On the efficiency of selected machine learning algorithms in predefined and non-defined environments, acting on polymorphic entities with n legs in a simulated biomechanical manner

Modulnummer:	CMN6302
Modulname:	Major Project
Abgabedatum:	23.08.2019
Abschluss:	Games Programming
Semester:	März 2019
Name:	Jan Kuhlmann
Campus:	Hamburg
Land:	Deutschland
Wortanzahl:	8815

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken (dazu zählen auch Internetquellen) entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht.

Elmshorn, 20.08.2019
Ort, Datum

Unterschrift Student

Abstract

This paper handles the topic of simulating n-legged agents with movements in different bodies and environments, exploring the difficulties of machine learning and its application within the game engine ‘Unity3D’. The goal will be a collection of trained and autonomous machine learning brains that can manipulate agent bodies, allowing them to walk by collecting observations about the agent’s state and the immediate environment, as well as acting on the agents muscles or joints.

After establishing the specific target goals of the project, the basics of machine learning will be reviewed, going further in depth with relevant focal points such as the types of applicable learning algorithms. Before the experimentation phase, possible branches for the project’s outcome will be discussed and prepared to be tested during the execution phase; this includes the possible libraries and environments that are applicable, along with theoretical blueprints for the agents, discussing the process with which they are going to be constructed.

The execution phase aims to experiment using said principles as a foundation and build the project on them further, exploring the possibilities and acting on the experiments’ results, documenting the process and decision making throughout the execution. The results will be summarized and evaluated, while also critically reviewing the execution process, along with utilized methods and strategies.

Content Overview

Illustration and Gif Index

1 Introduction

1.1 Topic

This paper will explore the use of machine learning in creating simulated biomechanical entities, which are aimed to behave similar to biological muscles and joints. The project will be conducted within the Unity3D engine, which lays the groundwork for rendering, physics and various helpful programming tools.

1.2 Red String

The first steps will consist of explaining the overarching concepts of machine learning, while keeping the flood of information to an acceptable level. Only major concepts will be explained, further details will be made clear over the course of the execution phase.
The implementation of machine learning will be done either on a clean C# base or premade python libraries in Anaconda3, made available through an official Unity SDK project, which would therefore eliminate the effort of writing custom neural networks and learning algorithms.

1.3 Motivation

Machine learning is and has been a hot topic for some time. Large companies like Google and Facebook are making advancements in actual applications for their trained neural networks for purposes like image analysis and search predictions to e.g. prohibit sharing illegal material (see Vision AI, 2017). Another area of interest are robotics companies like Boston Dynamics, who do not make use of machine learning for their walking and realistically behaving robots (see Boston Dynamics, 2017), which is the opposite of my approach to the same issue of creating walking robots, thus providing a possible gateway to the implementation of machine learning in actual biomechanical robots.
Aside from goals for application, machine learning is also a topic that interests me personally; I have previously completed a different machine learning project which involved a simple scenario of an agent moving through a grid to a target goal, while avoiding harmful obstacles. Said project was relatively simple, so I wanted to expand upon

it, which lead to the idea of using deep learning to enable walking for various n-legged entities, offering a much more complicated environment to work with.

1.4 Must-Have Goals

The final product must include a scene, which has an environment, either set or freely generated, as well as at least one type of agent with n legs, which has to navigate to a goal by utilizing its muscles to move. The agent will have to be aware of its environment by getting information about the distance of body parts to the ground, velocities, rotations, as well as direct contacts. The brain of the agent needs to either be purely completed via C# or created by using python libraries.

1.5 Nice-to-Have Goals

Various things, which can be learned by the agents would include pathfinding, environment-specific movement like jumping over a hole or crouching underneath an obstacle. An additional goal is to expand the manners of input, which can be perceived by the agents, e.g. allowing them to use raycasts in multiple directions to map the direct surroundings, or even react to audio cues.

Should the environment be easy to use and overall applicable in game development, the project should be converted to an asset and made available through the Unity Asset Store. If the machine learning aspect of the project gets done in C#, then this can also be publicized as a separate asset for others to use.

2 Basics

In order to understand the thought processes in the following chapters, a few basics need to be understood. Since I will be using machine learning as the main concept for this project, it needs to be understood on the surface level. I will present basic machine learning terminology, common learning-types and the environment I will be using to create the project in.

2.1 Neural Networks

The idea behind neural networks is to try and simulate the neurons of actual brains and mimic their ability to learn various tasks. The general flow of information can be broken down into three stages (ill. 1: Neural Network):

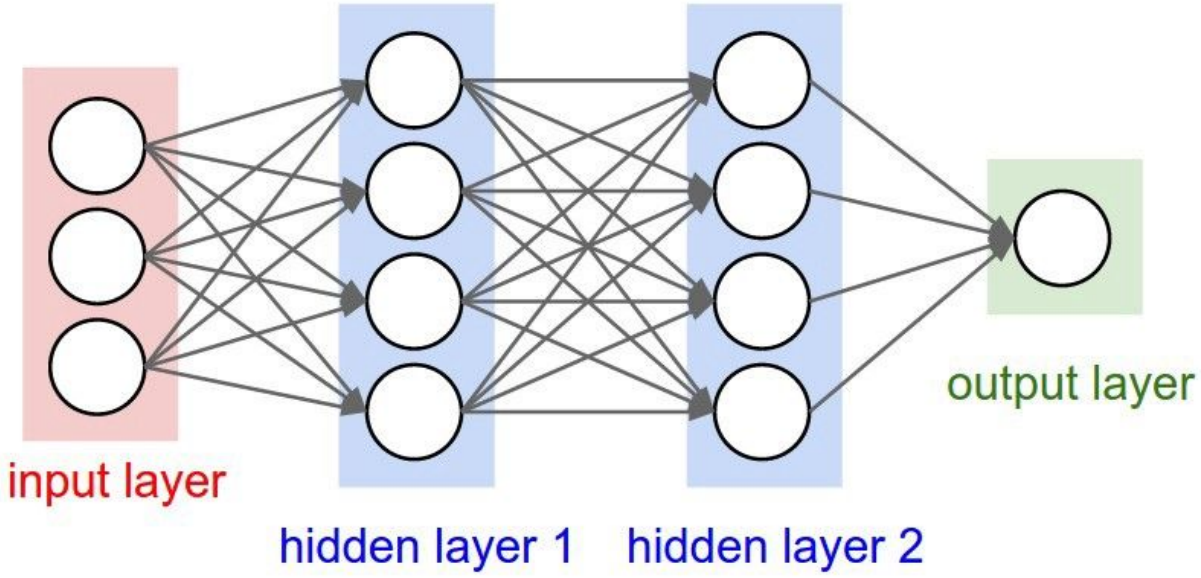


Illustration 1: Neural Network (Dormehl, 2019)

- A neural network will first need an array of input values, also called **State Array**, which can e.g. be interpreted as the **five senses** in the human body. These input values represent distances, positions, image details or other environmental states that the program has to feed to the network; these are basically its eyes and ears and represent the first layer of each neural network.

- These inputs get processed throughout one or multiple layers of neurons, which are called **Hidden Layers** and can be viewed as the **brain**. Each neuron has three main components; it has a weighted connection to a previous neuron, an activation function and a cached value. The weighted connections modify the value of the previous node, feed it into the activation function f(x) and caches its result to be read by the nodes connected to it.

- The last layer of each neural network is the output layer, called **Action Array**, which contains the resulting actions or decisions based on the initial inputs. These values will control **muscle movement** and allow the agent to move.

That is the general outline of every neural network. They have differences based on their use cases, with the most important ones being:

- Input Type. There are two main types of information that are fed to neural networks. In most cases, one needs to decide on which type is the most suited. The first is a **discrete input array**, which describes a set state, like the state of all pieces on a chess board.
  In complicated or 3D environments, this approach is not sufficient, with the other option being a **continuous input array**, which can handle distances, positions, velocities and other types of input that does not have a predefined magnitude.

