ECE 275 Final Project - Fetch

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

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

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

Abstract

The ever-expanding field of artificial life has presented many realistic animations of animals. Here, we present Fetch: a dog simulation model that aims to have a dog play fetch with a bone. Through neural network based reinforcement learning, the dog is able to make the correct action in regards to retrieving the bone. Ml-Agents was also utilized in order to draw from a wide variety of reinforcement algorithms. With regards to vision, we used 2 ONVs, one for each of the dog. Results stuff.

describe fetch game train locomotion and vision results - slightly walk and vision NN results coming in

1. Introduction

1.1. Motivation & Goal

This project began as a study of learning in animal simulation to help understand how learning can be utilized to instantiate artificial life form of mammals. This paper is nowhere near a complete, comprehensive artificial life study of an animal, but the project began as a first stepping stone that can serve as a cornerstone for more complex and realistic modeling of an animal.

As the first stepping stone, this project attempted to study the area of visual perception and locomotion. So, out of many animals, our team decided upon dog because dogs have sophisticated eyes that requires high level of visual perception to understand the world, compare to say spider or salamander. Also, Dogs have multi-jointed body part with center of mass that are located relatively higher than other animals like turtle or snake, which means it needs to solve non trivial task of balance while locomoting its body frame. These characteristics of dogs made it a compelling choice of model that was general enough and complex enough for us to explore the effect of learning in vision and locomotion.

As the goal for this project, we wanted to explore diverse set of vision and perception simulation and come up with two module: (1) vision module that is able to perceive the world and decide on its action, (2) locomotion module that is able to activate its multi joints to carry out that action. As a more concrete, short-term, practical goal, we envisioned a project called Fetch where the simulated dog uses vision module to perceive bone trajectory to follow it while avoiding trees, and locomotion module that can move its body according to the planned direction to finally fetch the bone.

Though we divided the simulation into two module, We necessarily didn't define what kind of vision module or locomotion module we will use. We did this so that we would have greater freedom in exploring different idea for each modules, as long as in the end, these module can talk to each other in a consistent interface. The interface is two variable: direction and speed. The vision module return these two values as output, and locomotion module consumes them as input. But aside from these constraints, any experimental module can be studied and utilized. Design and details of these modules will be given in the Method section.

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 was 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, the 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, including the overall scene, the dog, and the retina.

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.

The dog's physical model is simplistic. As shown in the Figure 1, the model features 22 links and 18 joints. The model divides up it main body frame into three section: Shoulder, Abdomen, and Pelvis. the head and tail is ornamental and the 18 moving joints connect 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 to four segments, which corresponds roughly to thigh, calves, sole, and toe in terms of human leg. Each joints can move with single degree of freedom (pitch). Assuming the angle for each joints is at zero degrees in figure 1, the range of each joint is given in the following table.

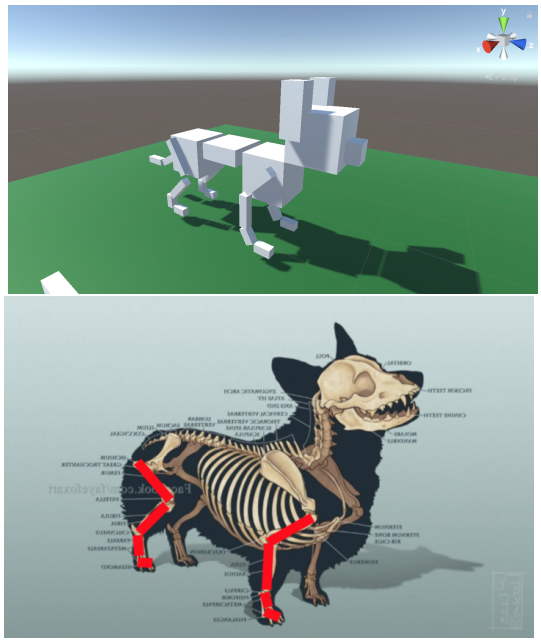


Figure 1: Dog Model design and Skeletal comparison

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

Instead of making a hyper-realistic model of a dog, the authors of this project wanted to start this project simple, and more like a first stepping stone and guided direction to build on top of. Hence, we decided to upon this more simplistic model of dog that is created with cubes and spring joints while still capturing most prominent feature of skeletal design of a dog. We were hoping that by being able to apply neural network learning in simplistic model, we would be able to extrapolate the same technique on more complicated, close to life like physical models.

Our model of the eye starts with spheres in unity. We resized the spheres to be about the size of a real eyeball rel-

ative to Unity units, but found that this was difficult to train given our time constraint and so we gave the eyes larger diameters than in real life. 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 rgb intensities to work with as input to our neural network.

