APML Project – Snake

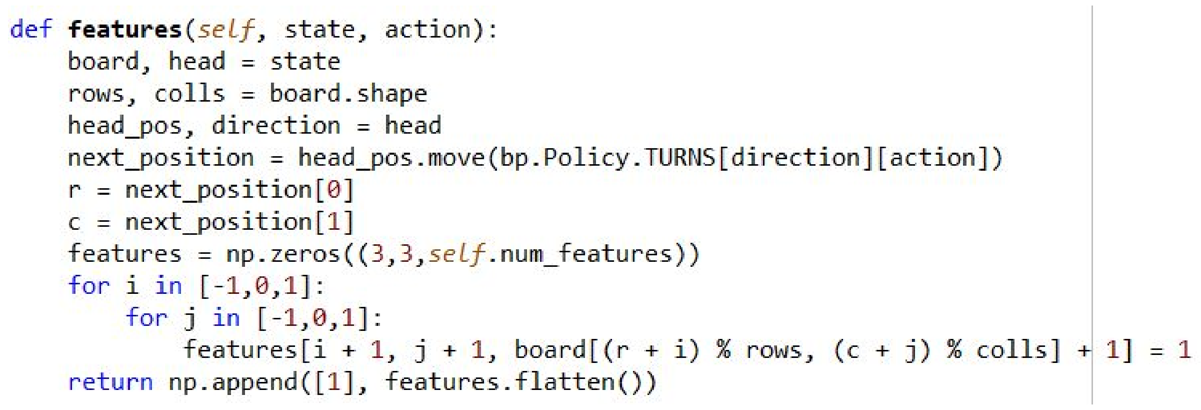
ID1: 204685200 ID2: 305248791
Name1: Joseph Jubran Name2: Christian Mtanes

The Linear Agent:

To use Q-values with function approximation, we need to find features that are functions of states and actions. This means in the linear function regime, we have:

$$Q(s,a) = \theta_0 \cdot 1 + \theta_1 \phi_1(s,a) + \cdots + \theta_n \phi_n(s,a) = \theta^T \phi(s,a)$$

We represent every (state, action) with a feature vector. Our feature vector is three dimensional. We create a window of size (3, 3) around the next position of the head of the snake in the board after applying the action in the current state. So the first two dimensions of the vector is a position in this window the third dimension is the board value plus one (s.t that the values between [-1,9] ,will be mapped to values [0,10]). Added to these features is the value one for bias. Below is the function the takes (state, action) and turns it to features vector. The code explains what we have done better:



```python
def features(self, state, action):
    board, head = state
    rows, colls = board.shape
    head_pos, direction = head
    next_position = head_pos.move(bp.Policy.TURNS[direction][action])
    r = next_position[0]
    c = next_position[1]
    features = np.zeros((3,3,self.num_features))
    for i in [-1,0,1]:
        for j in [-1,0,1]:
            features[i + 1, j + 1, board[(r + i) % rows, (c + j) % colls] + 1] = 1
    return np.append([1], features.flatten())
```

Experiments and results:

we started off with the following parameters:

the learning rate, alpha = 0.01.

the discount rate, gamma = 0.9.

the initial parameter for the epsilon greedy policy, epsilon = 1

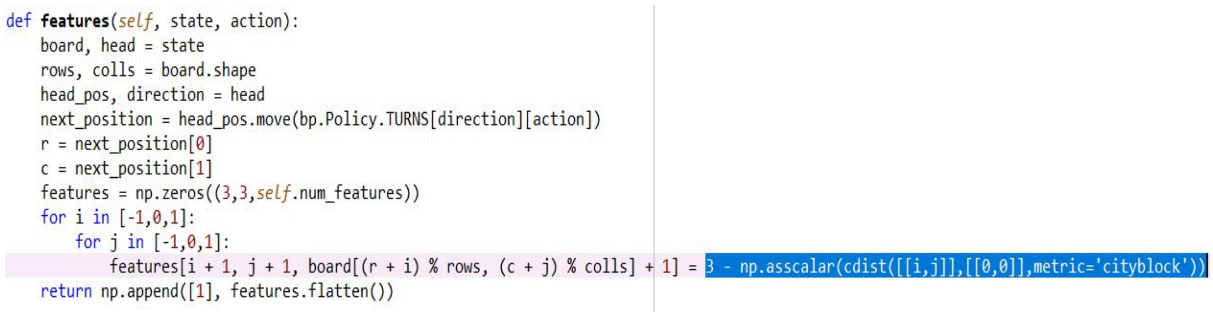
we used epsilon decay on epsilon parameter such that it reduces every round by 0.01.

