Report

June 29, 2019

1 Navigation Project

This notebook describes the results of the reinforcement learning project. Deep Reinforcement Learning Nanodegree.

1.1 Models

In this project, we evaluate five different architectures. The baseline DQN architecture, the Double DQN architecture, the Double DQN architecture with Prioritized Experience Replay (PER), the Dueling DQN architecture, and finally the Dueling DQN architecture with PER. Each of these architectures use the same hyperparameters to ease the comparison between architectures noting that those models with PER have two additional hyperparameters. *Note: as implemented, the Dueling DQN architecture implies Double DQN as well.*

1.1.1 Neural Networks

DQN Neural Network The DQN neural network model is used for the Baseline DQN, Double DQN, and Double DQN + PER architectures. This network is a simple feed-forward network with two hidden layers. Each of those two hidden layers is 128 nodes wide. The input feature vector has a width of 37 and the output action vector has a width of four corresponding to the four possible actions that the agent may take. Each of the two hidden layers is followed by a ReLU layer.

Dueling DQN Neural Network Dueling DQN is described in this paper. The Dueling DQN network starts with the same two 128 node wide hidden layers. The outputs of the second hidden layer are passed into two separate networks. The first network, the value network, adds a 32 node wide hidden layer followed by a single activation node representing the value of the best action. The second network, the advantage network, adds a 32 node wide hidden layer followed by a 4-node wide activation layer. The relative advantage of each of the nodes in the advantage layer is computed by subtracting the mean advantage from each node. Finally, the value of the best action is added to the relative advantage of each action to produce the output vector.

Optimizer The Adam optimizer is used in all cases. A description of the Adam optimizer may be found here.

1.1.2 Replay Buffer

Two replay buffers are used. The source code for the buffers is taken from the OpenAI baseline using the MIT license. The baseline replay buffer was defined in the DQN paper. It is simply a fixed length buffer from which the agent may select a batch of random samples.

The Prioritized Experience Replay buffer stores each sample with a priority corresponding to the error in the calculated value of the action versus the prediction. The idea is that the amount of learning available from a sample is higher from those samples with higher errors. To reduce the priority given to samples with higher errors, the error is raised to an exponent with a value less than 1. The sampling priority for each sample is then normalized across all samples in the buffer. A second hyperparameter, beta, allows for the amount of prioritization to increase as more and more training samples have been seen. Prioritized Experience Replay is defined here.

1.1.3 Agent Design

Two identical neural networks are constructed within the agent. These are the *target* and *local* networks respectively. The local network learns the optimal parameters while the target network is periodically updated with the parameters from the local network. The agent provides two methods to form the external API.

The *act* method returns the recommended action. With some probability, epsilon, the agent returns a random action.

The *step* method takes the state, action taken from this state, the next state following execution of the selected action, the reward provided by the environment, and whether or not the terminal state has been reached. These five parameters are added as a tuple into the replay buffer along with the error depending on whether or not the agent is using Prioritized Experience Replay. After a minimum number of samples have been collected, the agent starts to train the neural network.

Network Training The neural network is built to estimate the value of a state-action pair. The samples provided include the reward from the environment along with the next state. We can estimate the error in our neural network as `error = abs(Q(s,a) - reward * gamma * Q(s',a'))` where s and a are the current state and action and s′ and a′ are the next state and next action. The next action is not part of the data available in the sample so it is computed. Depending on whether [Double DQN] (https://arxiv.org/pdf/1509.06461.pdf) is in use, the a′ action is generated using the local network or the target network respectively. Once both of these estimated Q values are computed, the *loss* is calculated as the mean squared error between these two Q vectors. That loss is then fed through the optimizer to update the weights of the network.

1.1.4 Agent Training

Each architecture is trained over 500 episodes and the training for each architecture is executed three times using a different random seed each time.

1.2 Hyperparameters

The hyperparameters are set by default in rl_config.py

1.2.1 General Reinforcement Learning Hyperparameters

Learning Rate The learning rate is set to 0.0001. Testing with learning rates of higher values led to poor performance.

