generate new model files. The *mlagents-learn* command provides a convenient way to configure and initiate training.

1. Set Up the Training Environment:

- Open Command Prompt (or your terminal of choice);
- Move to your workspace folder (e.g.: *cd D:\Workspace\Unity\ML-Agents*);
- Activate your environment (e.g.: *venv\Scripts\activate*);

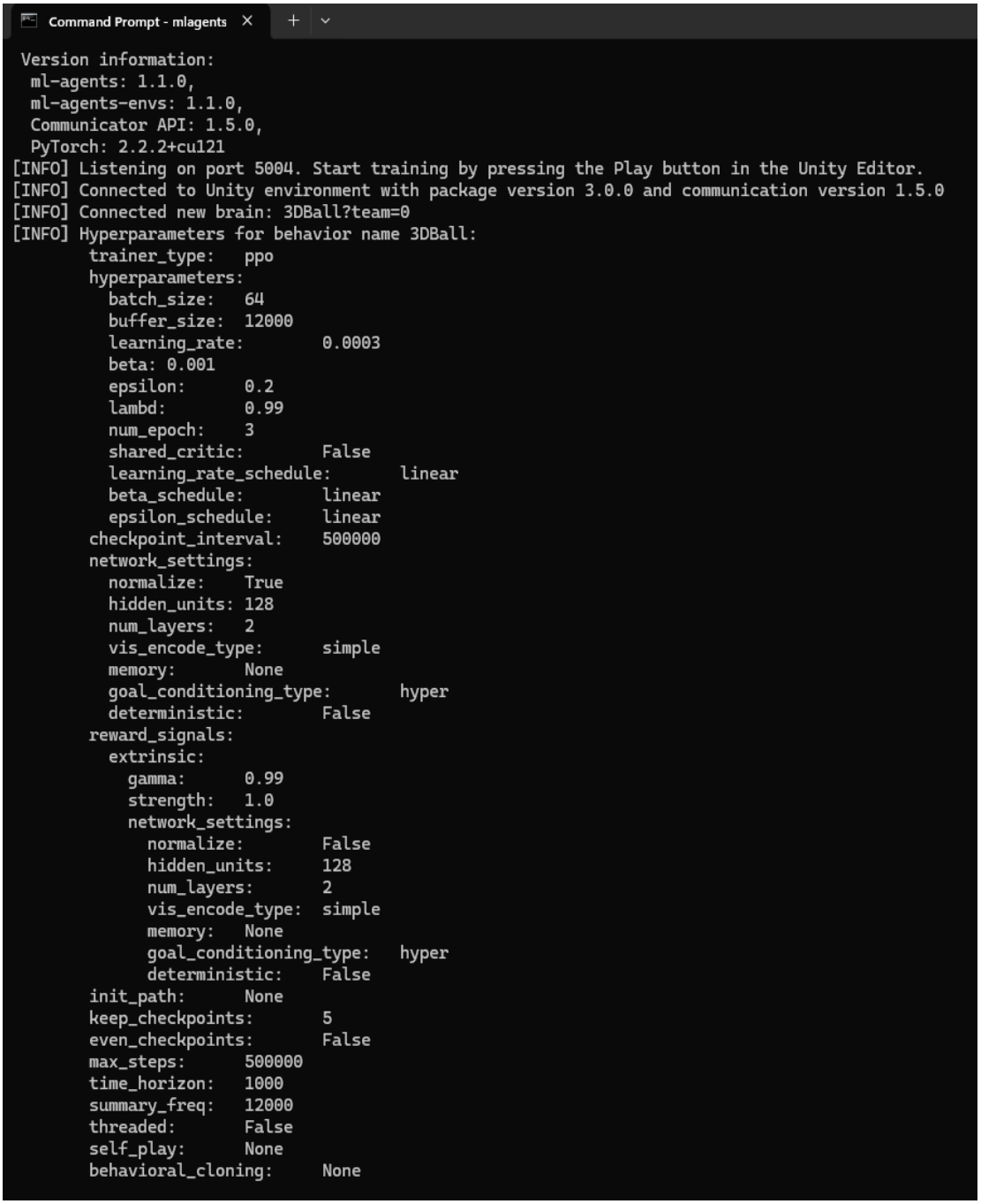
2. Setup Python API for Training:

- Run the Training Command: "*mlagents-learn config/ppo/3DBall.yaml --run-id=first3DBallRun*"
- "*config/ppo/3DBall.yaml*" is the path to the training configuration file. The config/ppo folder contains configuration files for all example environments, including 3DBall.
- "*--run-id*" specifies a unique name for the training session.

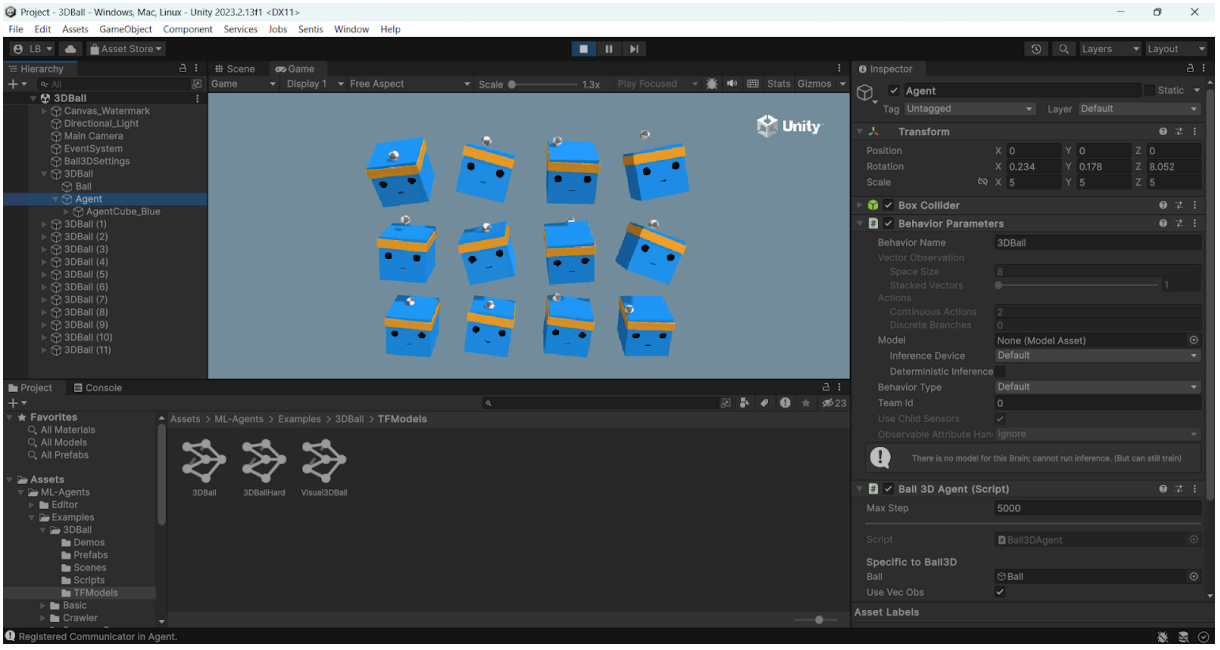
3. Start Training in Unity:

- Make sure that the prefab of the Agent is not using any model.
- When the terminal displays "Start training by pressing the Play button in the Unity Editor", you can press Play in Unity to begin training.

If mlagents-learn runs correctly and starts training, you should see something like this:



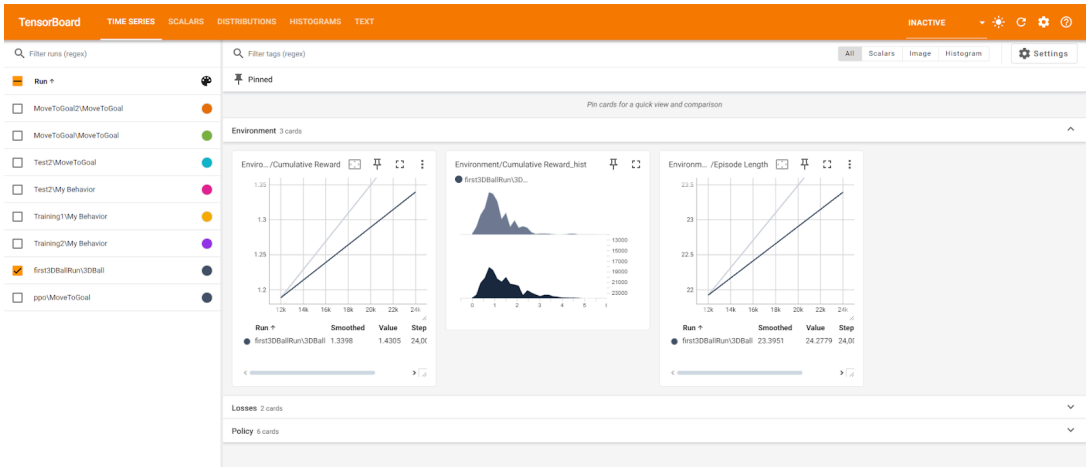
Observing the Training Progress



Once you start training using mlagents-learn in the way described in the previous section, the ml-agents directory will contain a results directory. In order to observe the training process in more detail, you can use TensorBoard. From the command line run:

- "*tensorboard --logdir results*"

Then navigate to *localhost:6006* in your browser to view the TensorBoard summary statistics as shown below. For the purposes of this section, the most important statistic is *Environment/Cumulative Reward* which should increase throughout training, eventually converging close to 100 which is the maximum reward the agent can accumulate in this scenario.



Embedding the Model into the Unity Environment

Once the training process completes, and the training process saves the model (denoted by the *Saved Model* message) you can add it to the Unity project and use it with compatible Agents (the Agents that generated the model).

- **Note:** Do not just close the Unity Window once the Saved Model message appears. Either wait for the training process to close the window or press Ctrl+C at the command-line prompt. If you close the window manually, the .onnx file containing the trained model is not exported into the ml-agents folder.

If you've quit the training early using Ctrl+C and want to resume training, run the same command again, appending the --resume flag:

- "*mlagents-learn config/ppo/3DBall.yaml --run-id=first3DBallRun --resume*"

Your trained model will be at results/<run-identifier>/<behavior_name>.onnx where <behavior_name> is the name of the Behavior Name of the agents corresponding to the model. This file corresponds to your model's latest checkpoint. You can now embed this trained model (either for further training or for inference) into your Agents by following the steps below, which is similar to the steps described above.

1. Move your model file into *Project/Assets/ML-Agents/Examples/3DBall/TFModels/*.
2. Open the Unity Editor, and select the 3DBall scene as described above.
3. Select the 3DBall prefab Agent object.
4. Drag the <behavior_name>.onnx file from the Project window of the Editor to the Model placeholder in the Ball3DAgent inspector window.
5. Press the Play button at the top of the Editor.

Toolkit Overview

The Unity Machine Learning Agents Toolkit (ML-Agents Toolkit) is an open-source platform for using games and simulations as environments to train intelligent agents. It supports various machine learning methods, including **reinforcement learning**, **imitation learning**, and **neuroevolution**, through a user-friendly Python API.

The toolkit provides PyTorch-based implementations of advanced algorithms, enabling developers and hobbyists to train agents for 2D, 3D, and VR/AR games. These trained agents can be utilized for tasks such as **controlling NPC behavior** (in scenarios like multi-agent or adversarial setups), **automated game testing**, and **evaluating game design decisions** pre-release.

Running Example: Training NPC Behaviors

To explain the concepts and terminology of ML-Agents, we'll use the example of training an NPC medic in a multiplayer, war-themed game. In this scenario, two teams of five players each have one NPC medic tasked with finding and reviving wounded teammates. The medic must balance avoiding danger, prioritizing injured teammates, and positioning itself for optimal assistance.

Manually coding such complex behavior is challenging due to the variety of environmental conditions and possible actions. Using ML-Agents, we can train the medic's behavior with **reinforcement learning** by defining three key entities in the game environment:

- **Observations**: These represent what the medic perceives about its environment. Observations can be **numeric** (e.g., teammate positions, injuries, and enemy locations) or **visual** (camera images showing what the medic "sees"). Observations are **limited to the medic's awareness**, excluding unseen aspects of the environment, like hidden enemies.
    - **The Agent Observations are different from the environment (or game) state.** The environment state represents information about the entire scene containing all the game characters. The agents observation, however, only contains information that the agent is aware of and is typically a subset of the environment state.
- **Actions**: These define what the medic can do. Actions can be **discrete** (e.g., move north, south, east, or west) or **continuous** (e.g., adjust direction and speed for movement). The complexity of the environment determines the type of action space.
- **Reward Signals**: Rewards are scalar values indicating performance. For example, the medic might receive negative rewards for dying, positive rewards for reviving

teammates, and negative rewards if a teammate dies. Rewards align with the task objectives, guiding the agent to learn optimal behavior.

With these entities defined, the medic learns its behavior by running simulations during a **training phase**, refining its decisions to maximize future rewards. The learned behavior is called a **policy**, mapping observations to optimal actions. During the **inference phase**, the trained NPC uses this policy to make real-time decisions in the game.

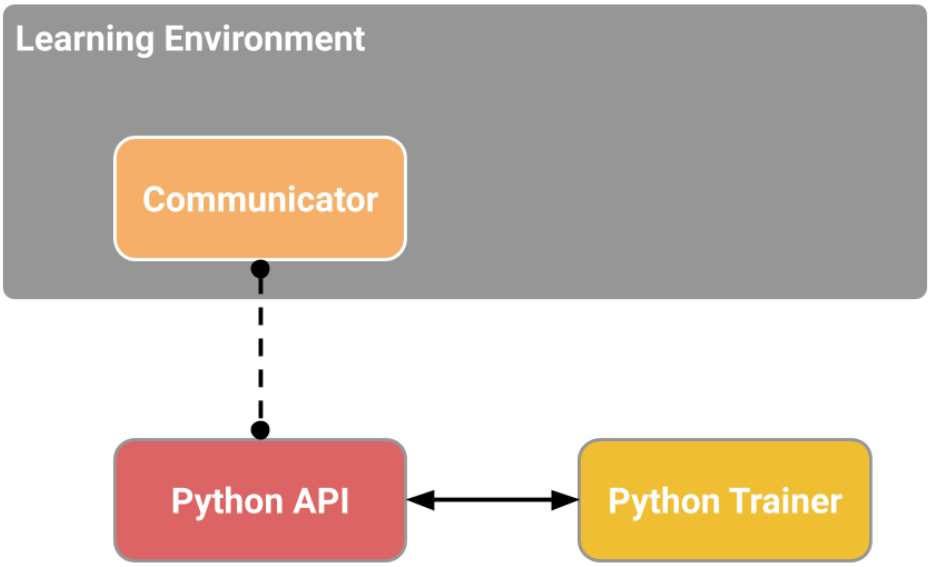
The ML-Agents Toolkit provides the necessary tools to use Unity as a simulation engine, enabling the training of agent behaviors across various game objects. Subsequent sections will explore its features and functionalities in detail.

Key Components

The ML-Agents Toolkit is structured around **five main components**, each contributing to the process of training intelligent agents:

1. **Learning Environment:** The Learning Environment consists of the Unity scene and all its game characters. It provides the environment in which agents observe, act, and learn.
    - Unity scenes can be configured for specific reinforcement learning problems or broader simulations.
    - For complex training, it may be practical to create a separate purpose-built training scene.
    - The **ML-Agents Unity SDK** (com.unity.ml-agents) transforms any Unity scene into a learning environment by defining agents and their behaviors.
2. **Python Low-Level API:** This API allows interaction with and manipulation of the Learning Environment from Python.
    - It exists outside Unity, communicating through the **External Communicator**.
    - Found in the mlagents_envs Python package, it supports training and alternative uses, such as integrating Unity with custom machine learning algorithms.
3. **External Communicator:** The External Communicator acts as a bridge, connecting the Learning Environment (Unity) with the Python Low-Level API. It facilitates data exchange during training.
4. **Python Trainers:** This includes all the machine learning algorithms used to train agents.
    - Implemented in the mlagents Python package.
    - Provides the mlagents-learn command-line tool to handle training methods and configurations.
    - Trainers interact directly with the Python Low-Level API to optimize agent behavior.

5. **Gym and PettingZoo Wrappers (not pictured)**
    - **Gym Wrapper**: Integrates Unity environments with algorithms utilizing OpenAI's gym API. More information about that can be found here.
    - **PettingZoo Wrapper**: Provides a gym-like interface for multi-agent simulation environments, expanding compatibility with machine learning frameworks. More information about that can be found here.

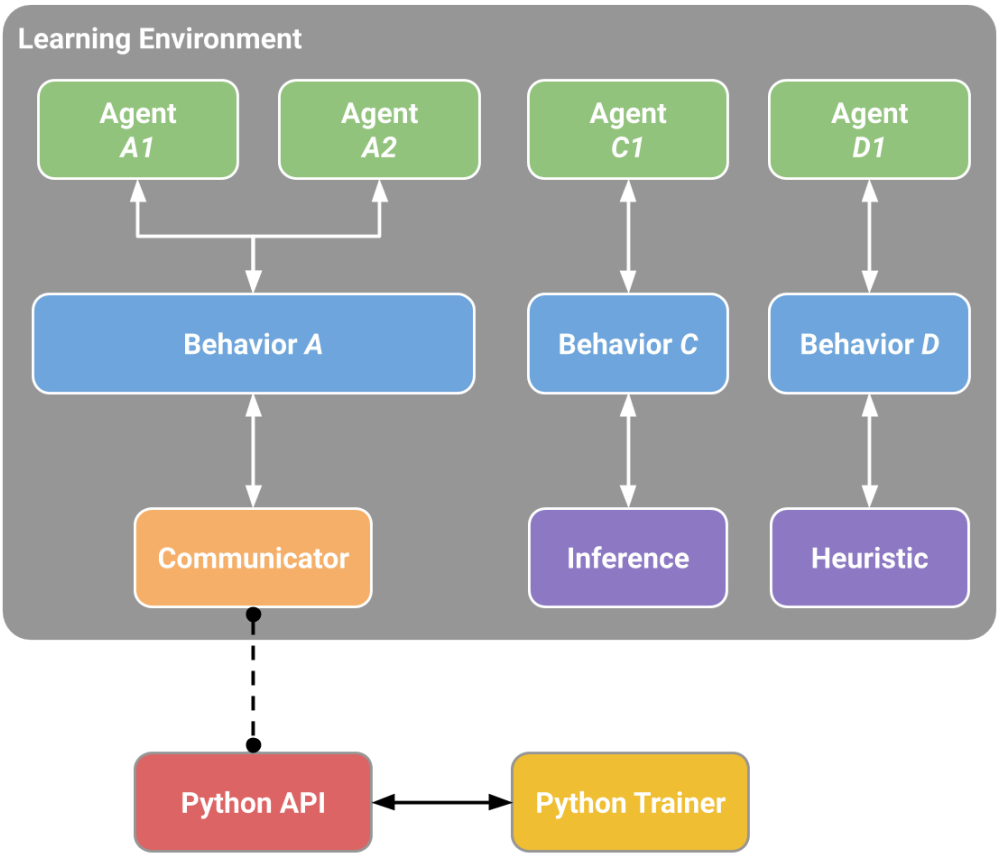


Simplified block diagram of ML-Agents.

The Learning Environment relies on **two key Unity components** to organize scenes:

- **Agents**: Attached to GameObjects, agents generate observations, perform actions, and assign rewards. Each Agent is associated with a Behavior.
- **Behavior**: A Behavior can be thought as a function that receives observations and rewards from the Agent and returns actions. It also specifies an Agent's **action space** and **decision-making process**. Behaviors can be:
    - **Learning**: Untrained and prepared for training.
    - **Heuristic**: Defined by hardcoded rules in code.
    - **Inference**: Includes a trained neural network for decision-making.

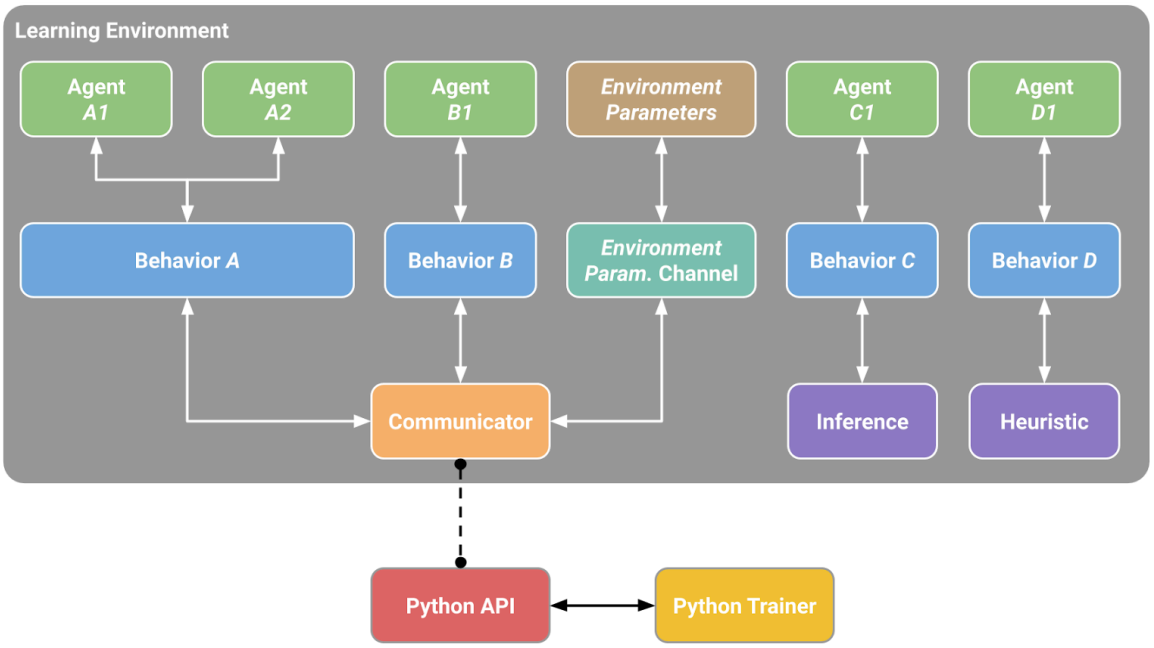
Every Learning Environment will always have one Agent for every character in the scene. While each Agent must be linked to a Behavior, it is possible for Agents that have similar observations and actions to have the same Behavior. In the example game, two medics (Agents) would **share a single Behavior but operate independently**, with unique observations and actions tailored to their situations.



Example block diagram of ML-Agents Toolkit for our sample game.

In the **Example ML-Agents Scene Block Diagram** for our sample game, the two medics share a Behavior. However, future expansions, such as adding NPC tank drivers, would require separate Behaviors for different roles. The Academy (not shown in the diagram) ensures synchronization between Agents and manages environment-wide settings.

Lastly, it is possible to exchange data between Unity and Python outside of the machine learning loop through **Side Channels**. Side Channels allow for data exchange between Unity and Python outside the training loop. For example, they can dynamically modify environment parameters during training. One example of using Side Channels is to exchange data with Python about Environment Parameters. The following diagram illustrates the above:



Training Modes

The ML-Agents Toolkit provides flexible ways to approach training and inference, allowing agents to learn and operate efficiently within Unity environments. During the training process, agents like medics send their observations to the Python API through the External Communicator. The API processes these observations and returns exploratory actions that help the model learn the optimal policy for each agent. This iterative process allows agents to refine their behavior by maximizing rewards. Once training concludes, the learned policy is exported as a model file. In the inference phase, the agents continue to generate observations, but instead of sending them to the Python API, they use their internal embedded models to determine optimal actions in real time.

It is important to note that the ML-Agents Toolkit leverages the Sentis to run the models within a Unity scene such that an agent can take the optimal action at each step. Given that Sentis support most platforms that Unity does, this means that any model you train with the ML-Agents Toolkit can be embedded into your Unity application that runs on any platform. More information about that can be found here.

For developers with custom needs, the toolkit allows for alternative training workflows. Instead of relying on built-in algorithms, users can implement their own machine learning solutions. In these cases, agent behaviors are controlled entirely within Python, offering greater flexibility and customization. Developers can even integrate Unity environments with machine learning frameworks like gym to experiment with novel training approaches. While

there is no specific tutorial for custom training, the Python API documentation provides detailed guidance on leveraging these advanced capabilities.

Flexible Training Scenarios

The ML-Agents Toolkit supports various training scenarios, enabling developers to create unique and complex environments. Below are some examples:

- **Single-Agent**: A single agent operates with its own reward signal, the traditional training setup.
  - **Example:** a character in a single-player game like *Chicken*.
- **Simultaneous Single-Agent**: Multiple independent agents with identical reward signals and Behavior Parameters operate in parallel. This speeds up and stabilizes training, especially when many agents must learn similar behaviors.
  - **Example:** several robotic arms learning to open doors simultaneously.
- **Adversarial Self-Play**: Two agents interact with opposing reward signals, competing against each other to improve. This setup helps agents refine their skills by facing an evenly matched opponent: themselves.
  - **Example:** training agents for games like *AlphaGo* or *Dota 2*.
- **Cooperative Multi-Agent**: Multiple agents share a reward signal and work together to achieve a common goal. This setup is ideal for tasks requiring collaboration or shared knowledge.
  - **Example:** solving puzzles or coordinating tasks with partial information.
- **Competitive Multi-Agent**: Agents compete for limited resources or strive to win in a competitive setting, with opposing reward signals.
  - **Example:** team sports or capture-the-flag games.
- **Ecosystem**: Multiple agents with independent reward signals coexist in a shared environment, simulating dynamic systems.
  - **Example:** wildlife on a savanna or an urban environment for autonomous driving simulations.

Reward Signals

Reinforcement learning centers on training an agent to discover a **policy** that maximizes its total reward. These rewards come in two types:

- **Extrinsic Rewards**: Defined by the environment, these rewards correspond to achieving goals, such as completing a task or reaching a target. They are external to the learning algorithm and directly tied to the problem the agent is solving.
- **Intrinsic Rewards**: Defined outside the environment, intrinsic rewards encourage certain behaviors or assist in learning extrinsic rewards. These signals can guide exploration or shape behavior, especially in environments where rewards are sparse.

In the ML-Agents Toolkit, rewards are modular and can be mixed to influence the agent's learning process. The toolkit provides four reward signal types:

- **extrinsic**: The default reward, based on the goals and tasks defined in the environment.
- **gail**: A reward signal derived from Generative Adversarial Imitation Learning (GAIL), which is used for imitation learning.
- **curiosity**: Encourages exploration in environments with sparse extrinsic rewards by rewarding the agent for visiting novel states (see below).
- **rnd**: Also used for exploration in sparse-reward settings, this reward signal comes from the Random Network Distillation (RND) module (see below).

Training Methods

The ML-Agents Toolkit includes state-of-the-art machine learning algorithms designed to train agents regardless of the specifics of the environment. These algorithms can often be treated as "black boxes," meaning that users do not need an in-depth understanding of the underlying methods to train agents effectively, although there are a few training-related parameters to adjust inside Unity as well as on the Python training side.

Environment-agnostic

This section specifically focuses on the training methods that are available regardless of the specifics of your learning environment.

Deep Reinforcement Learning

The ML-Agents Toolkit provides implementations of two major reinforcement learning algorithms: Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC).

PPO is the default algorithm due to its general-purpose applicability and stability. It operates on-policy, meaning it learns directly from the most recent experiences. SAC, on the other hand, is off-policy, allowing it to learn from a replay buffer containing past experiences. This makes SAC more sample-efficient, often requiring 5-10 times fewer samples than PPO for the same task. However, SAC needs more model updates and is well-suited for slower environments (around 0.1 seconds per step or longer). As a "maximum entropy" algorithm, SAC also intrinsically encourages exploration, enabling agents to discover new strategies naturally.

Curiosity for Sparse-Reward Environments

Sparse-reward environments, where rewards are rare or infrequent, pose a challenge for reinforcement learning. In such scenarios, intrinsic rewards can guide exploration, and Curiosity is a key tool for this purpose.

The curiosity reward signal activates the **Intrinsic Curiosity Module (ICM)**, which encourages the agent to explore novel states. This approach is based on the method described in *Curiosity-driven Exploration by Self-supervised Prediction* by Pathak et al. The ICM consists of two networks:

- **Inverse Model**: Encodes the current and next observations and predicts the action taken between them.
- **Forward Model**: Predicts the next encoded observation based on the current observation and action.

The loss of the forward model, which reflects the difference between predicted and actual encoded observations, serves as the intrinsic reward. The greater the model's surprise, the higher the reward, motivating the agent to explore unknown states.

Random Network Distillation (RND) for Sparse-Reward Environments

Similar to Curiosity, **Random Network Distillation (RND)** aids exploration in sparse-reward settings by rewarding the agent for visiting novel states. The RND module follows the method described in *Exploration by Random Network Distillation*.

RND uses two networks:

- **Fixed Random Network**: Generates encodings from observations using fixed, random weights.
- **Trainable Network**: Predicts the outputs of the fixed network, using the agent's observations as training data.