the following is the score results over 5 sessions:

Agents	Session 1	Session 2	Session 3	Session 4	Session 5	Average
Linear agent	0.293	0.255	0.193	0.224	0.207	0.234
Avoid(epsilon =0)	0.173	0.081	0.163	0.068	0.159	0.128
Avoid(epsilon =0)	0.078	0.135	0.125	0.11	0.041	0.0978

Our linear agents won in each session with an average score of 0.234.

what we tried next is to stay with same parameters as above but changing the feature value to 3 – ManhattanDistance(head_possition, position in the windows). The feature function becomes the following:



```python
def features(self, state, action):
    board, head = state
    rows, colls = board.shape
    head_pos, direction = head
    next_position = head_pos.move(bp.Policy.TURNS[direction][action])
    r = next_position[0]
    c = next_position[1]
    features = np.zeros((3,3,self.num_features))
    for i in [-1,0,1]:
        for j in [-1,0,1]:
            features[i + 1, j + 1, board[(r + i) % rows, (c + j) % colls] + 1] = 3 - np.asscalar(cdist([[i,j]],[[0,0]],metric='cityblock'))
    return np.append([1], features.flatten())
```

the following is the score results over 5 sessions:

Agents	Session 1	Session 2	Session 3	Session 4	Session 5	Average
Linear agent	0.393	0.44	0.43	0.247	0.27	0.302
Avoid(epsilon =0)	0.012	0.003	0.071	0.069	0.069	0.074
Avoid(epsilon =0)	0.065	0.006	0.056	0.132	0.085	0.0688

Our linear agents won in each session with an average score of 0.302 a better average than before.

Now we try to enlarge the windows to size (5,5) and the feature value will be 4 – ManhattanDistance(head_possition, position in the windows).

Agents	Session 1	Session 2	Session 3	Session 4	Session 5	Average
Linear agent	0.447	0.42	0.38	0.441	0.43	0.42
Avoid(epsilon =0)	0.028	0.015	0.01	0.056	0.049	0.0316
Avoid(epsilon =0)	0.053	0.005	0.034	0.045	0.054	0.0382

Our linear agents won in each session with an average score of 0.42 a better average than before.

Now we try the linear agent with different gamma values (discount rate) while other parameters are the same as above: (the linear agent is evaluated against two avoid policy agent with epsilon = 0 their results are not documented for they always lose against our agent)

Agents	Session 1	Session 2	Session 3	Session 4	Session 5	Average
Linear agent gamma = 0.01	0.39	0.33	0.124	0.391	0.41	0.3288
Linear agent gamma = 0.1	0.37	0.33	0.317	0.458	0.418	0.37
Linear agent gamma = 0.5	0.453	0.438	0.415	0.479	0.383	0.433

We get a better result than with gamma=0.9 (see the previous table) only for gamma=0.5.

Now we stay the same gamma = 0.5 for the rest of the experiments.

Now we run our agent against with two avoid agents with epsilon=0, changing the learning rate alpha:

Agents	Session 1	Session 2	Session 3	Session 4	Session 5	Average
Linear agent alpha = 0.01	0.453	0.438	0.415	0.479	0.383	0.433
Linear agent alpha = 0.001	0.34	0.351	0.435	0.462	0.246	0.3668
Linear agent alpha = 0.1	-0.22	-0.036	-0.214	-0.053	-0.117	-0.128

We get the best results with alpha = 0.01.