Exploration Epsilon The epsilon begins with a value of 1.0 and decays at a rate of `epsilon*0.9996**total_steps` where total_steps is cumulative across all played games. The minimum value is limited to 0.01 to ensure some residual level of exploration in all games.

Parameter Update Rate The model parameters are updated as `params = tau*new_params + (1.-tau)*params` and tau is set to 0.001 so that the model does not adjust too quickly to new parameters.

Future Discount Gamma reflects a discount given to future rewards versus current rewards. The discount factor is set to 0.999 for a very small discount.

Learning Interval The number of training steps between each learning step. A value of 1 is used so that learning occurs on every training step.

Parameter Update Interval The number of steps between updates to the parameters. The value is set to 4 so that parameter updates occur every fourth training step.

Batch Size The number of samples together in a mini-batch for training the model. A value of 64 is used.

Buffer Size The number of training samples to store in the buffer. A value of 100000 is used.

Samples Before Learning The number of training samples added into the buffer before the model begins learning. The minimum value would be the batch size or 64. A value of 1000 is used.

1.2.2 Prioritized Experience Replay Hyperparameters

Alpha The priority of each sample is equal to the error in the model prediction. Alpha is an exponent applied to this priority when sampling from the distribution. A value of 1 for alpha would mean that the full error is used as the priority and samplas with high losses would be sampled much more heavily than others. A value of 0 for the exponent forces all values to 1 with the result that samples are chosen uniformly. Here, a value of 0.2 is used providing a small amount of prioritization.

Beta Alpha is applied after the loss is calculated and is stored with the sample. Beta has a similar effect accept that it intends to encompass how we desire more prioritization after many training steps as the error losses are more meaningful then. Beta is linearly annealed from 0.1 to 1.0 over the course of 100k training steps.

[11]:
```
import matplotlib.pyplot as plt
import numpy as np
```

[13]:
```
def get_running_means(data, interval):
    return [np.mean(data[max(i-interval+1,0):i+1]) for i in range(len(data)-1)]
```

[14]:
```
def get_average_score_at_episode(data):
    return [np.mean([data[j][i] for j in range(len(data))]) for i in␣
 ↪range(len(data[0]))]
```

[15]:
```
def get_smoothed100_score(data):
    return get_running_means(get_average_score_at_episode(data), 100)
```

[16]:
```
def get_scores(data):
    return [d[0] for d in data]
```

[17]:
```
import pickle
def load_perf_data():
    dqn_data_and_time=pickle.load(open('dqn.pkl','rb'))
    ddqn_data_and_time=pickle.load(open('ddqn.pkl','rb'))
    ddqn_per_data_and_time=pickle.load(open('ddqn_per.pkl','rb'))
    dueling_dqn_data_and_time=pickle.load(open('dueling_dqn.pkl','rb'))
    dueling_dqn_per_data_and_time=pickle.load(open('dueling_dqn_per.pkl','rb'))
    return (get_scores(dqn_data_and_time),
            get_scores(ddqn_data_and_time),
            get_scores(ddqn_per_data_and_time),
            get_scores(dueling_dqn_data_and_time),
           get_scores(dueling_dqn_per_data_and_time))
```

[18]:
```
dqn_scores, ddqn_scores, ddqn_per_scores, dueling_dqn_scores,␣
 ↪dueling_dqn_per_scores = load_perf_data()
```