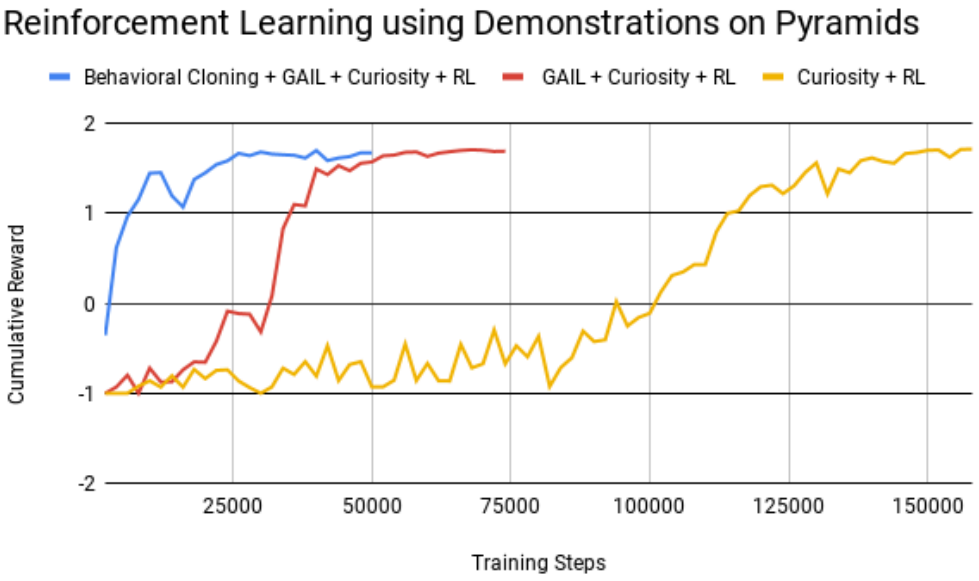
The intrinsic reward is derived from the prediction loss, calculated as the squared difference between the predicted and actual encodings. Over time, as the agent repeatedly visits the same state, the prediction becomes more accurate, reducing the reward. This mechanism encourages the agent to explore new states where prediction errors are higher.

Both Curiosity and RND are powerful tools for tackling environments where extrinsic rewards are sparse, guiding agents to discover meaningful strategies through exploration.

Imitation Learning

Imitation learning provides an intuitive way to train agents by demonstrating desired behaviors rather than relying solely on trial-and-error methods. For example, instead of using a reward function to train a medic, you can provide real-world examples of game observations paired with actions from a controller to guide its behavior. This method involves learning a policy based on demonstration data—pairs of observations and actions.

Imitation learning can work independently or alongside reinforcement learning. Used alone, it can teach specific behavior styles for solving tasks. When combined with reinforcement learning, it can significantly reduce the time required to train agents, particularly in sparse-reward environments. For instance, using six demonstration episodes in the Pyramids environment can reduce training steps by more than fourfold, as can be seen below:



The ML-Agents Toolkit allows agents to learn directly from demonstrations or use them to speed up reward-based training. It implements two key algorithms:

- **Behavioral Cloning (BC)**: This algorithm trains the agent to mimic actions directly from demonstrations. It works well when demonstrations cover nearly all the states the agent may encounter or when combined with GAIL and extrinsic rewards. Can be used alone or as a pre-training step for GAIL or RL.
- **Generative Adversarial Imitation Learning (GAIL)**: GAIL rewards agents for behavior that closely matches the provided demonstrations. It uses a discriminator network to differentiate between agent-generated actions and expert demonstrations, pushing the agent to "fool" the discriminator by mimicking the demonstrations. Works with or without extrinsic rewards to guide agents based on demonstrations.

Demonstrations are critical for both BC and GAIL. These can be recorded in the Unity Editor or build and saved as assets containing observations, actions, and rewards. These demonstration assets can be managed in the Editor and used during training.

Imitation learning can be combined with reinforcement learning (PPO or SAC) to guide agents more effectively:

1. For environments with sparse rewards, demonstrations can guide agents toward learning faster. Both GAIL and BC can be used at low strengths alongside extrinsic rewards, as demonstrated in the PushBlock environment configuration.
2. For training solely from demonstrations, GAIL and BC can work without an extrinsic reward signal, as shown in the CrawlerStatic environment configuration.

When using GAIL, it is important to balance the GAIL reward signal and extrinsic rewards carefully. GAIL introduces a survivor bias, incentivizing agents to align closely with demonstrations, which might conflict with goal-oriented tasks. In such cases, use a low GAIL reward signal and sparse extrinsic rewards to ensure task completion is prioritized.

Environment-specific

In addition to the three environment-agnostic training methods introduced in the previous section, the ML-Agents Toolkit provides additional methods that can aid in training behaviors for specific types of environments.

Self Play

ML-Agents supports training agents in competitive environments using Self-Play, suitable for both symmetric and asymmetric games. Symmetric games (e.g., Tennis, Soccer) feature agents with identical observations, actions, and objectives, allowing for shared policies. Asymmetric games (e.g., Hide and Seek) involve agents with different roles, making separate policies more effective.

Self-play allows agents to compete against fixed, past versions of opponents to ensure a stable learning process. This avoids instability caused by constantly changing, improving opponents. PPO is recommended over SAC for self-play due to SAC's challenges with non-stationary dynamics caused by changing opponents. For further reading on this issue in particular, see the paper Stabilising Experience Replay for Deep Multi-Agent Reinforcement Learning.

MA-POCA

ML-Agents enables training cooperative behaviors where agents work as a team toward shared goals. Agents typically receive collective rewards, making it challenging to identify individual contributions.

MA-POCA addresses this by using a centralized critic to guide learning, helping agents understand how to contribute effectively to the team's success. Rewards can also be

individualized, encouraging team collaboration to achieve personal goals. Agents can join or leave mid-episode (e.g., spawn or die) while still learning the impact of their actions on team performance, including self-sacrifice for team benefit.

MA-POCA can also be combined with self-play to train teams competing against one another. For more details, refer to the paper On the Use and Misuse of Absorbing States in Multi-Agent Reinforcement Learning.

Curriculum Learning

Curriculum learning gradually introduces more difficult aspects of a task, allowing the agent to build foundational skills before tackling complex challenges. This approach mirrors how humans learn—for example, mastering arithmetic before progressing to algebra or calculus.

In machine learning, simpler tasks act as scaffolding for harder tasks. For instance, instead of immediately training a medic to scale a wall to reach a wounded teammate (a task they would initially fail), we start with an easier scenario, like moving toward an unobstructed teammate. Once the simpler task is mastered, the difficulty is progressively increased, such as incrementally raising the wall's height. Over time, the medic learns to complete the once-impossible task.

The ML-Agents Toolkit supports curriculum learning by allowing custom environment parameters to be adjusted dynamically during training. This enables gradual task difficulty progression, helping agents achieve complex goals effectively.

Parameter Randomization

Agents trained on fixed environments often struggle to adapt to unseen variations, a problem known as overfitting. To counter this, ML-Agents provides Environment Parameter Randomization, a method where environment parameters are randomly varied during training.

This approach, inspired by Domain Randomization, exposes agents to a wide range of scenarios, making them more robust and capable of generalizing to new environments. By randomizing environment elements such as object properties or conditions, agents become better equipped to handle variations, improving their adaptability and performance in real-world applications.

Environment Parameter Randomization complements curriculum learning by focusing not just on task complexity but also on variability, ensuring agents can generalize across diverse scenarios.

Model Types

The ML-Agents Toolkit supports training various model types depending on the type of agent observations, such as vector, ray cast, and visual observations. These flexible observation definitions allow agents to learn from diverse inputs, making the platform versatile for different scenarios.

Vector Observations

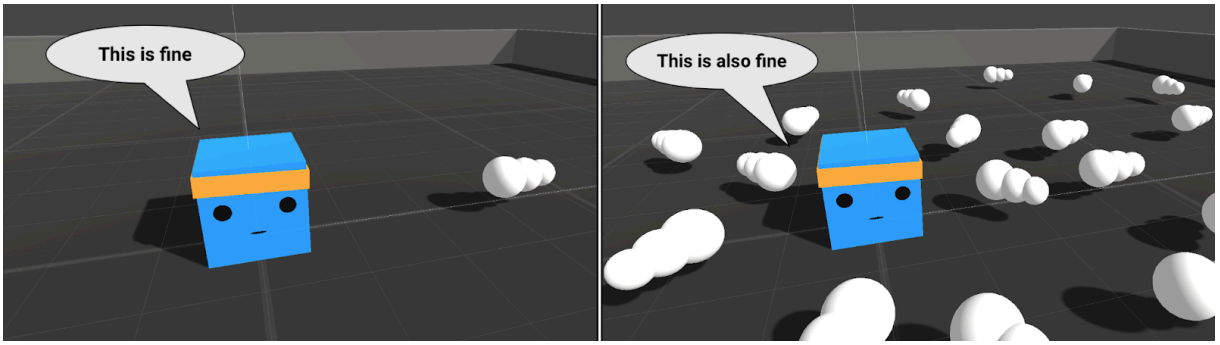
For ray cast or vector-based observations, the ML-Agents Toolkit uses fully connected neural networks. Users can configure the network's hidden layers and units during training, tailoring it to their needs.

CNN

Agents can utilize multiple cameras for observations, enabling the integration of multiple visual streams. For example, this can be applied to self-driving cars using different camera angles or navigational agents combining aerial and first-person views. ML-Agents provides three CNN architectures for processing visual observations:

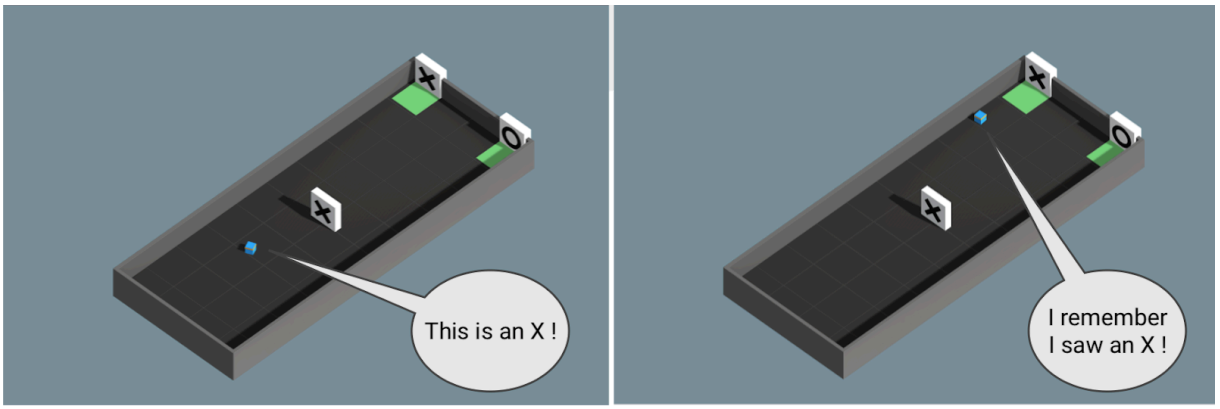
- A simple encoder with two convolutional layers.
- A three-layer architecture proposed by Mnih et al.
- The IMPALA ResNet, featuring three stacked layers with residual blocks for more complex visual inputs.

Attention



Agents can process observations of varying lengths, such as tracking different numbers of entities at each step. For instance, an agent learning to avoid projectiles can handle changing numbers of projectiles throughout an episode. Using attention networks, agents focus on relevant entities while ignoring irrelevant ones, adapting their behavior to dynamic contexts.

RNN



For tasks requiring memory, where agents must remember past observations to make decisions, ML-Agents leverages LSTMs (Long Short-Term Memory networks). These networks allow agents to track and retain important information over time, especially in environments with partial observability. This ensures agents can adapt to complex scenarios without manually defining what to remember.

The flexibility of these models allows ML-Agents users to design agents suited to diverse tasks and challenges, from simple vector-based inputs to multi-camera observations and dynamic, memory-intensive environments.

Additional Features

The ML-Agents Toolkit includes several additional features to enhance the training process by improving flexibility and interpretability:

- **Concurrent Unity Instances**: Developers can run multiple Unity executables in parallel during training, significantly speeding up the process for certain scenarios. Detailed instructions on enabling this feature are available here.
- **Recording Statistics from Unity**: The toolkit allows developers to record and aggregate statistics from Unity environments during training, providing insights into the training process.
- **Custom Side Channels**: Developers can create custom side channels to manage data transfer between Unity and Python, tailoring workflows to specific training needs or environments.
- **Custom Samplers**: Custom sampling methods can be defined for Environment Parameter Randomization, allowing users to adapt this feature to suit their unique training environments.

Training ML-Agents

Once your learning environment has been created and is ready for training, the next step is to initiate a training run. Training in the ML-Agents Toolkit is powered by a dedicated Python package, **mlagents**. This package exposes a command "*mlagents-learn*" that is the single entry point for all training workflows (e.g. reinforcement leaning, imitation learning, curriculum learning). Its implementation can be found at ml-agents/mlagents/trainers/learn.py.

Starting Training

The command "*mlagents-learn*" is the main training utility in the ML-Agents Toolkit. It uses a YAML configuration file to define hyperparameters and settings, which are critical for agent performance. To view a description of all the CLI options accepted by mlagents-learn, use:

- *mlagents-learn --help*

The basic command for training is:

- *mlagents-learn <trainer-config-file> --env=<env_name> --run-id=<run-identifier>*
  - **<trainer-config-file>:** Path to the YAML file with training settings and hyperparameters.
  - **<env_name> (optional):** Path to the Unity executable for training. If omitted, training happens in the Unity Editor.
  - **<run-identifier>:** Unique name to identify the training results.

Observing Training

During training, three key artifacts are generated in the "*results/<run-identifier>*" folder. These artifacts are updated in real time and finalized when training completes or is interrupted:

- **Summaries:** Training metrics that help monitor progress, viewable with tools like TensorBoard.
- **Models:** Includes periodic checkpoints and the final model file (.onnx) after training completes or is interrupted.
- **Timers file:** Aggregated metrics on training performance, useful for profiling.

Stopping and Resuming Training

To interrupt training and save the current progress, hit *Ctrl+C* once and wait for the model(s) to be saved out.

To resume a previously interrupted or completed training run, use the *--resume* flag and make sure to specify the previously used run ID.

If you would like to re-run a previously interrupted or completed training run and re-use the same run ID (in this case, overwriting the previously generated artifacts), then use the *--force* flag.

Loading an Existing Model

Existing models can be used for inference in Python by combining the --resume and --inference flags. Note that if you want to run inference in Unity, you should use the Sentis. If changes are made to the network architecture (e.g., adding a new reward signal or modifying hidden units), parts of the existing model can still be loaded, with new components initialized from scratch.

For initializing new training runs with an existing model (pre-trained), use the *--initialize-from=<run-identifier>* flag. This is useful for adapting models to updated environments while retaining previously learned behaviors.

Training Configurations

The Unity ML-Agents Toolkit supports diverse training scenarios and methods, requiring tailored configurations for each use case. Configurations are managed through YAML files and **mlagents-learn** command-line flags. These settings determine training hyperparameters, environment parameters (e.g., **Curriculum Learning**, **Environment Parameter Randomization**), and agent behaviors.

A **trainer configuration file** (<trainer-config-file>) specifies hyperparameters for each Behavior in the scene. Sample configuration files for environments like 3D Balance Ball are available in the config/ directory, including setups for **PPO**, **SAC**, and **GAIL** training methods. When creating a configuration file, consider the functionalities you plan to use:

- Choosing between **PPO** and **SAC** trainers.
- Adding memory to agents using **Recurrent Neural Networks (RNNs)**.
- Utilizing the **Intrinsic Curiosity Module** for exploration.
- Ignoring environment rewards entirely.
- Pre-training with **Behavioral Cloning** or adding **GAIL intrinsic rewards** (requires demonstrations).
- Enabling **self-play** for multi-agent scenarios.

These choices guide the file's structure and settings. For a full list of training configurations, refer to the **Training Configuration File** documentation.

Adding CLI Arguments to the Training Configuration file

In addition to YAML configurations, CLI arguments (e.g., --num-envs) can be included directly in the file for convenience. CLI arguments are grouped into **environment**, **engine**, **checkpoint**, and **torch** settings:

Environment settings

Configure simulation behavior, such as environment paths, parallel environments, and timeouts.

```
env_settings:
  env_path: FoodCollector
  env_args: null
  base_port: 5005
  num_envs: 1
  timeout_wait: 10
  seed: -1
  max_lifetime_restarts: 10
  restarts_rate_limit_n: 1
  restarts_rate_limit_period_s: 60
```

Engine settings

Define Unity environment rendering and performance parameters.

```
engine_settings:
  width: 84
  height: 84
  quality_level: 5
  time_scale: 20
  target_frame_rate: -1
  capture_frame_rate: 60
  no_graphics: false
```

Checkpoint settings

Manage run IDs, model loading, and training states.

```
checkpoint_settings:
  run_id: foodtorch
  initialize_from: null
  load_model: false
  resume: false
  force: true
  train_model: false
  inference: false
```

Torch settings

Specify hardware preferences for training.

```
torch_settings:
  device: cpu
```

Behavior Configurations

The trainer config file defines settings for each Behavior in your Unity scene under the behaviors section. Each Behavior's configuration includes hyperparameters, network settings, reward signals, and optional functionalities like memory, curiosity, behavioral cloning, GAIL, or self-play. Detailed parameter descriptions and their defaults are available in the Training Configuration File guide from Unity.

Below is a summary of the main components and their roles, as well as an example file with all of the configurations included:

- **Trainer Type**: Specifies the algorithm, such as **PPO** or **SAC**.
- **Hyperparameters**: Includes algorithm-specific settings like batch size, learning rate, and PPO or SAC-specific parameters.
- **Network Settings**: Configures the neural network, including hidden units, layers, and memory parameters (e.g., sequence length).
- **Trainer Configurations**: General settings like max_steps, time_horizon, and checkpoint frequency.
- **Behavioral Cloning**: Enables pre-training with demonstration data, useful for accelerating training in sparse-reward environments.
- **Reward Signals**: Adds intrinsic or extrinsic rewards, including:
    - **Extrinsic rewards** (default environment rewards).
    - **Curiosity** for exploration.
    - **GAIL** for imitation learning from demonstrations.
- **Self-Play**: Configures parameters for multi-agent training in adversarial settings.

Example File:

```yaml
behaviors:
  BehaviorPPO:
    trainer_type: ppo

    hyperparameters:
      # Hyperparameters common to PPO and SAC
      batch_size: 1024
      buffer_size: 10240
      learning_rate: 3.0e-4
      learning_rate_schedule: linear

      # PPO-specific hyperparameters
      beta: 5.0e-3
      beta_schedule: constant
      epsilon: 0.2
      epsilon_schedule: linear
      lambd: 0.95
      num_epoch: 3
      shared_critic: False

    # Configuration of the neural network (common to PPO/SAC)
    network_settings:
      vis_encode_type: simple
      normalize: false
      hidden_units: 128
      num_layers: 2
      # memory
      memory:
        sequence_length: 64
        memory_size: 256

    # Trainer configurations common to all trainers
    max_steps: 5.0e5
    time_horizon: 64
    summary_freq: 10000
    keep_checkpoints: 5
    checkpoint_interval: 50000
    threaded: false
    init_path: null

    # behavior cloning
    behavioral_cloning:
      demo_path: Project/Assets/ML-Agents/Examples/Pyramids/Demos/ExpertPyramid.demo
      strength: 0.5
      steps: 150000
      batch_size: 512
      num_epoch: 3
```

```
      samples_per_update: 0

    reward_signals:
      # environment reward (default)
      extrinsic:
        strength: 1.0
        gamma: 0.99

      # curiosity module
      curiosity:
        strength: 0.02
        gamma: 0.99
        encoding_size: 256
        learning_rate: 3.0e-4

      # GAIL
      gail:
        strength: 0.01
        gamma: 0.99
        encoding_size: 128
        demo_path:
Project/Assets/ML-Agents/Examples/Pyramids/Demos/ExpertPyramid.demo
        learning_rate: 3.0e-4
        use_actions: false
        use_vail: false

    # self-play
    self_play:
      window: 10
      play_against_latest_model_ratio: 0.5
      save_steps: 50000
      swap_steps: 2000
      team_change: 100000
```

Default Behavior Settings

For environments with multiple Behaviors or procedurally generated Behavior names, a default_settings section can simplify configuration. Behaviors not explicitly defined in the file will use the default_settings, and unspecified parameters within individual Behaviors will default to those in default_settings. To specify a default configuration, insert a default_settings section in your YAML. This section should be formatted exactly like a configuration for a Behavior:

```
default_settings:
  # < Same as Behavior configuration >
behaviors:
  # < Same as above >
```

Environment Parameters

Environment Parameters allow control over various aspects of the Unity simulation during training. These parameters can be set in the environment_parameters section of the YAML configuration file. For example, to set a parameter called my_environment_parameter to 3.0:

```
behaviors:
  BehaviorY:
    # < Same as above >

# Add this section
environment_parameters:
  my_environment_parameter: 3.0
```

Within Unity, these parameters can be accessed using:

```
Academy.Instance.EnvironmentParameters.GetWithDefault("my_environment_parameter",
0.0f);
```

Environment Parameter Randomization

To enhance agent robustness, environment parameter randomization introduces variability during training. Instead of setting a fixed value for an environment parameter, you can specify a sampler that determines its range or distribution. Here is an example with three environment parameters called *mass*, *length* and *scale*:

```
behaviors:
  BehaviorY:
    # < Same as above >

# Add this section
environment_parameters:
  mass:
    sampler_type: uniform
    sampler_parameters:
        min_value: 0.5
        max_value: 10

  length:
    sampler_type: multirangeuniform
    sampler_parameters:
        intervals: [[7, 10], [15, 20]]

  scale:
    sampler_type: gaussian
    sampler_parameters:
        mean: 2
        st_dev: .3
```

Supported Sampler Types

Each parameter uses a sampler type to define how values are generated:

- **uniform:** Samples a value uniformly between min_value and max_value.
- **gaussian:** Samples a value from a normal distribution with mean and st_dev.
- **multirange_uniform:** Chooses an interval from a list of ranges and samples uniformly within the selected interval.

The implementation of the samplers can be found in the Samplers.cs file.

Training with Environment Parameter Randomization

Once randomization is defined, run training as usual with the updated YAML file. For instance, to train the 3D Ball agent with parameter randomization:

```
mlagents-learn config/ppo/3DBall_randomize.yaml --run-id=3D-Ball-randomize
```

Progress and metrics can be monitored in TensorBoard, allowing you to observe the impact of randomization on training outcomes. By incorporating randomization, agents can better generalize to diverse environments.

Curriculum

Curriculum learning allows agents to progress through increasingly complex tasks during training. This is configured in the *environment_parameters* section of the YAML file by adding a *curriculum* subsection. Each curriculum is defined as a series of **lessons**, each with:

- **name**: A user-defined identifier displayed when the lesson changes.
- **completion_criteria**: Defines when the lesson is complete, including the measure of progress, threshold values, and other conditions.
- **value**: The environment parameter value during the lesson, which can be a static float or a sampler.

Example configuration:

```yaml
behaviors:
  BehaviorY:
    # < Same as above >

# Add this section
environment_parameters:
  my_environment_parameter:
    curriculum:
      - name: MyFirstLesson # The '-' is important as this is a list
        completion_criteria:
          measure: progress
          behavior: my_behavior
          signal_smoothing: true
          min_lesson_length: 100
          threshold: 0.2
        value: 0.0
      - name: MySecondLesson # This is the start of the second lesson
        completion_criteria:
          measure: progress
          behavior: my_behavior
          signal_smoothing: true
          min_lesson_length: 100
          threshold: 0.6
          require_reset: true
        value:
          sampler_type: uniform
          sampler_parameters:
            min_value: 4.0
            max_value: 7.0
      - name: MyLastLesson
        value: 8.0
```

Completion Criteria:

- **measure**: Tracks progress using *reward*, *progress* (ratio of steps/max_steps), or *ELO* (for self-play).
- **behavior**: Tracks progress for a specific behavior.
- **threshold**: The value of the measure at which the lesson advances.
- **min_lesson_length**: Minimum episodes required before the lesson can change.
- **signal_smoothing**: Smooths progress by considering prior values.
- **require_reset**: Specifies if the environment must reset when advancing lessons (default: *false*).

Training with a Curriculum

Run mlagents-learn with the YAML file containing curriculum settings. For example:

```
mlagents-learn config/ppo/WallJump_curriculum.yaml --run-id=wall-jump-curriculum
```

Progress can be monitored in TensorBoard, and interrupted training can be resumed using *--resume*, retaining lesson progress.

Training Using Concurrent Unity Instances

To speed up training, you can run multiple Unity instances simultaneously using the *--num-envs=<n>* argument in *mlagents-learn*. Optionally, set *--base-port* to configure the starting port for these instances.

Considerations:

- **Buffer Size**: Increase *buffer_size* in the YAML file proportionally to *num-envs* to stabilize training.
- **Resource Constraints**: Ensure your machine has enough resources to support the chosen number of instances.
- **Result Variation**: Changing *--num-envs* while keeping other hyperparameters constant may lead to variations in results and models.

Example command:

```
mlagents-learn config/ppo/3DBall.yaml --run-id=3D-Ball --num-envs=4
```

Concurrent Unity instances and curriculum learning can be combined for faster and more efficient training processes.

Learning Environment

A **Learning Environment** is a simulation scene designed for training autonomous agents. It consists of a Unity scene orchestrated by the **Academy**, which manages episode resets, controls environment parameters, and coordinates agents' decision-making processes.

Agents in the environment collect observations, perform actions, and receive rewards, allowing them to learn through trial and error. Features like **curriculum learning**, **environment parameter randomization**, and multiple training areas enhance the environment's adaptability and efficiency, supporting diverse and robust agent training scenarios.

Designing an effective learning environment involves setting up your Unity scene and simulation to enable agent training. This differs from designing the agents themselves, which involves defining observations, actions, rewards, and teams. To learn how to make a Learning Environment from scratch, check out the Making a New Learning Environment page from Unity.

The Simulation and Training Process

Training and simulation are managed by the **ML-Agents Academy** class, which coordinates the agent simulation loop in collaboration with the Python training process. During training, the Python process interacts with the Academy to run episodes, gather data, and optimize neural networks. Once training is complete, the resulting model can be added to the Unity project for use. The simulation loop follows these steps:

1. **Environment Reset**: Calls *OnEnvironmentReset()*.
2. **Episode Start**: Calls *OnEpisodeBegin()* for all Agents.
3. **Observation Collection**: Uses *CollectObservations(VectorSensor sensor)* and sensors to gather environment data.
4. **Policy Decision**: Uses each Agent's policy to select an action.
5. **Action Execution**: Executes actions via *OnActionReceived()*.
6. **Episode End**: Calls *OnEpisodeBegin()* for Agents that reach *Max Step* or ended their episode with *EndEpisode()*.

To create a training environment, extend the **Agent** class and implement these methods as needed for your scenario.

Organizing the Unity Scene

To train and use the ML-Agents Toolkit in a Unity scene, add to the scene as many Agent subclasses as you need. Agent instances should be attached to the GameObject representing that Agent.

Academy

The Academy is a singleton which orchestrates Agents and their decision making processes. Only a single Academy exists at a time.

Academy resetting

To modify the environment at the start of each episode, use the Academy's OnEnvironmentReset action:

```
public class MySceneBehavior : MonoBehaviour
{
    public void Awake()
    {
        Academy.Instance.OnEnvironmentReset += EnvironmentReset;
    }

    void EnvironmentReset()
    {
        // Reset the scene here
    }
}
```

Use this to reset Agents, reposition goals, or randomize elements like maze configurations to promote generalization and avoid overfitting to a single scenario.

Multiple Areas

Using multiple copies of the training area in a single scene can accelerate training by gathering experiences in parallel. This can be achieved by instantiating multiple Agents with the same Behavior Name.

Environments

To create a training environment in Unity, ensure the scene is properly configured to work with the external training process. Key considerations include:

- **Automatic Start**: The training scene should start immediately when launched by the training process.

- **Episode Reset**: Use the **Academy** to reset the scene to a valid starting point at the beginning of each episode.
- **Definite End**: Define episode termination either by setting a **Max Steps** value or by calling *EndEpisode()* when appropriate.

Environment Parameters

For features like curriculum learning and environment parameter randomization, use the EnvironmentParameters C# class to retrieve parameter values defined in the training configuration. Update environment parameters at every step to reflect the correct values. It is recommended to modify the environment in the Agent's *OnEpisodeBegin()* function using *Academy.Instance.EnvironmentParameters*. See the WallJump example environment for a sample usage (specifically, WallJumpAgent.cs).