Now we stay with gamma = 0.5 and alpha = 0.01 and explore different decay for the epsilon in the epsilon greedy policy.

Agents	Session 1	Session 2	Session 3	Session 4	Session 5	Average
Linear agent epsilon decay = 0.01	0.453	0.438	0.415	0.479	0.383	0.433
Linear agent epsilon decay = 0.001	0.415	0.386	0.452	0.439	0.31	0.400
Linear agent epsilon decay = 0.1	0.452	0.453	0.401	0.336	0.414	0.411

The best results are with epsilon decay = 0.01.

So the best parameters for our linear agent are:

the learning rate, alpha = 0.01.

the discount rate, gamma = 0.5.

the initial parameter for the epsilon greedy policy, epsilon = 1

we used epsilon decay on epsilon parameter such that it reduces every round by 0.01.

with these parameters our agent gets the best results and wins by a large margin against two avoid policies with epsilon = 0.

The Second Agent

Deep Reinforcement Learning

We chose to implement a deep reinforcement learning model because we saw its success on several Atari games, and since we are playing a game similar in concept to those Atari games we chose to use this model.

link for deep q learning on Atari games paper

the algorithm we implement is quoted from this paper:

```
Algorithm 1 Deep Q-learning with Experience Replay
  Initialize replay memory $\mathcal{D}$ to capacity $N$
  Initialize action-value function $Q$ with random weights
  for episode = $1, M$ do
      Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
      for $t = 1, T$ do
          With probability $\epsilon$ select a random action $a_t$
          otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
          Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
          Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
          Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
          Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
          Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
          Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
      end for
  end for
```

In Q-Learning Algorithm, we have a function called Q Function, which is used to approximate the reward based on a state. We call it Q(s,a), where Q is a function which calculates the expected future value from state s and action a. Similarly in Deep Q Network algorithm, we use a neural network to approximate the reward based on the state.

We want to decrease the gap between the prediction and the target (loss). We will define our loss function as follows:

$$loss = \left(\underbrace{\overset{\text{Reward}}{r} + \overset{\text{Decay Rate}}{\gamma} \max_{a`} \hat{Q}(s, a`)}_{\text{Target}} - \underbrace{Q(s, a)}_{\text{Prediction}}\right)^2$$

in our implementation we used the Logarithm of the hyperbolic cosine of the prediction error. log(cosh(x)) is approximately equal to (x ** 2) / 2 for small x and to abs(x) - log(2) for large x. This means that 'logcosh' works mostly like the mean squared error, but will not be so strongly affected by the occasional wildly incorrect prediction.

we used the RMSProp optimizer which was used in the Atari games paper.

We first carry out an action a, and observe the reward r and resulting new state s`. Based on the result, we calculate the maximum target Q and then discount it so that the future reward is worth less than immediate reward. Lastly, we add the current reward to the discounted future reward to get the target value. Subtracting our current prediction from the target and then applying Logarithm of the hyperbolic cosine gives the loss.

We just need to define our target:

```
target = reward + self.gamma * np.amax(self.model.predict(next_state)[0])
```

Keras does all the work of subtracting the target from the neural network output and then applying Logarithm of the hyperbolic cosine . It also applies the learning rate we defined while creating the neural network model. This all happens inside the fit() function. This function decreases the gap between our prediction to target by the learning rate. The approximation of the Q-value converges to the true Q-value as we repeat the updating process. The loss will decrease and score will grow higher.

<u>State representation:</u>

Similarly to the linear model, the state representation is comprised of a window of indicators around the head for each board feature with a few adjustments:

Asymmetrical window: Pretty straight forward, we use window_height > window_width
with the height axis being parallel to the direction of the snake.

Direction Oriented Representation: The representation of the board is is oriented using the direction, and thus we get a consistent state for the board cells compared to the snake.