[19]:
```
thirteens=[13 for i in range(500)]
```

1.3 Results

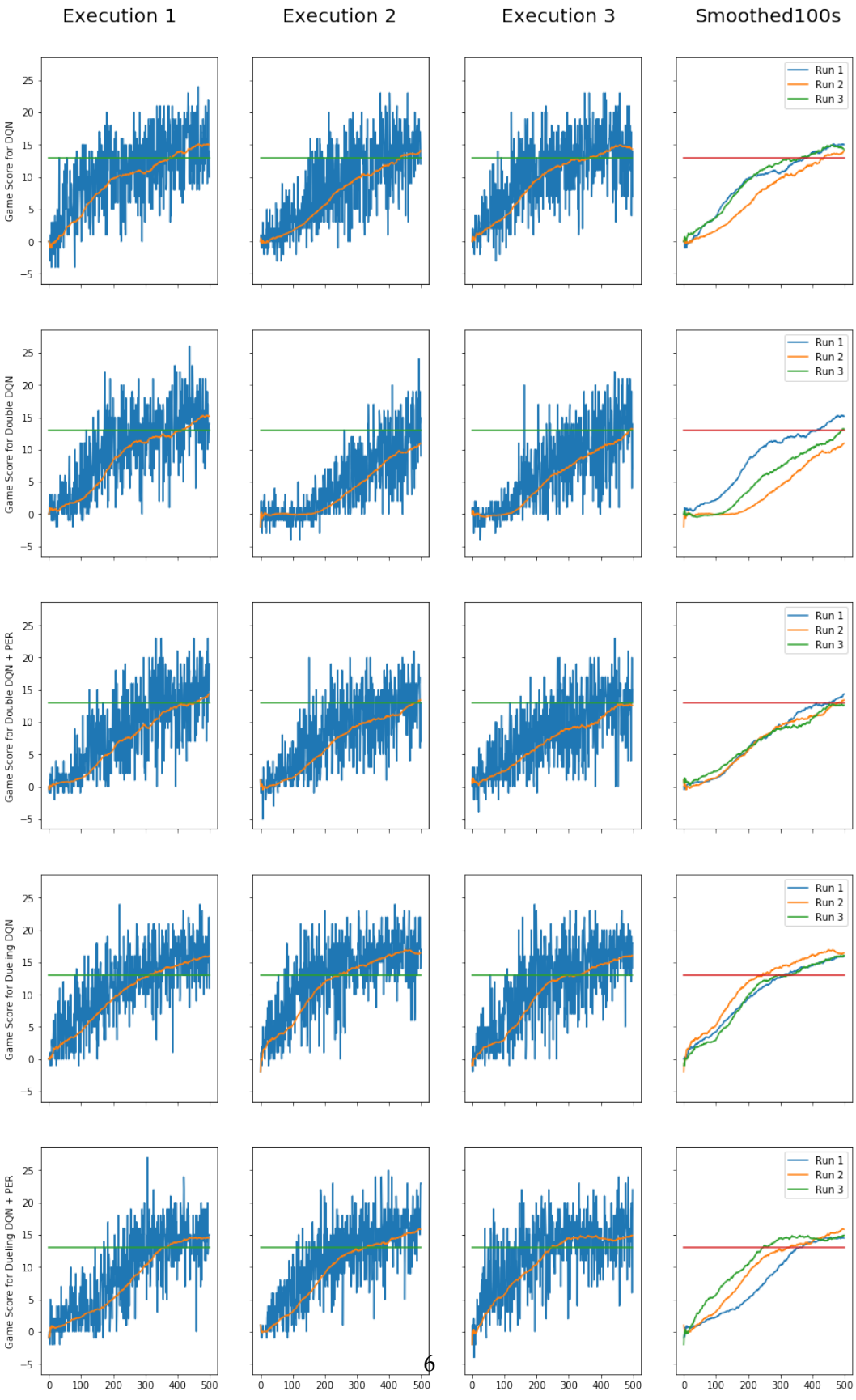
1.3.1 Raw Game Scores and Last 100 Games Moving Average

Here we display the raw scores along with the smoothed mean where the averaging interval is over the last 100 games. This is done for each of the five models: the baseline DQN, Double DQN, Double DQN with Prioritized Experience Replay, Dueling DQN, and Dueling DQN with Prioritized Experience Replay. Within these limited samples, it seems that the Dueling DQN architecture provides a clear advantage over the Double DQN architecture.

