Reinforcement Learning - Snake Game

Yasmin Heimann, 311546915

December 27, 2020

1 Q-Learning

Q-Learning is an off-policy reinforcement learning algorithm that seeks to find the best action an agent can take, given the current state it is in.

The Q stands for Quality, which represents how useful is a given action by approximating the future reward of this action. From Q-Learning, we conclude the policy to act by.

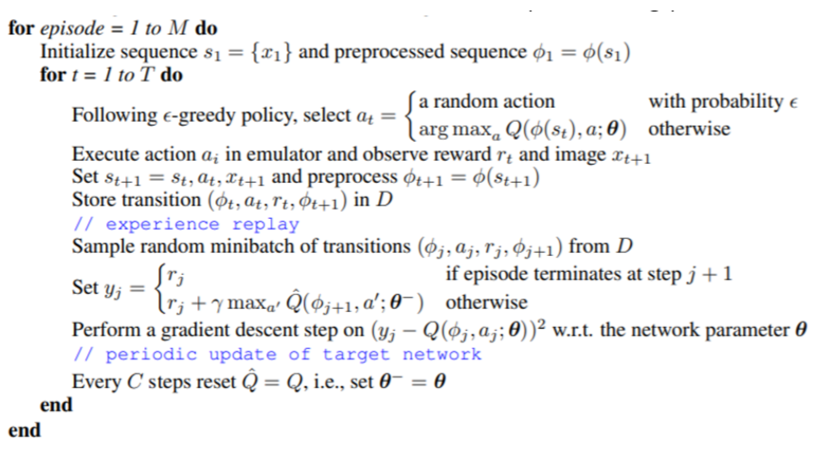
1.1 Model Specifications

I chose to implement the algorithm showed in class using a DQN - Deep Q-Network, which is useful in cases where the state space is large. Thus, instead of implementing a straight-forward q-table for each state and action, the network learns the Q-values of a state and an action given a state.

1.1.1 Q-Networks and Optimization

1. **Network Architechture:** I decieded to test 3 networks variations for the Q Learning:
   (a) **Simple Model:** 1-Layer network, that uses a linear layer, and lowers the state dimension from 9x9 squares of the board, to only 3x3 squares around the Snake's head.
   (b) **Full Linear Model:** 3-Layer network, that uses linear layers, and the full 9x9 squares of the board.
   (c) **CNN:** a network with 1 convolution layer and a linear out-layer
2. **Optimization:** For the optimization step - which is the actual learning step - I decieded to apply 2 approaches for avoiding gradients explosion:
   (a) **RMSprop optimizer** - I have used the RMSprop as the optimization algorithm to avoid gradient problems. RMSprop is using a moving average of squared gradients to normalize the gradient. This normalization balances the step size - momentum, decreasing the step for large gradients to avoid exploding, and increasing the step for small gradients to avoid vanishing. Thus, this loss provides a solution to possible large gradients, which might occur especially at the beginning of the training.
   (b) **Gradient Clipping** - a technique to prevent exploding gradients, by preventing any gradient to have norm greater than a given threshold. I have used the Pytorch function **clamp**, to normalize the gradients and avoid explosion.
3. **Target Network:** I have decieded to implement a separate target network, as it is a good practice in general, even though it is not required here. The target network is initialized with the random weights of the main model, and then its weights are updated only after every optimization step is done.

I have used the algorithm explained in class, with DQN and a target-network, with the help of this Tutorial:
https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html , and the following Pseudo-Code:



```
for episode = 1 to $M$ do
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    for $t$ = 1 to $T$ do
        Following $\epsilon$-greedy policy, select $a_t = \begin{cases} \text{a random action} & \text{with probability } \epsilon \\ \arg\max_a Q(\phi(s_t), a; \boldsymbol{\theta}) & \text{otherwise} \end{cases}$
        Execute action $a_i$ 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 $D$
        // experience replay
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \boldsymbol{\theta}^-) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \boldsymbol{\theta}))^2$ w.r.t. the network parameter $\boldsymbol{\theta}$
        // periodic update of target network
        Every $C$ steps reset $\hat{Q} = Q$, i.e., set $\boldsymbol{\theta}^- = \boldsymbol{\theta}$
    end
end
```

Figure 1: Q-Learning Pseudo-Code, reference:https://arxiv.org/pdf/1701.07274.pdf

1.2 Parameters Tuning

In this model, we have parameters we would like to optimize: ϵ, γ, number of steps, learning rate, batch size and more.

We will focus on ϵ and γ, as they play the most cruicial role here.

The learning rate will be set as default to 0.01, as we saw in earlier exercises, is a good learning rate estimate.

As for the number of steps, for this algorithm we will easily see that convergence will appear around $\sim 5 - 8K$, thus $10k$ will be a good estimate for this parameter.

1.2.1 The ϵ Role

We use a ϵ-greedy technique to balance between exploration and exploitation of the actions per each state, and that ϵ will decay with time (as the learning progresses).

1. In each action selection, we choose between 2 options:

   - **Exploration** - with probability ϵ, we will choose an action $randomly$.
   - **Exploitation** - with probability $1 - \epsilon$, we will choose the best action based on the highest q-value from the Q-network output.

2. At each step, we apply ϵ**-decay** which lowers the ϵ value.

   At the beginning, we have no knowledge of the model and we use random weights for the Q-Network, thus we would tend to do more exploration - and ϵ is the highet in the beginning.

   The more we learn, the more confident we will be in our model, so we would tend to choose the action our model predicted has the highest q-value. The exploitation rate grows greater as ϵ decays through out the learning process, thus we will rely on our Q-Network much more.

1.2.2 The γ Role

γ is the discount parameter we use to determine how much of the future reward will be taken.

We saw in class that the discounted reward is as follows:

$$R(\overrightarrow{s}) = \sum_{i=0}^{T} \gamma^i reward_i(s_i, a_i)$$

Where T is the finite steps of the game, $\overrightarrow{s}$ is the states series, and $\gamma \in (0, 1]$, where $\gamma = 1$ is equivalent to the non-discounted reward.

We shall make the following distinction, to understand the meaning of γ. A high γ will result in high impact of future rewards, as the decay rate will be slow. Counter-wise, a small γ, will result in lower impact of the future rewards.

We would like to explore the right discount parameter empiricaly, and analyse the range of γ over the learning success. We would expect to see moderate values of $\gamma \in (0, 1)$, which balances the impact of the future rewards.

1.2.3 ϵ, γ Optimization

I have extracted multiple graphs so that the impact of these parameters can be analysed.

I chose a series of ϵ values, and for each ϵ value, I have trained the model with different γ values.

We shall remember that ϵ is dacying with time, thus the ϵ values noted below is the strating maximal ϵ.

I ran all samples with batch size of 32 and 10k iterations, over the simple network model.

Low Maximal Exploration Rate - $\epsilon = 0.05, 0.1$

We can see from the graph below that when using a very low max-exploration rate, $\epsilon = 0.05$, the convergence rate is slower for $\gamma < 0.5$, $\sim 7k$, and a greater γ is required for better results:

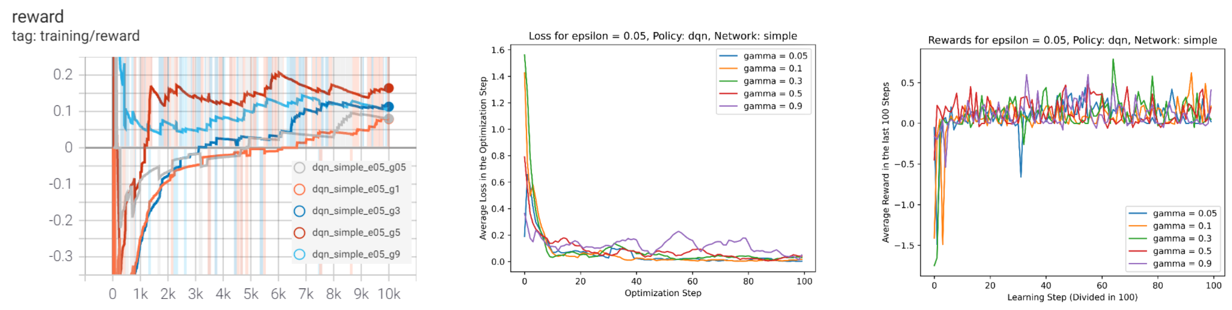


Figure 2: Q-Learning Algorithm performed with the simple DQN (model 1), $\epsilon = 0.05$ and various γ. **LEFT:** smoothed reward trendline using tensorboard, for each γ value. **MIDDLE:** Loss convergence as an average loss for each optimization step. **RIGHT:** Reward Average every 100 iterations for each γ value.

We can see that the algorithm converges to a positive value in each case. We would expect a moderate-high γ to perform the best rewards, as it balances how much to take the future into account. Low γ might not be efficient enough as it goes to 0 very fast.

We see empirically, that indeed, $\gamma = 0.3, 0.5, 0.9$ gives high results and $\gamma = 0.5$, gives the fastest convergence rate (~2k) combined with high results.

More intersting, we see that even when our exploration rate is low, we achieve positive results.

For $\epsilon = 0.1$, I got fairly similar results.

Moderate Maximal Exploration Rate - $\epsilon = 0.3, 0.5$

We can see that for moderate exploration rate, we converge to a positive reward around $\sim 2 - 5k$, and that $\gamma = 0.5$ gives the highest results:

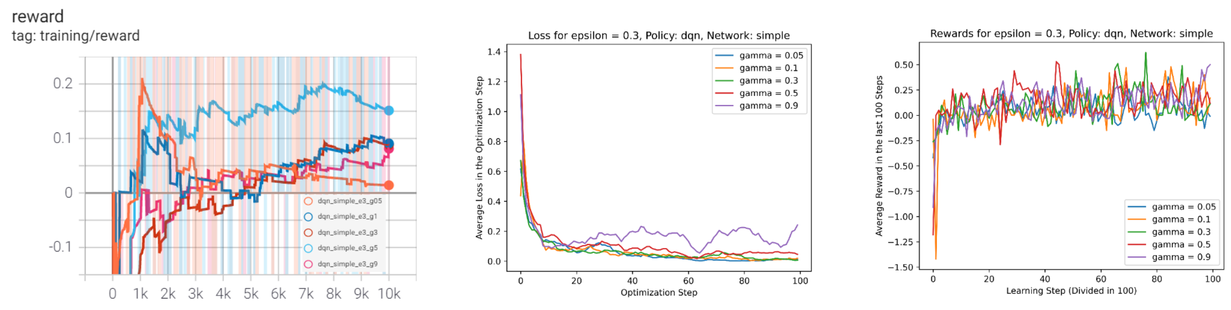


Figure 3: Q-Learning Algorithm performed with the simple DQN (model 1), $\epsilon = 0.3$ and various γ. **LEFT:** smoothed reward trendline using tensorboard, for each γ value (in the labels, $\gamma = 0.05$ is referred as 05, 0.1 as 1 and so on). **MIDDLE:** Loss convergence as an average loss for each optimization step. **RIGHT:** Reward Average every 100 iterations for each γ value.

Overall, the final converged reward is similar to the low-ϵ section, and again a moderate $\gamma = 0.5$ shows good balance and highest results.

$\epsilon = 0.5$, had shown fairly similar results.

High Maximal Exploration Rate - $\epsilon = 0.9$

We can see from the graph below that when using a very high max-exploration rate, $\epsilon = 0.9$, the convergence rate is very slow, $\sim 7k$ iterations, and for $\gamma = 0.05, 0.1$ does not converge to a positive reward. That can be explained by the fact that a very high exploration rate and very low γ do not reach enough iterations to learn from the model and the future rewards is not considered enough due to the γ value.

Again as expected, $\gamma = 0.3, 0.5$ shows the best results:

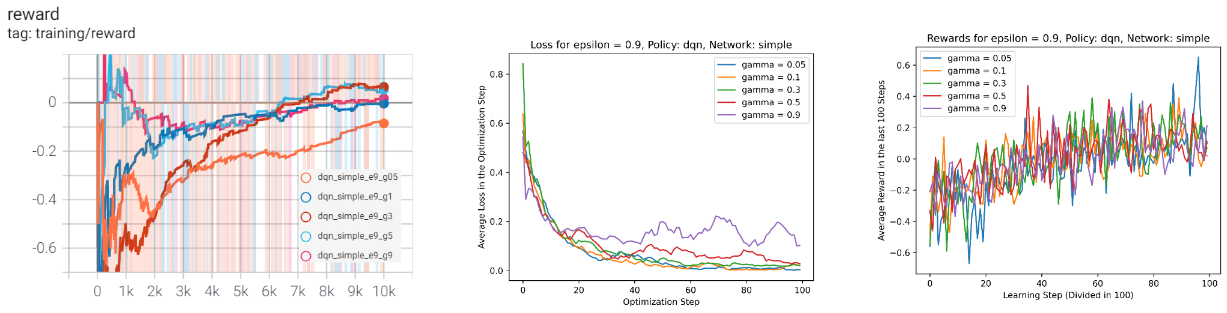


Figure 4: Q-Learning Algorithm performed with the simple DQN (model 1), $\epsilon = 0.9$ and various γ. **LEFT:** smoothed reward trendline using tensorboard, for each γ value. **MIDDLE:** Loss convergence as an average loss for each optimization step. **RIGHT:** Reward Average every 100 iterations for each γ value.

Here we can see that the loss of high γ values, has even more difficulty to converge than before, which indicates that a very high γ is inefficient - we should take into account only a reasonable amount of future rewards, where 0.9 takes most of the future rewards.

Overall, we conclude that $\epsilon = 0.9$, is too high for an exploration rate, as we need to rely more on our model for better learning.

Epsilon Decay

In the following graph, we can see the $\epsilon = 0.05, 0.3, 0.9$ decay rate over $10k$ iterations:

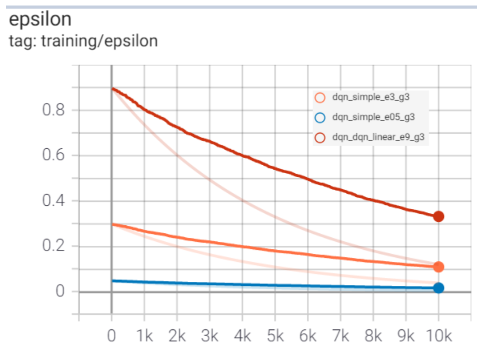


Figure 5: ϵ-Decay Rate

1.2.4 Summary

The following concludes the best results for each ϵ, with reward comparison:

ϵ	best γ	Convergence	Reward Convergence
0.05	0.5	2k	0.15
0.3	0.5	2k	0.15
0.9	0.3, 0.5	7k	0.1

Table 1: ϵ, γ Performance Comparison

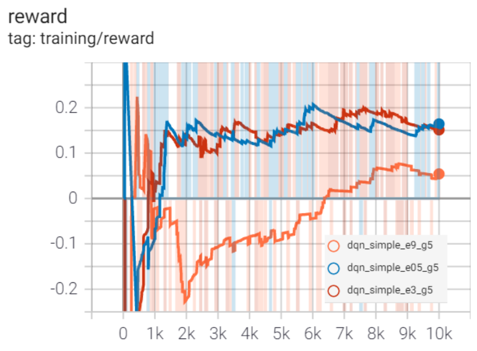


Figure 6: ϵ, γ Performance Graph Comparison

1.3 Model Evaluation

After analysing the best range of ϵ and γ parameters on the simple model , we would like to perform more complicated networks analysis, and compare between these networks.

1.3.1 CNN and Full Linear Model Evaluation

The **CNN** network, gave much higher results in general. Similar to the simple network parameters tuning results, I found out that $\gamma = 0.5, \epsilon = 0.05$ shows the highest results, with reward converging to **0.3**, and convergence at $\sim 6k$. The results of the best ϵ run is given below:

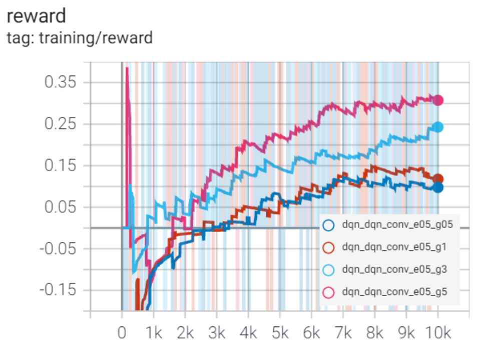


Figure 7: Q-Learning Algorithm performed with the CNN (model 3), $\epsilon = 0.05$ and various γ. Reward converging to **0.3**.

The **Full Linear Model** did worse than the simple model, and did not converge consistently to a positive reward. The best parameters were $\epsilon = 0.5, \gamma = 0.9$ that converged to 0.07 reward.

The best run is shown below:

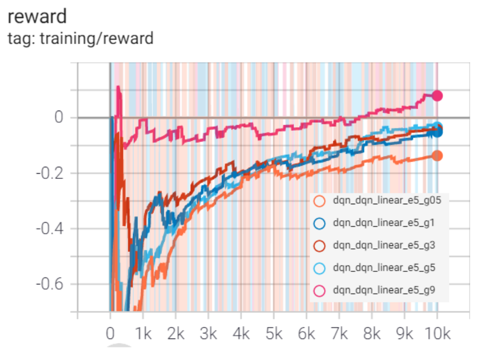


Figure 8: Q-Learning Algorithm performed with the Full Linear Model (model 2), $\epsilon = 0.5$ and various γ. Reward converging to **0.07**.

We might assume that the full linear model may need more steps (greater than 10k) as it takes the full 9x9 board, and it is more slowly converging. Also, only linear operaions, especially multiple linear layers, may not be the best strategy when building the network, and it might just "slow down" the process compared to 1 linear layer.

We saw that a convolution layer can highly improve the learning reward with good convergence rate, which indicates that the state space works well with convolutions, and is less efficient with linear layers - especially multiple linear layes that slow the convergence.

1.3.2 Models Comparison

Model	ϵ	best γ	Convergence	Reward Convergence
Simple	0.05	0.5	2k	0.15
Full Linear	0.5	0.9	8k	0.07
CNN	**0.05**	**0.5**	**6k**	**0.3**

Table 2: Models Comparison Between the Best Performances

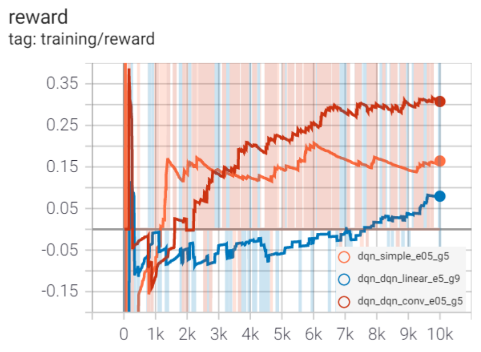


Figure 9: Models Graph Comparison of the Best Performances over $10k$ Iterations

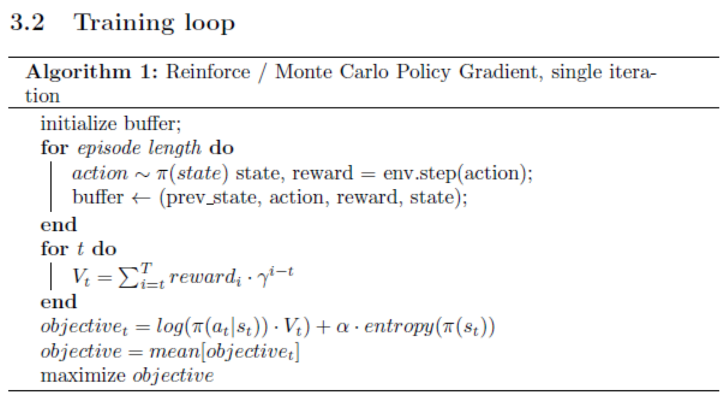
2 Monte Carlo Policy Gradient

In this model, we would like to learn an objective - learn a policy that maximizes the cumulative future reward to be received starting from any given time t until a final time T.

Since this is a maximization problem, we optimize the policy by taking the gradient ascent with the partial derivative of the objective with respect to the policy parameters.

2.1 Model Specifications

I have implemented the algorithm presented in class, with the following pseudo-code:



3.2 Training loop

Algorithm 1: Reinforce / Monte Carlo Policy Gradient, single iteration

```
initialize buffer;
for $episode\ length$ do
    $action \sim \pi(state)$ state, reward = env.step(action);
    buffer ← (prev_state, action, reward, state);
end
for $t$ do
    $V_t = \sum_{i=t}^{T} reward_i \cdot \gamma^{i-t}$
end
$objective_t = log(\pi(a_t|s_t)) \cdot V_t) + \alpha \cdot entropy(\pi(s_t))$
$objective = mean[objective_t]$
maximize $objective$
```

Figure 10: MC Policy Gradient Algorithm

2.1.1 Network and Optimization

For the prediction of actions I have used 2 networks that given a state s, learns the probability for each action.

1. **Network Architechture:**
   (a) **Simple Model:** 1-Layer network, that uses a linear layer, and lowers the state dimension the state from 9x9 squares of the board, to only 3x3 squares around the Snake's head.
   (b) **CNN:** a network with 1 convolution layer and a linear out-layer.
2. **Optimization:** For the optimization step, as in Q-learning, I decieded to use RMSprop as the optimization algorithm and Gradient Clipping method to avoid gradient explosion.

2.2 Parameters Tuning

As in Q-Learning, we have parameters we would like to optimize: α, γ, number of steps, learning rate, batch size and more.

We will focus on α and γ, as they play the most cruicial role here, and will use the same parameters as in Q-Learning, for the same justifications noted in the first section.

2.2.1 Entropy & Entropy Coefficient α

The entropy added to each objective has the purpose of regularization, and is calculated by the predicted actions' distributions p as $ent = p \cdot log(\frac{1}{p})$. Thus, as reviewed in Q-Learning section, we want to explore more in the beginning and exploit more as we progress. For that purpose, I have used a decaying α parameter for the entropy, similar to the ϵ used in Q-Learning. (The α parameter will be related as ϵ)

2.2.2 α, γ Optimization and Model Evaluation

I have extracted multiple graphs so that the impact of these parameters can be analysed.

As in the previous section, I chose a series of maximal starting $\epsilon = \alpha$ values, and for each ϵ value, I have trained the model with different γ values.

Simple Pg Model:
I ran all samples with batch size of 32 and $10k$ iterations, over the simple pg network model.
The following graphs show the best γ for each ϵ, alongside with epsilon decay rate and loss convergence:

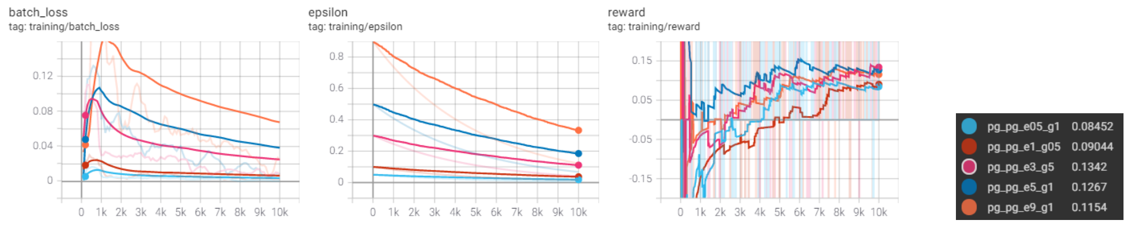


Figure 11: REINFORCE Algorithm with a simple linear network model, $\alpha = \epsilon = 0.05, 0.1, 0.3, 0.5, 0.9$ and the best performing γ for each parameter ϵ. **left**: loss convergence for each represented run. **middle:** epsilon decay. **right**: reward convergenc.

We can see that the REINFORCE algorithm requires around $8k$ steps to convergence.

Additionaly, We can see that the best reward achieved is 0.13 when using $\alpha = \epsilon = 0.3$ and $\gamma = 0.5$.

We do not need a very big exploration rate for getting good convergence, as the model itself samples from the distributions in the first step when selecting an action. Though, a moderate $\epsilon = 0.3$ is giving a farely promising results compared to other values, and also higher ϵ values show nice results.

I also ran the model on 50k iterations, to check for better convergence, though the results that were shown were fairly the same, with some small improvements in some cases. (I did not run a full analysis over 50k iterations, as my 4-yesrs old computer could not perform it well within a reasonable amount of time).

CNN Pg Model:
I ran all samples with batch size of 32 and $10k$ iterations, over the CNN pg network model.
The following graphs show the best γ for each ϵ, along side with epsilon decay rate and loss convergence:

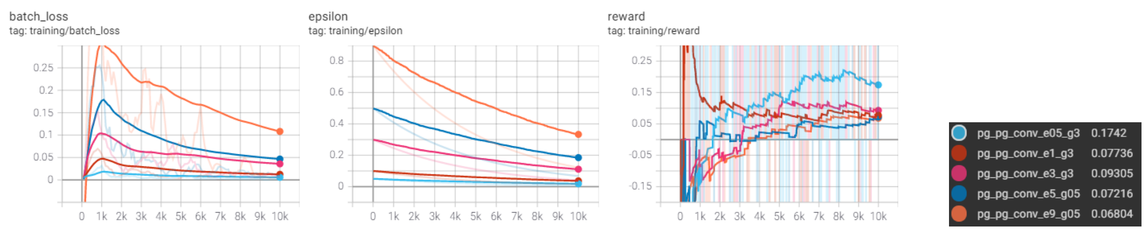


Figure 12: REINFORCE Algorithm with a CNN model, $\alpha = \epsilon = 0.05, 0.1, 0.3, 0.5, 0.9$ and the best performing γ for each parameter ϵ. **left**: loss convergence for each represented run. **middle:** epsilon decay. **right**: reward convergenc.

We can see that the REINFORCE algorithm requires around $6k$ steps to convergence, and that the parameters $\epsilon = 0.05, \gamma = 0.3$ show significantly higher results than the others.

In general, $\gamma \leq 0.3$ has performed best in all ϵ range, which is a reasonable discount factor, not too high nor too low.

2.3 Model Comparison

We saw empirically that the best runs of the simple and CNN models are as follows:

Model	best ϵ	best γ	Convergence Steps	Reward
Simple Linear	0.3	0.5	7k	0.13
CNN	0.05	0.3	6k	0.17

Table 3: Models Comparison Between the Best Performances

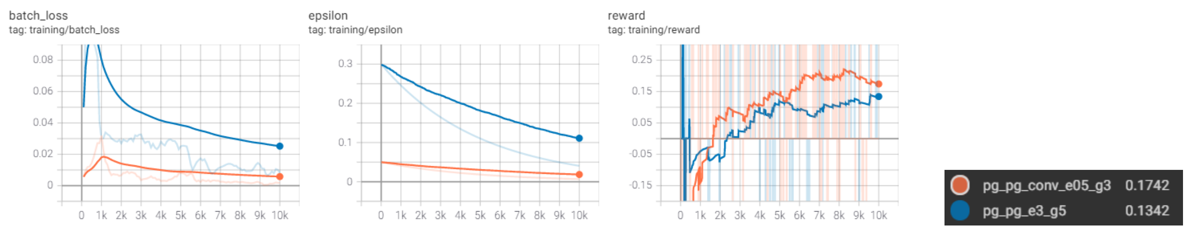


Figure 13: Models Graph Comparison of the Best Performances

We see that the CNN model needs less entropy - less regularization and random walks - as it learns the 9x9 board quite well and converges around ~6k.

The simple model is performing quite well too, as it converges around 7k and reaches a positive reward after 2.5k iterations, where the CNN model gets it by 1.7k steps.

3 Monte Carlo Policy Gradient and Q-Learning Comparison

Both models showed convergence to a positive reward, after 1-2k steps, and varied around the reward success rate, based on the α, ϵ and γ parameters, along with random processes that differ runs with the same parameters within the algorithms.

The following summary presents the best 2 models generated by the Q-Learning algorithm and the Monte Carlo Policy Gradient:

Algorithm	Model	ϵ	γ	Convergence Steps	Reward Convergence
Q-Learning	CNN	0.05	0.5	6k	0.3
MC Gradient Policy	CNN	0.05	0.3	6k	0.17

Table 4: Q-Learning vs. MC Gradient Policy with the parameters that gave the best performance

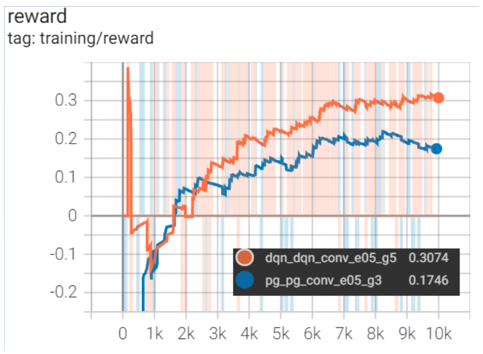


Figure 14: Q-Learning vs. MC Gradient Policy Graph over 10k Iterations, using CNN models

We can see that the CNN had shown the best results in both models.

Moreover, small $\epsilon = 0.05$ was greatly satisfying for both algorithms, when used with a CNN, and γ varied betweem 0.3 and 0.5.

That stands up to our expectations, as a CNN is theoratically suppose to capture and learn a state much better than only linear layers. Moreover, a moderate γ is reasonable, as we dont want to overly-consider the future rewards or the opposite.

As for the ϵ, it was interesting to see that in most cases a low ϵ gave enough exploration for this game's states.

Now, for completing the comparison, we would like to also examine the best results when taking the simple linear model, with both algorithms:

Algorithm	Model	ϵ	γ	Convergence Steps	Reward Convergence
Q-Learning	Simple Linear	0.05	0.5	7k	0.16
MC Gradient Policy	Simple Linear	0.3	0.5	7k	0.13

Table 5: Q-Learning vs. MC Gradient Policy

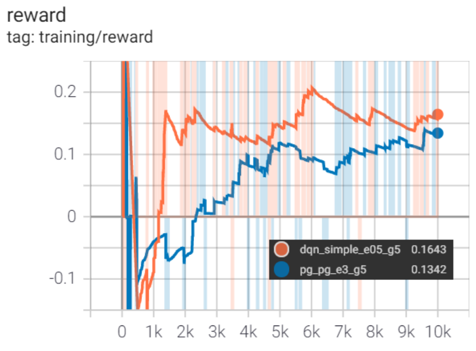


Figure 15: Q-Learning vs. MC Gradient Policy Graph over 10k Iterations, using Linear models

We see also in here, that both model satisfy with a moderate $\gamma = 0.5$, and now $\epsilon \leq 0.3$ gives high results for both - again a fairly low exploration rate.

We conclude that the CNN model improved both methods, though the Q-learning showed a more significant improvement when using a CNN, which makes sense as the runs of MC-PG are more probabilistic, and may require more iterations for convergence.

Overall, the Q-Learning algorithm using a CNN model, shows the best performance in learning, and will be my best choice.