ECE 275 Final Project - Fetch

Sangjoon Lee
U.C. Los Angeles
aaccjjt@g.ucla.edu

Taasin Saquib
U.C. Los Angeles
taasinsaquib@g.ucla.edu

Nima Zaghari
U.C. Los Angeles
nimazaghari@gmail.com

Abstract

We present Fetch - a simulation of a dog playing fetch with a bone. We set out to create locomotion and vision subsystems that, through neural network based learning, would allow the dog to track the bone, which will be thrown, and move towards it. Ml-Agents was utilized in order to draw from a wide variety of reinforcement learning algorithms to train the locomotion network. With regards to vision, we trained fully connected and convolutional neural networks that take unstructured pixel data as input. Time constraints prevented us from combining the subsystems, but we successfully trained the dog to walk using a neural network learning algorithm and utilized a retina neural network that can identify where the bone is.

1. Introduction

1.1. Motivation & Goal

We began the project as an exploration in how learning could be utilized to instantiate an artificial mammal. This paper is nowhere near a complete, comprehensive artificial life study of an animal, but the project is a stepping stone for more complex and realistic models of animals.

We first decided to study the areas of visual perception and locomotion. So, out of many animals, our team decided upon a dog because dogs have sophisticated eyes that require a high level of visual perception when compared to, say, a spider or salamander. Dogs also have a multi-jointed body with a center of mass that is located relatively higher than other animals, making balancing while locomoting a challenge. These characteristics of dogs made them a compelling choice that was both general and complex enough for us to explore the effects of learning in vision and locomotion.

Our goal was twofold: (1) create a vision module that is able to perceive the world and decide on an action, and (2) create a locomotion module that is able to activate multiple joints to carry out that action. As a more practical and short-term goal, we envisioned a simulation where a bone is thrown and the dog uses vision to perceive its trajectory. The dog follows the bone using the locomotion module while avoiding trees.

The output of the vision module, which is the direction that the bone is in relative to the dog, serves as input to the locomotion module. After determining where the bone is, the dog can decide how fast to approach its target and what gait to use.

1.2. Related Literature

There has been plenty of related literature on animal models that we took inspiration from. Ramakrishnananda 1999 [1] explored the aerodynamic principles for bird flight. We considered many parameters that were brought up in that paper, which too aimed to have an animal gravitate toward a target object.

There has also been work done on a musculo-mechanical model of a salamander [2]. This paper served to illustrate many of the notable gait parameters needed for walking. This proved to be helpful in our own experiment with getting the dog to move around.

Lastly, and perhaps most similar to our work, is a paper focused on locomotion skills for simulated quadrupeds [3]. Their model consisted of a flexible spine that connects front and back leg frames. Our work had a similar amount of links, with the additional feature of focusing on a vision task.

2. Methods

2.1. Unity

We decided to use Unity as our focus was on learning more about Artificial Life and not to create our own physics engine. However, we designed the model of the dog ourselves. This is because the design of the locomotive structure has very close ties to the control of the locomotive structure. Since learning encompasses control, this custom structure had to be designed. Additionally, we used three very helpful packages in Unity which are briefly described here. We trained our locomotion model using Unity's built in reinforcement learning package, called ML-

Agents. The library is relatively simple to use; we simply define an agent, inputs and outputs to the model, and can train many instantiations of our agent in parallel. The perception package allows for automatic creation of training data for computer vision. We planned to use bounding box labeled images to train a yoloV3 model to test our project workflow, but decided to skip this step to focus on developing the retina model. Perception also has a semantic segmentation labeler, which will be useful for the unstructured data that our retina will take in from the scene. Finally, we needed a way to train a neural network and then use it in our game. Barracuda enabled this functionality as long as we exported our network to ONYX format.

2.2. Physical Models

Here we describe the models that we built in Unity, which includes the overall scene, the dog, and the eye with a retina.