<u>Memory replay:</u>

we need a list of previous experiences and observations to re-train the model with the previous experiences. We will call this array of experiences replay_memory and use remember() function to append state, action, reward, and next state to the memory.

we also tried <u>Prioritized memory replay:</u>

DQN samples transitions from the replay buffer uniformly. However, we should pay less attention to samples that is already close to the target. We should sample transitions that have a large target gap:

$$\left| r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta) \right|$$

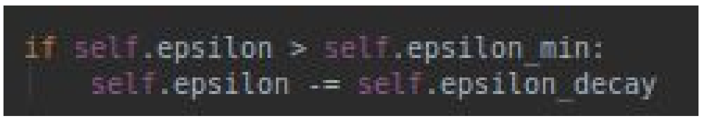
Therefore, we can select transitions from the buffer based on the error value above (pick transitions with higher error more frequently). [Prioritized Experience Replay paper](#).

<u>Decaying ϵ-greedy:</u>

Our agent will randomly select its action at first by a certain percentage, called 'exploration rate' or 'epsilon'. This is because at first, it is better for the agent to different states before it starts to see the patterns. When it is not deciding the action randomly, the agent will predict the reward value based on the current state and pick the action that will give the highest reward,

```
def act(self, round, prev_state, prev_action, reward, new_state, too_slow):
    if np.random.rand() < self.epsilon:
        return np.random.choice(bp.Policy.ACTIONS)
    new_state_features = self.preprocess_state(new_state)
    Q_value = self.model.predict(new_state_features)
    return bp.Policy.ACTIONS[np.argmax(Q_value[0])]
```

We also decayed the epsilon so in the latter stages it becomes less and less random until it reaches a minimum threshold thus we had both eaten the cake and took it too, using the exploration in the first stages to learn the environment, and once the model is sufficient enough it's mostly exploitation.

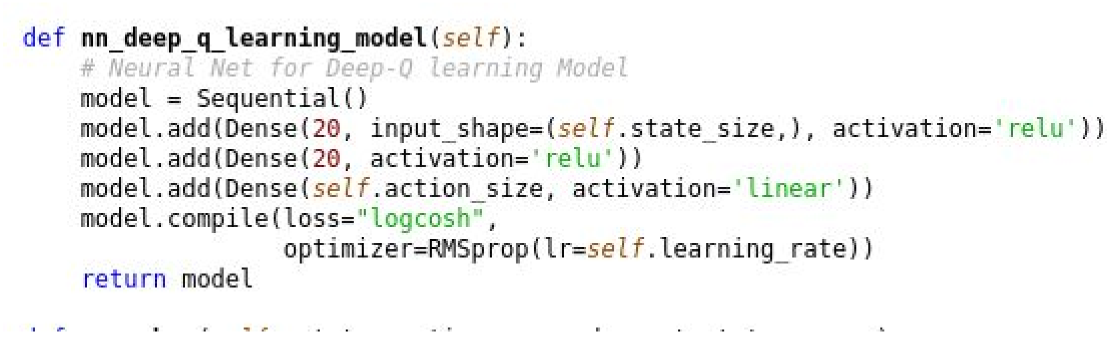


```
if self.epsilon > self.epsilon_min:
    self.epsilon -= self.epsilon_decay
```

This happened in the learning Function so it the decay happened every 4 steps or so.