Agent

The **Agent** class represents an actor in the scene responsible for observations and actions. Typically, it is attached to a GameObject representing the actor (e.g., a football player or vehicle). The Behavior Parameters assigned to an Agent should align with its implementation. To create a custom Agent:

- **Extend the Agent Class**: Implement the following key methods:
  - *CollectObservations(VectorSensor sensor)*: Gather observations from the environment.
  - *OnActionReceived()*: Execute actions based on the Agent's policy and assign rewards.
- **Define Episode Termination**:
  - Use *EndEpisode()* in *OnActionReceived()* when the Agent completes or fails its task.
  - Set **Max Steps** to automatically terminate episodes after a fixed number of steps.
- **Restart Episodes**: Use the *OnEpisodeBegin()* function to reset the Agent for a new episode.

Recording Statistics

Use the *StatsRecorder* C# class to record and aggregate training statistics directly within Unity. These statistics help track the training process and can be visualized later. See the FoodCollector example environment for a sample usage (specifically, FoodCollectorSettings.cs).

Agents

An **Agent** in Unity ML-Agents is an entity that observes its environment, decides on actions, and executes those actions. The primary goal of an Agent is to learn the optimal decision-making policy for completing its tasks, guided by observations and rewards. Agents are created by extending the **Agent** class, which provides key methods for implementing behavior, decision-making, and reward logic.

An Agent's **Policy** handles decision-making based on its **Behavior Parameters**. The Policy can rely on a neural network model or manual logic via the **Heuristic()** method. The choice depends on whether the Behavior Type is set to **Heuristic Only** or a trained Model is provided. The Policy abstraction allows the same Policy to be reused across multiple Agents.

Core Agent Methods:

1. **OnEpisodeBegin()**: Called at the start of each episode, resetting the Agent and environment. Randomization during resets helps generalize training to varied scenarios.
2. **CollectObservations(VectorSensor sensor)**: Collects environmental data for decision-making. Observations are added to the *VectorSensor*, which defines the Agent's observation space.
3. **OnActionReceived(float[] action)**: Executes the chosen action and assigns rewards. Actions affect the Agent's behavior, and the method also ends episodes if conditions are met.
4. **Heuristic()**: Generates actions manually or based on predefined logic when the Behavior Type is set to **Heuristic Only**. Avoid creating new arrays within this method, as it writes to the provided action array.

Example: In the **Ball3DAgent**, *OnEpisodeBegin()* resets positions, *CollectObservations()* tracks orientation and relative positions, and *OnActionReceived()* adjusts cube rotation, providing rewards for balancing the ball and penalizing for dropping it.

Decisions

The observation-decision-action-reward cycle occurs each time the Agent requests a decision using *Agent.RequestDecision()*. To automate this process at regular intervals, add a **Decision Requester** component to the Agent's GameObject. This is ideal for physics-based simulations requiring frequent control adjustments or games with synchronized decision-making.

- For **regular interval decisions**, set the **DecisionStep** parameter in the Decision Requester.

- For **event-driven decisions**, manually call *Agent.RequestDecision()* when specific game events occur, such as in a turn-based game.

Observations and Sensors

Observations are critical for enabling agents to learn effectively in their environments. They should include all relevant information necessary for completing tasks. Poorly chosen or incomplete observations can hinder or prevent learning. A good starting point is to consider the data a human or algorithm would need to solve the problem analytically.

Generating Observations

ML-Agents provides three primary methods for generating observations:

1. **Agent.CollectObservations()**: A customizable approach for adding numerical and non-visual data.
2. **Observable Fields and Properties**: Automatically expose fields or properties using the [Observable] attribute.
3. **ISensor Interface and SensorComponents**: Advanced techniques for implementing custom sensors or leveraging prebuilt components.

For most scenarios, CollectObservations() should be used for numerical data, SensorComponents for specialized observation needs, and [Observable] for simplicity. Advanced users can leverage ISensor for custom implementations.

Agent.CollectObservations():

This method allows you to explicitly collect observations and add them to the VectorSensor. Observations must remain consistent in size and order for effective learning. For example, the 3DBall example uses the rotation of the platform, the relative position of the ball, and the velocity of the ball as its state observation:

```
public GameObject ball;

public override void CollectObservations(VectorSensor sensor)
{
    // Orientation of the cube (2 floats)
    sensor.AddObservation(gameObject.transform.rotation.z);
    sensor.AddObservation(gameObject.transform.rotation.x);
    // Relative position of the ball to the cube (3 floats)
    sensor.AddObservation(ball.transform.position - gameObject.transform.position);
    // Velocity of the ball (3 floats)
    sensor.AddObservation(m_BallRb.velocity);
    // 8 floats total
}
```

As an experiment, you can remove the velocity components from the observation and retrain the *3DBall agent*. While it will learn to balance the ball reasonably well, the performance of the agent without using velocity is noticeably worse.

Additionally, when you set up an Agent's *Behavior Parameters* in the Unity Editor, you must adjust the **Vector Observations > Space Size** in **Behavior Parameters** to match the number of observations. If the number of entities varies, use padding or limit observations to a subset (e.g., closest five enemies).

Observable Fields and Properties:

Adding the *[Observable]* attribute to fields or properties automates observation generation. For example, in the *Ball3DHardAgent*, the difference between positions could be observed by adding a property to the Agent:

```
using Unity.MLAgents.Sensors.Reflection;

public class Ball3DHardAgent : Agent {

    [Observable(numStackedObservations: 9)]
    Vector3 PositionDelta
    {
        get
        {
            return ball.transform.position - gameObject.transform.position;
        }
    }
}
```

Control the behavior of *[Observable]* attributes via **Behavior Parameters**:

- **Ignore**: Fastest initialization; ignores all attributes.
- **Exclude Inherited**: Examines only declared class members (recommended for most cases).
- **Examine All**: Includes inherited members, useful for subclasses.

No adjustment to **Space Size** is required when using *[Observable]*.

ISensor interface and SensorComponents:

For advanced use cases, the **ISensor** interface offers full control over observation generation. The **SensorComponent** class creates ISensor instances and attaches them to GameObjects. Prebuilt SensorComponents include:

- **CameraSensorComponent**: Uses camera images as observations.
- **RenderTextureSensorComponent**: Observes RenderTexture contents.
- **RayPerceptionSensorComponent**: Provides data from ray casts.
- **GridSensorComponent**: Observes a grid of box queries.

- **Match3SensorComponent**: Encodes Match-3 game boards.

No **Space Size** adjustment is necessary for SensorComponents.

Vector Observations

Vector observations represent data as lists of floats and are critical for enabling agents to make decisions. These observations are produced by methods like **Agent.CollectObservations()**, **ObservableAttributes**, and **ISensors**. Below are best practices and techniques for effective vector observation design.

One-hot encoding categorical information:

Categorical data should use **one-hot encoding**, where each category is represented as a unique binary vector. For example:

- For categories [Sword, Shield, Bow], if the observed item is Bow, the vector would be [0, 0, 1].
- Use *VectorSensor.AddOneHotObservation()* for simplicity or annotate enum fields with *[Observable]* to handle this automatically.

```
enum ItemType { Sword, Shield, Bow }

public class HeroAgent : Agent
{
    [Observable]
    ItemType m_CurrentItem;
}
```

Normalization:

To improve training efficiency, normalize observation values to ranges like [0, 1] or [-1, 1].

- **Normalization Formula**:

$$normalizedValue = \frac{currentValue - minValue}{maxValue - minValue}$$

- Apply normalization to each component of vectors like Vector3 (e.g., positions or rotations). For angles, normalize based on the expected range (e.g., divide by 360 for [0, 1]).

```
normalizedValue = (currentValue - minValue)/(maxValue - minValue)
Quaternion rotation = transform.rotation;
Vector3 normalized = rotation.eulerAngles / 180.0f - Vector3.one;  // [-1,1]
Vector3 normalized = rotation.eulerAngles / 360.0f;  // [0,1]
```

Stacking:

Stacking refers to repeating observations from previous steps as part of a larger observation. For example, consider an Agent that generates these observations in four steps:

```
step 1: [0.1]
step 2: [0.2]
step 3: [0.3]
step 4: [0.4]
```

If we use a stack size of 3, the observations would instead be:

```
step 1: [0.1, 0.0, 0.0]
step 2: [0.2, 0.1, 0.0]
step 3: [0.3, 0.2, 0.1]
step 4: [0.4, 0.3, 0.2]
```

The observations are padded with zeroes for the first *stackSize-1* steps. This is a simple way to give an Agent limited "memory" without the complexity of adding a recurrent neural network (RNN). Enable stacking by setting Stacked Vectors in the Behavior Parameters or using *[Observable(numStackedObservations: N)]* for ObservableAttributes. ISensors can be wrapped in a *StackingSensor* for similar functionality.

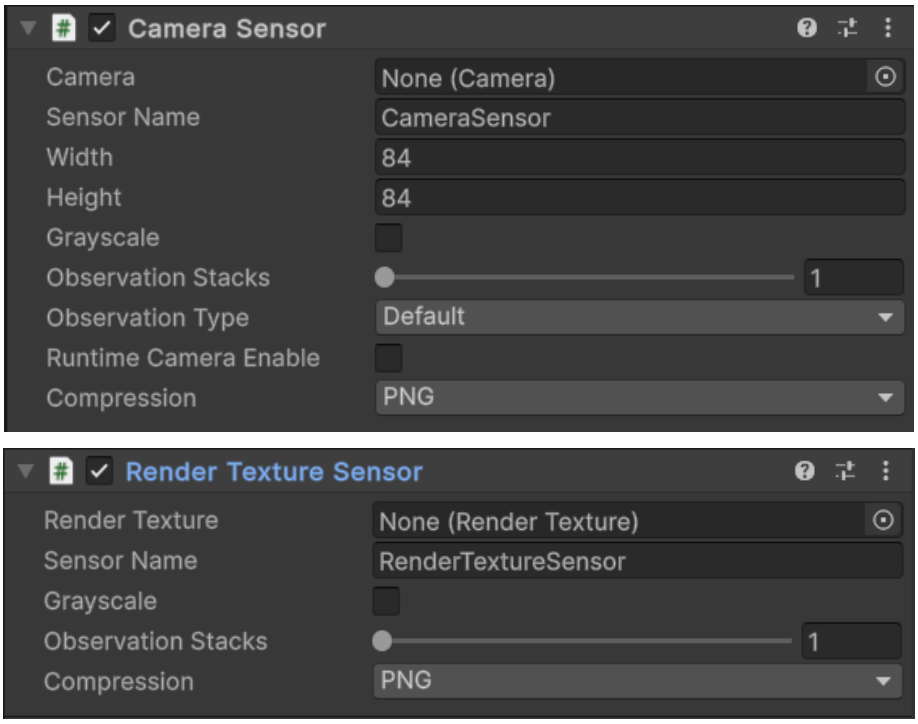
Vector Observation Summary & Best Practices:

- **Include Relevant Data**: Vector observations should capture all necessary information for decision-making without extraneous noise.
- **Relative Positions**: Encode positional data relative to the agent to simplify learning. For example, record a target's position as *target.position - agent.position*.
- **One-hot Encoding:** Categorical variables such as type of object (Sword, Shield, Bow) should be encoded in one-hot fashion (i.e. 3 -> 0, 0, 1). This can be done automatically using the AddOneHotObservation() method of the VectorSensor, or using [Observable] on an enum field or property of the Agent.
- **Use Stacking or RNNs**: For temporal dependencies, use stacked observations or incorporate Recurrent Neural Networks (RNNs).
- **Normalize Consistently**: Ensure all inputs are scaled to a uniform range for faster and more stable learning.

Visual Observations

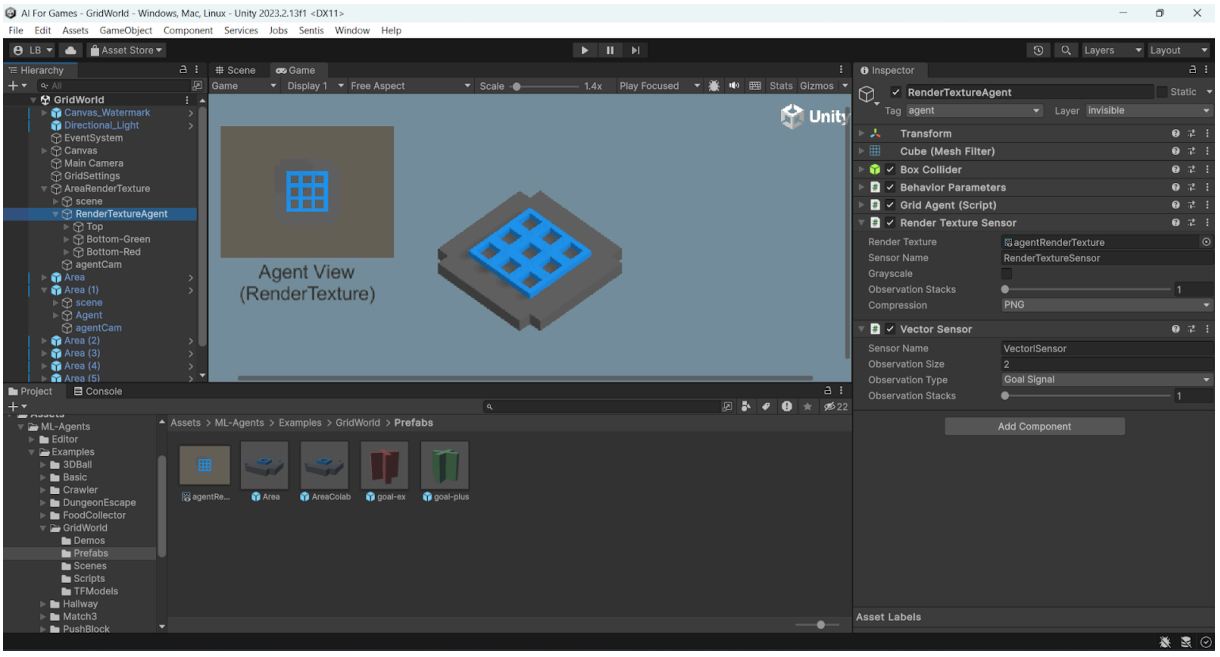
Visual observations allow agents to learn from spatial patterns within images by transforming inputs from Cameras or RenderTextures into 3D tensors, which are processed by Convolutional Neural Networks (CNNs). These observations are useful for capturing complex states that are hard to represent numerically, but they are often less efficient and slower to train compared to vector observations. Use them only when vector or ray-cast observations cannot adequately define the problem.

Using Visual Observations:



1. **Adding Visual Sensors**:
   - Attach **CameraSensor** or **RenderTextureSensor** components to the Agent.
   - Assign the desired camera or render texture to the component.
   - Configure the **image resolution** (width and height) and select **color** or **grayscale** as appropriate.
2. **Handling Multiple Sensors**:
   - Agents sharing a **Policy** must have the same number of visual observations, with identical resolutions and color settings.
   - Assign **unique names** to each Sensor Component on an Agent for deterministic sorting.
3. **Stacking Observations**:
   - Enable stacking by setting **Observation Stacks** greater than 1. This stacks observations from the last few steps along the channel dimension, providing temporal context.

Debugging RenderTexture Observations:

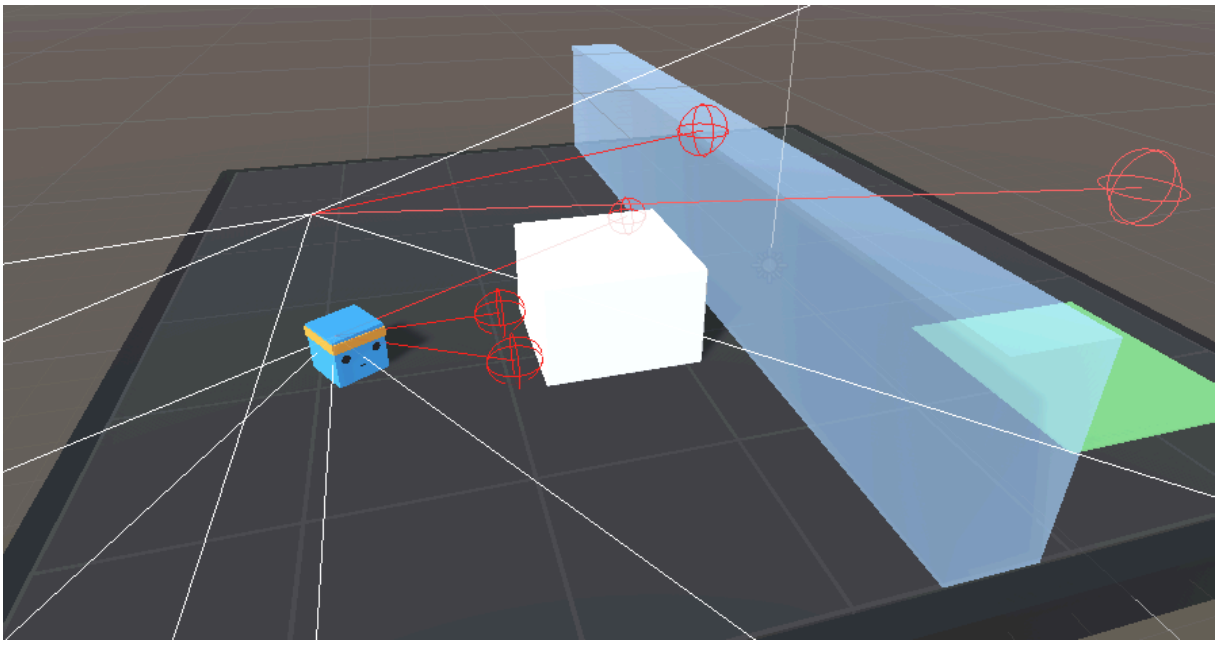


- Add a **Canvas** to the scene, and use a **Raw Image** component with its texture set to the Agent's RenderTexture. This allows you to view the Agent's perspective directly on the game screen.
- Update the RenderTexture by manually rendering the Camera during each decision step. For Cameras used directly, this happens automatically.

Visual Observation Summary & Best Practices:

- **Optimize Image Size**: Keep image resolutions as small as possible without losing crucial details for decision-making.
- **Use Grayscale**: Prefer grayscale images if color information is unnecessary, as it reduces computational load.
- **Minimize Use**: Only use visual observations when vector observations cannot sufficiently describe the environment.

Raycast Observations



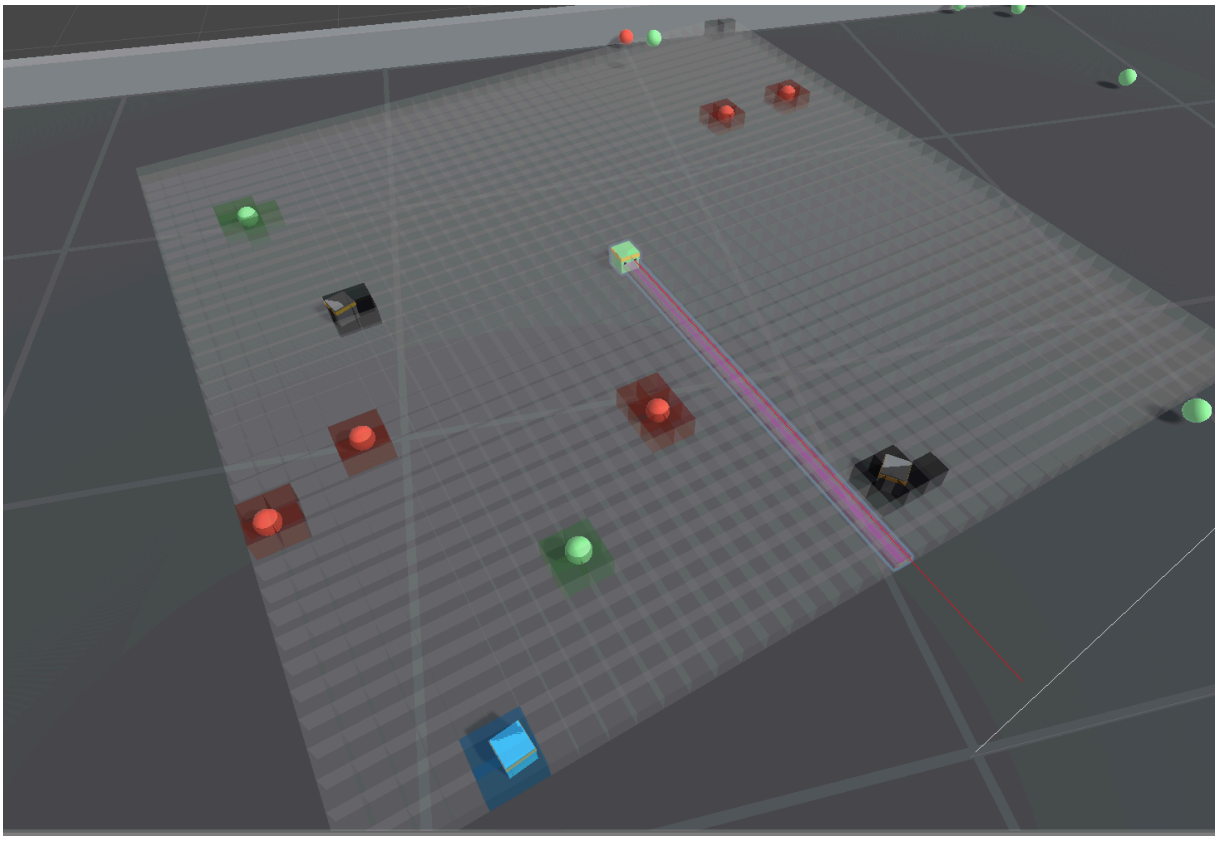
Raycast observations are an efficient way to provide spatial information to an agent without requiring full visual input. By using **RayPerceptionSensorComponent3D** or **RayPerceptionSensorComponent2D**, agents can detect objects and encode them into observation vectors. For an overview of the sensor components settings, check out this page. This method is ideal for detecting spatial relationships like proximity or obstacles without the overhead of visual observations.

To implement raycast observations, attach a RayPerceptionSensorComponent to the agent and configure settings such as detectable tags, number of rays, angles, and ray length. Use batched raycasts for better performance in 3D environments.

RayCast Observation Summary & Best Practices:

- Attach *RayPerceptionSensorComponent3D* or *RayPerceptionSensorComponent2D* to use.
- This observation type is best used when there is relevant spatial information for the agent that doesn't require a fully rendered image to convey.
- Use as few rays and tags as necessary to solve the problem in order to improve learning stability and agent performance.
- If you run into performance issues, try using batched raycasts by enabling the Use Batched Raycast setting. (Only available for 3D ray perception sensors.)

Grid Observations



Grid observations combine the spatial representation of visual observations with the object-detection flexibility of raycasts. The **GridSensorComponent** provides a top-down 2D view around the agent, encoding the presence of objects in each grid cell using one-hot representation. For an overview of the sensor components settings, check out this page.

To implement grid observations, attach a GridSensorComponent to the agent and configure grid size, cell scale, detectable tags, and whether the grid rotates with the agent. The resulting 3D tensor is fed into the agent's CNN policy.

Grid Observation Summary & Best Practices:

- Attach *GridSensorComponent* to use.
- This observation type is best used when there is relevant non-visual spatial information that can be best captured in 2D representations.
- Use as small grid size and as few tags as necessary to solve the problem in order to improve learning stability and agent performance.
- Do not use GridSensor in a 2D game.

Variable Length Observations

Variable length observations allow agents to collect information from a dynamic number of entities using a **BufferSensorComponent**. This is ideal for scenarios where the number of entities (e.g., enemies or projectiles) varies across episodes. The BufferSensor uses attention mechanisms to process the observations, enabling comparative reasoning between entities. However, training with variable length observations can be slower than using fixed vector observations.

To use the BufferSensor, define:

- **Observation Size**: Fixed size (in floats) to represent each entity's observation.
- **Maximum Number of Entities**: The maximum number of entities to observe. Observations are padded with zeros if fewer entities are present.

Add entities' observations to the BufferSensor by calling *BufferSensorComponent.AppendObservation()* within the *Agent.CollectObservations()* method. Normalize the observations between -1 and 1 for better training performance.

Variable Length Observation Summary & Best Practices:

- Attach *BufferSensorComponent* to use.
- Call *BufferSensorComponent.AppendObservation()* in the *Agent.CollectObservations()* method to add the observations of an entity to the *BufferSensor*.
- Normalize the entities observations before feeding them into the *BufferSensor*.

Goal Signal

A **goal signal** is a specialized observation that conditions the agent's policy. This allows agents to adapt their behavior dynamically based on changing goals, enabling them to generalize across similar tasks. For instance, an agent trained with goal signals can reuse learnings from one task to solve variations more effectively.

To implement goal signals:

- Attach a **VectorSensorComponent** or **CameraSensorComponent** to the agent and set the **Observation Type** to "Goal".
- In the training configuration, set the conditioning_type parameter:
  - **hyper**: Uses a HyperNetwork to condition the policy based on the goal. This is computationally intensive, so reduce the number of hidden units in the network for efficiency.
  - **none**: Treats the goal as a regular observation.

Goal signals are particularly useful in multi-task settings where shared learnings improve generalization. For practical examples, refer to the GridWorld environment.

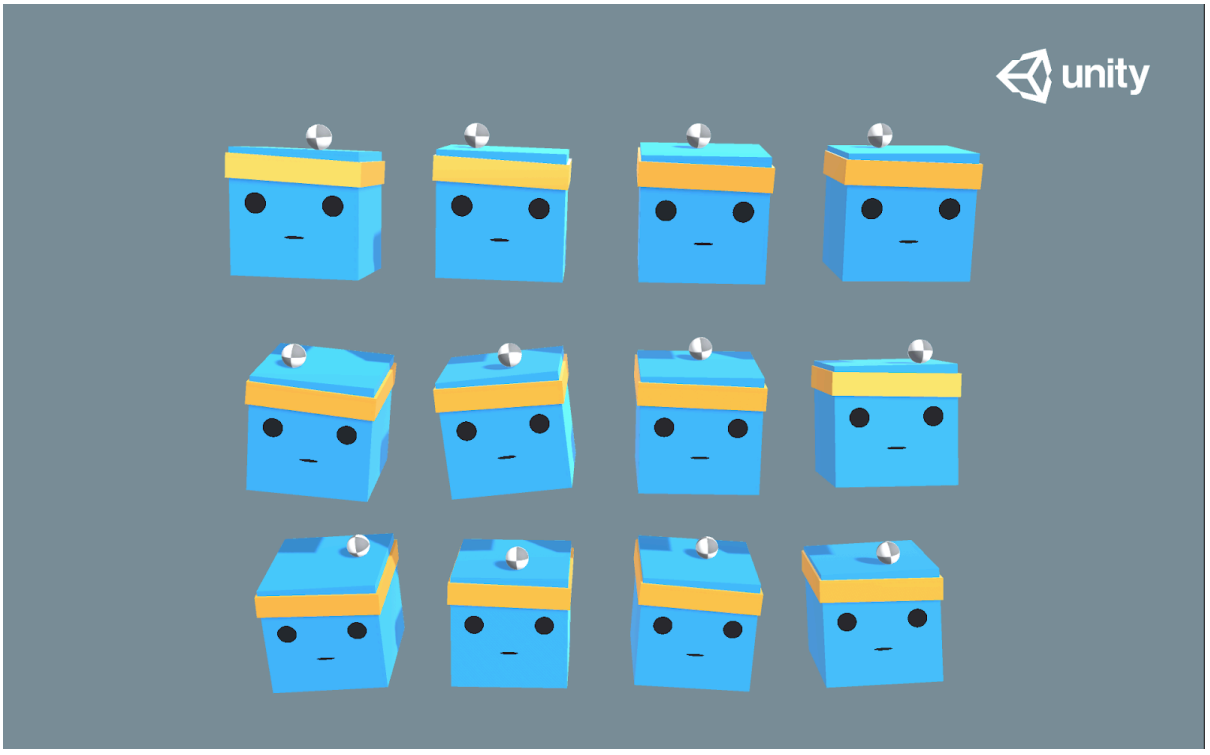
Goal Signal Summary & Best Practices:

- Attach a *VectorSensorComponent* or *CameraSensorComponent* to an agent and set the observation type to goal to use the feature.
- Set the *conditioning_type* parameter in the training configuration.
- Reduce the number of hidden units in the network when using the HyperNetwork conditioning type.

Actions and Actuators

Actions are instructions generated by an Agent's Policy and executed through the *OnActionReceived()* method. These actions can be **Continuous** (floating-point values) or **Discrete** (categorical values). The Policy itself does not understand the meaning of actions—it simply learns to maximize rewards through trial and error. Actions are defined in the *OnActionReceived()* function of the Agent or an IActuator.