- Output Type. The output array has the same main types of arrays. The **Discrete output array** can output discrete actions like a button press or a certain type of action. This type is suited for cases, in which the neural network should have the same options of game input as a human user.
  The other option is a **continuous output array**, which can output target rotations and target positions, which are not discrete or certain to be reachable. This option is suitable for more dynamic neural networks, which can work directly with the environment, instead of interacting with its body via virtual key presses or discrete actions. The output array is frequently called action array/space.

- Activation Functions f(x). Each neural network has either one or multiple types of activation functions. These are supposed to mimic a neurons property to only fire a signal, if the incoming signal exceeded a certain value. The most popular activation functions are the **Sigmoid Function**, **Hyperbolic Tangent Function (tanh)**, **ReLu** (Rectified Linear Unit) and their variants. See ‘Illustration 2: Hyperbolic Tangent’ for a standard tanh function.

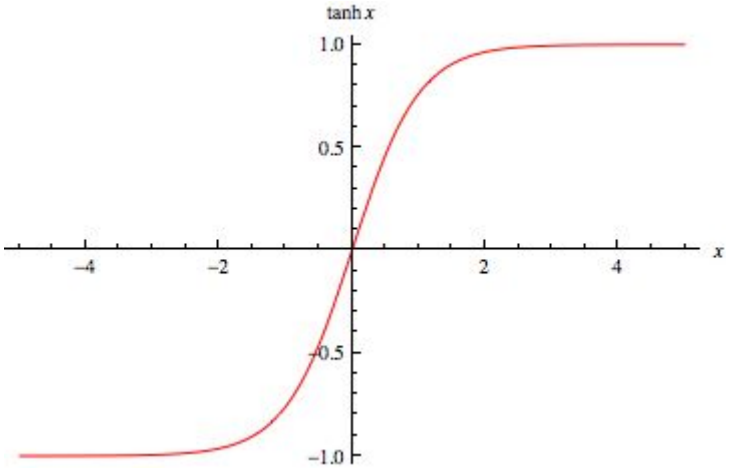


Illustration 2: Hyperbolic Tangent (Weisstein, n.d.)

In the scope of this project, the input and output arrays will most likely be continuous, but this, as well as the activation functions, will be discussed in the project/experimentation phase (see Zerium, 2018).

2.2 Learning Algorithms

The neural networks by themselves are not useful untrained. There are various different types of training methods, each having specific use cases, sometimes with subcategories of major learning algorithms for a specific learning task.

For the purpose of teaching a neural network to walk, two main learning algorithms are the most useful, namely Reinforcement Learning and Imitation Learning.

2.2.1 Reinforcement Learning

This algorithm is the most common and widely-known through its simplicity. The main idea is to have a neural network brain control multiple agents, performing random actions based on their current states. The programmer has to define rewards and punishments, which are values between -1 and 1 and allow the brain to learn appropriate actions through trial and error. When working purely with neural networks as the brains, a simple way of mimicking evolution would be to give all bodies in the game the same neural network and mutate them randomly, then selecting the most rewarded brain to be copied onto the other bodies, mutate them and repeat the whole process. This evolution-like

method works for more simple problems and handles discrete input and output for simple projects (see Simonini, 2018).

2.2.2 Imitation Learning

As the name suggests, this learning algorithm is based on imitation. An agent observes the actions of a “teacher” and derives its own decision-making to imitate the teacher’s actions in similar or identical situations. The teacher can be another ML agent, a human player or a recording of (one or) many past human players. The idea of imitation learning is based on a human’s actions in completely new environments. They do not blindly poke for the best solutions through trial and error alone; they always measure in their past experiences, which could include having seen someone else perform an action in a similar environment that they then try out themselves. This is helpful for robotics, since a living creature’s movements can be recorded and forwarded to an ML agent, who can then proceed to imitate them. Imitation learning would be helpful for this project, but since it is not focused solely on two- (humanoid) or four-legged (animals) movement, but rather n-legged movement, teachers cannot always be provided for the agents in my environment.
(see Kurin, 2017).

2.3 ML Agents

The most accessible option to utilize machine learning is the machine learning environment that was created by unity developers specifically for the Unity3D engine, called ‘**ML-Agents**’. It utilizes the **Anaconda3** library for the machine learning portion, which is a library written in **Python** (see Unity Technologies, 2018).

2.3.1 Python

Python is an object-oriented, high-level programming language that is popular with scientific applications, including machine learning scenarios. It is easy to read and quick to write, especially when properly utilizing its modular functionality, which allows for implementation of third party modules (see Python Software Foundation, n.d.).

2.3.2 Anaconda3

Anaconda is the easiest way to utilize machine learning using Python. It features the core Python language along with more than 1500 optional libraries and a package manager, “conda”, to manage those optional packages. The most important ones for this project will be **Tensorflow**, which handles the machine learning, and optionally **Jupyter**, which can be used to supervise the learning progress in the form of Jupyter Notebooks (see Anaconda, [no date]).

2.3.3 Unity3D

Unity Technologies created a GitHub repository by the name “ml-agents” that allows anyone to utilize machine learning within their Unity3D engine. Since the engine already features rendering, audio, physics and input handling, machine learning projects in this environment are easy to create and test. This repository uses Anaconda3 and therefore grants access to all Tensorflow and Jupyter tools. Since the engine’s top-level code base requires C#, access to the machine learning compartments from within the Unity3D-Project requires it, as well as basic engine knowledge (see Unity Technologies, 2018).

3 Methodology

There are various ways to complete the goal of this project; getting n-legged agents to walk. To get there effectively, however, requires some planning and vital decisions to be made, which are going to be discussed during the course of this chapter in order of execution.

3.1 Machine Learning

3.1.1 C# vs Anaconda3

There currently exist two options to use as a base for the machine learning part of this whole project. The first is to use a clean C# code-base and build the machine learning algorithms from the ground up, the second is to use Unity’s “ML-Agents” repository on

GitHub, which utilizes Anaconda3/Python; thus, the first decision of the project will be the choice between either of the two. Using the pre-made repository will skip the creation of my own machine learning algorithms in C#, so the task will be to find out if making my own would be plausible. The steps here will be to **analyse** other projects that utilize simple machine learning with discrete input and action spaces and try to **recreate** them. Should this succeed, the algorithms need to be upgraded to handle dynamic input and action spaces. If this turns out to be too time consuming, I will instead be using the pre-made repository “ML-Agents” in order to save time and ensure the success of the project.

3.1.2 Learning Algorithm

The two optional algorithms for my project are reinforcement learning and imitation learning and I need to decide on which one to use. They both theoretically have a place in this project, but imitation learning can be ruled out, due to being inflexible:

In order to utilize imitation, an existing agent must be present, which in this case would either be a player, computer-controlled brain or an animation. Having a player control an agent can already be ruled out, since a very large action space is necessary to control all the various muscle movements, which a human cannot all control at the same time via keyboard, mouse or other input devices that are available to most. Having another agent be controlled by a computer would defeat the purpose of this project, since that very agent is the goal. The last option of creating an animation and having the algorithm learn based on it would be an option, but it requires extra work and expertise in animation, if no fitting animations are available for a specific n-legged entity. Since the project is supposed to be easy to use from the get-go, this will not be an option.

Reinforcement learning will provide a more flexible algorithm, due to not being restricted by anything but the length of input and action spaces, which will increase the learning time with large input- and output-arrays. Since the reinforcement algorithm has nothing to go on but episodic rewards, unlike imitation learning, which imitates an already existing concept, reinforcement takes longer to learn new behaviours, but is necessary in the scope of this project; thus I will be using **reinforcement learning** as the main training algorithm purely through the **process of elimination**. This will be upgraded to ‘**Deep Q-Learning**’, should a custom C# machine learning algorithm be used for the final project.

3.2 Agents

The agents will be representing the bodies that the machine learning brains can control. They will mainly require body parts with **rigidbodies (physics)** (ill. 3: Rigidbody component for physics) and **joints** (ill. 4: Type of joint component 'hinge joint').