<u>the neural network:</u>

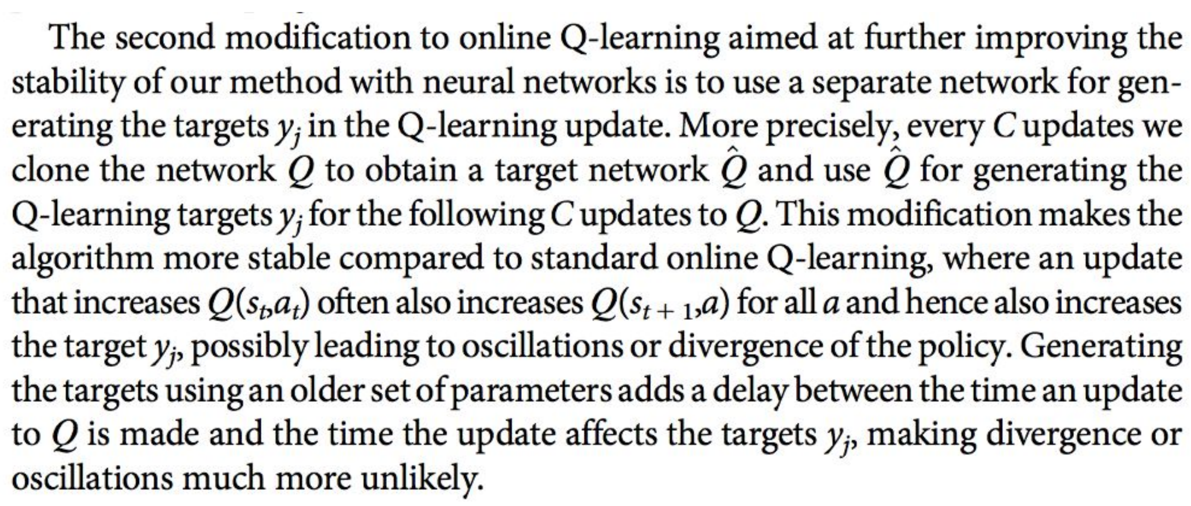
our neural network consists of two dense layers with 20 neurons each with a relu activation and a final dense layer with 3 neurons (size of possible actions) with a linear activation.



```
def nn_deep_q_learning_model(self):
    # Neural Net for Deep-Q learning Model
    model = Sequential()
    model.add(Dense(20, input_shape=(self.state_size,), activation='relu'))
    model.add(Dense(20, activation='relu'))
    model.add(Dense(self.action_size, activation='linear'))
    model.compile(loss="logcosh",
                  optimizer=RMSprop(lr=self.learning_rate))
    return model
```

<u>Freezing Q:</u>

DeepMind introduces in their paper [Human-level control through deep reinforcement learning]the notion of a target network. In their own words:



The second modification to online Q-learning aimed at further improving the stability of our method with neural networks is to use a separate network for generating the targets y_j in the Q-learning update. More precisely, every C updates we clone the network Q to obtain a target network $\hat{Q}$ and use $\hat{Q}$ for generating the Q-learning targets y_j for the following C updates to Q. This modification makes the algorithm more stable compared to standard online Q-learning, where an update that increases $Q(s_t,a_t)$ often also increases $Q(s_{t+1},a)$ for all a and hence also increases the target y_j, possibly leading to oscillations or divergence of the policy. Generating the targets using an older set of parameters adds a delay between the time an update to Q is made and the time the update affects the targets y_j, making divergence or oscillations much more unlikely.

<u>Hall Of Shame:</u>

For the first linear model we tried using its own weights for each board value and averaging over the window. This yielded great results but it didn't seem linear nor did it seem enough to submit although it often did better than the Deep Q learning! (results for this Learner are below)

For the second model we tried multiple other features to improve the second model, alas it was either futile or too computationally demanding.

The usage of the weights in the first layer to construct a general approximation of the 'goodness' of a window the same as we tried for the linear model, we used different statistics on the weights to try extract a general weight for each board value.

We also tried to reduce the cases of self suffocating, as in the cases that the snake would lock itself in its own body, but without any details on the snake representation in the board we had to use only the movements but without knowing when the snake died as we couldn't assume the penalty made it problematic and too hard to teach the model as a self engineering, a better approach would have been a different state representation which included different paths for the snake to traverse, but the deadline was closing in and we had to stop exploring and start exploiting our existing features.

experiments and results:

for each experiment we have fixed other parameter while changing the parameter we want to test.

(In Green is better)

With vs Without Memory prioritization:

	session 1	session 2	session 3	Avg
W Prioritized Memory	**0.2918**	**0.3102**	**0.2962**	**0.2994**
W/O Prioritized Memory	**0.3060**	**0.2516**	**0.2444**	**0.2607**

Different C_step values for Freezing Q:

	session 1	session 2	session 3	Avg
C_step = 50	**0.2910**	**0.2906**	**0.2652**	**0.2822**
C_step = 150	**0.2522**	**0.2582**	**0.2758**	**0.2620**
C_step = 500	**0.2738**	**0.2372**	**0.2700**	**0.2601**
C_step = 1000	**0.2940**	**0.2966**	**0.3032**	**0.2979**

-Something that we noticed as a result of these two features that while it increase the score maximum score somewhat, the more serious improvement was the stability/consistency of the learner.

Different Learning Rate values:

	session 1	session 2	session 3	Avg
LR value = 0.001	**0.3252**	**0.3174**	**0.3098**	**0.3174**
LR value = 0.01	**-0.0402**	**0.0836**	**0.1196**	**0.0543**
LR value = 0.1	**-0.0620**	**-0.0540**	**-0.0606**	**-0.056**

Different linear epsilon decay values:

	session 1	session 2	session 3	Avg
Eps Decay = 0.001	0.2884	0.2940	0.3284	0.3036
Eps Decay = 0.0001	0.2912	0.2634	0.2556	0.2700
Eps Decay = 0.00001	-0.2168	-0.1638	-0.1478	-0.1761

Different Gamma Values:

	session 1	session 2	session 3	Avg
Gamma = 0.99	0.2890	0.2612	0.2696	0.2732
Gamma = 0.95	0.3142	0.2510	0.2524	0.2725
Gamma = 0.8	0.2524	0.2738	0.3152	0.2804

-we expected to need high Gamma values due to the large window thus we only used large values, it also fits the values that were used in papers cited above.

Different batch sizes

	session 1	session 2	session 3	Avg
batch_size = 3	0.2586	0.3094	0.2552	0.2744
batch_size = 4	0.2458	0.2540	0.2892	0.2630
batch_size = 5	0.2682	0.2228	0.2948	0.2619

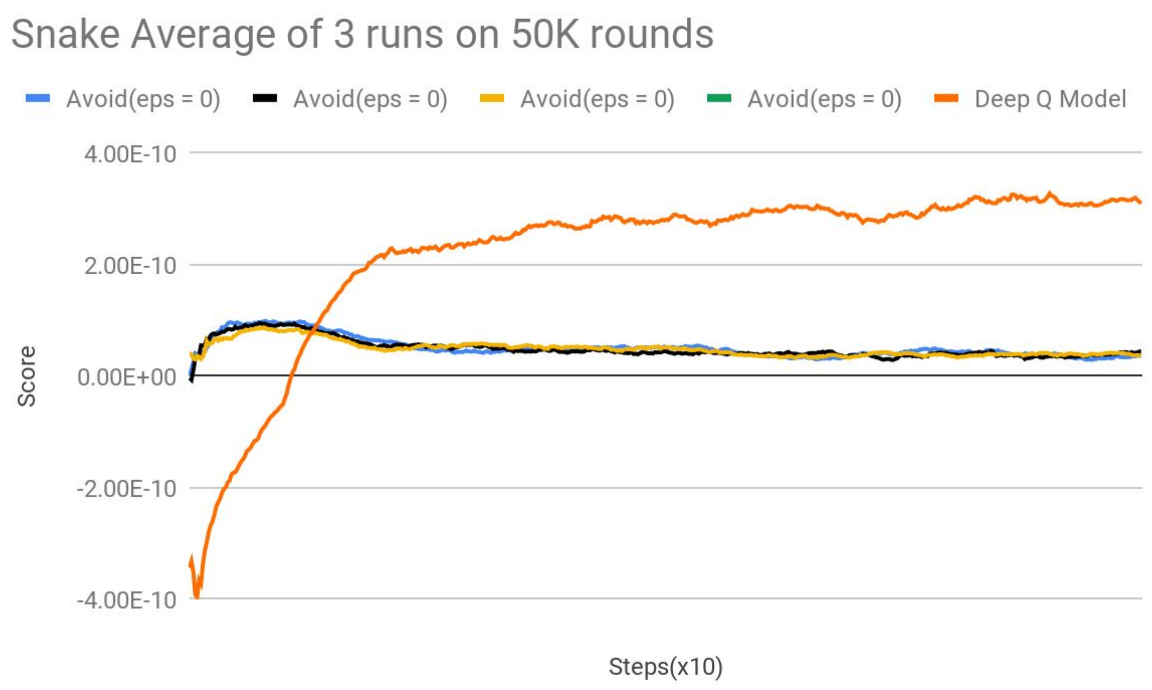
-We chose small batch sizes due to the time limitations, although given more time we could have optimized our feature extraction and used larger batches.

Different Replay Memory size

	session 1	session 2	session 3	Avg
mem_size = 200	0.2956	0.2662	0.2818	0.2812
mem_size = 300	0.3464	0.2528	0.2796	0.2929
mem_size = 400	0.2770	0.2508	0.2890	0.2722

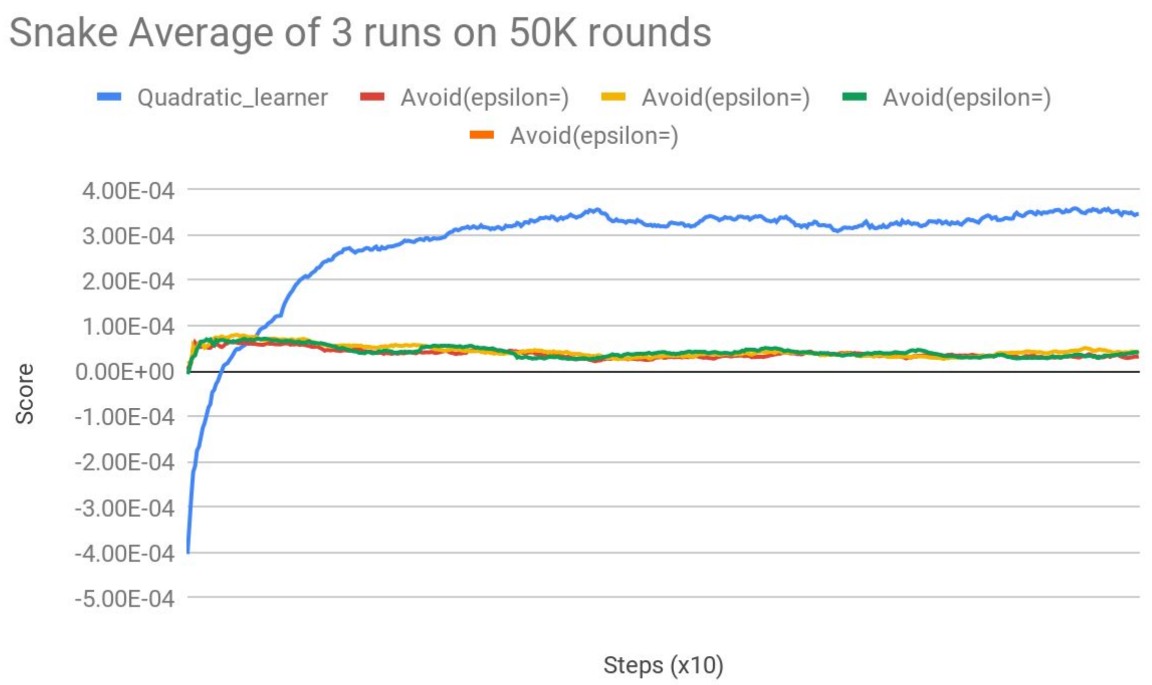
-We chose small batch sizes due to the time limitations, although given more time we could have optimized our feature extraction and used larger batches.

An average on 3 games of the whole score progress for the model with the best parameters:



Important to notice is that our model converges in the general proximity of 0.3, so even given 100,000 rounds it would still get a similar score.

Results of the “Quadratic” Agent mentioned in the whole of shame:


Snake Average of 3 runs on 50K rounds
Quadratic_learner
Avoid(epsilon=)
Avoid(epsilon=)
Avoid(epsilon=)
Avoid(epsilon=)
4.00E-04
3.00E-04
2.00E-04
1.00E-04
0.00E+00
-1.00E-04
-2.00E-04
-3.00E-04
-4.00E-04
-5.00E-04
Score
Steps (x10)