Continuous Actions



Continuous actions are arrays of floats, each representing a control variable like speed or direction. These values are typically clamped between -1 and 1 and can be scaled as needed. For instance, in the **3DBall** example, two continuous actions are used to control the platform's rotation:

```
public override void OnActionReceived(ActionBuffers actionBuffers)
{
    var actionZ = 2f * Mathf.Clamp(actionBuffers.ContinuousActions[0], -1f, 1f);
    var actionX = 2f * Mathf.Clamp(actionBuffers.ContinuousActions[1], -1f, 1f);

    gameObject.transform.Rotate(new Vector3(0, 0, 1), actionZ);
    gameObject.transform.Rotate(new Vector3(1, 0, 0), actionX);
}
```

Discrete Actions

Discrete actions are arrays of integers representing specific choices, grouped into **branches**. Each branch can handle multiple options. For example, an agent controlling movement and jumping might have two branches: one with five options for movement and another with two options for jumping. The OnActionReceived() method would look something like:

```
// Get the action index for movement
int movement = actionBuffers.DiscreteActions[0];
// Get the action index for jumping
int jump = actionBuffers.DiscreteActions[1];

// Look up the index in the movement action list:
if (movement == 1) { directionX = -1; }
if (movement == 2) { directionX = 1; }
if (movement == 3) { directionZ = -1; }
if (movement == 4) { directionZ = 1; }
// Look up the index in the jump action list:
if (jump == 1 && IsGrounded()) { directionY = 1; }

// Apply the action results to move the Agent
gameObject.GetComponent<Rigidbody>().AddForce(
    new Vector3(
        directionX * 40f, directionY * 300f, directionZ * 40f));
```

Masking Discrete Actions:

Discrete actions can be dynamically disabled using the *WriteDiscreteActionMask()* method. This ensures the Agent avoids invalid or irrelevant actions in specific states, improving training efficiency:

```
public override void WriteDiscreteActionMask(IDiscreteActionMask actionMask)
{
    actionMask.SetActionEnabled(branch, actionIndex, isEnabled);
}
```

Where:

- *branch* is the index (starting at 0) of the branch on which you want to allow or disallow the action
- *actionIndex* is the index of the action that you want to allow or disallow.
- *isEnabled* is a bool indicating whether the action should be allowed or now.

For example, if you have an Agent with 2 branches and on the first branch (branch 0) there are 4 possible actions : "do nothing", "jump", "shoot" and "change weapon". Then with the code bellow, the Agent will either "do nothing" or "change weapon" for their next decision (since action index 1 and 2 are masked):

```
public override void WriteDiscreteActionMask(IDiscreteActionMask actionMask)
{
    actionMask.SetActionEnabled(0, 1, false);  // Disable action 1 on branch 0
    actionMask.SetActionEnabled(0, 2, false);  // Disable action 2 on branch 0
}
```

Notes:

- You can call *SetActionEnabled* multiple times if you want to put masks on multiple branches.
- At each step, the state of an action is reset and enabled by default.
- You cannot mask all the actions of a branch.
- You cannot mask actions in continuous control.

IActuator interface and ActuatorComponents

Actuators abstract action processing away from the Agent itself, allowing complex behaviors to be modularized. By implementing the *IActuator* interface, advanced users can create reusable action logic. Actuators are initialized during *Agent.Initialize()* and can be attached to the Agent's GameObject or a child GameObject.

Actions Summary & Best Practices:

- Agents can use *Discrete* and/or *Continuous* actions.

- Discrete actions can have multiple action branches, and it's possible to mask certain actions so that they won't be taken.
- In general, fewer actions will make for easier learning.
- Be sure to set the Continuous Action Size and Discrete Branch Size to the desired number for each type of action, and not greater, as doing the latter can interfere with the efficiency of the training process.
- Continuous action values should be clipped to an appropriate range. The provided PPO model automatically clips these values between -1 and 1, but third party training systems may not do so.

Rewards

Rewards in reinforcement learning signal to an Agent that it has performed correctly. The training algorithm, such as PPO, optimizes actions to maximize cumulative rewards over time. Rewards are unused during inference or imitation learning, serving exclusively as guidance for training.

A good practice is to start with simple reward systems and incrementally add complexity. Rewards should reflect results rather than specific actions believed to lead to success. Use the *AddReward()* method to incrementally assign rewards or *SetReward()* to overwrite previous rewards. Keep rewards within the range of *[-1, 1]* to ensure stable training. For multiple calls to *AddReward()*, the values are summed, while *SetReward()* overrides prior rewards.

Examples:

GridWorld Example: The agent earns **+1** for reaching a goal and **-1** for falling into a pit. This sparse reward system requires extensive exploration to find infrequent rewards.

```
Collider[] hitObjects = Physics.OverlapBox(trueAgent.transform.position,
                                           new Vector3(0.3f, 0.3f, 0.3f));
if (hitObjects.Where(col => col.gameObject.tag == "goal").ToArray().Length == 1)
{
    AddReward(1.0f);
    EndEpisode();
}
else if (hitObjects.Where(col => col.gameObject.tag == "pit").ToArray().Length ==
1)
{
    AddReward(-1f);
    EndEpisode();
}
```

Area Example: A small negative reward (**-0.005**) is given for every step, incentivizing task completion. Falling off results in a larger penalty (**-1**).

```
AddReward( -0.005f);
MoveAgent(act);

if (gameObject.transform.position.y < 0.0f ||
    Mathf.Abs(gameObject.transform.position.x - area.transform.position.x) > 8f ||
    Mathf.Abs(gameObject.transform.position.z + 5 - area.transform.position.z) > 8)
{
    AddReward(-1f);
    EndEpisode();
}
```

3DBall Example: The agent gets a small positive reward (**+0.1**) for balancing the ball and a negative penalty (**-1**) if the ball falls off the platform.

```
SetReward(0.1f);

// When ball falls mark Agent as finished and give a negative penalty
if ((ball.transform.position.y - gameObject.transform.position.y) < -2f ||
    Mathf.Abs(ball.transform.position.x - gameObject.transform.position.x) > 3f ||
    Mathf.Abs(ball.transform.position.z - gameObject.transform.position.z) > 3f)
{
    SetReward(-1f);
    EndEpisode();
}
```

Each example uses *EndEpisode()* to terminate episodes upon goal completion or failure, regardless of the Max Step property.

Rewards Summary & Best Practices:

- Use *AddReward()* to accumulate rewards between decisions. Use *SetReward()* to overwrite any previous rewards accumulate between decisions.
- The magnitude of any given reward should typically not be greater than 1.0 in order to ensure a more stable learning process.
- Positive rewards are often more helpful to shaping the desired behavior of an agent than negative rewards. Excessive negative rewards can result in the agent failing to learn any meaningful behavior.
- For locomotion tasks, a small positive reward (+0.1) for forward velocity is typically used.
- If you want the agent to finish a task quickly, it is often helpful to provide a small penalty every step (-0.05) that the agent does not complete the task. In this case completion of the task should also coincide with the end of the episode by calling EndEpisode() on the agent when it has accomplished its goal.

Destroying an Agent

Agents can be destroyed during simulation but ensure at least one Agent is active at all times. This can be achieved by spawning new Agents upon destruction or resetting the environment.

Defining Multi-agent Scenarios

Self-play is triggered by including the self-play hyperparameter hierarchy in the trainer configuration. To distinguish opposing agents, set the team ID to different integer values in the behavior parameters script on the agent prefab.

Adversarial Scenarios (Teams):

- Symmetric games (e.g., Tennis, Soccer): Opposing teams share the same Behavior Name and Policy.
- Asymmetric games: Different Behavior Names and Policies are required for each team.
- Enable self-play by including self-play settings in the trainer configuration.

Cooperative Scenarios (Groups):

- Use *SimpleMultiAgentGroup* to enable group-level rewards and episode management.
- Register agents to a group using *RegisterAgent()* and add rewards using *AddGroupReward()*.
- Use *EndGroupEpisode()* to end the group's episode or *GroupEpisodeInterrupted()* for mid-episode resets.
- Multi-agent groups are compatible with MA-POCA for cooperative training and can coexist with self-play for hybrid scenarios.

See the Cooperative Push Block environment for an example of how to use Multi Agent Groups, and the Dungeon Escape environment for an example of how the Multi Agent Group can be used with agents that are removed from the scene mid-episode.

NOTE: Groups differ from Teams (for competitive settings) in the following way - Agents working together should be added to the same Group, while agents playing against each other should be given different Team Ids. If in the Scene there is one playing field and two teams, there should be two Groups, one for each team, and each team should be assigned a different Team Id. If this playing field is duplicated many times in the Scene (e.g. for training speedup), there should be two Groups per playing field, and two unique Team Ids for the entire Scene. In environments with both Groups and Team Ids configured, MA-POCA and

self-play can be used together for training. In the diagram below, there are two agents on each team, and two playing fields where teams are pitted against each other. All the blue agents should share a Team Id (and the orange ones a different ID), and there should be four group managers, one per pair of agents. Please see the SoccerTwos environment for an example.

Cooperative Behaviors Notes and Best Practices:

- **Single Group Registration:** Agents can only belong to one *MultiAgentGroup* at a time. Unregister an agent before assigning it to another group.
- **Unified Behavior Names:** All agents in a group must share the same Behavior Name and Behavior Parameters.
- **Group-Managed Max Steps:** Set individual agents' Max Steps to 0 and use *GroupEpisodeInterrupted()* to handle episode limits for the entire group.
- **Agent Removal:** Avoid calling *EndEpisode()* for removed agents. Instead, disable or destroy agents and re-register them when the next episode starts.
- **Group vs Individual Rewards:** Use *AddGroupReward()* for group-level reinforcement. Individual rewards (*Agent.AddReward()*) can still be used for active agents.
- **Training Considerations:** Use PPO or SAC for training multi-agent groups, but note group rewards won't influence deactivated agents' learning.

Recording Demonstrations

- Add the Demonstration Recorder component to a GameObject containing an Agent to record demonstrations.
- Specify the name of the demonstration and enable *Record*. Use *Num Steps To Record* to control the recording duration or allow manual termination.
- The demonstration file is saved in the *Assets/Demonstrations* folder and contains metadata viewable in the Inspector.
- Use the recorded *.demo* file for training with imitation learning by providing its path in the training configuration.

Glossary

Agent Class

The **Agent** class in Unity ML-Agents is the core component for creating AI agents in your environment. It provides the functionality to define an agent's behavior, observations, and interactions with the environment, enabling reinforcement learning through customizable actions, rewards, and training episodes.

The common methods listed bellow can be overriden for adding custom behavior, as will be seen in the following guides.

Common methods of the Agent Class:

- ***public void OnActionReceived(ActionBuffers actions)***
  Defines the agent's response to actions received from its policy or heuristic. Use this to implement behaviors such as movement, interactions, or triggering events in the environment. You can also calculate rewards based on the actions to guide the learning process.
- ***public void CollectObservations(VectorSensor sensor)***
  Collects data about the environment and adds it to the provided sensor. Observations, such as positions, distances, and velocities, inform the agent's decision-making process and are critical for training its policy.
- ***public void OnEpisodeBegin()***
  Resets the agent and the environment at the start of each episode. Use this to reposition objects, clear variables, or reinitialize states to ensure consistent conditions for learning.
- ***public void Heuristic(in ActionBuffers actionsOut)***
  Provides a way to manually define actions for the agent, often based on user input or predefined logic. This is useful for debugging or creating baseline behaviors to compare against learned policies.
- ***public void Initialize()***
  Called once when the agent is created. It’s typically used to set up components, initialize variables, or configure any static properties required for the agent’s operation.
- ***public void WriteDiscreteActionMask(ActionMasker actionMasker)***
  Specifies which actions the agent is allowed or disallowed to take in discrete action spaces. This helps prevent invalid actions and can guide the agent towards valid and efficient strategies.
- ***public float GetReward()***
  Returns the current reward for the agent. This represents immediate feedback based

on the agent's most recent actions, helping measure its success in achieving objectives.

- ***public void SetReward(float reward)***
  Sets the reward for the agent in the current step. Use this to provide positive or negative feedback for specific actions or outcomes.
- ***public void AddReward(float increment)***
  Increments the agent's reward by a specified value. This is useful for adding rewards throughout the episode, such as for partial progress or minor achievements.
- ***public void EndEpisode()***
  Ends the current episode prematurely. This can be used to signal that the agent has reached a terminal state, either by achieving its goal or failing.
- ***public void RequestDecision()***
  Requests a new decision from the agent's policy. This is used when decisions are made at specific intervals rather than at every step, allowing for variable decision timing.
- ***public void RequestAction()***
  Asks the agent to execute an action immediately without waiting for a decision from its policy. Useful for situations requiring instant responses.
- ***public void OnCollisionEnter(Collision collision)***
  Triggered when the agent collides with another object. You can use this to assign penalties, rewards, or other responses to collisions, depending on your environment design.
- ***public void OnTriggerEnter(Collider other)***
  Triggered when the agent enters a trigger zone. This is commonly used to detect goals, restricted areas, or checkpoints in the environment.
- ***public float GetCumulativeReward()***
  Returns the total reward accumulated by the agent over the current episode. This provides a measure of overall performance and progress during training.

Behavior Parameters

Regarding the various options that will appear on the Behavior Parameters component:

Vector Observation

- **Space Size:** This represents the **length of the state vector** for the agent. In a **continuous state space**, it refers to the number of floating-point observations the agent collects per decision. In a **discrete state space**, it refers to the number of distinct possible states.
- **Stacked Vectors:** Indicates the number of previous states (frames) to stack together before being fed into the neural network. Useful in time-dependent tasks where the current state alone doesn't fully capture the situation (e.g., velocity or momentum in

physics simulations). Higher values mean the network receives multiple observations at each step, providing more context.

Actions

- **Continuous Actions:** Represents the number of **continuous control outputs** (e.g., values between -1 and 1 for steering or throttle).
- **Discrete Branches:** Specifies the number of different actions the agent can perform, each branch being one action space.
    - **(Branch Size):** Specifies the number of **choices or actions** available in a discrete action space (e.g., move left, move right, jump).

Model: Points to the pre-trained neural network model asset that the agent should use for inference. If left empty, the agent trains from scratch.

- **Inference Device:** Selects the way the inference will be performed. Options: “Default”, “Compute Shader”, “Burst”, “Pixel Shader”.
- **Deterministic Inference:** Ensures deterministic (reproducible) actions during inference. Only applies for Inferences from within Unity.

Behavior Type: Determines how the agent behaves:

- **Default:** Learns or performs using the set policy/model.
- **Heuristic Only:** Lets you manually specify actions for debugging or custom rules.
- **Inference Only:** Only uses the provided model for decision-making.

Team ID: Assigns the agent to a specific team. Useful in multi-agent environments for categorizing agents and enabling team-specific objectives.

Use Child Sensors: If enabled, the agent can use sensors attached to child objects for observation.
Observable Attribute Handling: Specifies how the agent’s observation data is handled. Options: “Ignore”, “Exclude Inherited”, or “Examine All”.

Episodes

Episodes are the building blocks of reinforcement learning in ML-Agents, enabling the agent to repeatedly learn, fail, and improve its behavior over time.

Key Characteristics of an Episode:

1. **Starting Point:**
    - An episode begins with the environment and agent being reset to their initial conditions (e.g., the agent is placed at a starting position, and the goal is placed in a specific location).
2. **Agent Actions and Rewards:**
    - During the episode, the agent takes actions based on its current policy, receives observations about the environment, and earns rewards (or penalties) based on its behavior.
3. **Ending Conditions:**
    - The episode ends when:
        - The agent achieves the desired goal (e.g., reaches the target or completes a task).
        - The agent fails (e.g., collides with an obstacle or runs out of time).
        - A predefined maximum step limit is reached.
4. **Learning Process:**
    - At the end of each episode, the collected data (observations, actions, rewards) is used to update the agent's policy (if training). The agent then starts a new episode, learning iteratively over multiple episodes.
5. **Episode Length:**
    - The duration of an episode depends on the task's complexity and the conditions for success or failure. Shorter episodes are generally better for faster learning but may require more training to master complex tasks.

Why Are Episodes Important?

- **Evaluation of Progress:** Episodes provide a clear boundary for measuring the agent's performance, such as whether it can consistently achieve the goal.
- **Reward Feedback:** They define when cumulative rewards are calculated, helping the agent understand long-term consequences of actions.
- **Exploration and Reset:** By resetting the environment at the start of each episode, the agent can explore different scenarios and avoid getting stuck in suboptimal states.

Events

Events in Unity ML-Agents enable custom behaviors to trigger specific actions or logic in other parts of the environment, using a publisher-subscriber model. Events provide a mechanism for communication between different scripts or components, enhancing modularity and flexibility.

Publisher-Subscriber Model

The publisher-subscriber pattern connects an event (published by one script) to handlers (subscribed methods in another script) that execute when the event is invoked. For example, consider the following scenario:

1. **Agent Script (Publisher):** The agent publishes an event, *OnEpisodeBeginEvent*, which is triggered at the start of each episode:

```
public event EventHandler OnEpisodeBeginEvent;
public override void OnEpisodeBegin()

{
    transform.localPosition = new Vector3(
        UnityEngine.Random.Range(-2.5f, +2.5f),
        0,
        UnityEngine.Random.Range(-2.0f, +2.0f)
    );

    // Invoke the event to notify subscribers
    OnEpisodeBeginEvent?.Invoke(this, EventArgs.Empty);
}
```

2. **FoodButton Script (Subscriber):** Another script subscribes to the agent's *OnEpisodeBeginEvent* and executes specific logic when the event is triggered:

```
[SerializeField] private PressButtonAgent agent;

private void Start()
{
    // Subscribe to the agent's event
    if (agent != null)
    {
        agent.OnEpisodeBeginEvent += HandleEpisodeBegin;
    }
}

private void HandleEpisodeBegin(object sender, EventArgs e)
{
    // Logic to execute when the event is triggered
    Debug.Log("Episode has begun!");
}
```

Key Concepts:

- **Publisher:** The agent script declares an event (OnEpisodeBeginEvent) and invokes it at a specific point (OnEpisodeBegin method).
- **Subscriber:** The FoodButton script registers a handler (HandleEpisodeBegin) to the agent's event, ensuring it executes custom logic when the event occurs.
- **Invocation:** The event is invoked using ?.Invoke, which checks if there are any subscribers before calling the event, preventing null reference exceptions.

Best Practices:

- **Null Checking:** Use the null conditional operator (?.Invoke) to avoid errors if there are no subscribers.
- **Encapsulation:** Expose events using public access modifiers but keep the invocation logic within the publisher script.
- **Unsubscribe:** Always unsubscribe from events to avoid memory leaks, especially when objects are destroyed or disabled:

```
private void OnDestroy()
{
    if (agent != null)
    {
        agent.OnEpisodeBeginEvent -= HandleEpisodeBegin;
    }
}
```

Events Summary:

- Events allow seamless communication between scripts, enabling modular and scalable design.
- The publisher-subscriber model lets multiple components react to an event, improving code maintainability.
- Proper handling of subscriptions and unsubscriptions ensures efficient resource usage and prevents runtime issues.

APÊNDICE 5

Termo de Aceite de Entrega

Objetivo deste documento

Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião ("gate") de aprovação: 7 de out. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

Lucas Brandão

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

Durante esse Stage (Stage 10 - 051224), foram realizadas as seguintes atividades:

Finalização do material didático Unity ML-Agents :

- **Seção "Introdução" (1 página):** Breve introdução descrevendo quem eu sou, o que é esse projeto, onde encontrar os artefatos criados durante o projeto e outras informações.
- **Seção "Imitation Learning" (15 páginas):** Minha implementação de um cenário demonstrando o uso de Imitation Learning, com base no tutorial Teach your AI! Imitation Learning with Unity ML-Agents! .
- **Seção "Flappy Bird" (10 páginas):** Minha implementação de RL para um clone do jogo Flappy Bird (disponível em um repositório no github com tutorial de implementação), com base no tutorial AI Learns to play Flappy Bird! .
- **Seção "Conclusões" (1 páginas):** Reflexão sobre o processo de pesquisa e aprendizado, identificando áreas de melhoria e possíveis próximos passos para aprofundar o estudo do tema.

Criação de um repositório no GitHub permitindo acesso aos projetos práticos desenvolvidos durante o processo da Residência.

Planejamento: [descrever o que pretende fazer para realizar a próxima ENTREGA]

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

ACEITE DA ENTREGA:

CEDRIC LUIZ DE CARVALHO: Go!

Instituto de Informática (INF) - UFG
Alameda Palmeiras, Quadra D, Câmpus Samambaia
Fone: (62) 3521-1181 / Fax: (62) 3521-1182

Material Didático - Seções Práticas

Durante as últimas semanas da Residência em IA, criei um material didático que aborda conceitos teóricos e práticos de Aprendizado por Reforço e Imitation Learning, com foco na aplicação prática utilizando Unity. O Unity ML-Agents é uma das opções mais populares e bem documentadas para integrar aprendizado por reforço em jogos. A engine Unity é amplamente utilizada na indústria de desenvolvimento de jogos, e o ML-Agents Toolkit oferece um conjunto robusto de recursos para criar agentes inteligentes em ambientes 3D interativos. Além disso, existem muitas informações disponíveis, incluindo tutoriais oficiais [1, 2, 3], vídeos explicativos [1, 2, 3], e exemplos práticos [1, 2, 3, 4] que facilitam o aprendizado e a aplicação rápida.

O material criado inclui tutoriais passo a passo, exemplos comentados e observações pessoais sobre as principais dificuldades e soluções encontradas. As seções cobrem desde a configuração do ambiente de desenvolvimento e a criação de agentes simples até a implementação de cenários mais complexos, como o controle de agentes em jogos e a análise de desempenho. Esse material serve como um guia de apoio para futuros estudos e projetos relacionados à área de IA aplicada a jogos.

A **parte prática** do material didático desenvolvido consiste nas seguintes seções:

Seção "Scene: Basic": Minhas observações sobre um dos exemplos de Environment da Unity chamado "Basic", seguindo o tutorial How to use Machine Learning AI in Unity! (ML-Agents). **Subseções:**

- **Agent:** Configuração e criação de um Agente no Unity, incluindo ações (discretas ou contínuas), observações do ambiente e atribuição de recompensas e penalidades.
- **Testing the Script:** Testes do comportamento do modelo treinado e ajustes do ambiente para validação.
- **Training:** Execução de treinamento básico, ajustes de hiperparâmetros e monitoramento do progresso com TensorBoard.

Seção "Imitation Learning": Minha implementação de um cenário mais complexo demonstrando o uso de Imitation Learning através da gravação de demos, com base no tutorial Teach your AI! Imitation Learning with Unity ML-Agents!. **Subseções:**

- **Scenario:** Explicação do ambiente de aprendizado, incluindo o design e implementação das ações, observações, recompensas e critérios de término do episódio.
- **Demonstration:** Gravação de demonstrações para o aprendizado do agente.

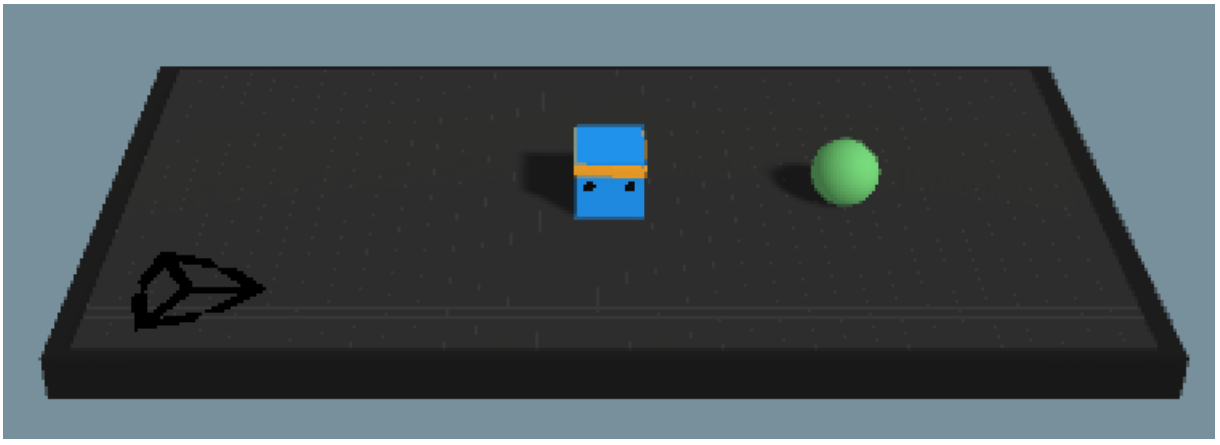
- **Training:** Configuração de métodos de treinamento (Behavioral Cloning, GAIL e Recompensas Extrínsecas) e ajuste de parâmetros.
- **Results:** Análise dos resultados e desempenho obtido ao longo dos testes.

Seção "Flappy Bird": Minha implementação de RL para um clone do jogo Flappy Bird (disponível em um repositório no github com tutorial de implementação), com base no tutorial AI Learns to play Flappy Bird! . **Subseções:**

- **Scenario:** Explicação do ambiente de aprendizado, incluindo o design e implementação das ações, observações, recompensas e critérios de término do episódio.
- **Training:** Configuração de métodos de treinamento (Behavioral Cloning, GAIL e Recompensas Extrínsecas) e ajuste de parâmetros.
- **Results:** Análise dos resultados e desempenho obtido ao longo dos testes.

Scene: Basic

The goal of this section is to make a simple Agent (blue cube) and teach it to reach a Goal (green ball)[12], using one of the example environments from Unity, as can be seem in the image below:



[12] For this section, I will be following the tutorial
How to use Machine Learning AI in Unity! (ML-Agents) .

Agent

An agent is an autonomous actor that observes and interacts with an *environment*. In the context of Unity, an environment is a scene containing one or more Agent objects, and, of course, the other entities that an agent interacts with.

An Agent, for our context, is basically a C# script that will be a component of a GameObject (in this case, the “player” of the game).

To create an agent in your Unity Project, do the following:

- Right click on the “Scripts” folder and choose *Create -> C# Script*;
- We will name it “MoveToGoalAgent”;

Open the script and then do the following:

- Add the line “*using Unity.MLAgents;*” to the top, to import the necessary libraries;
- Replace the inheritance class from “*MonoBehaviour*” to “*Agent*”[13];
- Delete the default methods *“Start()”* and *“Update()”*.

Adding the script to the Agent

Add the script to the Agent by selecting the Agent GameObject on the Hierarchy and dragging the script to the Inspector on the right side. As a result, two new components will be added to the Inspector: the MoveToGoalAgent (Script) component and the Behavior Parameters[14] component, for setting the AI model hyperparameters. Change the behavior name to “MoveToGoal” for now.

Some important parameters that we will be using for defining the Agent actions are:

- **Continuous Actions:** Represents the number of **continuous control outputs** (e.g., values between -1 and 1 for steering or throttle).
- **Discrete Branches:** Specifies the number of different actions the agent can perform, each branch being one type of action (action space).
    - **(Branch Size):** Specifies the number of **choices or actions** available in a discrete action space (e.g., move left, move right, jump).

[13] For more information on the Agent class, check out the ML-Agents -> Glossary section.
[14] For more information on Behavior Parameters, check out the ML-Agents -> Glossary section.

Observations

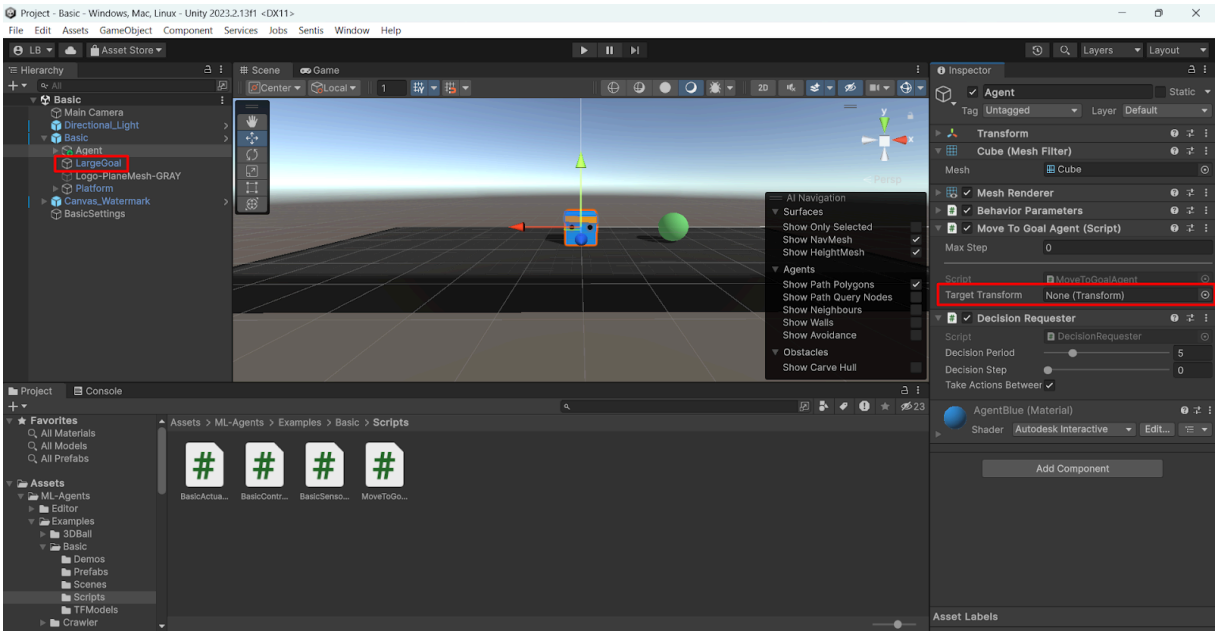
Observations are how the agent perceives its environment. We need to think about “what data does the AI model needs in order to solve the problem we are giving to it?”. Our goal in this example is to move the Agent to the Goal. In order to do that, the Agent may need to know things like:

- Where it is in the environment (Agent position coordinates);
- Where the target is (Goal position coordinates);

The way Observations are collected are by overriding a function: *void CollectObservations(VectorSensor sensor){}*. To collect the Agent position, we can write:

```
public override void CollectObservations(VectorSensor sensor)
{
    sensor.AddObservation(transform.position); // Agent position
}
```

To collect the Goal position, we need to have a reference to the Goal GameObject. For that, add a private variable to the MoveToGoalAgent class: “*[SerializeField] private Transform targetTransform;*”. Then, to reference the Goal object, drag the Goal GameObject to the new field that should have appeared in the Inspector back in the Unity Project:

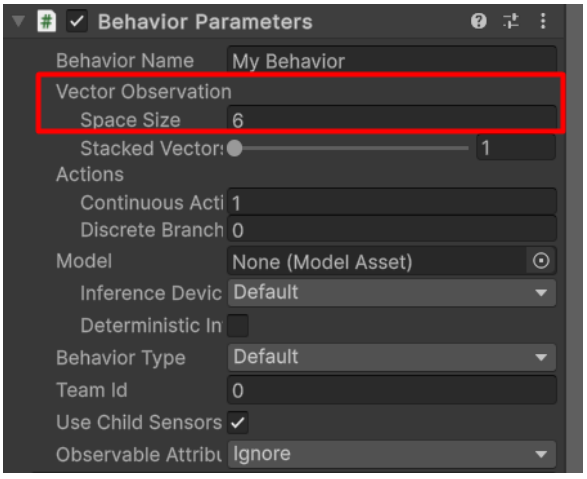


Now add that information to the function:

```
public override void CollectObservations(VectorSensor sensor)
{
    sensor.AddObservation(transform.position); // Agent position
    sensor.AddObservation(targetTransform.position); // Target position
}
```

Now, in the Unity Project, we need to set the correct Space Size for the Vector Observation on the Behavior Parameters component. The Space Size specifies how many inputs (Observations) are being passed to the Agent, in this case, how many floats we are giving it. Each transform.position variable is a XYZ position vector, so we are passing 6 variables in total.