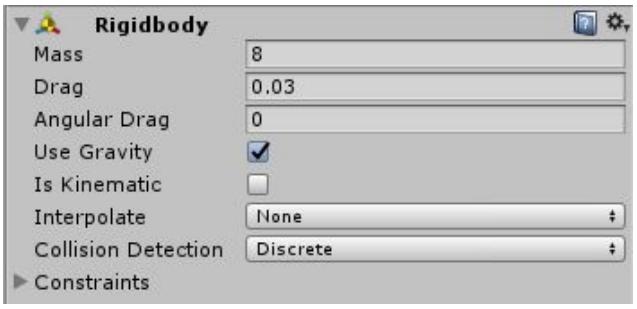


Illustration 3: Rigidbody component the physics

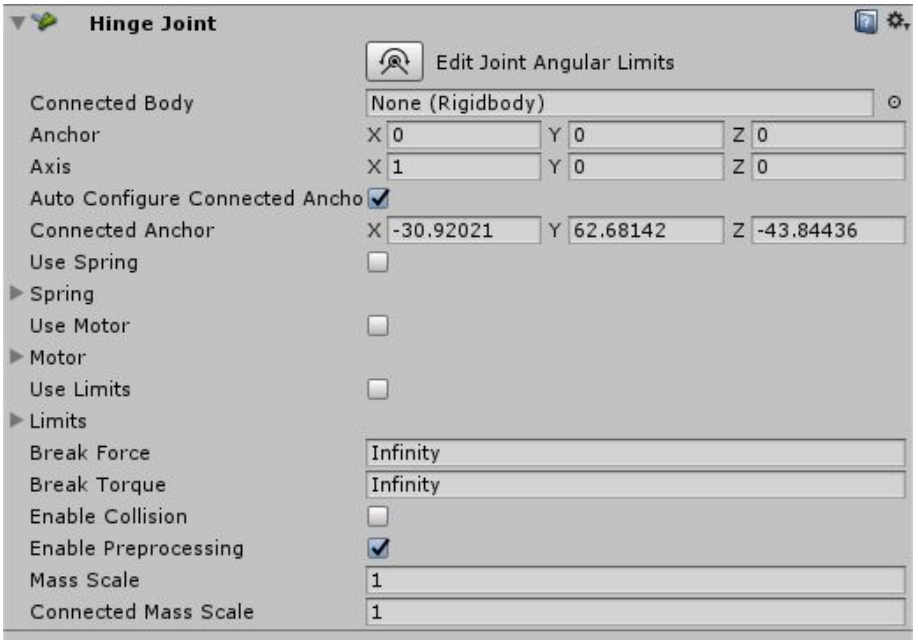


Illustration 4: Type of joint component 'hinge joint'

The rigidbodies are relatively straightforward; each body part needs to be assigned a rigidbody component with a set weight to try and recreate a realistic biological body. The body parts need to be connected via joints, of which multiple kinds are available in Unity. A goal during the execution phase will be to figure out which joints are the most fitting for what body parts, or if only one type of joint should be used for all use cases, in case they are controlled differently. After deciding on what joints to use, they each have a driver

component to them, which represents a motor/muscle that can move and rotate the joint along all three dimensions x, y and z. These motors have a main force, spring force and optional damping, which are all components that need to be evaluated for each muscle/joint. This will be done through **experimentation** with various values, as well as **analysis** of other projects utilizing joints with close-to-realistic bodies. The main projects to analyse will be the example scenarios ‘crawler’ and ‘walker’, which both feature an attempt to emulate forward movement with bodies and limbs. The crawler has four legs, while the humanoid walker has two; they will both provide valuable input for my own project that can use them as an entry point for my basic script behaviours, which will then be expanded upon. Said example projects can be found within the machine learning repository ‘ML-Agents’ that also contains the basic UnitySDK for machine learning using Anaconda3.

The resulting bodies should represent basic animalistic or humanoid shapes which includes a **main torso**, **limbs** and an optional **head**. The main torso will be the center of gravity, while the limbs are responsible for movement. The head will first only have its weight as an impact on the whole body, but can later serve as a starting point for the agent’s ‘eyes’, which will be a topic for the environment design.

All the body parts need to be **modular**, meaning that each agent can easily be extended upon with extra limbs and thus, the scripts needs to allow extra limbs to be added without any/too much manual change in the limbs’ script. The agents therefore need to be dynamic to the point where they can handle any amount of limbs. Every body part will have its own input and output/action array and can therefore act as a separate entity that merges into the full agent via its inputs and outputs, the rest of its logic stays hidden within the body part itself. While a foot will be able to rotate like other limbs, it should allow contact with the ground or give more specific info about the surface it is touching; these specific information sources, along with the basic joint and rigidbody settings need to be experimented with to make training as **efficient** and **effective** as possible.

The first target agent will be a **four-legged dog-like agent** with basic limb movement and design. Since the agent can be easily extended with extra legs, other possibilities would include creating a humanoid agent, a multi-legged centipede or spider and various other creatures with different leg counts that can be emulated within Unity.

3.3 Environment

Aside from the task of simply moving forward, the modular nature of the agents' body parts will have to allow for more customizability. In the case of a head, it would make sense to include a type of vision, that would allow the agent to construct a basic sense of its surrounding environment via raycasts or similar detection methods. In order to test the limits of this functionality, the agents need an environment to move around in; this will then be scaled up in difficulty to traverse in the form of obstacles, pathfinding or ground instability.

The agents' goals concerning the environment will be:

- Not touching the ground with anything but their feet
- Not touching any walls
- Moving towards the goal

How to set up the rewards/punishments for these rules and requirements will be done during the execution phase. The goals can be adjusted for each type of environment specifically and will very likely be expanded upon, but they currently merely represent the starting point for the main goals of every agent.

The first and most **basic environment** will be a plain surface with a goal on the opposite side of the agent, to test basic forward movement and optimize the parameters for walking or running. This or a very similar environment will be used during the first to middle phases of the project to set up the agents.

A more complicated environment would require the agents to get a basic sense of their surroundings by placing walls closely around and in the way of their path, requiring the agents to navigate more precisely. The goal of this environment is to not make the paths snake too much, only to vary the straightness of their path slightly to see if the agents can be taught to make slight turns according to their environment. The agent will still try to reach the goal here, so it would be best to not force any corner turns sharper than 90° to avoid 'necessary punishment' to be able to progress in some cases. An interesting environment to teach constant turning would be a slalom path with multiple walls in the way. In order to avoid the agents simply learning how to move through one specific path of walls, these can be **randomly generated** to some degree, forcing agents to rely more on their **emulated 'sight'**, rather than memorizing where to move at what location.

Should the previous environments succeed, their respective ideas can be extended upon; an example would be using more snaking paths with angles higher than 90° that require moving away from the goal for a short while. Another improvement would be narrowing the space between walls to enforce more precise movement and teach the agents to limit their muscle strength more precisely. These are all ideas that are only extensions, however, so they might not be as important as exploring new environments, like the following:

An environment can have uneven ground that enforces precise detection of the distance between the foot and the surface right below it, in order to progress without falling over. The pattern on the ground can be created using procedural noise or other randomization methods. Instead of making the ground uneven, some areas of it can be marked as ‘dangerous’ and will kill the agent once it steps on them. This is much easier to implement and would require detection of danger zones around the agent, as well as the ability to circumvent said danger zones.

An idea for interesting traversal would be a parkour-like course. This includes obstacles that need to be moved/jumped **over** and obstacles that need to be ducked **under**. This can make for interesting behavior for the agents, since ducking and jumping are extra behaviors that would not be explored in other environments. They will also require a more fine tuned detection of obstacles, in order for the agents to consistently progress towards the goal.

The idea of a static goal is not a necessity, it can be moved around. This can be used in an enclosed space with multiple types of obstacles from previous environments and goals at random positions. The agent would navigate straight towards one goal, deactivate it and activate another random goal to move towards. This would allow constant training of various types of obstacles and can make comparison between them easier when changing a brain’s parameters.