2.2.1 Park

We created a "park" that has our dog, a bone, and some trees. We experimented with the terrain component in Unity to make a more realistic scene, but we had no way of labeling the images automatically using the Perception package and therefore used simple block objects for trees. The bone obeys projectile motion, and the target shows up as a red box on the ground. The target is reset to a random location on subsequent throws to ensure that the dog does more than simply walk forward through reinforcement learning. The trees are generated randomly and are meant to be obstacles for the dog to avoid. Finally, we place the dog behind the bone so that it can do an initial scan of the scene to locate its target and hopefully follow the trajectory of the throw.

2.2.2 Dog

The dog's physical model is simplistic but captures most of the prominent features of a skeletal design of a dog. As shown in Figure 1, the model features 22 links and 18 joints. We divide its main body frame into three sections: Shoulder, Abdomen, and Pelvis. The head and tail are ornamental, and the 18 moving joints connect the shoulder, abdomen, and pelvis to each other and to each of their corresponding limbs.

The limbs were designed by referencing the model of a skeletal dog. Each limb is divided into four segments, which correspond roughly to thighs, calves, soles, and toes in a human leg. Each joint can move with a single degree of freedom (pitch). Assuming the angle for each joint is at zero degrees as in Figure 1, the range of each joint is given in the table below.

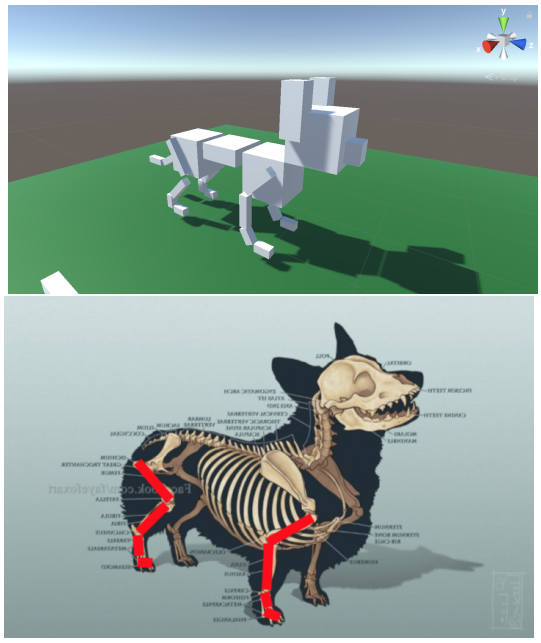


Figure 1: Dog Model design and Skeletal comparison

Col1	Min Degrees	Max Degrees
Abdomen	-50	50
Pelvis	-50	50
Front Thigh	-150	60
Front Calf	-150	60
Front Sole	-150	60
Front Toe	-60	60
Hind Thigh	-90	150
Hind Calf	-90	90
Hind Sole	-70	50
Hind Toe	-40	90

2.2.3 Eye Model

The goal for the vision subsystem was to, instead of using traditional computer vision algorithms to look at frames as input, create something more biologically inspired. We looked to a previous master's thesis [4] for inspiration on our retina model. Our retina casts rays into a scene to get a vector of pixel intensities, known as the optic nerve vector (ONV). We set out to teach our dog to identify bones and trees from this unstructured data. Given enough time, the plan was to then experiment with spiking neural networks by transforming the pixel intensities into spike trains [5].

Our model of the eye starts with a sphere in Unity. We resized the sphere to be about the size of a real eyeball relative to Unity units, but found that this model was difficult to train with our oversized scene. Therefore, we scaled up the eyes to have a diameter of 5 Unity units. The retinas

have some overlapping photoreceptors, but the distance between eyeballs is not to scale. From the center of the sphere, we utilize a noisy log-polar distribution to determine points to cast rays from. This gives us a vector of 11880 rgb intensities from each eye to work with as input to our neural network. The projected rays can be seen in Figure 2. We generate one such distribution and use it for each retina to make sure that data is input in a consistent order from different photoreceptors.