Set the Space Size to 6:



Actions

Now that the Agent nows its own position and the position of the Goal, it can take actions towards reaching the Goal. For that, let's prepare its actions.

First, on the Inspector, change the Actions -> Continuous Actions to 2 (we are moving on X and Z dimensions). Now, on the code for the MoveToGoalAgent class, we will add the actions:

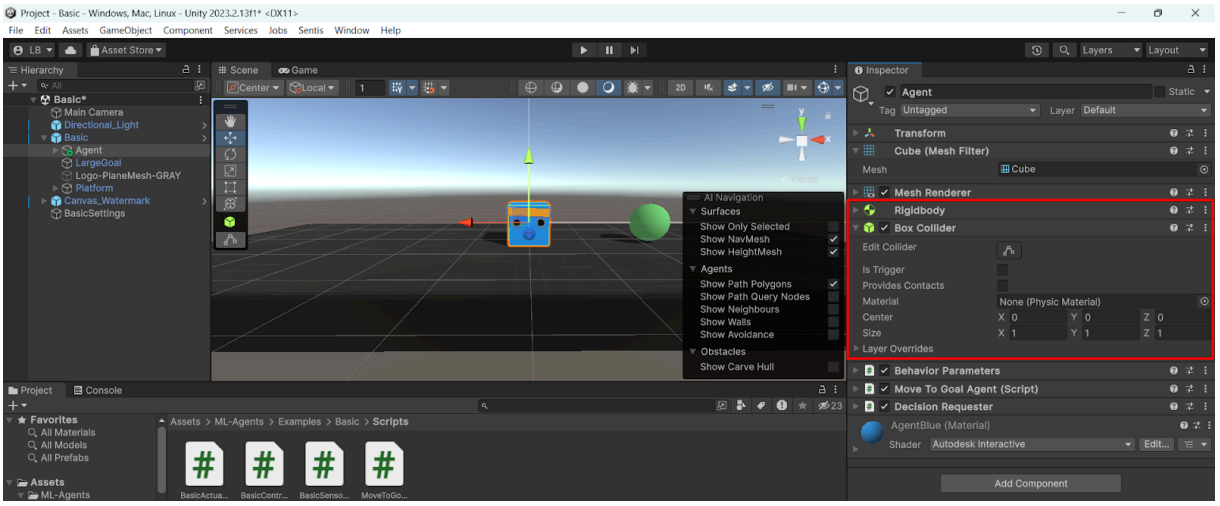
```
public override void OnActionReceived(ActionBuffers actions)
{
    float moveX = actions.ContinuousActions[0];
    float moveZ = actions.ContinuousActions[1];
    float movespeed = 1f;
    transform.position += new Vector3(moveX, 0, moveZ) * Time.deltaTime *
movespeed;
}
```

This is a very simple code that takes two actions (moveX and moveZ), creates a movespeed that we can adjust later, and then changes the position of the Agent based on the actions taken by the model.

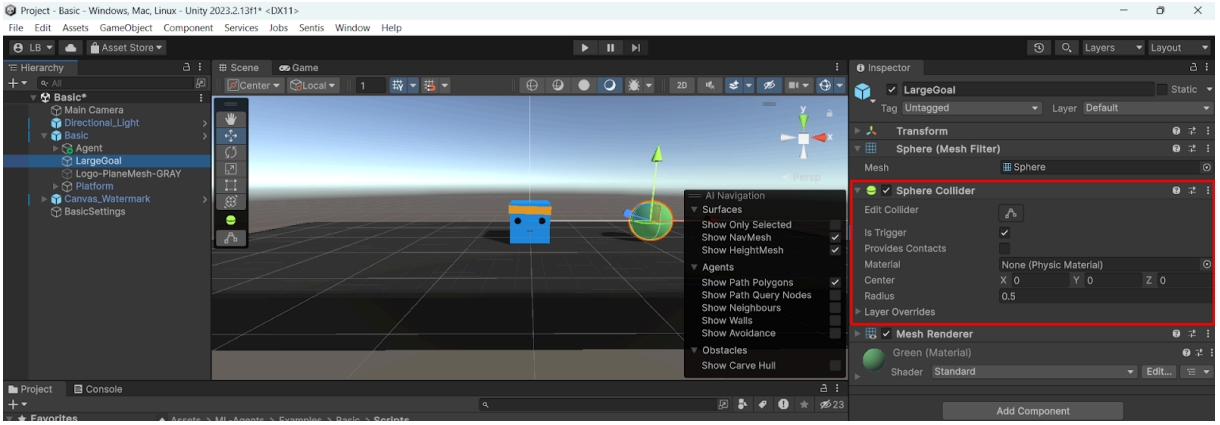
Reward

Finally, for the model to actually learn how to move in the correct direction, we need to ***reward it*** when it moves towards the Goal, and ***punish it*** when it moves in other directions. This way, it will eventually (hopefully) learn to walk and reach the Goal.
To check for collisions between the Agent and the Goal (so as to see if the Agent reached the Goal), we can use Unity's components. For that, the Agent needs two components:

- **RigidBody:** Enables the agent to interact with Unity's physics system, allowing it to move and be affected by forces like gravity and collisions.
- **Box Collider:** Defines the physical boundaries of the agent for detecting collisions with other objects in the environment.



The Goal should have a Sphere Collider with the "Is Trigger" box marked as True:



There are two ways to give a reward to the Agent:

- **SetReward():** sets the reward to an specific amount.
- **AddReward():** increments the current reward by an specific amount.

For this case, it's simple enough so we can just use SetReward. On more complex scenarios, with more goals and actions, it might be better to use AddReward.

Another important thing to note is the concept of "Episodes"[15]. One Episode is essentially one run, and the Episode should end when the character either achieves the final goal or loses.

To test for the collision, let's add the function:

```
public void OnTriggerEnter(Collider other){
    SetReward(+1f);
    EndEpisode();
}
```

After the Episode ends, we need a way to "restart" the game (and follow with training). For resetting the states in this example, we just need to position the Agent back to its initial position:

```
public override void OnEpisodeBegin()
{
    transform.position = Vector3.zero;
}
```

Or, for a bit of randomness, we make the Agent spawn in a random location around his starting position. Make sure to check the actual values on your Engine, as they might be different:

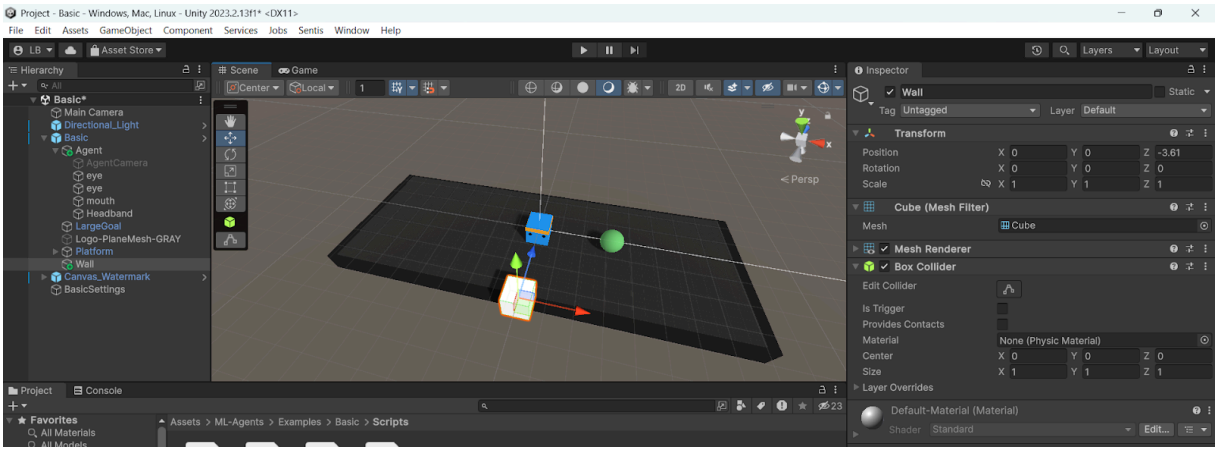
```
public override void OnEpisodeBegin(){
    transform.position = new Vector3(Random.Range(-4f, 4f), 0, Random.Range(-4f,
4f));
    targetTransform.position = new Vector3(Random.Range(-4f, 4f), 0,
Random.Range(-4f, 4f));
}
```

[15] For more information on Episodes, check out the ML-Agents -> Glossary section.

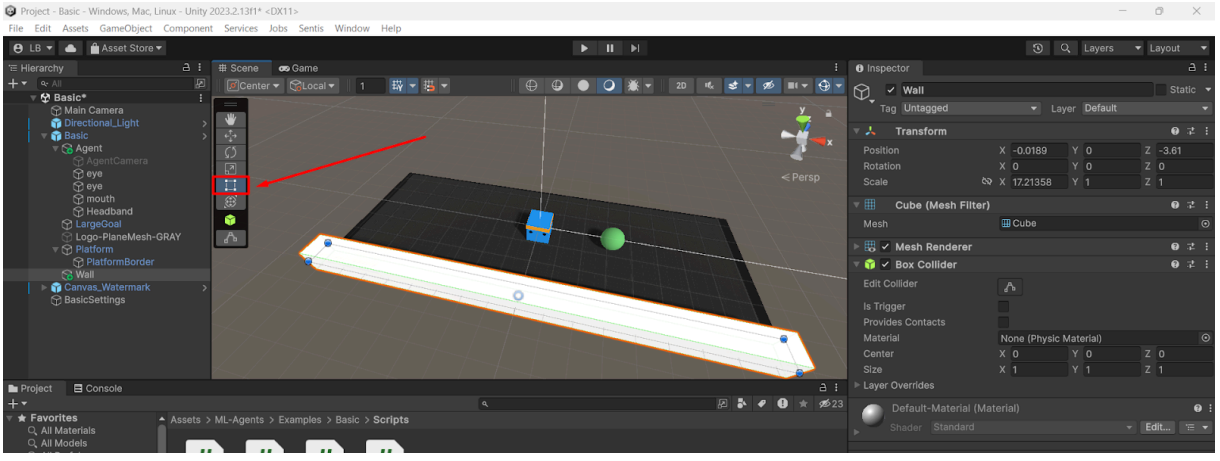
Penalty

For the penalty, let's first add some colliders on the edges of the platform, like walls that will detect if the characters goes out of bounds. If the Agent collides with the wall, it will get a negative reward and end the Episode.

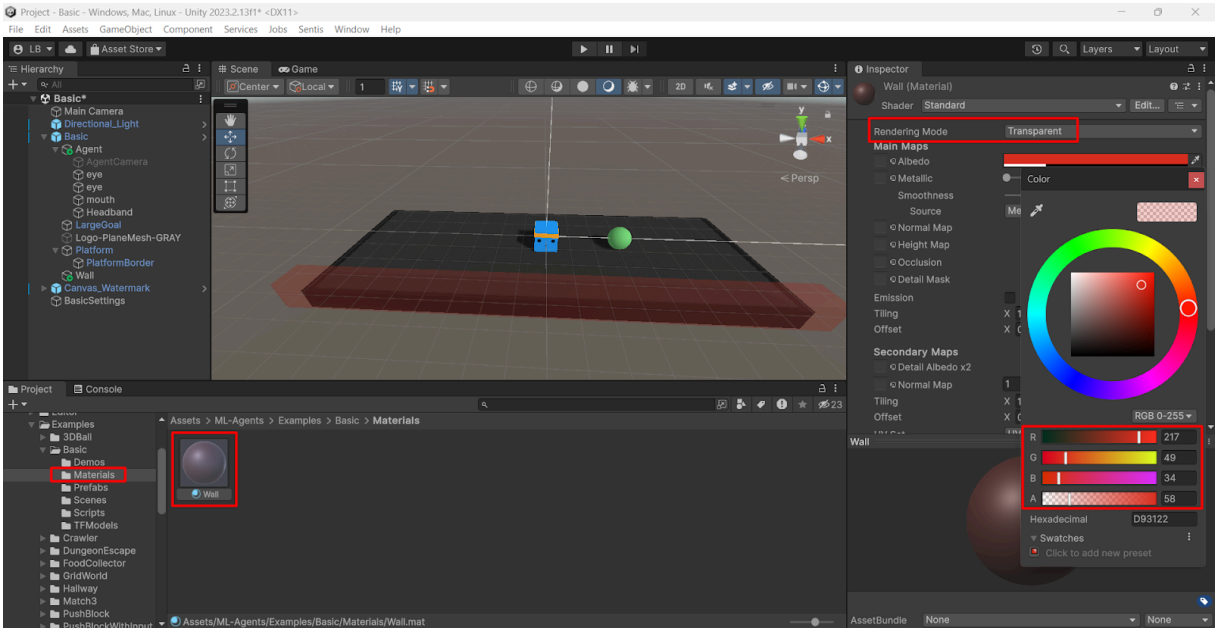
First, make a new GameObject: 3D Object -> Cube and name it "Wall". Position it over the edge of the platform:



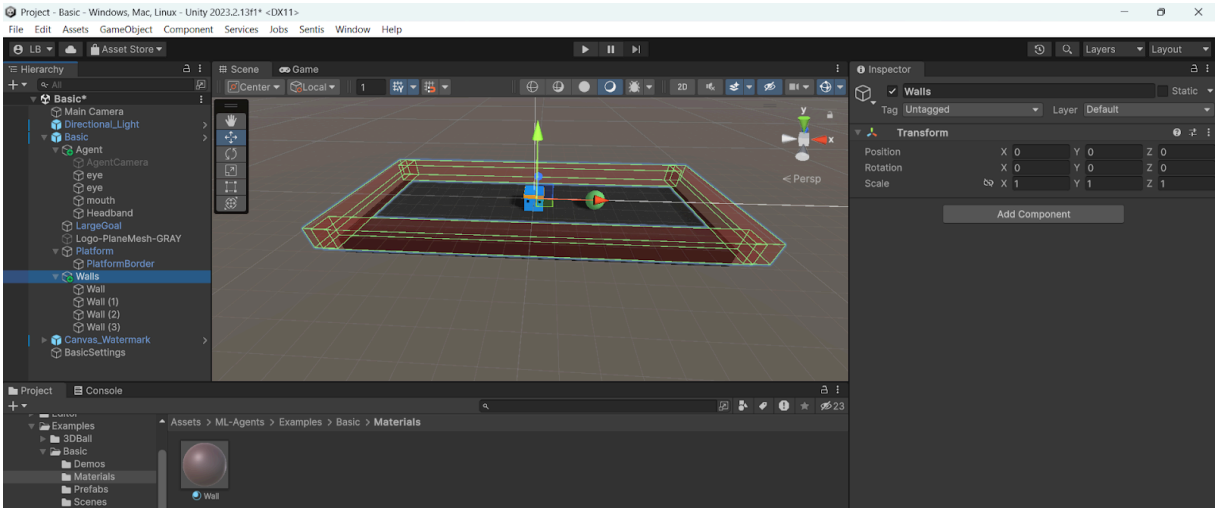
Then use the Rect tool to change it's shape to go around the platform:



Set the Box Collider "Is Trigger" box to True. If you want to change the appearance of the Wall, create a new Material object in your project Assets -> Materials folder (create one if you don't have it), name it "Wall" and drag it to the Wall GameObject that you created earlier. Then, change the color manually:



Do the same for all sides, copying the Wall object and positioning it on the edges. Finally, add it all under the same parent (“Walls”). The final result should look like this:



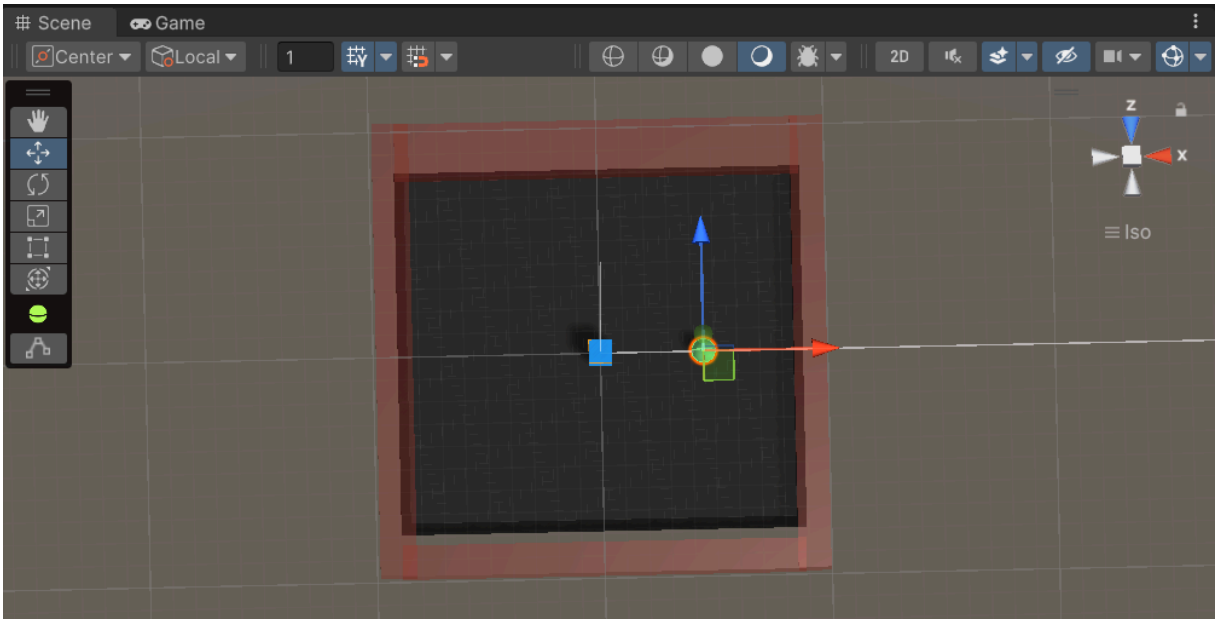
Now we need to identify if the player collides with either the Goal or the Walls. For that, we will add a script to each of them, to act as “Tags”, so we can identify the objects. Add two new C# scripts to the Scripts folder, named “Goal” and “Wall”, and then just drag each script to its corresponding object (the Goal and each of the 4 walls).

Now let's modify the Reward function to have a positive reward on reaching the Goal, and a negative Reward on hitting the walls:

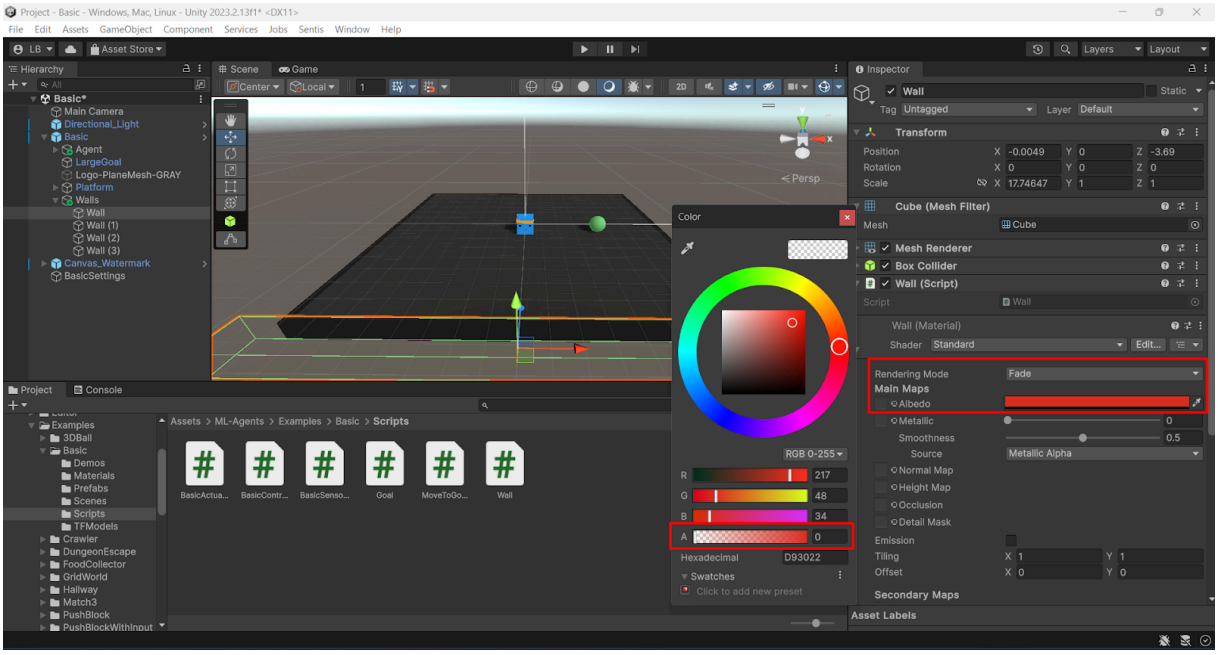
```
public void OnTriggerEnter(Collider other)
{
    if (other.TryGetComponent<Goal>(out Goal goal))
    {
        SetReward(+1f);
        EndEpisode();
    }
    if (other.TryGetComponent<Wall>(out Wall wall))
    {
        SetReward(-1f);
        EndEpisode();
    }
}
```

Adjusting The Scene

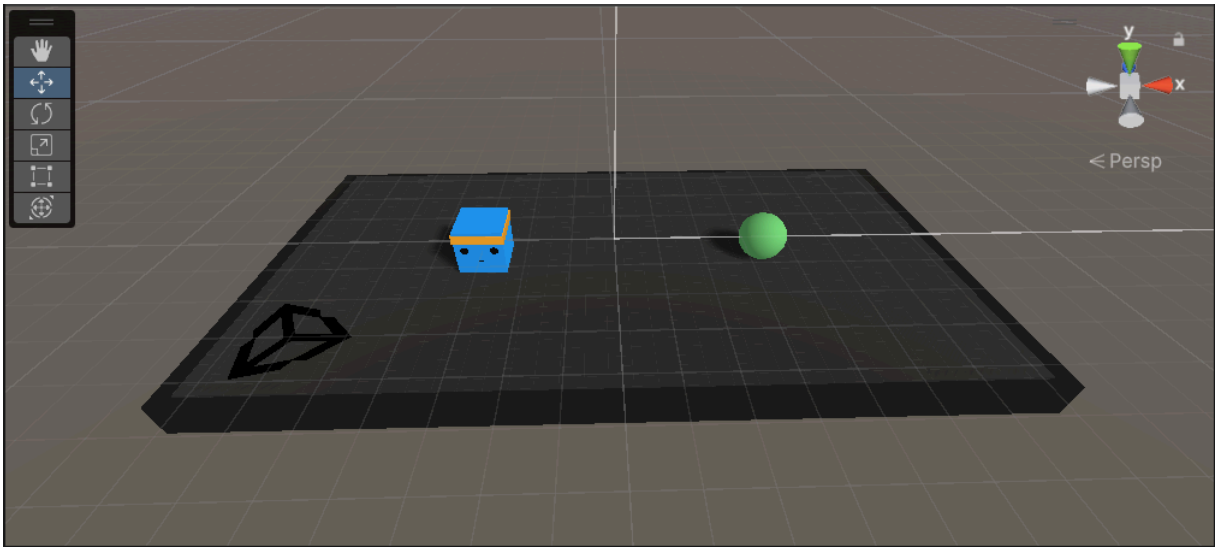
To improve the visuals a little bit, I did a bit of adjustments to the scene, making the platform longer and square, and also adjusted the Agent movespeed to 3f. The final result is this:



I also changed the Alpha value of the color to 0 and the Rendering mode to Fade, in order to hide the walls and make it visually a bit better:



Finally, I also rescaled things a bit, and added the Unity logo to the platform:



Final Script

The final script should look like this:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Unity.MLAgents;
using Unity.MLAgents.Actuators;
using Unity.MLAgents.Sensors;
```