The unity model is much more simple than the one in the thesis, including the fact that all of our rays are projected straight and there is no lens to refract the boundary ones. Also, the given parameters for the noisy log-polar distribution are $\alpha = 41$ and $\theta = 360$, but we reduced α to 11 to reduce the number of photoreceptors and make training and verifying our algorithms a bit easier.

2.3. Locomotion

The primary learning algorithm Locomotion used was the neural network based reinforcement learning algorithm. supervised learning was out of the option because we did not have labeled data of how the real dog walked. Even if we had access to these data from real dogs, our simplistic model of the dog could not easily correlate to the organic structure of the dogs. 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, time variant balance and motion that is evident in walking. Hence neural network based supervised learning and unsupervised learning did not seem competent in light of reinforcement learning.

So instead of relying preexisting data to learn how to walk, the agent relied on reward and cumulative future reward (Value function) to help realize whether taking one pose is good or bad to eventually come up with the best sequence of action it needs to take to locomote itself.

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

We tried many variation and improvement of reward shaping and observation space and action space to help the dog walk using PPO. However, they still shared general neural network architectures 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 hyper parameter were replay buffer size of 20480, with batch size 2048 for weight updates. It used learning rate of 0.0003, with exploration constant 0.2. and future reward discount of 0.995.

2.4. Vision

The input to our neural network will therefore be two ONVs, one from each eye. The output will be a vector of 3 values that correspond to x, y, and z distance to the bone. If we can successfully determine where the bone is, if it is in view, then we will 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.

Here we will discuss our choice of network architecture. We considered a fully connected network, but this would have resulted in a very large number of trainable parameters and would have been difficult to run alongside our game. We also wanted to experiment with convolutional layers to combine certain 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 could not avoid the fully connected network at the end, which we used to reduce our outputs to 3 numbers. We started with two fully connected layers but found that adding a third resulted in faster convergence to the desired output range. This architecture is summarized in Table 1. The network was created using pytorch and the optimizer was adam. We started with a learning rate of 0.1 as the loss was very high, but we moved to 0.01 and 0.001 if we saw that training was struggling to make progress.

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

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

Table 3: Difference Metric Results

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

Table 4: Direction Metric Results

Architecture	Training	Validation	Test
2xCNN	87.83%	54.37%	49.21%
RNN	95.39%	61.47%	54.63%
LSTM	95.33%	64.54%	58.92%
GRU	94.68%	66.90%	60.72%
CNN + RNN	91.67%	65.25%	50.56%
CNN + LSTM	89.60%	63.36%	54.85%
CNN + GRU	87.12%	64.78%	58.69%
2xCNN + GRU	87.71%	71.40%	70.14%

Table 5: Training, Validation, and Test Accuracies for Evaluated Architectures. Note that all architectures are followed by a Fully Connected Network, and that all RNN architectures are bi-directional.

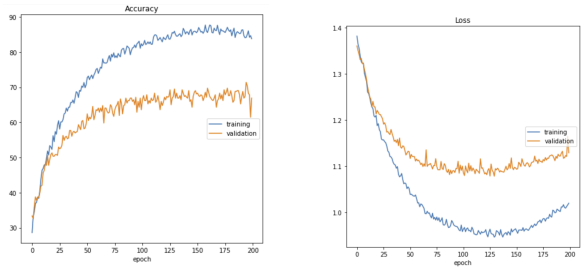
3. Results

4. Conclusion

4.1. Future Work

Throughout testing we were able to attain a large amount of data for the individual models and their combinations. These results are listed below in Table 1 where we can see the growth and change across different architectures. We found that individual models have relatively good training and validation accuracy, averaging around 55-60%. Test accuracy, however, averaged around 55% while our target was to reach 70%.

Once we begin to combine models we saw that simply adding more CNN layers didn't increase testing accuracy significantly. When we started adding RNN layers to the CNN architecture we saw gradual improvement in test accuracy, with the CNN+GRU model performing the best with 70.14% test accuracy. More specifically, this model used two CNN layers in series with a bidirectional, two-layer GRU. Furthermore, the model consisted of several batch normalization, dropout, and pooling layers, which will be summarized in the Methods and Architecture section. Below, in Figure 1, we can see the plot of accuracy and loss changing over time as we are able to achieve a test accuracy of above 70%.



(a) Accuracy (b) Loss

Figure 2: Training and Validation Results For Optimized 2xCNN + GRU Classifier

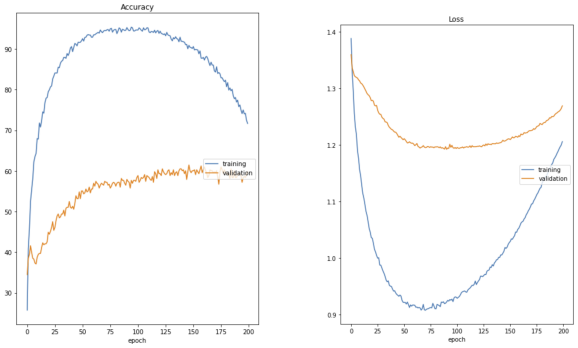
5. Discussion

5.1. Model Distinction

Our model choices investigated three Recurrent Neural Network architectures as opposed to optimizing a simple CNN, in an attempt to determine the optimal RNN configuration for this task. This strategy allowed us to determine if our Recurrent Neural Network Architectures would benefit if CNN layers preceded them. This section will break down these evaluations.