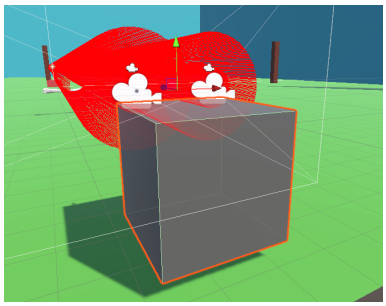


Figure 2: The retina model with visible ray casts into our scene

Our eye model is much more simple than the one used in Arjun's thesis; all of our rays are projected straight as there is no lens to refract the boundary ones. Also, the parameters for the noisy log-polar distribution are $\alpha = 41$ and $\theta = 360$ in the thesis, but we used $\alpha = 11$ to reduce the number of photoreceptors and make training our algorithms easier.

2.3. Locomotion

We used neural network based reinforcement learning to train the locomotion module. Supervised learning was out of the question because we did not have labeled data of a real dog walking. Even if we had access to this kind of data, the obvious differences between our simplistic model and a real dog would make data translation very difficult. One may suggest using forward and inverse kinematics to create labeled data between joint activation vs poses, but this too was not a viable option. The time invariant static pose alone could not solve the dynamically changing poses and balance that depended on previous states.

Instead, the agent relied on rewards and cumulative future rewards (value function) from reinforcement learning to decide whether a specific pose was good or bad. This way, the agent could generate a string of poses to eventually come up with the best sequence needed to locomote.

Unity's ML-Agents provides a diverse set of reinforcement algorithms including DQN, DDPG etc. Among these algorithms, we chose the state of the art, policy gradient algorithm known as Proximial Policy Optimization (PPO). The algorithm is a more sophisticated variant of a deep learning policy gradient algorithm that attempts to tune weights in a conservative manner to output the best policy that can maximize rewards.

We tried many variations of reward shaping, observation spaces, and action spaces to help the dog walk using PPO. However, each iteration still shared general neural network architecture hyperparameters and reinforcement learning hyperparameters. For the most part, the neural network in PPO consisted of 3 layers of 512 hidden neurons. The reinforcement learning hyperparameters were a replay buffer size of 20480, with batch size 2048 for weight updates. It used a learning rate of 0.0003, with exploration constant 0.2 and a future reward discount of 0.995.

2.4. Vision

The input to our vision neural network is two ONVs, one from each eye with dimension $\mathbb{R}^{11880}$. The output will be a vector of 3 values that correspond to the x, y, and z distance to the bone. If we can successfully determine where the bone is, we plan to add tree detection and conduct the aforementioned spiking neural network experiment.

To collect data, we simply ran our game and saved the ONV state while looking around at different objects through different angles, distances, etc. Each data point consists of, as mentioned above, 2 ONVs and a label that indicates how far the bone is from the center of the head. This label is set to (0, 0, 0) if the bone is not within the current field of view, so the model learns to differentiate when there is a bone and when there isn't. We collected around 2,000 training examples, of which 20% were set aside for testing and another 20% used for validation.

Next we chose a network architecture to implement this functionality. We trained two kinds of neural networks, one with convolution and one that was fully connected. We were interested to see if convolutional layers could combine inputs from each eye. To use convolution, we stacked the left ONV on top of the right one and used filters with height values equal to 2. We then added fully connected layers at the end, which reduced our output to 3 numbers. We started with two fully connected hidden layers but found that adding a third resulted in faster convergence to the desired output range.

We also implemented a fully connected network with 3 hidden layers. Our loss function was the mean squared error to the label vector. Both architectures are summarized in Tables 2 and 3.

3. Results

3.1. Locomotion result

It turned out that making the dog walk by learning was a very difficult task. We tried 8 different methods to help the dog walk. Each method consisted of a particular RL environment definition and a particular physical dog model

Layer(s) Type	Output Dimensions
Conv2d	(batch, 10, 1, 7001)
Dropout2d	(batch, 10, 1, 7001)
Conv2d	(batch, 50, 1, 5002)
Dropout2d	(batch, 50, 1, 5002)
Linear	(batch, 50, 1, 1000)
Linear	(batch, 1000)
Linear	(batch, 3)

Table 1: Retina CNN Architecture

Layer(s) Type	Output Dimensions
Linear	(batch, 1, 2, 8000)
Dropout2d	(batch, 1, 2, 8000)
Linear	(batch, 1, 2, 5000)
Dropout2d	(batch, 1, 2, 5000)
Linear	(batch, 1, 2, 1000)
Dropout2d	(batch, 1, 2, 1000)
Linear	(batch, 3)

Table 2: Retina Fully Connected Network Architecture

variant.

3.1.1 Initial Attempt: method 1 & 2

The initial attempt involved using the current physical model of the dog. We tried two RL environment definitions. The first method used 18 dimensional continuous action space [-1,1]x18 in additive manner, where each action dimension controls the relative degree of motion joint should make from its previous state ($\theta = \theta + \alpha[-1, 1]$). It takes in 21 observations at every timestep: they are 18 joint angles (θ) and body position defined by x,y,z. The reward is defined by the following.

```
at a time step:
    reward = + distance from origin
    reward = −1 if angle > joint range
```

The episode ends if the dog's body touches the ground. When we trained the RL neural net with this setting for 500,000 steps, the dog merely stretched forward and didn't bother walking. We quickly tried an action space of [-1,1] to span the entire range of each joint angle (i.e [-1,1] $\Longrightarrow$ [-150,60] for thigh θ) instead of additive definition. This did not give us good performance either.

Next we considered how the body moves in a cyclical motion while walking. We limited our actions to only increase joint angles in one direction until they reached the limit and reversed direction. This way the joints would swing back and forth between the maximum to minimum

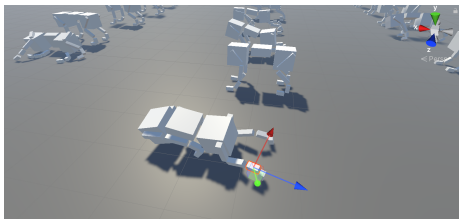


Figure 3: method 1: stretching dog

joint limits. In addition to this change, Since the dog needs to know which direction its limbs are moving in, we made the observation space a time series. We did this to include the current snapshot and previous state snapshot. We thought these additions would encourage the dog to learn to fully swing its limbs in sync and eventually walk. Fortunately, this enabled the dog to learn how to jump forward at the beginning. However, it did not enable it to walk after it landed on its stomach.

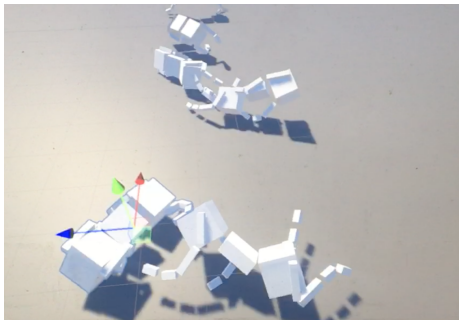


Figure 4: method 2: jumping dog

3.1.2 Reversal: method 3 & 4

We tried many variants with the above two methods. Unfortunately, none of them produced a satisfactory result. This discouraged us and made us think that maybe the physical model was at fault. There was the possibility that it was too complicated due to its many joints. On the other hand, there was the possibility that RL was implemented incorrectly. So, to test RL with a more simplified model, we decided to reduce the complexity of the problem by creating an even simpler version of the dog physical model.

As seen in the figure 5, we reduced the number of joints in the arm from 4 to 2. This meant less variables to control for the RL agent, which effecively reduces its search space. Also, to help the agent make a more informed decision, we added sensory feet that indicated whether they were touching the ground (0 = not touching,1 = touching). From this model, Method 3 was created. Method 3 is basically the same as Method 1 with the aforesaid changes + a reset trigger that occurs whenever the body rolls more than

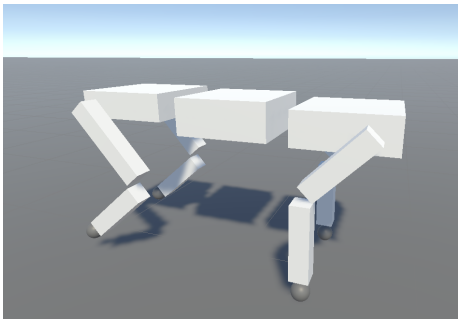


Figure 5: 2-jointed limb model

80 degrees, an indication that the dog fell on its side. RL still failed with this even simpler skeletal model.

To us, it appeared that the model was no longer the problem, but rather the fault was with the objective of the Reinforcement algorithm. We changed the objective to be more mathematically defined rather than simply relying on the distance from its original position as a reward.

The new objective was set by the following definition:

$$r = propulsion * correctdirection * bodyheight$$

$$propulsion = dot(velocity_{body}, forwardvector_{world})$$
$$correctdirection = dot(forwardvector_{body}, forwardvector_{world})$$
$$bodyheight = body.position.y$$

In addition to this reward, we gave a complementary reward for moving the feet forward. We gave a 0.1 balancing constant for moving the hind leg forward, and a 0.2 balancing constant for the fore leg.

$$r = r_{body} + 0.2 * r_{frontfeet} + 0.1 * r_{hindfeet}$$

Only with all these additional reward definitions did the dog start to walk. For the previously mentioned three RL methods and their variants, the mean reward grew as the dog learned to lurch forward. But after the first 100,000 training steps, the mean reward eventually plateaued (Figure 6), which meant the dog couldn't learn beyond this point even if we ran the training session for twice the amount we put in (Even if we trained for an additional 300,000 steps, the dog couldn't learn beyond taking one or two physical steps).

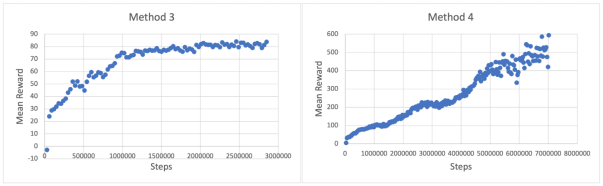


Figure 6: Method 3 & 4 reward over time

However, for method 4, the mean reward didn't plateau but consistently increased even after 500,000 training steps, which roughly meant the dog was learning to take more steps over time. To our amazement, Both the reward and the distance covered steadily grew even at 1 million to 2 million time steps. When the reward and distance grew without plateauing even at 7 million steps, we knew that RL was successful: the learning algorithm was going to teach the dog how to walk. We beheld the answer to our question about locomotive learning in artificial life.

3.1.3 Return: Method 5 & 6

With this huge success, we returned to the original dog model with 4 jointed limbs in method 5 with the same reward definition we used in method 4. Because we believed that the key to success in learning came from the redefined reward function and feet sensors, we implemented the very same features in the original dog model. With this set up, we ran the learning algorithm. Surprisingly, we didn't see success in the 7 million training steps.

Since the agent had to deal with a far larger action space (18 controllable joints compared to 10 joints in method 3 & 4), we gave the agent more time to train. But even after 20 million time steps, the dog couldn't make a single foot step.

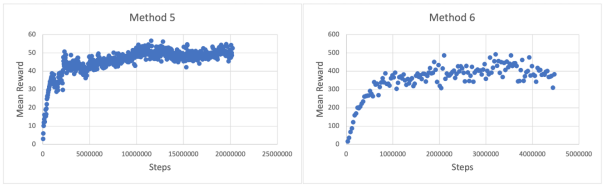


Figure 7: Method 5 & 6 reward over time

We tried to remedy this in method 6. We gave a greater incentive (x10) for moving the feet forward and also removed the penalty that comes from rolling because we thought it was making the dog conservative in its actions. But we didn't see any performance improvement.