[20]:
```
all_scores = [dqn_scores, ddqn_scores, ddqn_per_scores, dueling_dqn_scores,␣
 ↪dueling_dqn_per_scores]
labels = ["DQN", "Double DQN", "Double DQN + PER", "Dueling DQN", "Dueling DQN␣
 ↪+ PER"]
fig,ax = plt.subplots(nrows=5,ncols=4,sharex=True,sharey=True,figsize=(15,25))
for i in range(5):
    ax[i][0].set_ylabel("Game Score for " + labels[i])
    for j in range(3):
```

```
        ax[i][j].plot(all_scores[i][j])
        ax[i][j].plot(get_running_means(all_scores[i][j], 100))
        ax[i][j].plot(thirteens)
    for j in range(3):
        ax[i][3].plot(get_running_means(all_scores[i][j], 100), label="Run " +␣
 ↪str(j+1))
    ax[i][3].plot(thirteens)
    ax[i][3].legend()
fig.text(0.5, 0.95, 'Raw vs Smoothed100 Game Scores', ha='center', fontsize=28)
fig.text(0.5, 0.09, 'Game Number', ha='center', fontsize=20)
fig.text(0.20,0.9, 'Execution 1', ha='center', fontsize=20)
fig.text(0.41,0.9, 'Execution 2', ha='center', fontsize=20)
fig.text(0.62,0.9, 'Execution 3', ha='center', fontsize=20)
fig.text(0.82, 0.9, 'Smoothed100s', ha='center', fontsize=20)
```