The inclusion of Recurrent Neural Networks was due to the extensive research on the usage of RNNs[5], LSTMs[2],

and GRUs[3][4] in accordance with EEG data. As a starting point, we implemented a simple vanilla RNN that allowed us to attain roughly 95.39% training accuracy, 61.47% validation accuracy, and 54.63% testing accuracy. The very high training accuracy suggested significant overfitting, portrayed in Figure 2 below. Note that the plot does demonstrate that the training error decreases over a short period of time, which is primarily due to our inclusion of dropout.



(a) Accuracy (b) Loss

Figure 3: Training and Validation Results For Optimized RNN Classifier

Additionally, we found that the LSTM and GRU have nearly identical results with training accuracies of 95.33% and 94.68% (high overfitting) respectively, validation accuracies of 64.54% and 66.90% respectively, and testing accuracies of 58.92% and 60.72% respectively. Unfortunately, literature research and our own empirical results show that implementing regularization and personalization on both the LSTM and the GRU will not actually improve the accuracy with EEGs[2]; these architectures in isolation won't work as a final optimized classifier. Instead, our literature research demonstrated that CNN layers preceding these RNNs will actually improve the accuracy on the EEG dataset[3][4].

Further research also compared LSTMs and GRUs to demonstrate the preferred RNN for the EEG dataset. Based on generalized empirical results, the LSTM appears to be a better model for larger datasets because it is much more complex with more required gates.[1] However, for our task, we actually found GRUs to offer slightly better validation and test accuracies. Since the best performance of the individual models came from the GRU, it became our preferred classifier as we created and tested these multi-layer combinations with CNNs.

After optimization of both CNN and RNN architectures on the EEG dataset, combining the two allowed us to greatly optimize our results.

We first tested the combined architecture of one CNN layer in series with an RNN layer and a final FC network. This combination resulted in a training accuracy of 91.67%, a validation accuracy of 65.25%, and a testing accuracy of 50.56%. In comparison to the regular RNN implementation, we actually see that including the CNN layer improved overfitting as we see that the training accuracy is lower, but it still had a lower than desired testing accuracy. It was interesting to see that the inclusion of a CNN layer did help our model generalize better, as found to be thoroughly tested through research[4].

Due to this discovery, we implemented similar architectures with the LSTM and the GRU models. With the improved accuracy of the LSTM and GRU along with the generalizing ability of the CNN layer, we expected to see a higher test accuracy with a lower difference to show a lack of overfitting.

The CNN + LSTM + FC model resulted in a training accuracy of 89.60%, a validation accuracy of 63.36%, and a testing accuracy of 54.85%, and the CNN + LSTM + FC model offered noticeably better test results as shown in Table 1. While the overfitting issue was slightly improved upon, we still didn't have enough of an improvement on the validation or testing set to deem this model our most optimized.

Noting that we received the best results with the GRU, we spent significant amounts of time optimizing this architecture. Finally, we achieved our best results with a 2-layer CNN + 2 layer GRU + FC model. The 2xCNN + GRU + FC model,resulted in a training accuracy of 87.71%, a validation accuracy of 71.40%, and a testing accuracy of 70.14%. As shown previously in Figure 1, this architecture offers a more consistent reduced training loss and accuracy, resulting in our maximum test accuracy.

5.2. Hyperparameter and Architecture Optimization

We began our analysis by evaluating all subjects in the dataset, rather than evaluating each subject individually. Through this method, we hoped to find a model that would better generalize over various subjects rather than inadvertently overfit to a specific subject. Furthermore, we evaluated the temporal dependencies in the dataset by adjusting the hyperparameters in the CNNs and RNNs, rather than manually adjusting the number of time bins used in classification.

To test a certain architecture, we varied the learning rate by simply trying different values and choosing the model with the highest validation accuracy. We tuned other hyperparameters such as number of hidden units for RNNs in a similar way. In general we tuned each value independently as doing a grid search over many values would be very time intensive. Finally, we trained each model for 200 epochs but used the model with the highest validation accuracy on

the test set.

The two-layer CNN architecture utilized in our final architecture was designed by referencing the provided Shallow Convolutional Network. By adjusting the filter sizes and count, incorporating batchnorm, pooling, and dropout after each layer, and introducing deeper layers, we were able to optimize a CNN architecture for a CNN + GRU architecture. We found that using larger filters in the first convolution and pooling layers helped increase test accuracy, which is expected as this reduces our data dimensions early on and helps avoid "memorizing" the data as much. Increasing the stride of our first pooling layer also had a similar effect.