It is not guaranteed that all of these environments will be created, but rather be implemented as far as and if time allows.

3.4 Versioning

This project will be managed using ‘GitHub’. Git is a version control protocol that can be used to manage projects in teams or solo projects in this case. Its use in this project is to ensure access to past versions and allow review of the whole project’s progress for readers through the git protocol, as well as allow branching to test potentially volatile actions in the project like deleting or reworking certain code sections. GitHub can manage git online, acting as an intelligent storage, in case my local files ever get corrupted or end up lost. It also makes sharing the project much easier since it opens all versions and the final result to the public to view and use themselves. It also allows multiple developers to work on a single project, though that will not be the case for this project specifically (see Brown, 2017).

4 Execution

4.1 C# Based Machine Learning

In order to start working on the project itself, the base for all machine learning processes need to first be established. In order to decide between Anaconda3 and custom made machine learning algorithms in C#, an experimental project needs to be made to serve as an example of how quickly and effectively custom algorithms can be created for the purpose of deep learning.

4.1.1 Project Setup

In order to allow versioning, a new repository was created and hosted on GitHub with the name ‘Neural Networks’. This repository will contain the following experimental project. The online repository was cloned and setup with a ‘gitignore’ file, which limits the files that are managed by the git protocol. Inside the locally cloned repository, a new project was created; for this, the Unity3D game engine was used to allow pure focus on the functionality of my algorithms and eliminate the need to create a custom three-dimensional object rendering system. In order to eventually create a deep learning

algorithm, it is best to climb up in difficulty, starting with a basic reinforcement learning algorithm.

4.1.2 Reinforcement Learning

A fitting scenario for testing a reinforcement learning algorithm needs to contain agents that are simple to control and require strictly discrete input and output spaces. The chosen scenario for this was having multiple agents constantly move along their own forward vector and only give them the option to turn as a steering method. They should get rewards based on their angle between their forward vector and the vector between them and the current mouse position. Their goal would thus be to constantly turn and move towards the mouse (gif 5: agents following the players mouse).

Gif 5: Agents following the player's mouse

To achieve this, two main pieces were necessary: A neural network and the reinforcement learning algorithm.

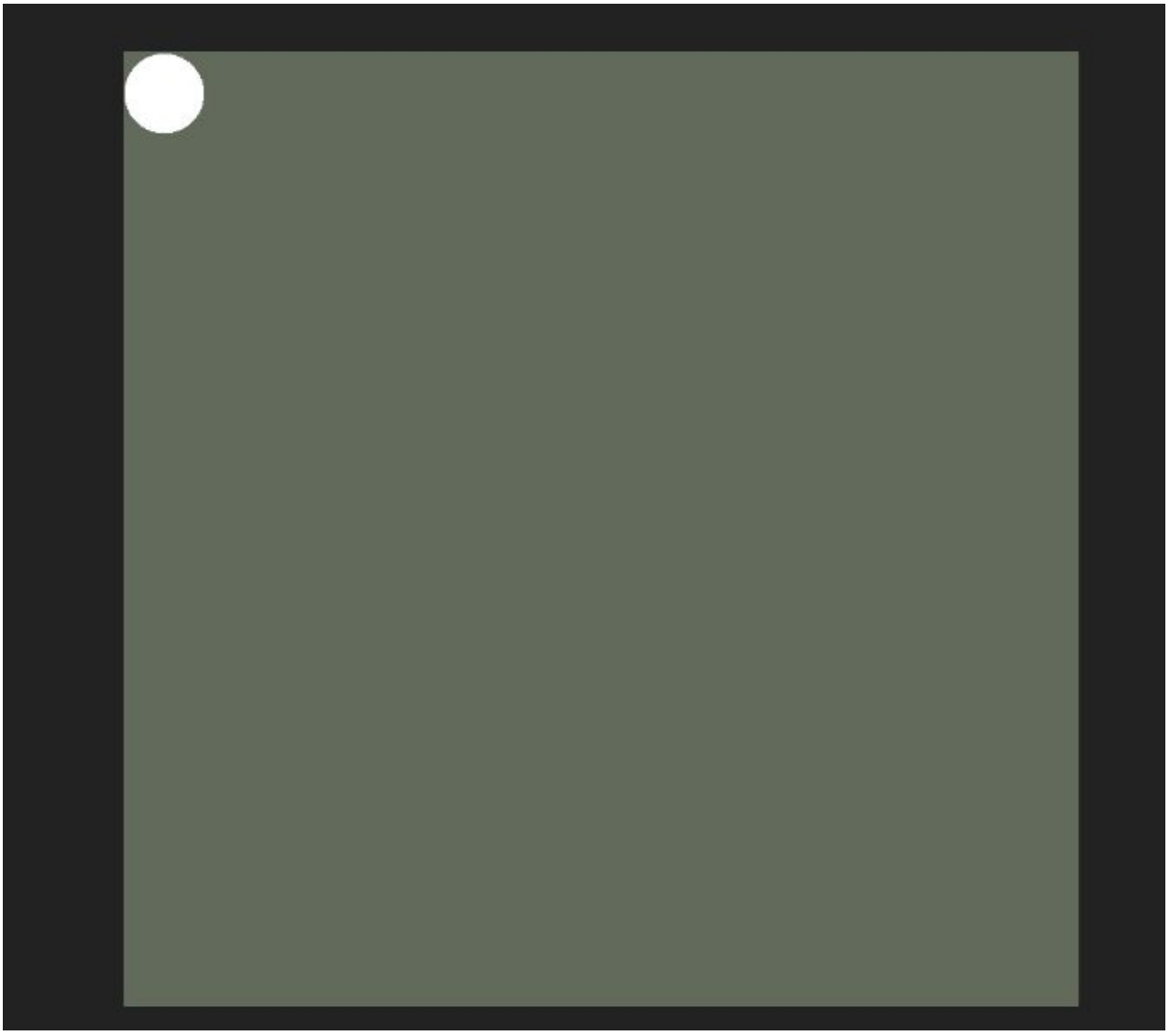
A tutorial on the internet platform ‘YouTube’ was used as inspiration for some of the ideas for the network and the application scenario (see The One, 2017).
The network consists of four main parts; the network’s fitness, the size of each layer (the amount of neurons per layer), the value of each neuron and the weight of neuron connections. The fitness is a float value stored for the reinforcement algorithm and comes into play later. The layer sizes are stored as arrays, while the neuron values and connections are both stored as jagged arrays; they are all easily iterated through, which is important for all kinds of neural networks that need to make frequent decisions and not take up too much performance during each iteration. The neural network also has two main methods; the first one is called ‘**RequestDecision**’ and accepts a float array as a parameter, which represents the input array. Each value is then fed to the first nodes in the network and get fed to the nodes in the next layer based on the connection weights, once the first layer is filled. This process continues until all output nodes have been fully written to and can return and output array as the method’s return value. The activation functions that were implemented were a flat function (no change to weighted values), hyperbolic activation (Tanh) and ReLU activation (Rectified Linear Unit), which all work for this example scenario. The second main method in the network ‘**Mutate**’ directly correlates to the algorithm that was implemented to utilize machine learning. It iterates through all weights in the network and attempts to randomly mutate each weight individually, based on a chance that can be set in the inspector. The resulting weight will be loosely based on the previous weight with a maximum and minimum offset that the random number generator can choose from.
Every agent will start out with a randomly mutated network and evolve it over the course of its lifespan. Every frame, a decision is requested with the angle between its forward vector and the vector between itself and the mouse as input, while interpreting the output as a z-rotation. After handling the network’s output, it needs to receive a reward in the form of positive or negative fitness, which gets added onto the network’s total fitness. The network is given a positive reward, if the agent is either very close to the mouse or aiming its forward vector nearly directly at it. Should both these cases evaluate to false, fitness is retracted and the ‘Mutate’ method is called to incentivize change in the next decisions.