```
public class MoveToGoalAgent : Agent
{
    [SerializeField] private Transform targetTransform;

    public override void OnEpisodeBegin()
    {
        transform.position = Vector3.zero;
    }
    public override void OnActionReceived(ActionBuffers actions)
    {
        float moveX = actions.ContinuousActions[0];
        float moveZ = actions.ContinuousActions[1];
        float movespeed = 3f;
        transform.position += new Vector3(moveX, 0, moveZ) * Time.deltaTime *
movespeed;
    }

    public override void Heuristic(in ActionBuffers actionsOut)
    {
        ActionSegment<float> continuousActions = actionsOut.ContinuousActions;
        continuousActions[0] = Input.GetAxis("Horizontal");
        continuousActions[1] = Input.GetAxis("Vertical");
    }

    public override void CollectObservations(VectorSensor sensor)
    {
        sensor.AddObservation(transform.position); // Agent position
        sensor.AddObservation(targetTransform.position); // Target position
    }

    public void OnTriggerEnter(Collider other)
    {
        if (other.TryGetComponent<Goal>(out Goal goal))
        {
            SetReward(+1f);
            EndEpisode();
        }
        if (other.TryGetComponent<Wall>(out Wall wall))
        {
            SetReward(-1f);
            EndEpisode();
        }
    }
}
```

Testing the Script

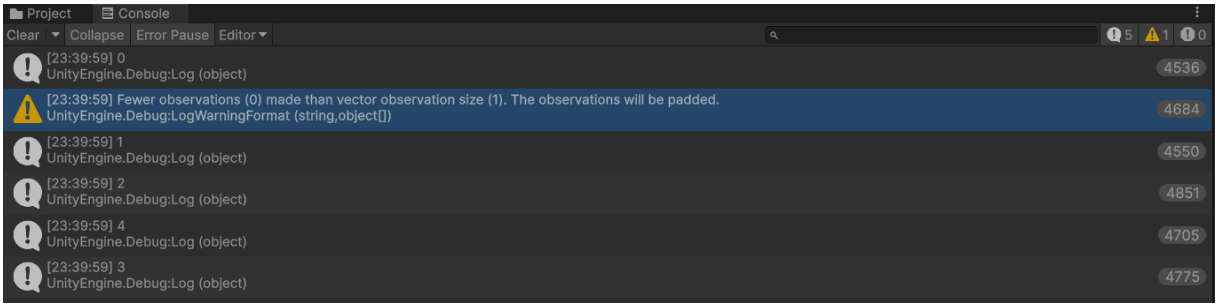
For a simple test, set the “Discrete Branches” parameter to 1 and “Branch 0 Size” to 5. What it means is that our agent has access to one type of action, and this type of action has 5 possible states. For example, it could be the action space for “Movement” and the states could be [“Stop”, “Left”, “Right”, “Up”, “Down”].

Now in the code, call “Debug.Log(actions.DiscreteActions[0]);” inside the function “OnActionReceived”. This should print out the actions vector when an action is executed. But, before this can work, we need to make sure the Agent has access to Observations, and can take Decisions based on them.

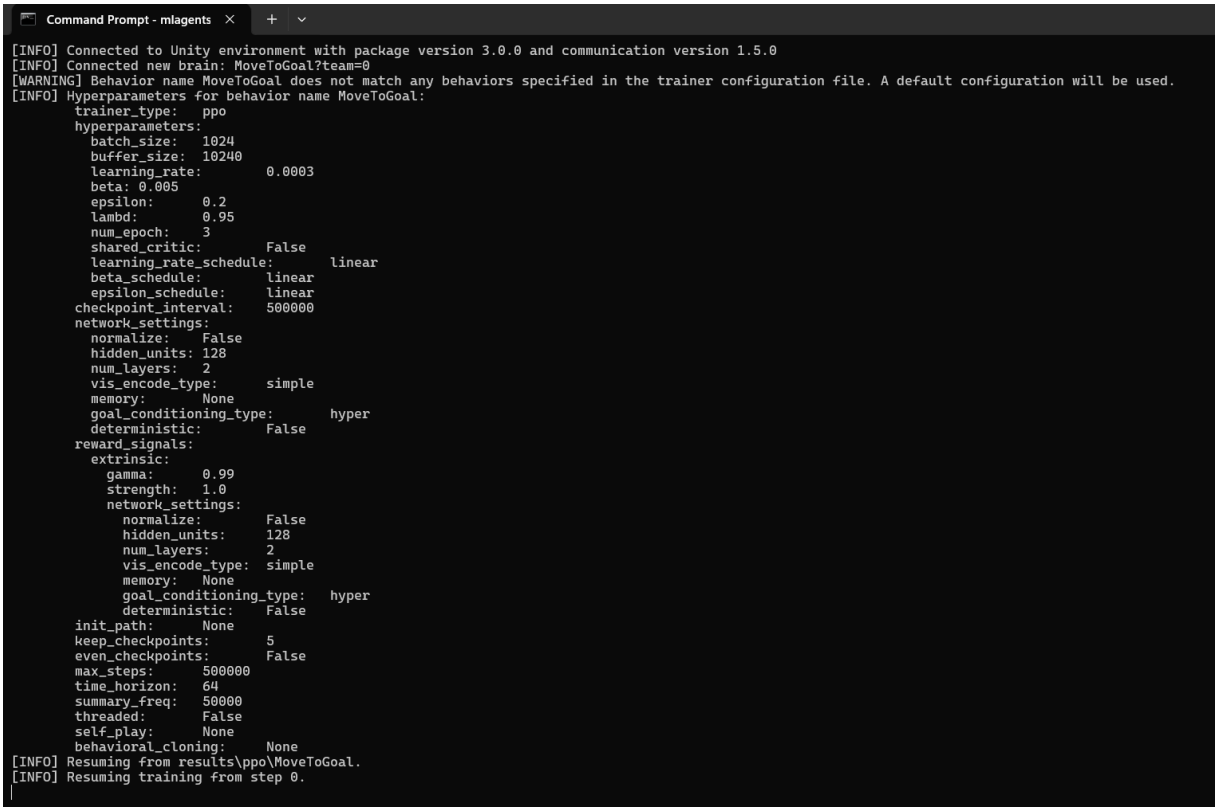
In order for an Agent to take an action, it must first **request a decision**. For that, add a component to your Agent: “*ML Agents -> Decision Requester*”. This component requests a decision every certain amount of time and then take actions based on that.

Now, open command prompt, make sure you are inside your virtual environment, and run the command “*mlagents-learn*”. After that, start training by pressing the Play button in the Unity Editor.

In the Unity Console we can see that our action vector indeed has 5 positions (the output of the model ranges between 0 and 4):



And in the command prompt we can check the hyperparameters of the model used:



For another test, let's change the behavior for continuous actions. Change the value of "Discret Branches" back to 0 and set the value of "Continuous Actions" to 1, meaning that we have one type of continuous action, with an action space ranging from -1 to 1. Then, change the code inside the "OnActionReceived" function to "Debug.Log(actions.ContinuousActions[0]);" and run the trainer again in command prompt, then press the play button on Unity Editor. This time, try adding a tag for the ID of the run with "--run-id=" (e.g. "run-id=Test2").

You can see in the console that now the model outputs consist of continuous values between -1 and 1:

Remember to **delete the log functions** after you are done testing!

Heuristics Testing

We can validate and test our code in another way, which is driving the actions ourselves. For that, we can override the function "*public void Heuristic(in ActionBuffers actionsOut)*", to modify the actions that will be received by the "*public void OnActionReceived(ActionBuffers actions)*" function mentioned before. For now, let's modify the way our Agent decides on an action by letting us move the character ourselves. We can do this by modifying the continuousActions inside the Heuristic function:

```
public override void Heuristic(in ActionBuffers actionsOut)
{
    ActionSegment<float> continuousActions = actionsOut.ContinuousActions;
    continuousActions[0] = Input.GetAxis("Horizontal");
    continuousActions[1] = Input.GetAxis("Vertical");
}
```

Now, for testing it, go to the Behavior Parameters component and change "Behavior Type" to Heuristic Only. After pressing the Run buttom, you should be able to move the character with the movement arrows of your keyboard. Try moving the character around and hitting the walls and the Goal to make sure that everything is working smoothly. The character should reset its position when hitting either the wall or the Goal.

Remember to **change the Behavior Type back to Default** after you are done testing, so as to not affect the actual training!

Training the Agent

Make sure that the Behavior Type is set to Default on the Behavior Parameters component. Then, open the command prompt and let's run a new test, just like we did when Testing the Script:

- Enter your environment where you installed the mlagents package;
- Run "*mlagents-learn --run-id=Training1*" (use "*--resume"* if you want to run the same id after);
- Press play on the Unity Engine.

With this, the basic training is done!! Congratulations 🎉 🥳 🎊 🎁.

Improving the Training

By doing the training the way we are doing right now, a few issues will show up:

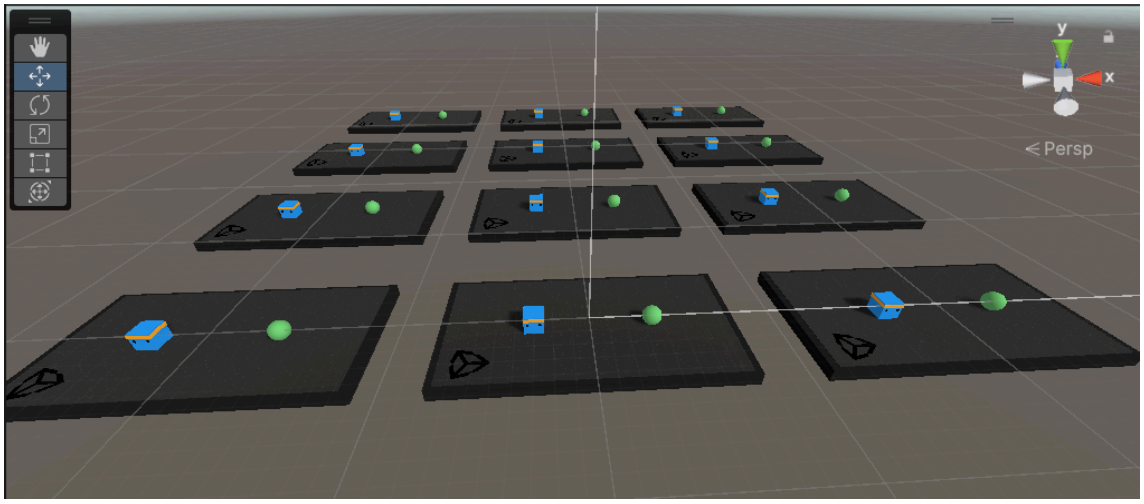
1. The Agent might simply learn to **avoid walls** if he never actually touches the Goal;
2. The training might take **too long**;
3. The Agent might **overfit to the specific location of the Goal**, since we are not making changes to positioning. This could look like the Agent only learning that *moving right* means getting a reward, and when we move the Goal, the Agent doesn't know how to find it anymore.
4. The **rewards and penalties might not best reflect our objective**. For example, touching the wall from very far from the Goal could be more punishing than touching the wall that is right next to the Goal.
5. There should be a penalty for how long the Agent takes to reach the Goal, to **encourage the Agent to be faster**.

1 - Max Step

We will tackle these one by one. Starting from Problem 1, we can solve it by adding a "Max Step" value on the MoveToGoalAgent Script that we created. By default, our training runs 50 times per second (same as the Unity physics update), so we can change that value to around 1000 steps. This will make sure that the **Episode will end after a maximum amount of time** (20 seconds) regardless of whether the Agent touched the Goal or the walls.

2 - Faster Training

For Problem 2, in order to speed up training, we can **use more than one Agent**, a training scenario called “Simultaneous Single-Agent”[16]. Make sure that all of your Scene is inside a single container (in our case, the “Basic” object), and then drag it into your Assets to create a Prefab[17]. Since Unity already gave us a Basic prefab, I will change the name of our container to be “Basic - Lucas”. Then, we can just copy and paste a lot of times, making more Agents train at the same time:



For that to work, we need to make sure that we are using localPosition in our code, and not global position. Change all references to “position” to “localPosition” (e.g.: *transform.position = Vector3.zero; -> transform.localPosition = Vector3.zero;)*.

For better visualization, we can make it so that the floor material will change depending on the results of the Agent action, red for losing and green for winning. We can do that by adding a few more fields to the class and adjusting the Trigger function:

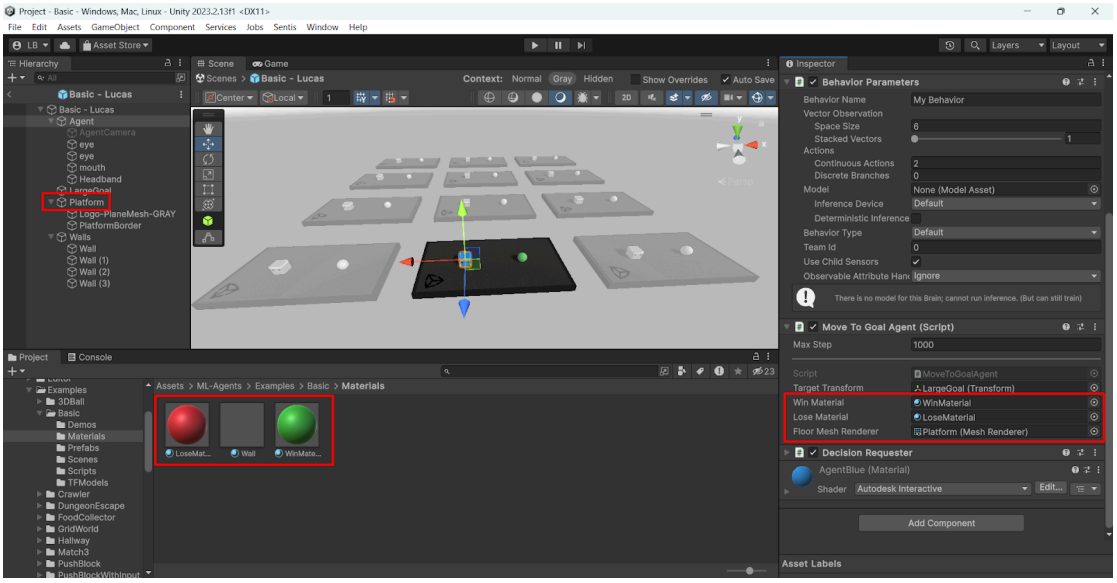
```
[SerializeField] private Material winMaterial;
[SerializeField] private Material loseMaterial;
[SerializeField] private MeshRenderer floorMeshRenderer;

public void OnTriggerEnter(Collider other)
{
    if (other.TryGetComponent<Goal>(out Goal goal))
    {
        SetReward(+1f);
        floorMeshRenderer.material = winMaterial;
        EndEpisode();
```

[16] For more information about the Simultaneous Single-Agent training method, check out the ML-Agents -> Toolkit Overview section.
[17] For more information on the Prefabs, check out the Basics -> Unity section.

```
    }
    if (other.TryGetComponent<Wall>(out Wall wall))
    {
        SetReward(-1f);
        floorMeshRenderer.material = loseMaterial;
        EndEpisode();
    }
}
```

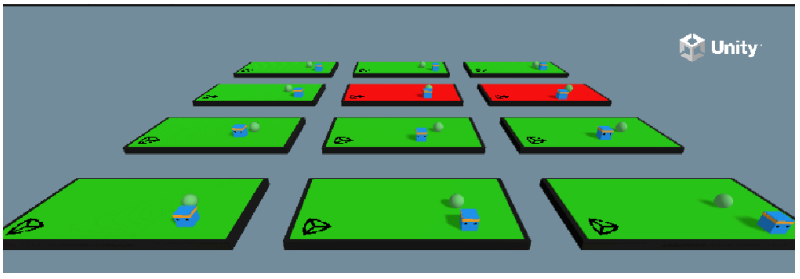
Back in the Editor, open the Prefab, select the Agent and drag the Platform object to the Floor Mesh Renderer field. Create materials for winning and losing and choose colors to represent them, then drag then to the corresponding fields:



Now, we can create a new run on the terminal:

- Enter your environment where you installed the mlagents package;
- Run “mlagents-learn --run-id=MoveToGoal” (use --resume if you want to run the same id after);
- Press play on the Unity Engine.

Now you can visualize when each of the Agents win or fail:

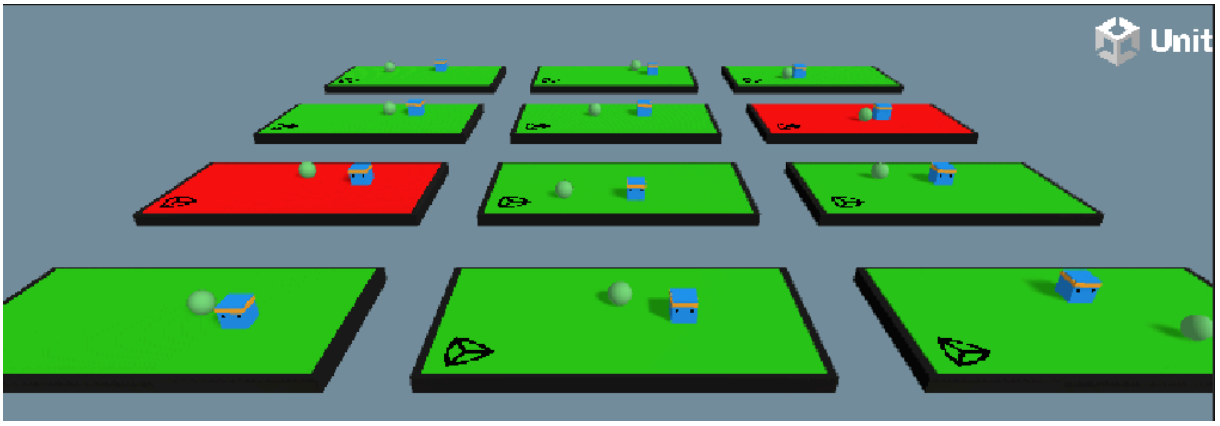


3 - Randomness

For Problem 3, we can add randomness to the initial position of both the Agent and the Goal. This ensures that our training environment is more robust to different situations, and helps avoiding overfitting. To do that, we can simply change the OnEpisodeBegin function:

```
public override void OnEpisodeBegin()
{
    transform.localPosition = new Vector3(Random.Range(-5f, 5f), 0, Random.Range(0, 6f));
    targetTransform.localPosition = new Vector3(Random.Range(-4f, 4f), 0, Random.Range(1, 5f));
}
```

Now if we run training again, we can see that both the Agent and the goal are being initialized in random positions, leading to a more robust environment:



4 - Adjusting Penalties

For Problem 4, we can adjust the penalties of touching the wall. Instead of giving the Agent the same penalty no matter where it touches the wall, we could base it on the distance from that point of the wall to the Goal, and scale it accordingly:

```
public void OnTriggerEnter(Collider other)
{
    if (other.TryGetComponent<Goal>(out Goal goal))
    {
        SetReward(+1f); // Positive reward for reaching the goal
        floorMeshRenderer.material = winMaterial;
        EndEpisode();
    }
    if (other.TryGetComponent<Wall>(out Wall wall))
    {
        // Calculate the penalty based on the distance to the Goal
        float distanceToGoal = Vector3.Distance(transform.localPosition,
targetTransform.localPosition);
        float penalty = Mathf.Clamp(1f - (1f / (distanceToGoal + 1f)), 0.1f, 1f);
        // Clamp ensures the penalty has a minimum value (e.g., 0.1) and doesn't go
too high.

        SetReward(-penalty); // Apply scaled penalty
        floorMeshRenderer.material = loseMaterial;
        EndEpisode();
    }
}
```

5 - Step Penalty

For Problem 5, to penalize the agent for taking too long to reach the goal, we can introduce a time step penalty. This is typically done in the OnActionReceived or similar method that runs every time the agent performs an action:

```
public override void OnActionReceived(ActionBuffers actions)
{
    // Apply a small negative reward for each step
    AddReward(-1f / MaxStep); // Small penalty for each step

    float moveX = actions.ContinuousActions[0];
    float moveZ = actions.ContinuousActions[1];
    float movespeed = 3f;
    transform.localPosition += new Vector3(moveX, 0, moveZ) * Time.deltaTime *
movespeed;
}
```

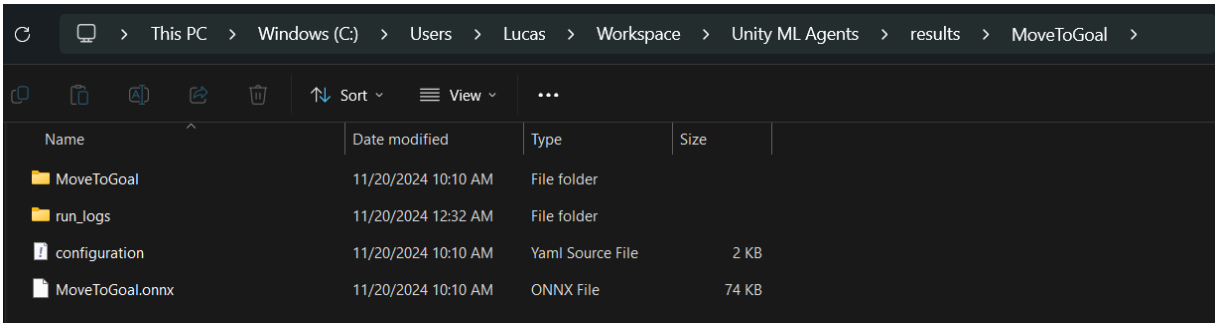
Results

Using a Trained Model

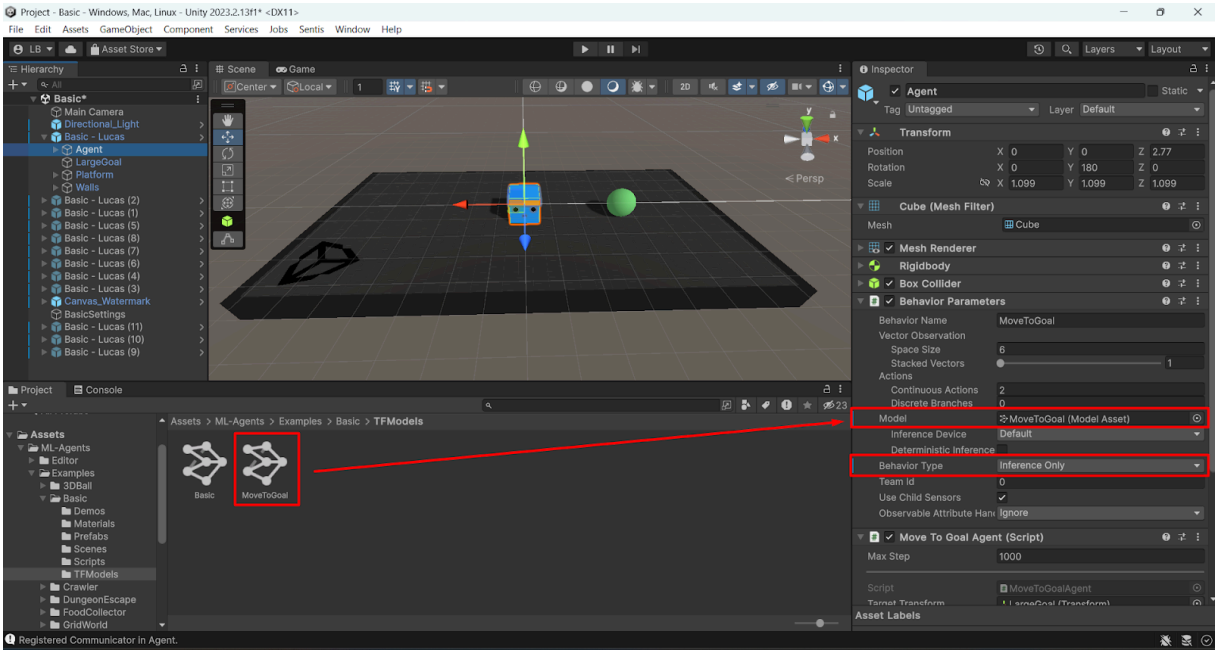
After finishing training, the trained model is saved in a default results folder, with the name of your run:

- *C:\Users\Lucas\Workspace\Unity ML Agents\results\MoveToGoal*

There you will find an .onnx file that represents the trained brain model:



To use it, copy the trained model and paste it on your Assets folder. Then, you can drag it to the Model parameter on the Behavior Parameters component. This will make sure that the Agent is using that particular model, even without running the python script on command prompt:



To test it, you can run the simulation by changing Behavior Type to “Inference Only” and pressing play.

Using Config File

Another way to make training more effective is to tune the hyperparameters[18] of the brain model. To do that, we need to create a “config” folder and place a .yaml configuration file inside, specifying the parameters of the model[19].
Let’s make a new folder on our project named “config” and create a “moveToGoal.yaml” configuration file inside. Make sure to change the second line to the name of your model:

```
behaviors:
  MoveToGoal:
    trainer_type: ppo
    hyperparameters:
      batch_size: 10
      buffer_size: 100
      learning_rate: 3.0e-4
      beta: 5.0e-4
      epsilon: 0.2
      lambd: 0.99
      num_epoch: 3
      learning_rate_schedule: linear
      beta_schedule: constant
      epsilon_schedule: linear
    network_settings:
```

[18] For more information on how to do that, check out the official documentation on this link.
[19] For more information on the Training Configuration File, check out this link.

```
    normalize: false
    hidden_units: 128
    num_layers: 2
  reward_signals:
    extrinsic:
      gamma: 0.99
      strength: 1.0
  max_steps: 500000
  time_horizon: 64
  summary_freq: 10000
```

To train a model using custom parameters, you need to add the config file path when starting the run:

- Enter your environment where you installed the mlagents package;
- Run "*mlagents-learn config/moveToGoal.yaml --run-id=MoveToGoal*" (use "*--resume"* if you want to run the same id after);
- Press play on the Unity Engine.

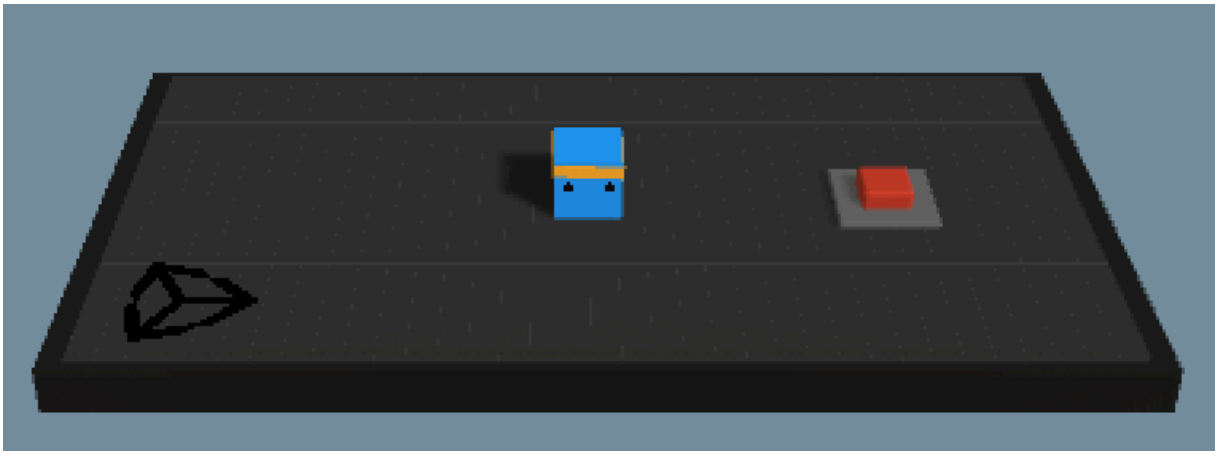
Initialize From Model

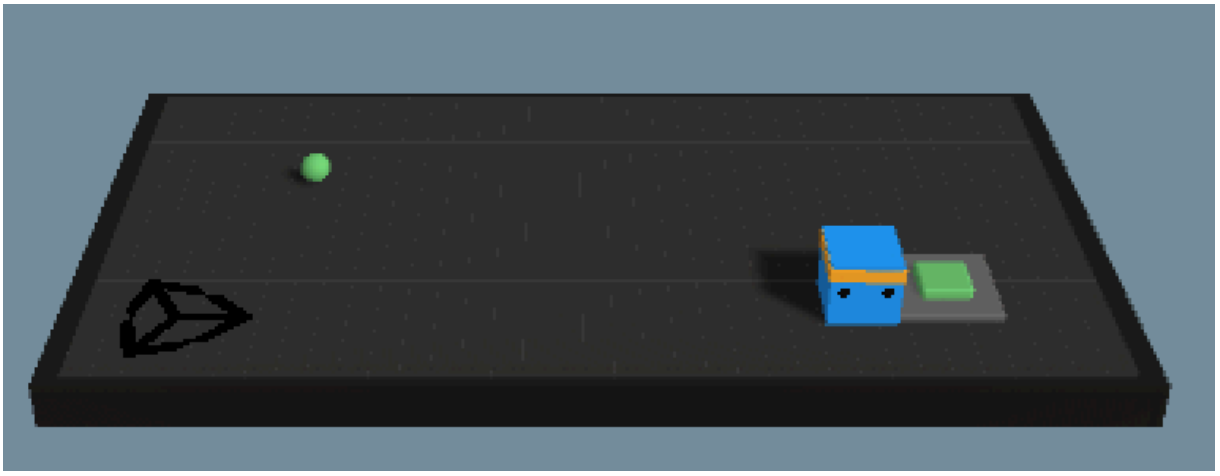
To learn from a previously trained brain, and improve on it, you can use the command "*--intialize-from=*" followed by the run-id of the model that you want to use. To make sure that it is a new run (even if it started from another run), we will also include a new run-id. In our case, we can run:

- "*mlagents-learn config/moveToGoal.yaml --initialize-from=MoveToGoal --run-id=MoveToGoal2*"

Imitation Learning

The goal of this section is to make a simple Agent (blue cube) and teach it to reach a Button, press it, and then get the food that spawns as a result.[20] It is built on top of the last scenario (Scene: Basic) with an added Button and Food assets, as can be seem below:





[20] For this section, I will be following the tutorial
How to use Machine Learning AI in Unity! (ML-Agents) .

Scenario

This scenario builds upon the previous one (Scene: Basic) by introducing a more complex, multi-step task. The Agent must first navigate to a button, press it to spawn food on another side of the environment, and then move toward the food to collect it. Unlike the straightforward task of reaching a single goal, this scenario requires the Agent to sequence multiple actions correctly, making it significantly more challenging.

Completing this task through classic Reinforcement Learning alone can be inefficient and slow. The Agent would need to stumble upon the right sequence of actions purely by chance: moving to the button, pressing it, and then locating and collecting the food. This random exploration approach would likely require a vast number of episodes before the Agent even begins to learn the basic structure of the task, let alone optimize its performance.

Imitation Learning offers a more efficient solution for such scenarios. By learning from recorded demonstrations of a player successfully completing the task, the Agent can bypass the need for random exploration to discover the first rewards. It learns the fundamental sequence of actions much faster. Once the Agent acquires these basic behaviors through Imitation Learning, it can transition to Reinforcement Learning to further refine its actions, ensuring it generalizes its policy rather than merely replicating the demonstrated behavior.

Designing the Agent

Actions

In this scenario, the Agent must complete a multi-step task: moving toward a button, pressing it to spawn food, and then navigating to collect the food. To achieve this, the Agent has access to three discrete action branches: one for horizontal movement (left, right, or no movement), one for vertical movement (forward, backward, or no movement), and one for interacting with the button (press or don't press). The complexity arises because the Agent must correctly sequence its actions to complete the task, making random exploration inefficient.

Actions:

- **Move X**: Discrete, 1 Branch of size 3 (don't move, left, right).
- **Move Z**: Discrete, 1 Branch of size 3 (don't move, back, forward).
- **Press Button**: Discrete, 1 Branch of size 2 (don't press, press).

Observations

The observations collected by the Agent are designed to guide it through the task. These include whether the button is usable, the direction to the button as normalized X and Z coordinates, whether food is present in the scene, and the direction to the food if it exists (also as normalized X and Z coordinates). If no food is spawned, the Agent observes zeros for the food-related information.

Observations:

- **Food Button**: Boolean indicating whether the button is usable.
- **Direction to Button**: Normalized X and Z coordinates pointing toward the button.
- **Food Spawned**: Boolean indicating whether food is present in the scene.
- **Direction to Food**: Normalized X and Z coordinates pointing toward the food (or zeros if no food is spawned).

Rewards

Rewards are structured to encourage efficient completion of the task. The Agent receives a positive reward for pressing the button successfully and for collecting the food. To discourage inefficiency, the Agent is penalized incrementally for each step it takes and receives a larger penalty for colliding with walls or going out of bounds.

Rewards:

- **Positive**:
  - +1 for pressing the button successfully.
  - +1 for collecting the food.
- **Negative**:
  - -1 for colliding with walls or going out of bounds.
  - -1 divided by MaxStep for every step taken to encourage efficiency.

End of Episode

The episode ends when the food is collected, the Agent collides with a wall, or the maximum number of steps is reached.

Episode Ends:

- Collecting the food.
- Colliding with a wall.
- Reaching the Max Steps limit.

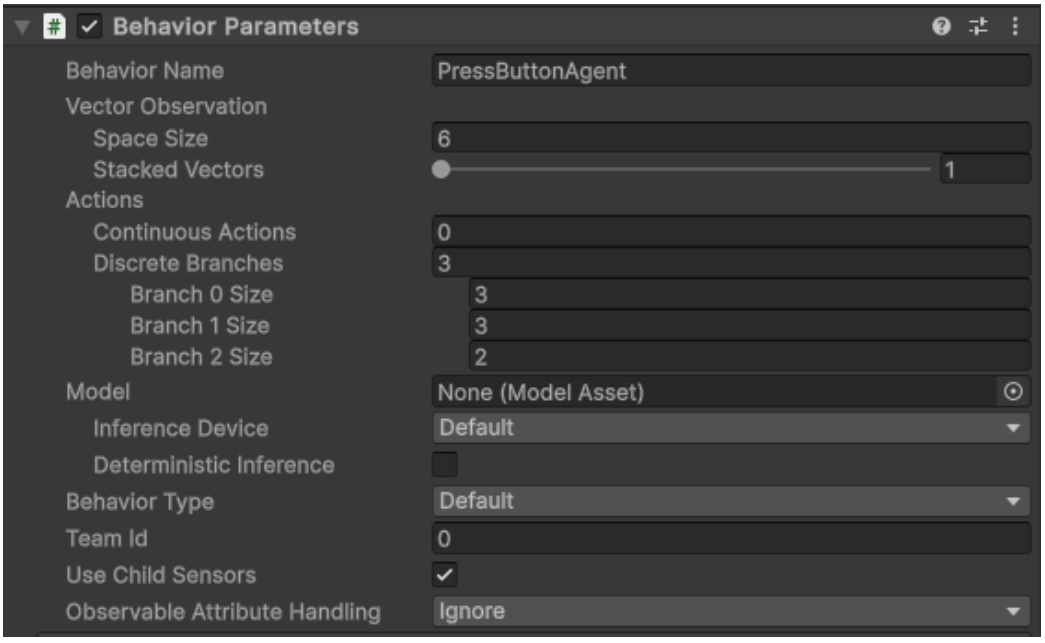
Scripts

For the Scripts, I won't go into as much detail as last section. Instead, I will just include a short description of each attribute and method included in the created classes. The complete code can be found in my GitHub, along with the assets used and other relevant files.

Agent

The **Agent** is responsible for learning and executing the sequence of actions required to complete the task: moving to the **FoodButton**, pressing it to spawn food, and collecting the spawned food. It interacts with the environment by observing the button's usability, the food's presence, and their respective directions, making decisions based on these observations. The agent's behavior is guided by **Imitation Learning** to learn the initial actions and subsequently reinforced by **Reinforcement Learning** to refine its strategy. It uses events like *OnEpisodeBeginEvent* to notify related objects of resets and *OnAteFood* to signal food collection.

Behavior Parameters:



Agent Script

Attributes:

- **foodSpawner:** Reference to the **FoodSpawner** object, enabling interaction with spawned food.
- **foodButton:** Reference to the **FoodButton** object, allowing the agent to use the button.
- **agentRigidBody:** Used for applying movement forces to the agent.

Methods:

- **OnEpisodeBegin():** Spawns the Agent in random positions at the start of the Episode. Invokes the *OnEpisodeBeginEvent*[21], allowing other scripts to do some setup of their own on the start of the Episode.
- **CollectObservations():** Collects observations about the FoodButton's usability, direction to the button, whether food is spawned, and direction to the food if present.
- **OnActionReceived():** Handles movement based on discrete actions, applies forces for navigation, and interacts with the FoodButton if in range, rewarding the Agent for successful use and penalizing it for each step.
- **OnCollisionEnter():** Detects collisions with food, rewarding the Agent and ending the episode, or with walls, penalizing the Agent and ending the episode.
- **Heuristic():** Maps keyboard input to discrete actions, allowing manual control for debugging or testing Agent behavior.

[21] For more information on Events, check out the ML-Agents -> Glossary section.

FoodButton

The **FoodButton** acts as the trigger for spawning food in the environment. It changes state when pressed, indicated visually through material and scale changes. It resets its position and state at the start of each episode, ensuring the agent must navigate to and interact with it afresh. The button emits an *OnUsed* event when pressed, notifying the **FoodSpawner** to spawn food, and listens for the agent's *OnEpisodeBeginEvent* to reset.

Food Button Script

Attributes:

- **foodSpawner:** Reference to the **FoodSpawner** to notify it to spawn food when the button is used.
- **pressedButtonMaterial:** Material used to visually indicate the button has been pressed.
- **startMaterial:** Material for the button's default state.
- **agent:** Reference to the **Agent** that interacts with the button.
- **buttonMeshRenderer:** Renderer controlling the button's visual appearance.
- **buttonTransform:** Transform for manipulating the button's scale and position.

Methods:

- **Awake():** Initializes references to the button's transform, mesh renderer, and state, setting *canUseButton* to *true*.
- **Start():** Resets the button's state and subscribes to the *OnEpisodeBeginEvent* of the associated Agent, ensuring the button resets at the start of each episode.
- **HandleEpisodeBegin():** Resets the button's appearance and state when an episode begins.
- **OnDestroy():** Unsubscribes from the Agent's *OnEpisodeBeginEvent* to prevent memory leaks.
- **CanUseButton():** Returns whether the button is currently usable.
- **UseButton():** Changes the button's appearance and state when pressed and invokes the *OnUsed* event, notifying other components.
- **ResetButton():** Restores the button to its original state and relocates it to a random position within a defined range.

FoodSpawner

The **FoodSpawner** manages the spawning of food objects when the **FoodButton** is pressed. It listens for the button's *OnUsed* event to trigger food spawning and ensures only one food object exists at a time. If the food is collected or destroyed, the spawner clears its reference to the last spawned food, allowing new food to be spawned.

Food Spawner Script

Attributes:

- **foodPrefab:** Prefab used to instantiate food objects in the environment.
- **foodButton:** Reference to the **FoodButton**, which triggers food spawning.
- **lastSpawnedFood:** Reference to the most recently spawned food object. This is used to track its existence and position.

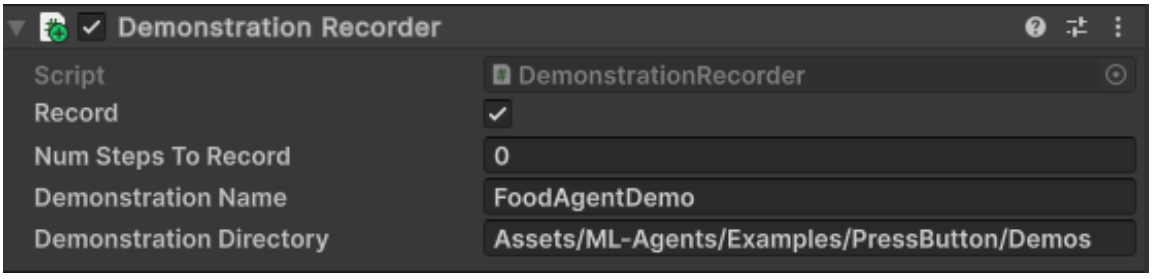
The script implements the following functions:

- **Start():** Subscribes to the *OnUsed* event of the associated FoodButton to trigger food spawning when the button is pressed.
- **HandleButtonUsed():** Spawns food in response to the button's *OnUsed* event.
- **OnDestroy():** Unsubscribes from the *OnUsed* event to prevent memory leaks.
- **HasFoodSpawned():** Returns *true* if food is currently spawned, otherwise *false*.
- **GetLastFoodTransform():** Retrieves the transform of the last spawned food or *null* if no food is spawned.
- **SpawnFood():** Instantiates a food prefab at a random position within a defined range, ensuring only one piece of food is spawned at a time.
- **DestroyFood():** Removes the last spawned food and clears its reference.

Demonstration

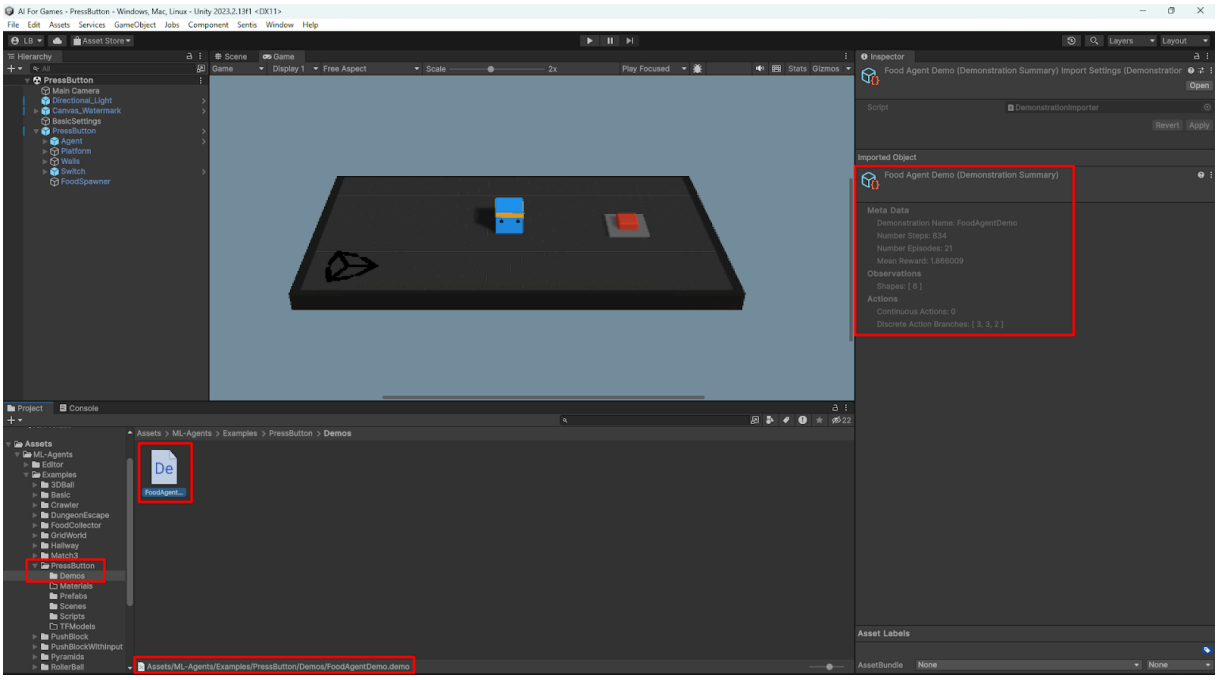
Recording demonstrations is a critical step in enabling Imitation Learning, particularly in complex scenarios where initial random exploration by the Agent is inefficient or unlikely to yield results. By capturing a series of successful actions performed by a player or heuristic, the Agent gains valuable data to emulate and learn from, effectively bypassing the need to randomly discover rewarding behavior. This significantly accelerates the learning process by providing a foundation upon which the Agent can refine its strategy using Reinforcement Learning.

To record a demonstration, add the **Demonstration Recorder** component to your Agent's GameObject. Configure the "Num Steps to Record" parameter to limit the recording duration, or set it to 0 for unlimited recording. Set the "Demonstration Name" to something descriptive, like **FoodAgentDemo**, and optionally update the "Demonstration Directory" to define where the file will be saved. By default, demonstrations are stored in *{Application.dataPath}/Demonstrations*.



Once configured, toggle the "Record" parameter to enable recording. Run the scene with the Agent's **Behavior Type** set to **Heuristic Only** in the Behavior Parameters. This allows the Agent to act based on predefined logic or player inputs. Stop the recording by exiting Play Mode.

The demonstration will be saved as *{Demonstration Directory}/{Demonstration Name}.demo*, which can later be used for training. The demonstration file contains valuable metadata, such as the total number of steps, which can be inspected in Unity's Inspector panel.



Best Practices for Recording Demonstrations:

- **Demonstrate Optimal Behavior:** Ensure that the recorded actions represent the desired strategy or behavior the Agent should learn. Poor or inconsistent demonstrations can mislead the Agent during training.
- **Record Multiple Sessions:** Provide varied demonstrations to expose the Agent to different scenarios and reduce overfitting to a specific path or strategy.
- **Limit Recording Length:** Avoid excessively long demonstrations as these can include redundant or non-essential data that might confuse the Agent during training.
- **Verify Recorded Demonstrations:** Review the metadata and validate the demonstration file to ensure it accurately reflects the intended behavior before using it for training.
- **Use Consistent Settings:** Ensure the environment configuration matches the training setup to avoid discrepancies that could hinder learning.

Training

We'll use three training methods to teach the Agent: **extrinsic rewards**, **Generative Adversarial Imitation Learning (GAIL)**, and **Behavioral Cloning (BC)**. Each method has a role to play at different stages of training. Early on, we'll rely on BC and GAIL to guide the Agent with pre-recorded demonstrations, helping it quickly learn the basic actions. Once the Agent knows how to complete the task, we'll shift focus to extrinsic rewards so it can refine its behavior on its own and adapt to new situations.

In ML-Agents, rewards can come from different sources, like extrinsic rewards or signals from GAIL or curiosity modules. These sources work together, with their importance determined by a strength parameter. For example, at first, GAIL and BC will have more influence, helping the Agent follow the demonstrations. Later, extrinsic rewards will take over as the main driver of learning, encouraging exploration and improving performance.

Training happens in phases. First, we'll focus on teaching the Agent to complete the task using imitation learning. During this phase, we'll turn off the time penalty for taking extra steps and set extrinsic rewards to a lower strength. Once the Agent is comfortable with the basics, we'll reduce the influence of imitation learning and increase the strength of extrinsic rewards. At this point, we'll also introduce a time penalty to motivate the Agent to complete tasks faster.

Generative Adversarial Imitation Learning (GAIL)

GAIL is like teaching the Agent by showing it how a task is done but with a twist. Instead of directly copying, the Agent learns to make its actions look as if they came from the demonstration, even if it's finding its own way to solve the problem. This balance helps the Agent generalize its learning to situations outside the demo.

In our setup, GAIL uses the demonstration file *FoodAgentDemo.demo* and starts with a strength of *0.5*. This helps the Agent mimic the demo early on while leaving room for it to learn on its own later.

Behavioral Cloning (BC)

Behavioral Cloning is a straightforward way of teaching by example. The Agent watches a demonstration and copies the actions step-by-step. This method is great for getting the Agent to learn tasks quickly, especially when exploring randomly would take forever to find the right behavior.

For this scenario, BC also uses *FoodAgentDemo.demo* with a strength of *0.5*. Early on, the Agent will heavily rely on this method to figure out how to press the button and get the food. As training progresses, we'll reduce BC's influence to encourage the Agent to explore and fine-tune its behavior.

Training Configuration File

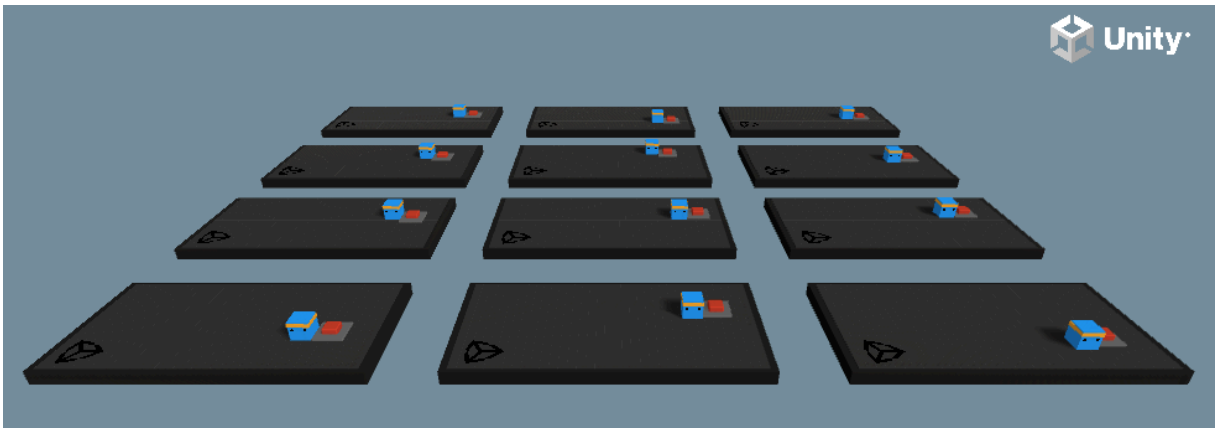
The training settings[22] for our Agent are outlined in the *PressButton.yaml* file. We're using the PPO trainer with balanced hyperparameters, such as a batch size of *256*, buffer size of *1024*, and a learning rate of *3.0e-4*. These values provide a stable and efficient foundation for training.

Initially, **extrinsic rewards** will have a strength of *0.1*, while **GAIL** and **Behavioral Cloning (BC)** will contribute equally with strengths of *0.5* each. This setup allows the Agent to focus on imitating the demonstration early on, ensuring it learns the basic actions needed to complete the task. The lower strength for extrinsic rewards prevents the Agent from prematurely overemphasizing exploration before it understands the fundamentals. These strength parameters for the first run were decided after a few tests:

[22] For more information on the configuration file, check out the ML-Agents -> Training ML-Agents section.

Performance Issues

While trying the some runs, I ended up in this situation:



The Agents started to just “hug” the Button without pressing it. To encourage the Agent to press the Button while penalizing it for staying near it, I have added the following changes to the Agent code:

- Increased reward for pressing the Button (+2f);
- Penalty for staying near the Button without pressing it (-0.001f per step);
- Reward for running towards the Food (+0.005f per step).

Training configuration file:

```yaml
behaviors:
  PressButtonAgent:
    trainer_type: ppo

    hyperparameters:
      batch_size: 256
      buffer_size: 1024
      learning_rate: 3.0e-4
      learning_rate_schedule: linear

      # PPO-specific hyperparameters
      beta: 5.0e-4
      epsilon: 0.2
      lambd: 0.99
      num_epoch: 3

    # Configuration of the neural network
    network_settings:
      normalize: false
      hidden_units: 128
      num_layers: 2

    # Trainer configurations common to all trainers
    max_steps: 50000
    time_horizon: 64
    summary_freq: 2000
    keep_checkpoints: 5
    checkpoint_interval: 10000

    reward_signals:
      # environment reward (default)
      extrinsic:
        strength: 0.1
        gamma: 0.99
      # GAIL
      gail:
        demo_path: Demos/FoodAgentDemo.demo
        strength: 0.5

    # behavior cloning
    behavioral_cloning:
      demo_path: Demos/FoodAgentDemo.demo
      strength: 0.5
```

Results

Training was conducted across 12 parallel environments to accelerate the learning process. The model was initially trained from scratch in the first run, and subsequent runs continued from where the previous run left off, leveraging the learned policy and building on it.

Run 1

Command: mlagents-learn config/ppo/PressButton_run1.yaml
--run-id=PressButtonAgent_ImitationLearning
Strength parameters: extrinsic=0.1, behavioral_cloning=0.5, gail=0.5
Time penalty: None
Steps trained: 40,000

This run focused on imitation learning, prioritizing behavioral cloning and GAIL to ensure the Agent learned the basic actions required to complete the task effectively. The extrinsic reward was set low to avoid overwhelming the imitation process early on.

Run 2

Command: mlagents-learn config/ppo/PressButton_run2.yaml
--run-id=PressButtonAgent_ImitationLearning2
--initialize-from=PressButtonAgent_ImitationLearning
Strength parameters: extrinsic=1.0, behavioral_cloning=025, gail=0.25
Time penalty: Enabled
Steps trained: 20,000

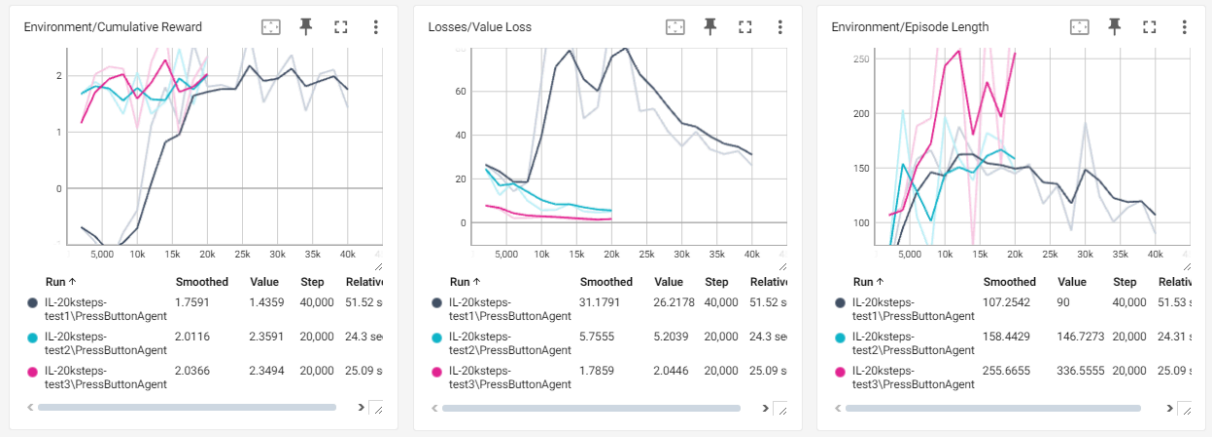
In this phase, the focus shifted toward reinforcement learning, increasing the weight of extrinsic rewards and decreasing reliance on GAIL and behavioral cloning. Adding a time penalty encouraged the Agent to complete the task more efficiently, rewarding speed and precision.

Run 3

Command: mlagents-learn config/ppo/PressButton_run3.yaml
--run-id=PressButtonAgent_ImitationLearning3
--initialize-from=PressButtonAgent_ImitationLearning2
Strength parameters: extrinsic=1.0, behavioral_cloning=0.1, gail=0.1
Time penalty: None
Steps trained: 20,000

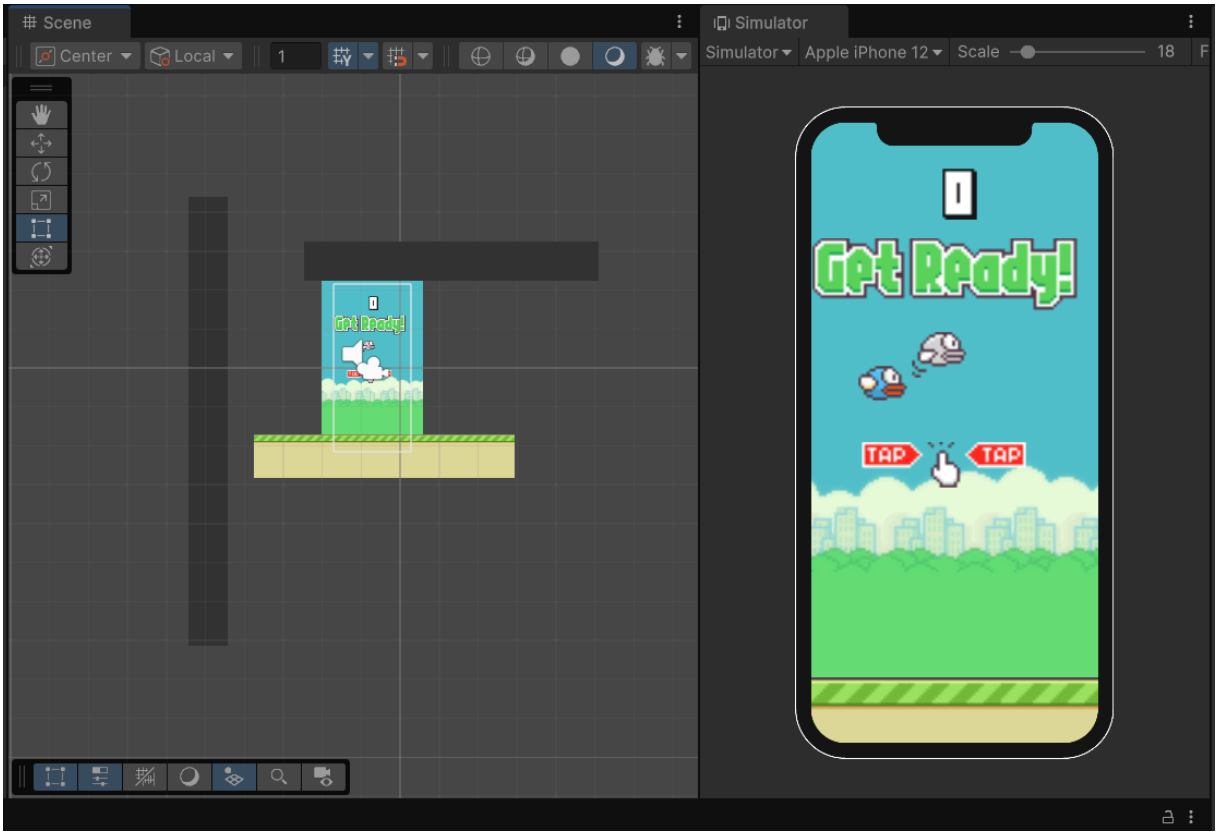
The final phase emphasized independent learning, with extrinsic rewards dominating while imitation-based signals played a minor supporting role. The Agent refined its policy to achieve optimal performance and adapt to more nuanced scenarios without overfitting to the demonstration data.

Final Results



Flappy Bird

In this section, instead of working with example environments, we will implement RL in an actual game. We will use a clone of Flappy Bird[23] made in Unity, created by Dimitris Gkanatsios and available in this Github repository.



[23] For this section, I will be following the tutorial AI Learns to play Flappy Bird!.

Scenario

The scenario is as simple as a game can get. In case you never played or heard about Flappy Bird (yeah right…), the game involves a small bird that continuously moves forward through a series of pipes with gaps. The player controls the bird's vertical movement by making it "flap" to ascend, while gravity pulls it back down.

The challenge is to navigate the bird through the gaps in the pipes without hitting the pipes or the ground, which causes the game to end. Players earn points by passing through the pipes, and the game increases in difficulty as it progresses.

Designing the Agent

Actions

The Agent is equipped with a single discrete action branch to mimic the bird's ability to jump or remain still. The simplicity of the action space reflects the game's core mechanic but requires the Agent to carefully time its jumps to avoid obstacles and maintain a steady flight path.

- **Jump**: Discrete, 1 branch of size 2 (jump, don't jump).

Observations

The Agent collects observations that provide it with critical information about its surroundings and state. These observations guide its decision-making process as it navigates through the pipes and avoids obstacles.

- **Ray Perception Sensor 2D**: Detects checkpoints (gaps in pipes) and walls.
- **Height**: Normalized Y position of the bird, indicating its vertical position relative to the scene.
- **Distance to the Next Pipe**: A normalized value representing how far the bird is from the next pipe obstacle.
- **Velocity**: A normalized value indicating the bird's current velocity.

Rewards

The reward system is designed to encourage the Agent to successfully navigate the obstacles, pass through gaps, and avoid penalties. Positive rewards are given for achieving milestones like passing checkpoints, while penalties discourage undesirable actions like hitting obstacles or the floor.

Positive Rewards:

- **+1** for passing a checkpoint (pipe gap).

Negative Rewards:

- **-1** for touching the ground.
- **-1** for colliding with pipes.

Episode Ends

The episode concludes under the following conditions:

- **Game Over**: The Agent touches the ground or collides with a pipe.
- **Max Steps**: The Agent reaches the maximum number of steps allowed in a single episode without completing any further checkpoints.

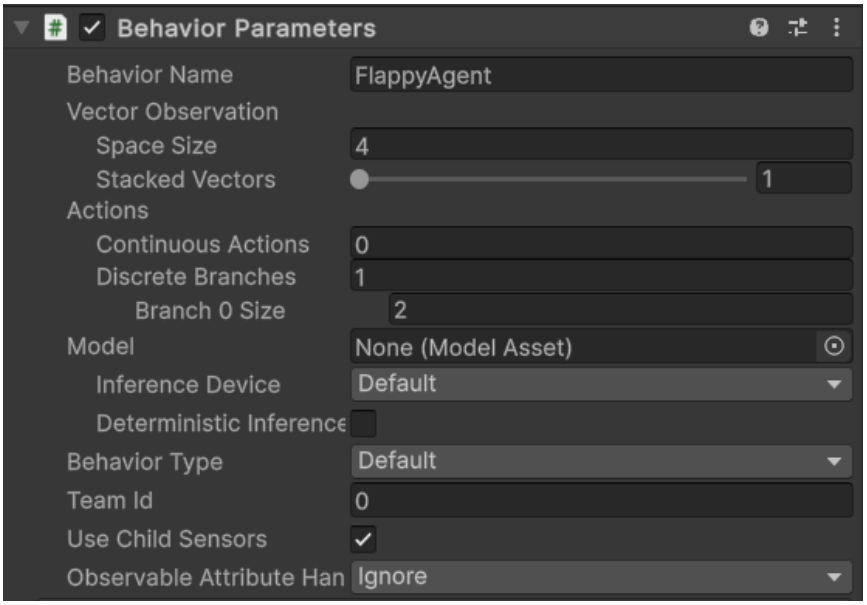
Creating the Agent

Unlike traditional ML-Agents environments where the game logic is designed with AI integration in mind, the Flappy Bird project was originally created as a self-contained game. The challenge was to maintain its original behavior while enabling it to be controlled by an AI Agent. Instead of rewriting the game logic, I used a modular approach. The FlappyAgent script was written separately and attached to the existing FlappyBird GameObject, allowing it to interact with the original script through public methods. This approach preserved the ability to play the game as usual, while also allowing AI control.

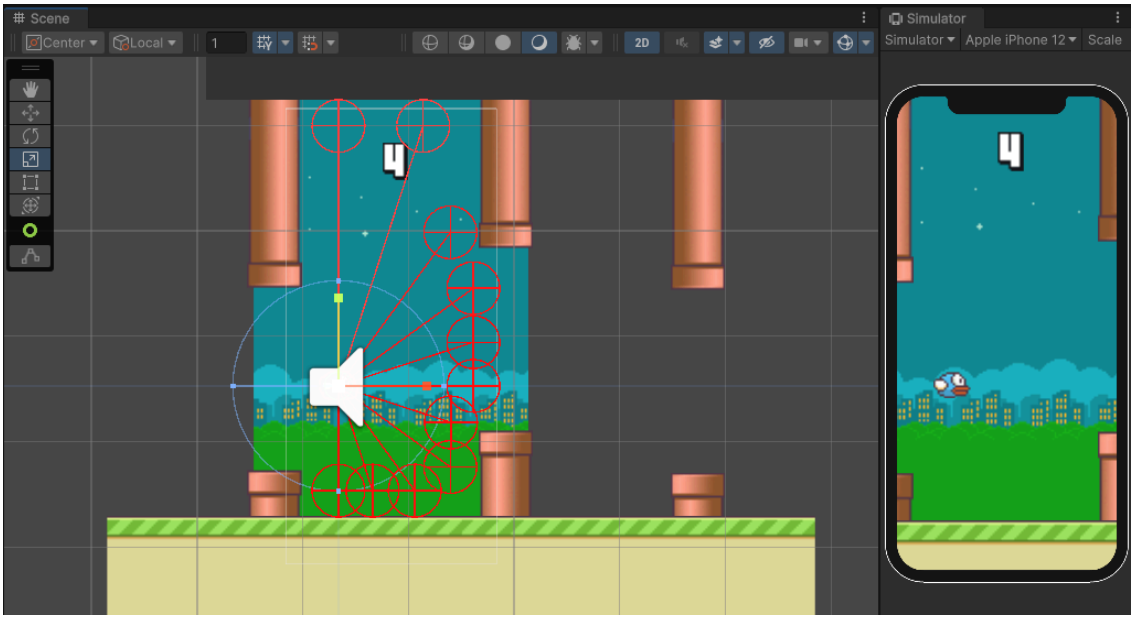
To make the **FlappyBird** script compatible with the ML-Agents framework, I introduced minor adjustments that did not alter its core behavior. The following changes were made:

- **Public API**: Added public methods to allow external scripts (like **FlappyAgent**) to access important functionalities, such as triggering jumps or resetting the game.
- **Event Handlers**: Modified collision logic to trigger an event whenever the bird hits an obstacle or the ground. This event is used by **FlappyAgent** to detect and respond to collisions.

Behavior Parameters:



Ray Perception Sensor 2D



To enable the Agent to observe its environment, a **Ray Perception Sensor 2D** was added to the **Flappy Bird** GameObject, allowing it to detect obstacles, pipe gaps, and ground contact. To make this sensor functional, several adjustments were made. Objects in the scene, such as the floor and ceiling, were tagged as **Wall**, while pipes were tagged as **Pipe**. Detected objects were placed on separate layers to allow the sensor to differentiate between them, and a **Layer Mask** was configured to ensure it only detected objects on the specified layers.

Agent Script

Attributes:

- **flappy**: Reference to the **FlappyScript** component, enabling interaction with the Flappy Bird character.

Methods:

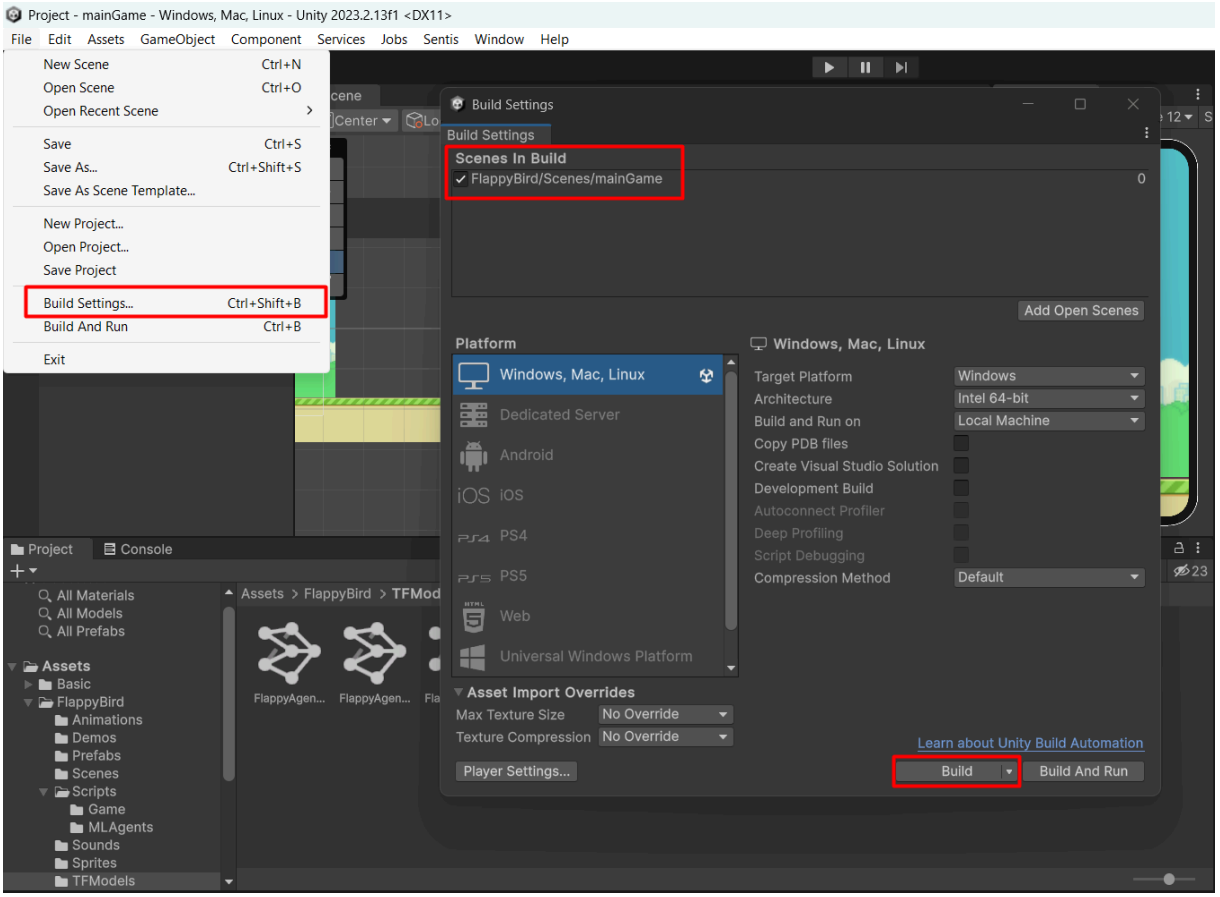
- **Initialize()**: Initializes the FlappyAgent by linking it to the **FlappyScript** component and subscribing to the collision event.
- **HandleFlappyCollision()**: Listens for collisions with pipes or walls, rewarding the Agent for passing a checkpoint and penalizing it for hitting obstacles.
- **CollectObservations()**: Collects observations on the bird's height, velocity, and distance to the next pipe, normalizing the values for input to the Agent.
- **OnActionReceived()**: Executes the jump action if the Agent decides to jump, based on the action buffer.
- **Heuristic()**: Maps player input to Agent actions, allowing manual control for testing.
- **GetDistanceToNextPipe():** Calculates and returns the distance to the nearest upcoming pipe.

Training

Training was conducted using **Imitation Learning** across multiple runs to progressively increase the complexity of the task. Each run focused on distinct learning strategies, starting with Imitation Learning and gradually shifting toward **Reinforcement Learning** as the Agent's skills improved. Since the original Flappy Bird game was not designed as an ML-Agents environment, using multiple training environments was not possible. Instead, multiple normal builds were run in parallel to accelerate training.

To simulate human behavior with the Jump buttom, I added a cooldown of 0.5 seconds between jumps. This helped the agent get through the first run faster, and also aligns with how it should play the game. I also made the GameState always be in the Playing state during training, to avoid interferences with the Agent decisions. Finally, I reduced the jump strength on y axis, to give the Agent more freedom of movement.

Running multiple envs with normal build



Make sure the game requires no input to start (starts from Playing state). Make sure that the Agent's Behavior Type is set to "Default", the Demonstration Recorder is not recording, and no Model is selected. Then, go to File -> Build Settings, make sure that the correct scenes are selected and press build.

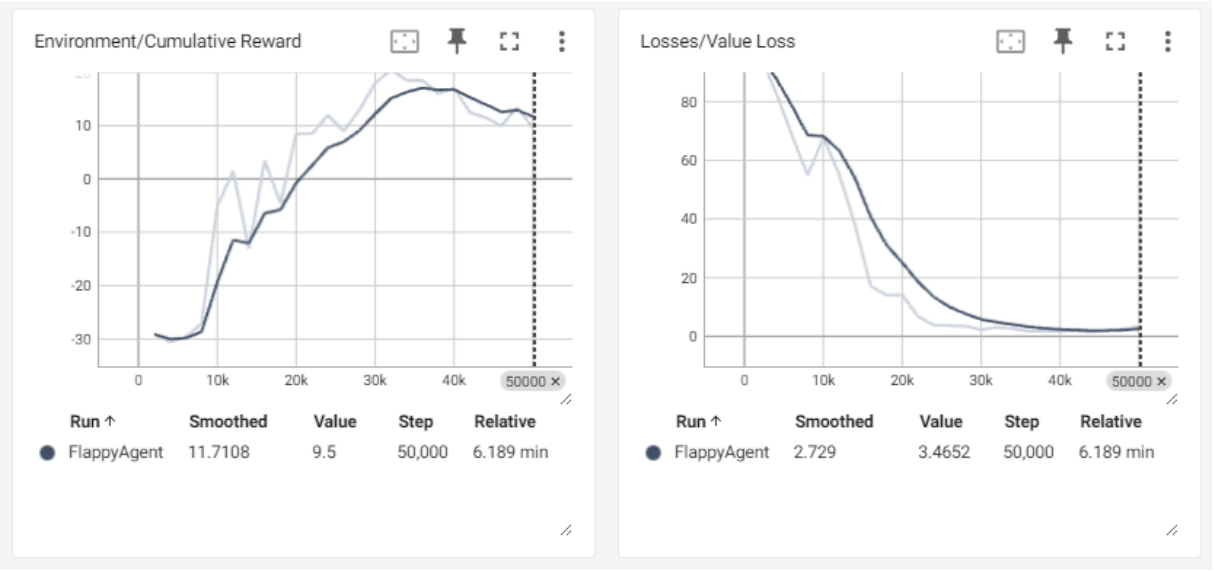
To run training with this build, you add the --env flag with the path to your build, and --num-envs with the number of concurrent environments to run.

Example Command:

```
mlagents-learn config/ppo/FlappyBird_run3.yaml
--run-id=FlappyAgent_ImitationLearning4
--initialize-from=FlappyAgent_ImitationLearning3
--env=_Builds/FlappyBirdLevel3/UnityEnvironment
--num-envs=8
```

Results

Run 1



Command: mlagents-learn config/ppo/FlappyBird_run1.yaml --run-id=FlappyAgent_ImitationLearning
Strength parameters: extrinsic=0.1, behavioral_cloning=1.0, gail=0.5
Environment Adjustments:

- **Pipe Position**: Fixed vertical position and fixed distance between pipes.
- **Learning Focus**: Heavy use of Imitation Learning through pre-recorded demonstrations.
- **Agent Behavior**: The walls do not kill the bird. Pipes don't kill, but still give negative rewards.
- **Steps trained**: 50,000

This initial run focused on familiarizing the Agent with the task using Imitation Learning as the primary driver. By maintaining fixed pipe positions and distances, the Agent could develop a basic understanding of how to navigate through gaps, leveraging behavioral cloning to copy the demonstrated actions. Extrinsic rewards were kept low to avoid interference with Imitation Learning.

Training configuration file:

```yaml
behaviors:
  FlappyAgent:
    trainer_type: ppo

    hyperparameters:
      batch_size: 256
      buffer_size: 1024
      learning_rate: 3.0e-4
      learning_rate_schedule: linear

      # PPO-specific hyperparameters
      beta: 5.0e-4
      epsilon: 0.2
      lambd: 0.99
      num_epoch: 3

    # Configuration of the neural network
    network_settings:
      normalize: false
      hidden_units: 128
      num_layers: 2

    # Trainer configurations common to all trainers
    max_steps: 50000
    time_horizon: 200
    summary_freq: 2000
    keep_checkpoints: 5
    checkpoint_interval: 10000

    reward_signals:
      # environment reward (default)
      extrinsic:
        strength: 0.1
        gamma: 0.99
      # GAIL
      gail:
        demo_path: Demos/Leve1FlappyAgentDemo.demo
        strength: 0.5

    # behavior cloning
    behavioral_cloning:
      demo_path: Demos/Leve1FlappyAgentDemo.demo
      strength: 1.0
```

Run 2

Command: mlagents-learn config/ppo/FlappyBird_run2.yaml --run-id=FlappyAgent_ImitationLearning2 --initialize-from=FlappyAgent_ImitationLearning
Strength parameters: extrinsic=1.0, behavioral_cloning=0.4, gail=0.4
Environment Adjustments:

- **Pipe Position**: Random heights and distances.
- **Learning Focus**: Shift from Imitation Learning to a balance between imitation and reinforcement.
- **Agent Behavior**: The walls do not kill the bird. Pipes don't kill, but still give negative rewards.
- **Steps trained**: 50,000

In this phase, randomness was introduced to the pipe heights and distances, increasing the difficulty of the task. The Agent could no longer rely solely on fixed patterns learned during Run 1. Instead, a more balanced approach was used by increasing extrinsic rewards and decreasing the influence of Imitation Learning.

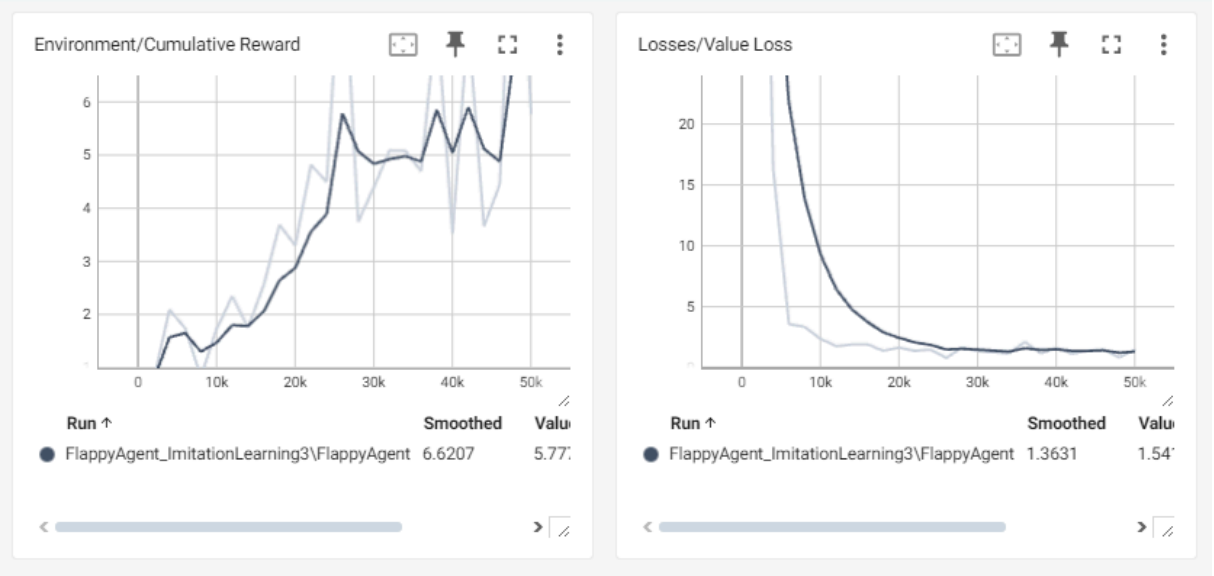
Training configuration file:

```
behaviors:
  FlappyAgent:
    # Same as before

    reward_signals:
      # environment reward (default)
      extrinsic:
        strength: 1.0
        gamma: 0.99
      # GAIL
      gail:
        demo_path: Demos/Leve2FlappyAgentDemo.demo
        strength: 0.4

    # behavior cloning
    behavioral_cloning:
      demo_path: Demos/Leve2FlappyAgentDemo.demo
      strength: 0.4
```

Run 3



Command: mlagents-learn config/ppo/FlappyBird_run3.yaml --run-id=FlappyAgent_ImitationLearning3 --initialize-from=FlappyAgent_ImitationLearning2
Strength parameters: extrinsic=1.0, behavioral_cloning=0.1, gail=0.1
Environment Adjustments:

- **Pipe Position**: Random heights and distances.
- **Learning Focus**: Full shift toward Reinforcement Learning.
- **Agent Behavior**: Walls kill the bird on collision. Pipes kill.
- **Steps trained**: 50,000

The final phase emphasized **Reinforcement Learning** as the primary learning method. Behavioral cloning and GAIL were significantly reduced, allowing the Agent to refine its strategy through exploration. By this stage, the Agent had to generalize its knowledge to solve more dynamic challenges, like variable pipe positions and distances, while also being careful with pipes and walls. This approach ensured the Agent could adapt to unseen pipe arrangements without relying on demonstration data.

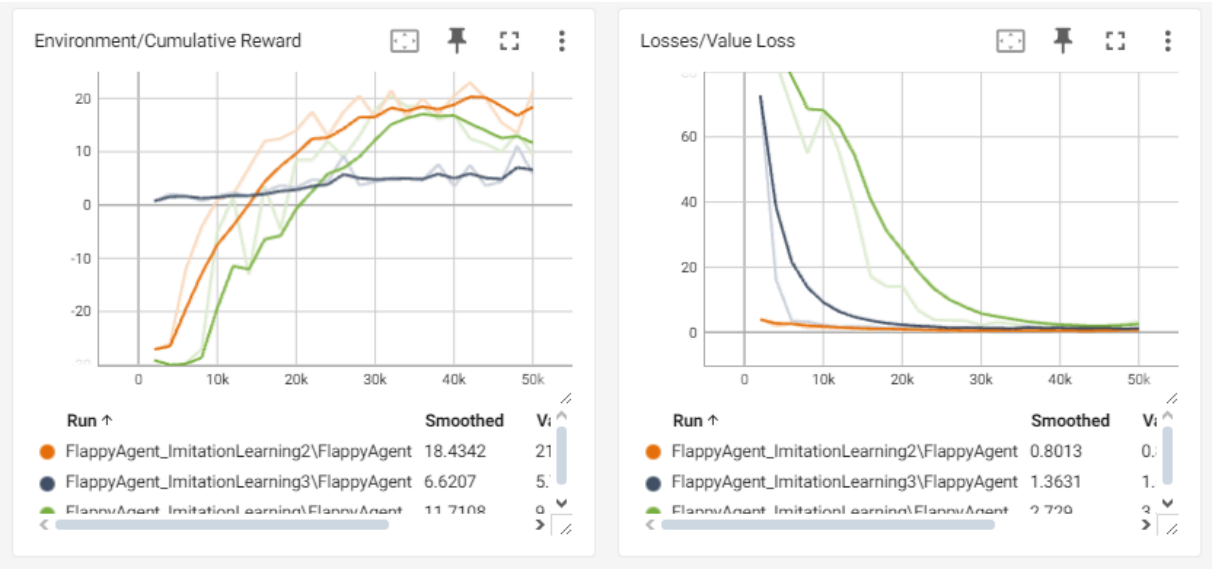
Training configuration file:

```
behaviors:
  FlappyAgent:
    # Same as before

    reward_signals:
      # environment reward (default)
      extrinsic:
        strength: 1.0
        gamma: 0.99
      # GAIL
      gail:
        demo_path: Demos/Leve3FlappyAgentDemo.demo
        strength: 0.1

    # behavior cloning
    behavioral_cloning:
      demo_path: Demos/Leve3FlappyAgentDemo.demo
      strength: 0.1
```

Final Results



APÊNDICE 6

Relatório de Entregas

Semana 1: 19/09

Durante esse primeiro Stage, foram realizadas as seguintes atividades:

- Pesquisa inicial CSCE : Pesquisa de tópicos de interesse em conferências do CSCE2024;
- Stage 1 - 190924 : Pesquisa de tópicos de interesse com base nos seguintes meios:
  - Livro 1 - AI for Games , 2nd edition, do autor Ian Millington;
  - Livro 2 - Artificial Intelligence and Games , dos autores Georgios N. Yannakakis e Julian Togelius;
  - Congresso 1 - SBGames .

Ao final do stage, decidi como foco da minha pesquisa o tema: **“Aprendizado por Reforço Aplicado a Games”.**

Semana 2: 26/09

Durante esse Stage (Stage 2 - 260924), foram realizadas as seguintes atividades:

- **Revisão Bibliográfica - RL :** Realização de uma busca por livros, artigos e publicações científicas sobre Aprendizado por Reforço, com foco na história e evolução do campo. Para isso, foi utilizada uma abordagem de revisão bibliográfica do tipo Screening, conforme apresentado pelo professor Federson em sala de aula.
- **Cronograma Residência em IA :** Com base no feedback do coach career e orientação do professor Federson, foi elaborado um planejamento geral para o processo da Residência em IA. **O planejamento servirá apenas como guia** para a execução das atividades, e está **sujeito a adaptações e mudanças** conforme o desenvolvimento do projeto.
- **Fundamentals of Artificial Intelligence (2020) :** Leitura do sumário e identificação dos capítulos relevantes para minha pesquisa.

Semana 3: 03/10

Durante esse Stage (Stage 3 - 031024), foram realizadas as seguintes atividades:

- **Levantamento de artigos e estudos que tratam da aplicação de Aprendizado por Reforço em Games,** através de uma revisão bibliográfica do tipo Screening, buscando entender a história e evolução do tema, além de aplicações atuais e estado da arte da área. **Foram levantados 53 artigos** durante esse processo.
  - Revisão Bibliográfica - Reinforcement Learning
  - Revisão Bibliográfica - RL for Games
  - Revisão Bibliográfica - AI for Games
- **Produção de um catálogo dos artigos levantados,** com destaque para as informações relevantes de cada artigo (ano de publicação, número de páginas, abstract, keywords), com o objetivo de facilitar referências e consultas futuras. Durante a produção do catálogo, **o número de artigos foi reduzido para 25,** levando em consideração a disponibilidade, relevância e aplicabilidade dos artigos para o projeto.
  - Catálogo de Artigos

- **Produção de resumos detalhados de três artigos relevantes em cada tópico pesquisado,** destacando os principais conceitos, algoritmos, ferramentas e técnicas abordados.
    - Resumos

Semanas 4 e 5: 17/10

Durante esse Stage (Stage 5 - 171024), foram realizadas as seguintes atividades:

- **Pesquisa de frameworks populares de aprendizado por reforço (Reinforcement Learning - RL):** OpenAI Gym, TensorFlow Agents (TF-Agents) e PyTorch RL.
- **Levantamento de ferramentas e frameworks utilizados para a integração de RL com engines de jogos:** Unity ML-Agents, Unreal Engine AirSim e Godot Engine + TensorFlow/PyTorch Integration.
- **Produção de um catálogo detalhando os frameworks e ferramentas identificadas,** incluindo uma breve explicação de como são utilizadas no contexto do desenvolvimento de IA para games:
    - Catálogo de Ferramentas e Frameworks

Semana 6: 31/10

Durante esse Stage (Stage 6 - 311024), foram realizadas as seguintes atividades:

- Escolha do framework **Unity ML-Agents** como principal ferramenta para o projeto final.
    - Configuração do ambiente de desenvolvimento;
    - Levantamento de tutoriais oficiais [1, 2, 3], vídeos explicativos [1, 2, 3], e exemplos práticos [1, 2, 3, 4] para guiar o estudo da ferramenta.

Semana 7: 07/11

Durante esse Stage (Stage 7 - 071124), foram realizadas as seguintes atividades:

- **Preparação do ambiente de desenvolvimento para usar Unity ML-Agents,** seguindo o tutorial How to use Machine Learning AI in Unity! (ML-Agents) e o guia de instalação da Unity.
- **Escolha de um jogo simples (open source) para implementar aprendizado por reforço:** um clone do Flappy Bird desenvolvido em Unity 5.6, disponível em um repositório no github com tutorial de implementação.
- **Criação de um guia didático com instruções de instalação e primeiros passos** **Flappy Bird RL ,** para documentar o processo de desenvolvimento da IA para o jogo.
- **Elaboração de um planejamento para as próximas semanas** **Planejamento Flappy Bird RL ,** detalhando os objetivos semanais para a implementação do modelo.

Semana 8: 14/11

Durante esse Stage (Stage 8 - 141124), foram realizadas as seguintes atividades:

Criação de um material didático Unity ML-Agents :

- **Seção “First Steps”:**

    - Instruções para criação de um projeto Unity com cenas de exemplo do tutorial Getting Started Guide for ML-Agents.
    - Instruções para instalação do pacote ML-Agents dentro do projeto Unity.
- **Seção "Basics":**
    - Explicação dos princípios de Reinforcement Learning, com base no tutorial Background: Machine Learning.
    - Introdução ao PyTorch e TensorBoard, com base no tutorial Background: PyTorch.
- **Seção "Scene: Basic":**
    - Minhas observações sobre um dos exemplos de Environment da Unity chamado "Basic", seguindo o tutorial .

Semana 9: 28/11

Durante esse Stage (Stage 9 - 281124), foram realizadas as seguintes atividades:

Foram adicionadas 80 páginas no material didático Unity ML-Agents :

- **Seção "Basics" (10 páginas):** Explicação dos conceitos usados na Unity Engine, com base no tutorial Background: Unity. Explicação sobre o que é Imitation Learning e Curriculum Learning.
- **Seção "ML-Agents" (58 páginas):** Instruções sobre o uso do pacote ML-Agents, com base em minhas observações e nos tutoriais Getting Started Guide, ML-Agents Toolkit Overview, Training ML-Agents, Designing a Learning Environment e Designing Agents.
- **Seção "Scene: Basic" (22 páginas):** Minhas observações sobre um dos exemplos de Environment da Unity chamado "Basic", seguindo o tutorial How to use Machine Learning AI in Unity! (ML-Agents).

Semana 10: 05/12

Durante esse Stage (Stage 10 - 051224), foram realizadas as seguintes atividades:

Finalização do material didático Unity ML-Agents :

- **Seção "Introdução" (1 página):** Breve introdução descrevendo quem eu sou, o que é esse projeto, onde encontrar os artefatos criados durante o projeto e outras informações.
- **Seção "Imitation Learning" (15 páginas):** Minha implementação de um cenário demonstrando o uso de Imitation Learning, com base no tutorial Teach your AI! Imitation Learning with Unity ML-Agents! .
- **Seção "Flappy Bird" (10 páginas):** Minha implementação de RL para um clone do jogo Flappy Bird (disponível em um repositório no github com tutorial de implementação), com base no tutorial AI Learns to play Flappy Bird! .
- **Seção "Conclusões" (1 páginas):** Reflexão sobre o processo de pesquisa e aprendizado, identificando áreas de melhoria e possíveis próximos passos para aprofundar o estudo do tema.

Criação de um repositório no GitHub permitindo acesso aos projetos práticos desenvolvidos durante o processo da Residência.