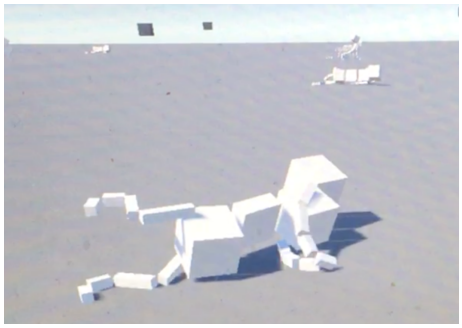
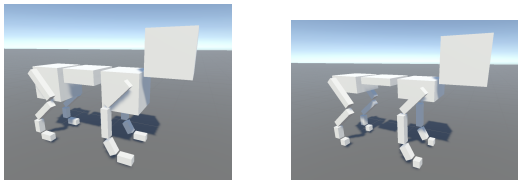


Figure 8: Method 5 & 6: dog that fell over; they are having a hard time getting back up

In both method 5 and 6, we noticed that the dog had a harder time getting back up once it rolled over to the side (compared to agents in methods 3 & 4, where the dog was able to right itself when it almost fell over). We thought that maybe the dog model's block shaped body was making it difficult to stand up again.

3.1.4 Slim body: method 7 & 8



(a) original dog model (b) slim body model

Figure 9: Comparing original model with slim body model

A new prototype model was developed where we halved the chest, abdomen, and pelvis thickness. We were hoping to stop the block-shaped nature of the chest and pelvis from preventing the dog from getting back up. The length of the toes was also halved as they seemed too long for a typical dog. They also seemed to merely drag across the terrain, making it more difficult to walk.

None of these improvements enabled the dog to walk even after training for millions of time steps. Each method was trained for 7 million steps and 14 million steps, respectively. But the graphs plateaued very early on, indicating a early halt in learning.

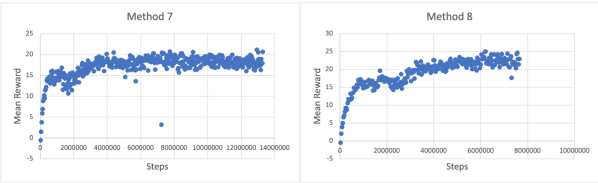


Figure 10: Method 7 & 8: Reward over time

3.1.5 Locomotion Result Overview

The details of each implementation on both hardware and reinforcement learning objectives are given in the table "Summary of 8 methods".

When plotted, each method had a different magnitude of mean reward due to how they have different definitions of reward objectives. But, when we normalize their reward over time, we can make a comparative analysis (Figure 11). As shown in this graph, even when we gave a greater training period for the dog model with 4 jointed limbs, they hit a

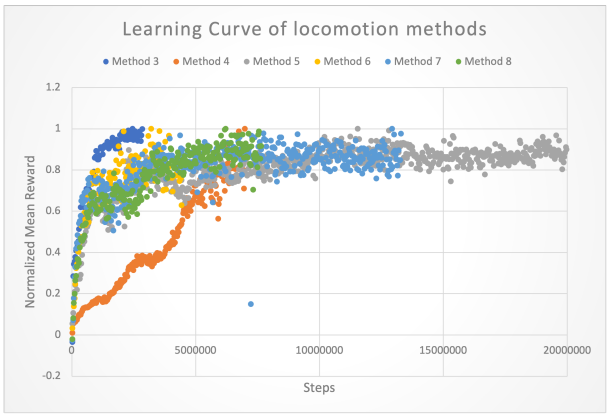


Figure 11: Normalized mean reward of 8 methods

plateau early on. This suggests that the reinforcement learning that does not work early on will likely not work even when trained for longer time.

In contrast, method 4, which was a successful run with 2 jointed limbs, created a steady increase in reward without any sign of plateauing during the entire duration of training period. This steady increase shows potential for learning, and it also shows that had we extended its learning period, it would have likely continued learning in steady fashion.

Method	Maximum Distance Covered
1	1.330459
2	2.483249
3	1.726174
4	10.69055
5	2.526709
6	2.824836
7	2.307596
8	2.834915

Undoubtedly, when we measured the distance covered by each method, Method 4 was the most successful.

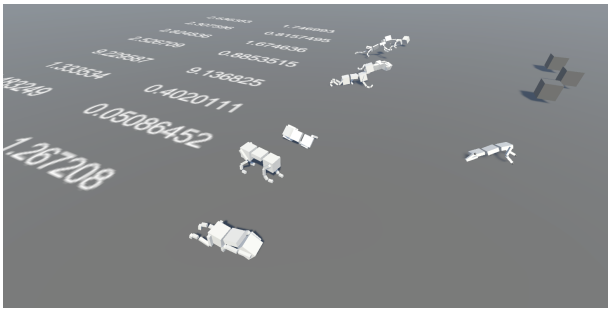


Figure 12: Testing distance for each methods

From this result, we can understand what model has potential for learning to work. Obviously, not all models do.

	Reinforcement Learning Environment				Physical Model
Method	Action	observation	reward	Reset trigger	
1	18; additive	21; 18 joints + pos(3)	+ [dist from origin] -1 for triggering reset	Exceed Joint range; Body Touch ground;	Original
[1.5]	18; range	“”	“”	“”	“”
2	18; additive, cyclic	48; time series; 18 joint + pos(3) + orientation(3) x 2	+ [dist from origin] -1 for triggering reset	Body Touch ground; Abs(Roll) > 80 degrees	“”
3	10; additive, cyclic	20; 10 joints + pos (3) + orientation(3) + Feet touch ground (4)	“”	“”	2 jointed
4	10; additive	“”	+ $v_{forward\_body}$ * $direction_{[-1,1]}$* $height_{body}$ + v_{bind_feet} *0.1f + v_{Fore_feet} * 0.2f -1 for triggering reset	Exceed Joint range; Abs(Roll) > 80 degrees Reward < 1 for 1000 steps	“”
5	18; additive	28; 18 joints + orientation(3) + correct Direction(3) + Feet touch ground(4)	“”	“”	Original
6	18; additive (joint angle speed x 2.5)	“”	+ $v_{forward\_body}$ * direction* $height_{body}$ * $Roll_{[1,-1]}$ + v_{bind_feet} *1f + v_{Fore_feet} * 2f	Exceed Joint range; Reward < 1 for 1000 steps	“”
7	“”	“”	“”	“” + Abs(Roll) > 120 degrees	Slim Body
8	“”	“”	“”	“” + Abs(Roll) > 80 degrees	Slim Body

Figure 13: Summary of 8 methods

Neither does any objective work. We can understand that while learning can be used to train models to locomote, both the physical structure of the agent as well as the reward objective must be carefully defined. The model cannot simply learn if either of the two element is poorly defined. Furthermore, we can conclude that there are myriads of factors that can both hinder or boost RL algorithms performance. It is very difficult to logically deduce which factors will lead to success. In this case, a smaller continuous action space and carefully designed reward space seemed to be the most important for using learning for artificial life. But, any other limitation could have been the key to success or failure. For instance, perhaps it was a lack of a roll joint in the simplified dog model that prevented the model from righting itself. For this reason, we can analyze that learning cannot be applied to any kind of model. Simplified imitation of a physical model may not be sufficient ground for learning to achieve success in locomotion for artificial life.

3.2. Vision Results

In vision, we ran into a Barracuda error that prevented us from running our model within our game and doing real-time inference. Therefore the vision results compare the test set performance between each of our network architectures. We had two data sets. one data set was collected raw from Unity. The other data set was manually created where we replaced the label with all zeros if the bone was not hit by any of the projected rays.

To analyze our outputs, we developed two metrics. The first metric - difference - tells us how many of our predicted labels were within 0.5 unity unit of the correct value across x,y,z dimension. We chose 0.5 as it was within the range

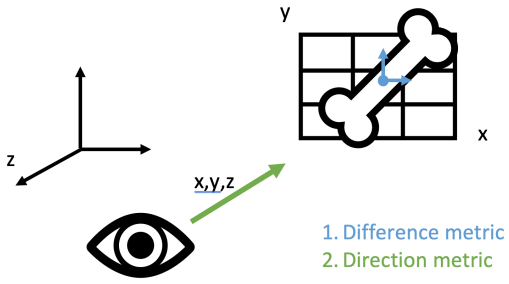


Figure 14: Difference metric and Direction metric pictogram

of error the dog could tolerate to play the game fetch. The second metric - direction - conveys whether the neural network is able to pick the right direction in which the agent should move to bring the bone to the center of the screen. Direction was a more loose metric to judge with, and this was our substitutionary feature that could be implemented for eyes shift in the future dog model. The accuracy of the trained networks are judged according to these metrics and are reported in Tables 3 and 4.

Network	X	Y	Z
Conv	0.143	0.023	0.0139
FC	0.804	0	0.686

Table 3: Difference Metric Results

Network	X	Y	Z
Conv	0.162	0.016	0.025
FC	0.827	0.838	1

Table 4: Direction Metric Results

As we can see, the fully connected network outperforms convolution for both metrics in almost all dimensions. Only in the y-dimension for difference metrics does the FC network actually get zero correct. This was an unexpected result. One explanation for zero accuracy across y axis is the lack of comprehensive data of bone distribution across y axis. Even though the bones were thrown across wide range of positions, their y position may have tended to be same when we collected the data.

The CNN's comparatively worse performance could be attributed to its filter design. We believe our convolution filters were often too large and could have had difficult time learning smaller features. For direction, Z predictions may have yield high accuracy because in all the dataset the bone is almost always in front of the dog or not visible. The accuracy revealed in difference metric for Z axis, however, indicates that supervised learning algorithm was able to predict some distance accurately.

4. Conclusion

While we weren't able to combine the locomotion and vision subsystems to complete our game of fetch, we were able to have most of the modules built out. We believe that the foundation we laid through this project will enable us to simulate and build more accurate form of artificial life. It was an exciting experience for us to create and train these models firsthand, and we were able to gain better understanding of how learning can be applied in artificial life. Primarily we were able to see the extent of learning as it relates to physical model and reward definition. Both good design and well-defined control objectives are key to using NN learning in simulating artificial life.

4.1. Future Work

For future work in locomotion, we would like to explore other areas of deep reinforcement learning algorithms. Proximial Policy Optimization (PPO) is state of the art algorithm. But, we would like to compare its performance with other learning algorithms, both from reinforcement learning (e.g DQN) and outside (e.g evolutionary algorithm). Now that the simplified model has been tried and implemented, we want to also try making a more realistic skeletal muscular model of the dog. We want to compare its performance because we think that perhaps the model wasn't able to learn very well because the model lied in some awkward range between too simple and too realistic. Perhaps using more spinal joints may have helped. In addition, using roll and yaw joint may have helped the dog learn to balance better.

For visual perception, we would like to enable Barracuda to run our models within Unity. This would complete our pipeline and allow us to see how the networks perform with real time data and verify predicted output directions and distances. We would also like to create a better dataset with more varied positions for the bone in the scene. With access to a more powerful machine, we would like to create a deeper CNN with smaller filters to truly test convolutions with unstructured data. Finally, we want to convert the ONV to time series data of neural spikes and convert our retina NNs to spiking neural networks to see if we can create object detection using spiking domain data.

References

[1] Balajee Ramakrishnananda and Kok Cheong Wong. Animating bird flight using aerodynamics. *The Visual Computer*, 1999.

[2] Nalin Harischandra, Jean-Marie Cabelguen, and Orjan Ekeberg. A 3d musculo-mechanical model of the salamander for the study of different gaits and modes of locomotion. *Frontiers in Neurorobotics*, 2010.

[3] Stelian Coros, Andrej Karpathy, Ben Jones, Lionel Reveret, and Michiel van de Panne. Locomotion skills for simulated quadrupeds. *ACM Transactions on Graphics*, 2011.

[4] Arjun Lakshmipathy. Biomimetic modeling of the eye and deep neuromuscular oculomotor control. Los Angeles, California, 2018. UCLA.

[5] Evangelos Stromatias, Miguel Soto, Teresa Serrano-Gotarredona, and Bernabé Linares-Barranco. An event-driven classifier for spiking neural networks fed with synthetic or dynamic vision sensor data. *Frontiers in Neuroscience*, 11:350, 2017.