After experimenting with different combinations, we found that the best results came when our layers were stacked in this order: conv-batchnorm-pool-dropout. Increasing p in subsequent dropout layers also helped to generalize our model.

For the RNN architectures we found optimal results by incorporating two layers of GRUs, which essentially consists of 2 GRUS in series. Furthermore, we found that 15 feature dimensions allowed all the RNN models best capture the temporal dependencies of the data. All of our optimal RNN architectures were bi-directional, heavily suggesting that the EEG dataset actually contains reversed classifiable temporal sequences, which played a key role in our final test accuracy. Finally, we incorporated dropout in all RNNs used to reduce overfitting and approximate ensembling, utilizing dropout with a probability of 0.3 in the optimized GRU architecture.

These architectures, including the optimized architecture, were then followed by another dropout and batchnorm layer before going into a Fully Connected Network. We experimented with various activation functions, notably tanh, but found the optimal results with relu for all activations. When trying to use the square and log activation functions as seen in the given ShallowConv network, we ran into computational problems as our loss function became "nan" and resulted in zero-ed out gradients and no learning.

6. Conclusion

We successfully achieved 70% testing accuracy with our combination of CNN and RNN architectures. This confirms our hypothesis that CNNs help with spatial locality and regularization while RNNs provide temporal recognition of neural signals. Techniques such as batch normalization and dropout in combination with intensive hyperparameter and architecture optimizations enabled us to design our general classifier. In future work we hope to develop a classifier utilizing completely different architectures such as GANs and VAEs, again building our designs off of previous literature.

7. Appendix: Methods and Architectures

In this section we outline our various architectures. Note that all other RNN architectures presented in Table 1 are analogous to these architectures, yet result in lower performances.

Layer(s) Type	Output Dimensions
Permute	(batch, 1000, 22)
2xGRU	(batch, 1000, 30)
Dropout	(batch, 1000, 30)
Batchnorm1d	(batch, 1000, 30)
Linear	(batch, 4)
Softmax	(batch, 1)

Table 6: Bi-directional Two-layer GRU Architecture

Layer(s) Type	Output Dimensions
Conv2d	(batch, 40, 22, 976)
Permute	(batch, 976, 22, 40)
2xGRU	(batch, 976, 30)
Dropout	(batch, 976, 30)
Batchnorm1d	(batch, 976, 30)
Linear	(batch, 4)
Softmax	(batch, 1)

Table 7: CNN + Bi-directional Two-layer GRU Architecture

Layer(s) Type	Output Dimensions
Conv2d	(batch, 10, 22, 901)
Batchnorm2d	(batch, 10, 22, 901)
Dropout2d	(batch, 10, 22, 901)
Relu	(batch, 10, 22, 901)
AvgPool2d	(batch, 10, 22, 401)
Conv2d	(batch, 100, 18, 397)
Batchnorm2d	(batch, 100, 18, 397)
Dropout2d	(batch, 100, 18, 397)
Relu	(batch, 100, 18, 397)
MaxPool2d	(batch, 100, 18, 196)
Permute	(batch, 196, 18, 100)
2xGRU	(batch, 196, 30)
Dropout	(batch, 196, 30)
Batchnorm1d	(batch, 196, 30)
Linear	(batch, 4)
Softmax	(batch, 1)

Table 8: Optimized Two-Layer CNN + Bi-directional Two-layer GRU (Final Classifier)

References

[1] Samuel P. Babaro. *Recurrent Neural Networks: building GRU cells VS LSTM cells in Pytorch*. 2020. URL: https://theaisummer.com/gru/.

[2] Erik Bresch, Ulf Großekathöfer, and Gary Garcia-Molina. "Recurrent Deep Neural Networks for Real-Time Sleep Stage Classification From Single Channel EEG". In: *Frontiers in Computational Neuroscience* (2018). DOI: https://doi.org/10.3389/fncom.2018.00085.

[3] Isuru Niroshana et al. "Sleep Stage Classification Based on EEG, EOG, and CNN-GRU Deep Learning Model". In: *2019 IEEE 10th International Conference on Awareness Science and Technology* (2019). DOI: https://doi.org/10.1109/ICAwST.2019.8923359.

[4] Theerawit Wilaiprasitporn et al. "Affective EEG-Based Person Identification Using the Deep Learning Approach". In: *IEEE Transactions on Cognitive and Developmental Systems* (2020). DOI: https://doi.org/10.1109/TCDS.2019.2924648.

[5] Shixiao Xu et al. "Using a deep recurrent neural network with EEG signal to detect Parkinson's disease". In: *Annals of Translational Medicine* (2020). DOI: 10.21037/atm-20-5100.