[20]:
```
Text(0.82, 0.9, 'Smoothed100s')
```

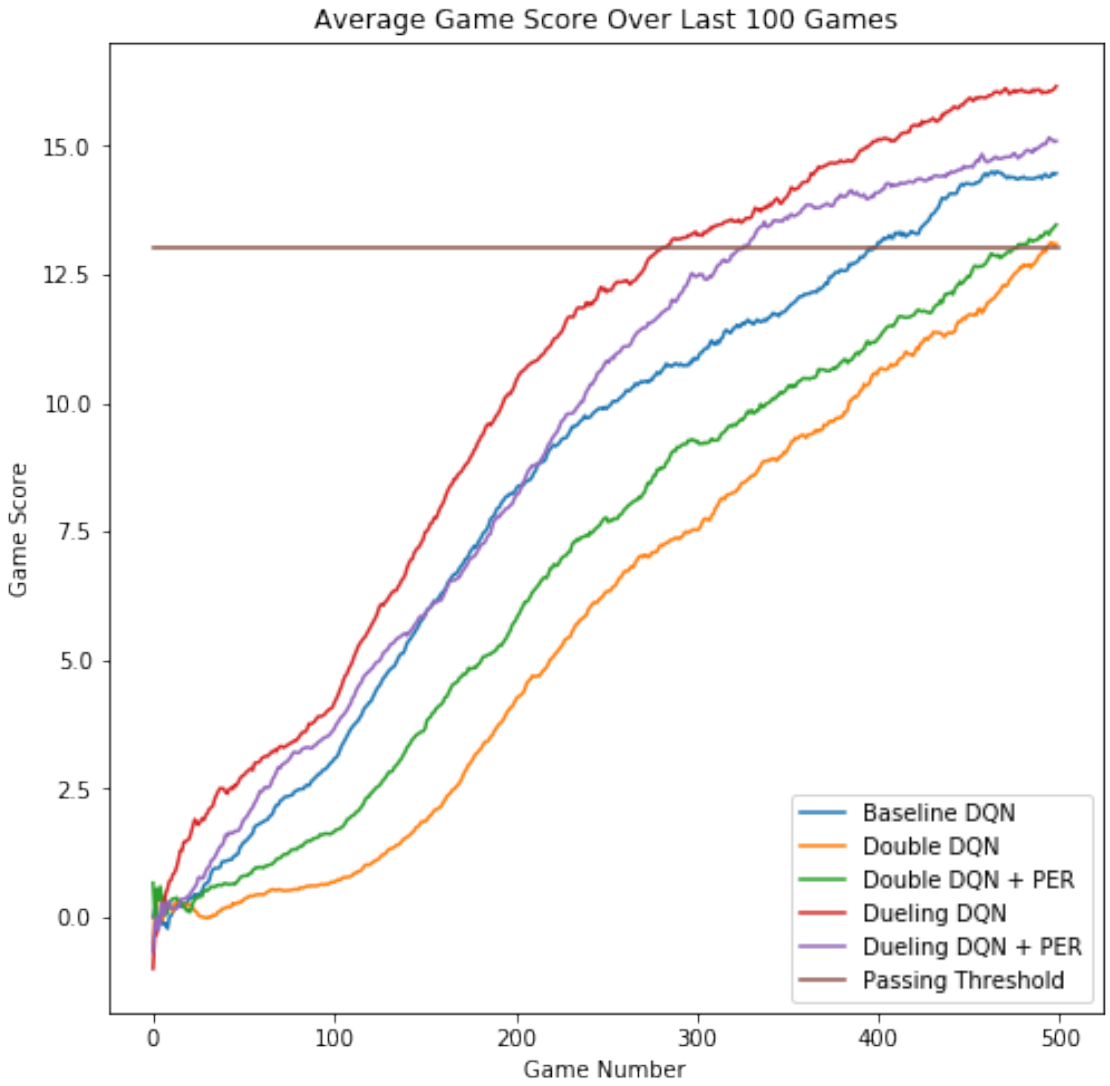

Raw vs Smoothed100 Game Scores
Execution 1
Execution 2
Execution 3
Smoothed100s
Game Score for DQN
Game Score for Double DQN
Game Score for Double DQN + PER
Game Score for Dueling DQN
Game Score for Dueling DQN + PER
Run 1
Run 2
Run 3
Game Number

1.3.2 Comparing Models

In order to try to limit the potential impact of a single lucky or unlucky run, each of the three runs on each architecture is averaged by taking the mean score at for each episode. This combined training sample for a given model can then be easily compared with the other considered models. It is clear that the Dueling DQN and Dueling DQN + PER models perform well. It may be that the single relatively poor run from the Dueling DQN + PER model results in this delta but that has not been established in this report.

[21]:
```
dqn_means=get_smoothed100_score(dqn_scores)
ddqn_means=get_smoothed100_score(ddqn_scores)
ddqn_per_means=get_smoothed100_score(ddqn_per_scores)
dueling_dqn_means=get_smoothed100_score(dueling_dqn_scores)
dueling_dqn_per_means=get_smoothed100_score(dueling_dqn_per_scores)
```

[22]:
```
plt.rcParams['figure.figsize'] = [8,8]
plt.plot(dqn_means, label="Baseline DQN")
plt.plot(ddqn_means, label="Double DQN")
plt.plot(ddqn_per_means, label="Double DQN + PER")
plt.plot(dueling_dqn_means, label="Dueling DQN")
plt.plot(dueling_dqn_per_means, label="Dueling DQN + PER")
plt.plot(thirteens, label="Passing Threshold")
plt.ylabel("Game Score")
plt.xlabel("Game Number")
plt.title("Average Game Score Over Last 100 Games")
plt.legend()
plt.show()
```



1.3.3 Number of Episodes to Requirement Satisfaction

Below we calculate the episode number at which the average score over the last 100 episodes exceeds the requirement of 13. We can see that the two dueling DQN networks clearly performed better while the two double DQN architectures performed worse.

[27]:
```
def get_episode_meeting_requirement(title, means, requirement):
    vals = [(i, means[i]) for i in range(len(means)) if means[i] >= requirement]
    if len(vals) > 0:
        print("Method {} meets requirement of {} at episode {} with mean {}".
↪format(title, requirement, vals[0][0], vals[0][1] ))
    else:
        print("Method {} failed to meet the requirement of {}".format(title,␣
↪requirement))
```