The networks are created and handled via a '**Neural Network Manager**', which, as the name implies, serves to manage all networks and apply the reinforcement learning algorithm every time the player presses down the 'Spacebar' key: All current neural networks on the agents are evaluated based on their **fitness** and the network with the highest fitness value gets passed on to the next generation of agents, repeating the whole process of neural networks mutating and eventually being evaluated until the player stops the process or unloads the scene. After around two or three iterations, a number of agents will successfully target the cursor, therefore proving this algorithm to be a success, allowing the project to move on to the next step, Deep Q-Learning.

4.1.3 Deep Q-Learning

Having successfully implemented a basic version of reinforcement learning, the project could now be upgraded to a more complicated but necessary learning algorithm, namely deep q-learning. In order to construct a custom deep q-learning algorithm specifically for this project, a third party library called 'REINFORCEjs' was used as a template (see karpathy, 2015). As the name implies, this library was written in javascript, but features exactly the features that would be necessary in this project. Thus, the task was to port the whole library with deep q-learning into C#, which required learning the basics of javascript. A new script called 'DeepQLearning' was created within the same project, along with the reinforcement learning scripts. Then began the process of converting the javascript library into C# within the newly created script; all classes were contained within a newly created 'DeepQLearning' namespace, containing seven classes that were ported from the original library, as well as a void-returning delegate without any parameters. In order to allow for more cleanly written code, an extra class 'Utils' was created to contain various math-related methods. The first class 'DQNInitSettings' contains all the settings that can be set before creating a new 'DQNAgent', another class that contains the main agent with references to the other classes. The two most important classes for deep q-learning are 'Experience', a class that stores past decisions and results as an experience, and 'Network', which contains four matrices, representing a neural network. The 'Network' class is handled by a 'Graph' class that is used for backpropagation, as well as matrix addition, multiplication and hyperbolic tangent activation.

In order to test the newly ported library in Unity, the main class ‘DeepQLearning’ was made to inherit from ‘MonoBehaviour’, thus enabling it to be added onto a Unity ‘GameObject’, attaching the whole script to a chosen agent, who was represented using a white circle. The agent was initiated with the default settings and had control over movement within two dimensions: horizontally and vertically. The goal of said agent was to move to the center of a box-shaped arena (gif 6: agent running against walls).



Gif 6: Agent running against walls

As seen in the resulting gif, the agent ended up running into walls, which prompted change in the code. It was attempted to change the reward system to punish wall contact or change the interpretation of the action array to handle discrete target positions instead of continuous values in the horizontal or vertical direction. Other attempts were to debug

code sections, which proved to be difficult and time consuming due to the size of the script.

After having invested too much time into attempting to fix the deep q-learning algorithm, it was decided that fixing the existing or creating a new algorithm from scratch would take up too much time, thus, the official ML-Agents repository would be used as the base for machine learning during the upcoming course of the project.

4.2 ML-Agents UnitySDK

After the decision to abandon a custom C# machine learning algorithm, the machine learning kit needs to be set up to be used for this project. The kit is part of the Unity Technologies 'ML-Agents' repository and is the project contained within is called 'UnitySDK' (Unity Software Development Kit) (vgl. Unity Technologies, 2019)

4.2.1 Anaconda3/Python Environment

In order to allow the UnitySDK to utilize the Anaconda3 libraries, these need to first be installed. Downloading and installing Anaconda3 includes the base Python. After installing Tensorflow via the conda console, the UnitySDK should get its own local environment to contain settings specifically for this project. It is optional, but recommended, to also setup CUDA for GPU computing (see NVIDIA CUDA, 2019).

4.2.2 Project Environment

In order to work with the ML-Agents repository, it needs to either be downloaded from the main repository or forked into a seperate one. While it is possible to clone the main repository and work within it, it is not recommended due to the lack of version control permissions; only local commits would be permitted, which is not ideal, since the progress of this project should be trackable online for the purpose of being able to easily share it. The repository was forked and locally saved in order to work within the Unity project.
The project folder within the repository root is called "UnitySDK". This project contains all necessary scripts and tools to utilize its machine learning capabilities, plus a number of

extra example scenes for various ml-application cases. Aside from the main project, multiple subfolders can be found within the repository's root, of which the folders 'ml-agents' and 'ml-agents-envs' correspond to the external Anaconda3 processes, along with the 'config' folder, which contains configuration files that are used for training.
A subfolder 'Polymorphic Project' was created within the 'Assets' folder of the 'UnitySDK', together with a corresponding C# assembly definition to reduce compilation time and folder clutter. It would also be possible to delete all examples that are provided for even faster compilation times, which is not recommended however, since these are helpful to learn about best practices or, for beginners, provide a starting point for custom environments and may even come in handy for advanced users at certain points. A new scene 'MainScene' was created to contain upcoming game objects such as the environment, agents and an academy.

4.2.3 Polymorphic Academy

The Academy is a MonoBehaviour component that needs to exist within a scene that utilizes machine learning; it controls the brains in the scene and can communicate with external processes, in this case the Anaconda3 process, which handles the machine learning portion and controls it from the outside. A new game object 'Polymorphic Academy' was created in the scene with the sole purpose to host the 'PolymorphicAcademy' component. A new brain 'PolymorphicLearningBrain' was created and assigned to the academy as a broadcasted brain in the broadcast hub, allowing it to be controlled via Anaconda3 processes. The inspector has additional settings for training and inference configurations, for which the default settings were used as a starter template, only with an adjusted training time scale of 25 due to local performance limitations. The resolutions do not matter outside of a built project, which is not necessary for training. (ill. 7: Polymorphic Academy).

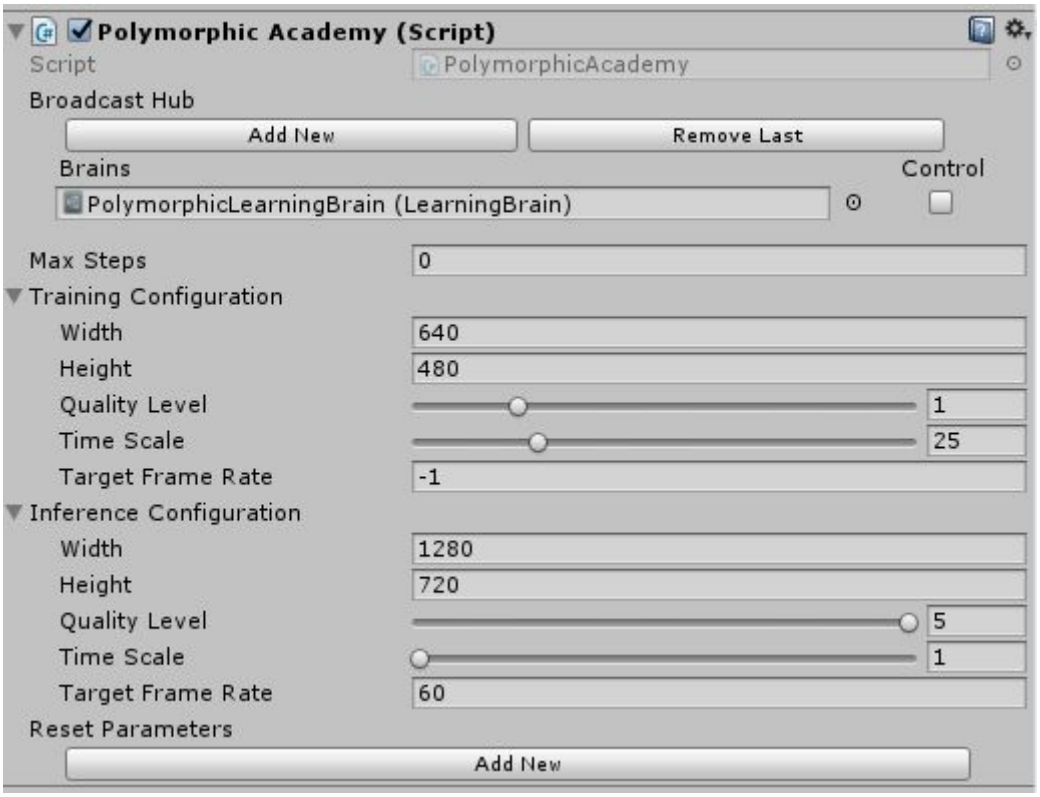


Illustration 7: Polymorphic Academy

Inside the script itself, the physics solver iterations were heightened to 12, along with a shortened fixed update interval of 0.013 seconds, similar to other physics-based example scenarios within the SDK. This ensures a higher physics quality for more precise movements and allows the brain to work with more precise values in the form of velocities and angular velocities.

4.2.4 Polymorphic Agent

In order to enable a modular agent to still collect observations and transmit actions to body parts, a system needs to be setup that allows any number of limbs or general body parts to be handled by a single agent. Along with the agent ‘PolymorphicAgent’, a base class ‘PolymorphicLimb’ was created to serve as a template for all modular body parts. An agent scans all children of its transform component for polymorphic limb components on the start of its lifecycle, then collects the references to each in a list. This list is iterated over to collect observations and transmit actions for each body part. A decision interval system was implemented, that only makes every x action relevant, x being five with the

current settings. This is to prevent stacking of new decisions, while the physics have not been updated properly yet, since they occur in a fixed time interval. The only observations that are seperate from the body parts are the rotational and velocity alignment of a center rigidbody component to the goal. The agent can be reset via method call, called by the academy, which calls an **'OnAgentDone'** method on all body parts, therefore leaving the specific reset controls to every individual body part.

4.2.5 Polymorphic Limbs

The 'PolymorphicLimb' class is a base class that serves as a template for all body parts of a polymorphic agent. It contains two properties **'ObsSize'** and **'ActSize'**, which need to be overwritten with an integer value for every limb type. 'Obs' stands for observation and corresponds to the amount of float values that are observed in this limb, while 'Act' stands for action and corresponds to the amount of floats required for this limb's actions, such as rotating a joint.

On the start of its lifecycle, the limb remembers its starting position and rotation in global coordinates, which it resets to, should 'OnAgentDone' be called, the method that gets called by the polymorphic agent. The agent can call two other methods: The first being **'CollectLimbObs'**, which collects all observations in the limb and writes them into a list that gets passed onto the next limb; the second being **'FeedActions'**, which reads float values from the action array with a starting index corresponding to the current limb, therefore acting on said float values. Two mono behavior methods 'OnCollisionEnter' and 'OnCollisionExit' were used to control a 'grounded' boolean, therefore being able to add ground contact as an observation during the observation call.

In addition to these methods, three utility methods were added that have the ability to add quaternions, vectors and booleans to the observation list by converting them to floats, e.g. a quaternion is split into four floats that get added to the observations.

Aside from the polymorphic limb, another base class 'PolymorphicJointedLimb' was created to accommodate all jointed limbs. It inherits directly from the polymorphic limb base class, but adds joint functionality to it. The reset system now includes the initial joint force and rotation. It also offers a method **'SetJointProperties'** that sets the joint motor to target a specific euler rotation, along with a target force. Since the action array contains float in the range between -1 and 1, these values need to be converted to the space 0 to 1

to work with interpolation. The joints have angular limits for each of the three axes x, y and z, therefore offering a low and high rotational limit, which is utilized in the lerp function. The force is set based on a maximum force, which represents the force set for the joint in the inspector.

Each limb can have an optional component ‘ContactRewarder’, which offers the ability to directly reward the agent with a custom negative or positive float value that can be set in the inspector, should the limb touch a collider with a specific tag. This was used to punish the agent, should a body part like the torso touch the ground.

4.2.6 Polymorphic Brain

The previously created brain ‘PolymorphicLearningBrain’ contains the settings for input/observation and output/action arrays. The amount of observations and action in every limb is summed up and compared to the brain parameters; both must match exactly, which can be checked by running the agent in the editor, which compares the brain parameters with the summed up observation and action sizes of every limb in an assert method. Aside from setting the correct vector sizes, the action space type must be chosen, which is continuous in this project’s case, as already explained. It is possible to stack observation vector, in order to remember the last x-amount of observation arrays, stack them with the current observation vector and make decisions based on the result. This allows the brain to interpret velocities and predict body part positions and rotations based on multiple observations; this settings was set to five stacked observation vectors.

4.2.7 Polymorphic Body

With the scripts complete, the agent’s body was created. In order to test forward movement, only a main body/torso and four legs were created.

The torso was outfitted with a ‘CenterTorso’ script, that inherited from ‘PolymorphicLimb’, along with a rigidbody for physics and ‘ContactRewarder’ script that was set to punish the agent for contact with any ‘ground’ tag.

The legs consist of three segments, the upper leg, lower leg and foot. The foot has the same components as the torso, except without punishments for ground contact. The upper and lower leg parts additionally received a ‘ConfigurableJoint’ component, along

with scripts that inherit from ‘PolymorphicJointedLimb’ to control said joint. The joints were initially different kinds of joints like hinge joints for example, but they are all controlled differently and since the configurable joint can be turned into any joint, it was used for all joint applications.

The available actions for the jointed limbs are a target rotation, along with a strength value, standard polymorphic limbs do not have any available actions. To reduce the action count for jointed limbs, only available rotation axes are offered as action inputs, excluding locked joint rotation axes.
The body part observations consist of:

- Boolean for being grounded
- Local position relative to the body’s center (torso)
- Rigidbody velocity
- Rigidbody angular velocity

Along with extra observations for jointed limbs:

- Limb rotation
- Joint motor force

4.2.8 Environment

The environment was set up to be simplistic and encourage basic forward movement for the agent; thus, it was built to contain an endless plain with a single goal that the agent has to move to, doing so in a single direction without obstacles. This was necessary to test the agent’s ability to handle the observations and action options given to it, as well as to test the learning parameters.

4.2.9 Learning Parameters

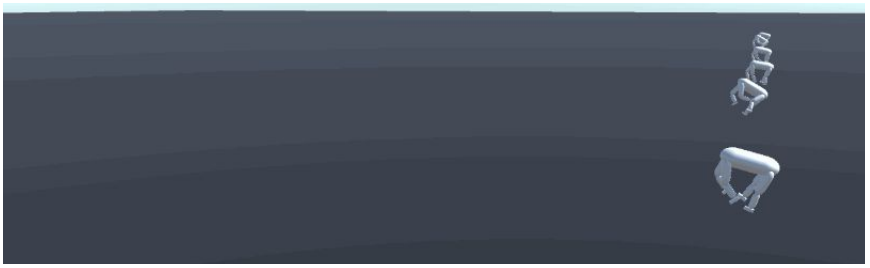
In order to train an agent using the Anaconda3 process, the machine learning algorithms need to be fed a settings sheet. An example sheet for all example scenarios in the SDK can be found within the repository’s root folder under ‘config’. The sheet is named ‘trainer_config.yaml’. This sheet was duplicated and renamed to ‘poly_config.yaml’ to freely cut it to this project’s needs without disrupting the example scenario settings. Some of the settings needed to be changed, for which the ml-agent docs were used to

understand the impact of each value and look for recommended value ranges. Due to the complexity of walking, the values responsible for the experience buffer were doubled; this includes the ‘batch_size’, ‘buffer_size’ as well as ‘time_horizon’, which was increased from 64 to 1000, as suggested in the docs when training with small periodic rewards, which applies to the polymorphic agents. The maximum amount of steps was raised to 2 million steps with a progress report every 3000 steps, since training the agents took a long time and 2 million turned out to be enough after running the training multiple times. In order to support the project’s complexity in the form of comprehending the various observations, the amount of hidden layers was raised to 3, while their depths were raised to 512. These values are within the recommended range that was given in the docs for more complex observations. Since the observations are not always the most accurate due to the nature of the state-driven physics system in Unity with fixed physics intervals, the observation normalization was enabled. With these settings set, the training phase could be started

4.2.10 Training

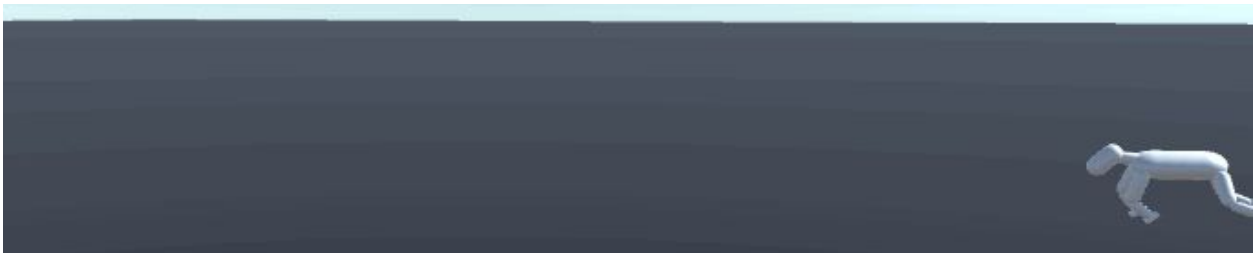
To start training, the ‘Anaconda Prompt’ was opened, which is the process used to control the Unity scene. If the environment was setup correctly, the ml-agents environment needs to be activated with ‘**activate ml-agents**’, followed by a navigation command to the ml-agents repository ‘**cd C:\[...]\Repositories\ml-agents**’. The process was then prepared to take control over a Unity scene with the command **‘mlagents-learn config/poly_config.yaml --run-id=PolymorphicAgent --train’**. The first part ‘mlagents-learn’ is the main command to run the training sequence, followed by the path to the previously created poly_config.yaml training settings sheet. The run-id is used to create a folder within the ‘models’ folder in the repository, along with the final trained brain. The last command ‘--train’ signals the training to start.

After 2 million iterations, the first brain was fully trained, resulting in the agents being able to move forward (gif 8: Four-legged agents moving forward).



Gif 8: Four-legged agents moving forward

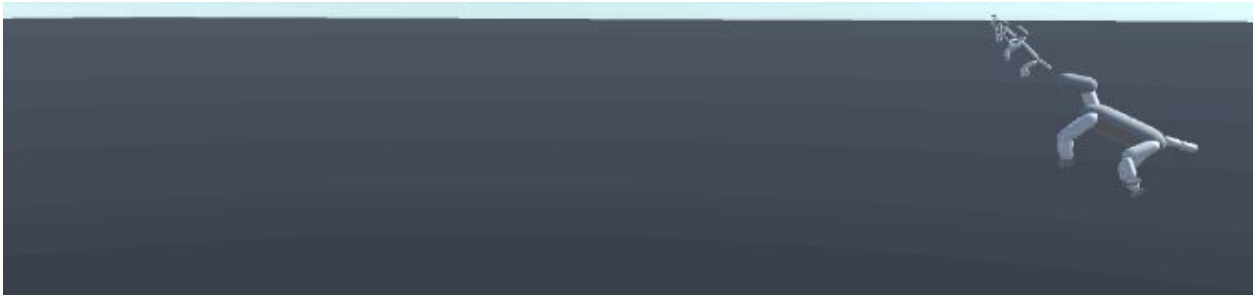
With the first experiment run being a success, the agent was expanded upon by adding a neck and a head to simulate a more realistic center of gravity for a dog-like animal. The training sequence was started anew with the same learning parameters (gif 9: four-legged agent with head).



Gif 9: four-legged agent with head

The resulting brain displays movement similar to the first one, though with more unstable moments. This can be due to the addition of the head and neck, or because of a difference in training experience. As a last change, a tail was added with two individual tail segments, meant to help the agent stabilize itself more reliably, like a tailed animal could (see Pradhan, 2015).

To test the addition of a tail, as well as the stability of training, the training sequence was run again with the same parameters for 2 million iterations (gif 10: Completed four-legged agent).



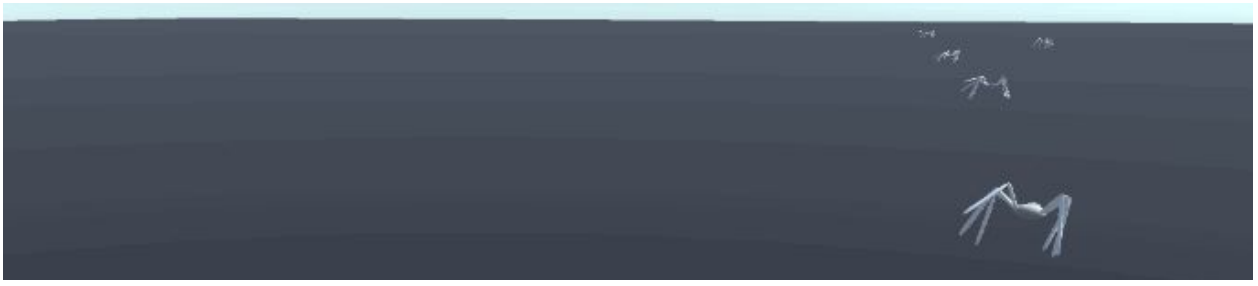
Gif 10: Completed four-legged agent

4.2.11 Expansions

With three successful tests on the four-legged agent, it needed to be expanded upon. A new agent with a different amount of legs, as well as a new environment are options for said expansion. Testing a new agent served as insurance that the modular agents work properly, as well as test their robustness in situations with a high amount of actions, while the environment provided tests for more advanced movements like jumping.

4.2.11.1 Agent

The new agent only needed to be built in the scene, since the scripts are modular. The goal was to create a spider-like creature with 8 individual legs. Each leg was similar to the four-legged agent's legs, only without feet and arranged slightly differently, to resemble spider legs. The batch of trained agents adapted a crab-like sideways walking strategy and could move towards the goal stablly without falling over (gif 11: Eight-legged agents moving 'forward') This behavior was shown repeatedly, even after multiple training attempts.



Gif 11: Eight-legged agents moving 'forward'

4.2.11.2 Environment

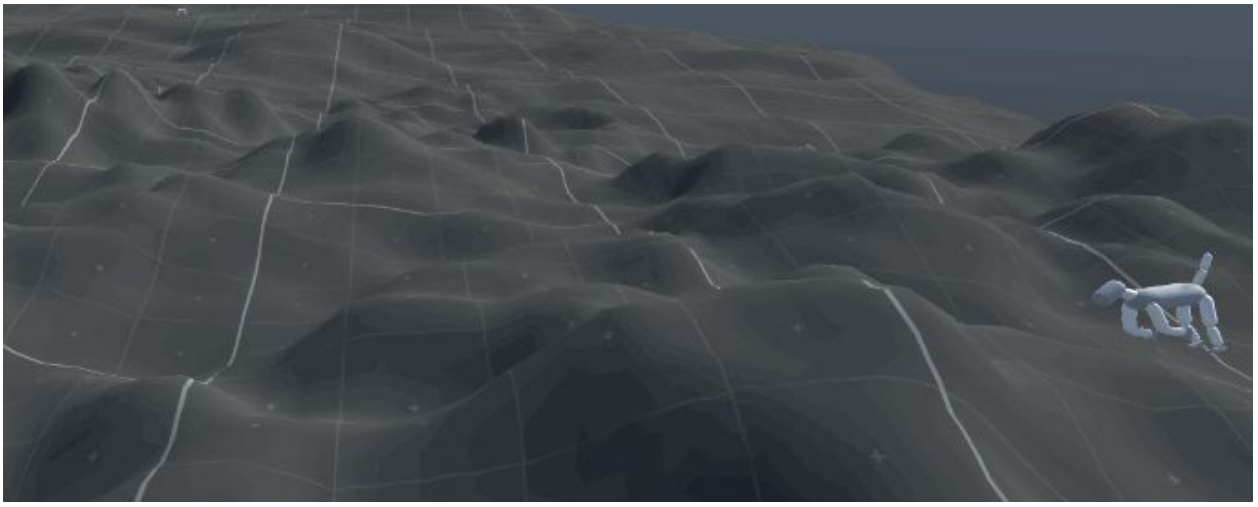
As an expansion to the environment, an uneven terrain was created with Unity's terrain tool. Its goal was to encourage more stable movement on hills and enable detection of ground shapes. To train more advanced movement for an agent, multiple stages are necessary, which were two in this case:

1. Train standard forward movement with chosen agent on flat terrain

2. After standard training, change terrain difficulty

Starting training in a difficult terrain would make learning the basics of movement difficult for the agent, thus an already trained agent is necessary to train properly. A new agent was trained from scratch, due to a change in the ground detection within the agent's code; every foot now scans the distance to the ground on the y-axis, instead of using a boolean that checks for ground contact. Having trained the new agent with the same procedure as done previously, the terrain was altered using a mountain brush with low opacity, to create small hills. The agents were placed to touch the terrain on their start cycle and the training was initiated. Since the Anaconda3 process should now load the partially trained agent, the command '**--load**' was required after the execution line.

Training stability in a hill-ridden environment was successful after 1 million training iterations, in addition to the 2 million iterations, resulting from the training for standard forward movement (gif 12: Four-legged agent on uneven terrain).



Gif 12: Four-legged agent on uneven terrain

5 Conclusion

5.1 Summary of Results

The required goals of the project have been reached;
a preset environment contains two agents with 4 and 8 legs respectively, which can navigate to their target goals and were implemented using the ML-Agents repository, which in turn utilizes Anaconda3.
The agents were implemented using the ML-Agents repository, utilizing Anaconda3.
They are aware of the environment through either ground detection or the distance from their feet to a surface. The agent makes additional observations about the rigidbody state, including rotational and directional velocity, as well as its position. The movement is performed via a normalized target rotation and a normalized force value.
Aside from the required goals, a more complicated environment with uneven ground was created to require agents to balance their bodies and thus reach their respective goals.

5.1.1 C# Machine Learning

The first part of the execution failed; the C#-based machine learning algorithm that was ported from javascript did not work properly, despite attempts to look for errors in the code or usage. The reason it failed is unknown, but it could have to do with either the structural difference of data types between javascript and C#, or due to an error in the porting procedure. The result was lost time and a malfunctioning project.

5.1.2 Four-Legged Agent

Moving on to using the ML-Agents repository was a success. The modular system was built to support many different types of n-legged agents, which proved to be in working order.
The first agent was provided with four legs, without head, neck or tail, just to test forward movement with a low count of observations and actions. The rotational reward worked, the

agents moved along their forward axis, while the velocity reward provided extra reward values for fast forward movement. The agents therefore tried to permanently run and would occasionally use too much strength, but still be stable enough to move for around 10 seconds.
The addition of a head was to provide a base for a type of visual system, that would allow the agent to see blocks in front of it, which ended up not being implemented. The addition of the head and neck, however, provided the agent with extra stability, due to being able to control the center of gravity, as well as its collective velocity more controllably.
This held true for the addition of a tail as well, which allowed the agent to reach its goal without falling over. The tail is visually being swung around to counteract the body's sway, so the agent adapted to the existence of a tail well.

5.1.3 Eight-Legged Agent

The second batch of agents was made to prove the abilities of the agent's modular system, which was done in the form of an eight-legged agent, built to look and work similar to a spider. The results from this agent were unprecedented, however; the expected spider-like forward movement did not occur, the agent moved sideways, favoring the speed reward over a correct rotational reward. This leads to assume that either the sideways movement is faster, or that the limbs restricted the agent in a way that made the crab-like movement faster than a spider's movement, this held true even after multiple attempts. The modular system now worked with two different agents, having different amounts of legs.

5.1.4 Unpredictable Terrain

To test a different environment, a terrain was created and altered via terrain tools, to resemble a hill-rich environment. The predictability of the environment was removed by placing multiple agents at different positions and in different situations within the environment, therefore forcing the brain to find a dynamic solution for every agent.
Further environments were not explored.

5.2 Critical Review of Execution

While the experimentation was overall successful, there was still room for improvement. Analyzing and rebuilding existing machine learning code was a helpful idea for learning the specifics of machine learning and allow the construction of a custom machine learning library, but the time investment was underestimated and therefore cost a large amount of time, which would have otherwise been invested in more agents and environment diversity; more research about machine learning libraries should have been made, to be able to estimate the required time for creating machine learning within C# and prevent time loss by abandoning a part of the project that had many hours invested in it.

The process of experimentation from a simple agent to a more advanced environment and agent was an effective approach, allowing a slow but steady progress, instead of trying to develop advanced mechanics like path-finding and parkour-elements right at the start.

To fully handle the topic of simulation in a realistic biomechanical manner, the implementation of an energy system would have been a feature, had the C# machine learning library not taken up too much time. The energy system would require agents to conserve energy and not permanently move their limbs, possibly resulting in more realistic behavior, especially during airborne phases, where current agents keep their limbs moving permanently.

Bibliography

Vision AI (2017) “Industry-leading accuracy for image understanding” [online] accessible via: https://cloud.google.com/vision/ [21.08.2019]

Boston Dynamics (2017) “Jobs” [archived online] accessible via: https://web.archive.org/web/20170528234112/http://bostondynamics.com/bd_jobs.html [21.08.2019]

Dormehl, Luke (2019) “What is an artificial neural network? Here’s everything you need to know” [online] accessible via: https://www.digitaltrends.com/cool-tech/what-is-an-artificial-neural-network/ [21.08.2019]

Weisstein, Eric W. “Hyperbolic Tangent” [online] accessible via: http://mathworld.wolfram.com/HyperbolicTangent.html [21.08.2019]

Zerium, Aegeus (2018) “Artificial Neural Networks Explained” [online] accessible via: https://blog.goodaudience.com/artificial-neural-networks-explained-436fcf36e75 [21.08.2019]

Simonini, Thomas (2018) “An introduction to Reinforcement Learning” [online] accessible via: https://www.freecodecamp.org/news/an-introduction-to-reinforcement-learning-4339519de419/ [21.08.2019]

Kurin, Vitaly (2017) “Introduction to Imitation Learning” [online] accessible via https://blog.statsbot.co/introduction-to-imitation-learning-32334c3b1e7a [21.08.2019]

Unity Technologies: Juliani, A., Berges, V., Vckay, E., Gao, Y., Henry, H., Mattar, M., Lange, D. (2018) “Unity: A General Platform for Intelligent Agents” [online] accessible via: https://github.com/Unity-Technologies/ml-agents [21.08.2019]

Python Software Foundation (n.d.) “What is Python? Executive Summary” [online] accessible via: https://www.python.org/doc/essays/blurb/ [21.08.2019]

Brown, Korbin (2017) “What is GitHub, and What Is It Used For?” [online] accessible via: https://www.howtogeek.com/180167/htg-explains-what-is-github-and-what-do-geeks-use-it-for/ [21.08.2019]

The One (2017) “Tutorial On Programming An Evolving Neural Network In C# w/ Unity3D” [online] accessible via: https://www.youtube.com/watch?v=Yq0SfuiOVYE [21.08.2019]

Kaparthy (2015) “REINFORCEjs” [online] accessible via: https://cs.stanford.edu/people/karpathy/reinforcejs/ [21.08.2019]

NVIDIA CUDA (2007) “CUDA Zone” [online] accessible via: https://developer.nvidia.com/cuda-zone [21.08.2019]

Pradhan, Rujuta (2015) “Why Do Animals Have Tails” [online] accessible via: https://www.scienceabc.com/nature/animals/why-animals-have-tails.html [21.08.2019]

Annex

The annex is contained on the added hard drive. The hard drive contains the following data in the folder “CMN6302_KuhlmannJan_MajorProject”:

- Abstract: Abstract in .pdf format
- Executable: Executables for Windows
- Major Project: Major Project in .pdf format
- Media: Collection of Gifs
- Online Sources: Online sources in .html format
- Project
    - C# Machine Learning: Full project for first part of the execution
    - ML-Agents: Full project for second part of the execution