[28]:
```
get_episode_meeting_requirement("DQN", dqn_means, 13.0)
get_episode_meeting_requirement("Double DQN", ddqn_means, 13.0)
get_episode_meeting_requirement("Double DQN + PER", ddqn_per_means, 13.0)
get_episode_meeting_requirement("Dueling DQN", dueling_dqn_means, 13.0)
get_episode_meeting_requirement("Dueling DQN + PER", dueling_dqn_per_means, 13.
↪0)
```

```
Method DQN meets requirement of 13.0 at episode 395 with mean 13.020000000000003
Method Double DQN meets requirement of 13.0 at episode 493 with mean
13.046666666666665
Method Double DQN + PER meets requirement of 13.0 at episode 474 with mean 13.0
Method Dueling DQN meets requirement of 13.0 at episode 280 with mean 13.0
Method Dueling DQN + PER meets requirement of 13.0 at episode 326 with mean
13.059999999999997
```

1.4 Pre-Trained Model Live Test

Here we pick the Dueling DQN + PER model and just run some games using the previously trained model.

[6]:
```
config=RLConfig()
config.title="Dueling DQN + PER"
config.usePER=True
config.useDoubleDQN=True
config.useDuelingDQN=True
config.modelSaveName = "dueling_dqn_per.pth"
agent = DQNAgent(state_size, brain.vector_action_space_size,config)
agent.qnetwork_local.load_state_dict(torch.load("dueling_dqn_per.pth"))
numGames = 10


total_score = 0
for i in range(numGames):
    env_info = env.reset(train_mode=False)[brain_name] # reset the environment
    state = env_info.vector_observations[0]            # get the current state
    game_score = 0
    while True:
        # get the action from the agent per the current state but do not train␣
↪the agent
        action = agent.act(state)
        env_info = env.step(action)[brain_name]
        next_state = env_info.vector_observations[0]
        reward = env_info.rewards[0]
        done = env_info.local_done[0]
        game_score += reward
        state = next_state
        if done:
            break
```

```
    print("Game {} completed with score {}".format(i+1, game_score))
    total_score += game_score
print("Smart Agent completed {} games with Total {} and Average Score {}".
 ↪format(numGames, total_score, total_score/numGames))

env.close()
```

```
Game 1 completed with score 15.0
Game 2 completed with score 22.0
Game 3 completed with score 17.0
Game 4 completed with score 23.0
Game 5 completed with score 17.0
Game 6 completed with score 15.0
Game 7 completed with score 14.0
Game 8 completed with score 19.0
Game 9 completed with score 18.0
Game 10 completed with score 16.0
Smart Agent completed 10 games with Total 176.0 and Average Score 17.6
```

Our pre-trained model seems to score well above the requirement.

1.5 Potential Improvements and Future Work

1.5.1 Further Evaluation of Double DQN

The two Double DQN networks without the Dueling DQN enhancement both performed significantly worse than the vanilla DQN network. It is unclear whether this is an artifact of small sample size or an artifact of the problem itself. The author also considers it possible that there is an implementation error in this Double DQN architecture. However, the author notes that the Dueling DQN architectures both performed dramatically better and the Dueling DQN architectures use Double DQN under the hood.

1.5.2 Further Evaluation of PER Impacts

In the set of three model convergence runs presented here, adding PER led to a decline in convergence time for the Dueling DQN case. PER led to an improvement in the Double DQN case but the gain was modest enough that it may be attributable to small sample size. It would be desirable to execute far more convergence runs in order to provide a more definitive answer on the effect of PER to this particular problem. Likewise, it would be desirable to test the impact of PER on the learning from pixels case which requires significantly more episodes to converge toward the solution. It should be noted that in this write-up, experience led to decreasing alpha and the starting beta, thereby limiting the impact of PER, as a necessary means towards a converging algorithm.

1.5.3 Grid Search of Hyperparameters

It would be desirable to perform some level of search over the range for each of the hyperparameters in order to identify optimal values. Some level of experimentation was performed in development with the learning rate, the epsilon decay and minimum, the number of samples before learning, PER-alpha, and PER-beta. However, these were not captured in a systematic fashion.

1.5.4 Model Search

The neural network architecture used here is a simple feed-forward architecture with two fully connected hidden layers with 128 nodes each. It is not clear whether a simpler model would perform as well or whether adding in additional learning capacity in the form of additional layers or nodes within the layer would lead to overfitting or better performance.

[ ]: