EX4 - Advanced Practical Course In Machine Learning Reinforcement learning Group Report

Mazor David 313263006
Daniel Afrimi 203865837
Asaf Korem 206281925
Eran Zecharia 311482830
Mika Schechter 204589832

December 27, 2020

Contents

1 Introduction

In this team report, we will present the training results of the following reinforcement learning algorithms (implemented by us) after trained on a given demo problem - the game Snake (we really enjoyed the exercise!):

- Dueling Network Architectures for Deep Reinforcement Learning
- Double q learning
- Q Actor Critic
- A2C (Advantage Actor Critic)
- Comparing performance with and without reward normalization

2 Algorithms

2.1 Dueling Network Architectures

Following the work done in 'Dueling Network Architectures for Deep Reinforcement Learning'[3], we implemented the Dueling Network architecture.

2.1.1 Motivation

In many games, given certain states some actions taken do not have an effect on the future reward. In order to deal with this,after a series of convolutional layers the network splits into two streams, one for the value function which produces a scalar (V: the payoff function which estimates the reward of a series of states) and one for the advantage function which produces an output with the dimension of the action space (Q function, which gives a value for each action from a given state - similar to q-learning), These are combines to calculate the Q function.
The model learns if it should prioritize the advantage function, or choosing a certain action, or if it should prioritize the value function, or the future reward disregarding the immediate action.

In **Figures 1** we can see the architecture of a single Q-net and Dueling Q-net.

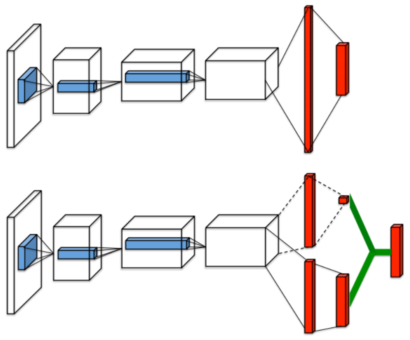


Figure 1: Single stream Q-network (top), Dueling Q-network (bottom)

The dueling architecture can learn which states are valuable (or not), without having to learn the effect of each action for each state. This is useful in states where its actions do not affect the environment in any relevant way.

2.1.2 Training the models

Parameters The training parameters and model's hyper-parameters we used for each training of the Double Q-Learning models are:

Table 1: Best Training parameters for Dueling Network

Parameter	Default value
Steps `N` steps	15,000
Optimize model every each `N` steps	128
Buffer size	1,024
Batch size	32
Learning rate	0.001
Epsilon	0.5
Gamma	0.3

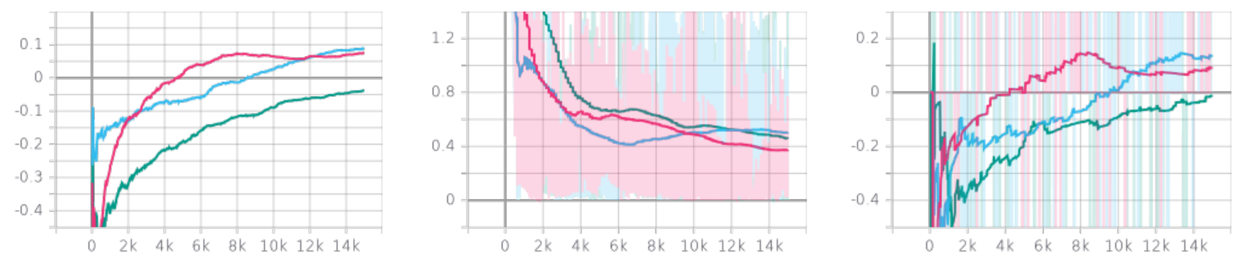
2.1.3 Results

In this section, we presented four different metrics:

- Average reward - cumulative reward on all samples until that step.
- Training loss at each optimization step.
- Reward - The reward at each step.

Exploration-Exploitation Trade-Off We handled the exploration-exploitation trade-of by following an ϵ-greedy policy during act time (choosing an action during training). We tried different values for ϵ, and looked at the graph of the rewards. As expected, the larger the epsilon is the lower cumulative reward is gained, but the final policy in the last score-scope iterations is better. The final value ϵ was chosen to be 0.5.

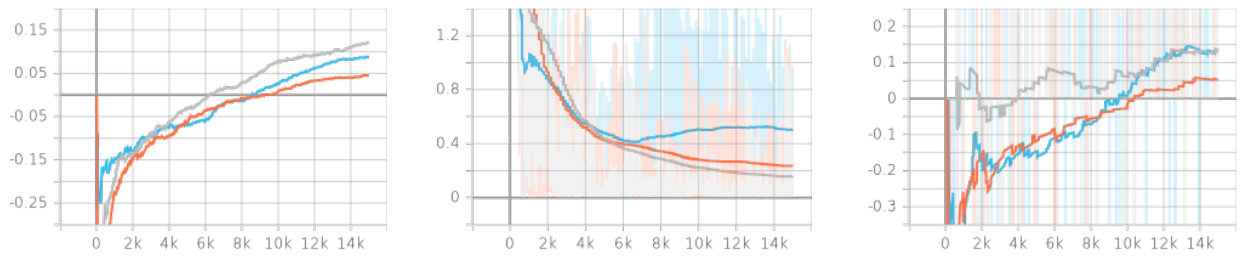
In **Figures 2** we can how different values of ϵ effect on the obtained reward of the model.



(a) Average Reward (b) Training Loss (c) Reward

Figure 2: Different Epsilon, ϵ=0.3 (pink), ϵ=0.5 (blue), ϵ=0.8 (green)

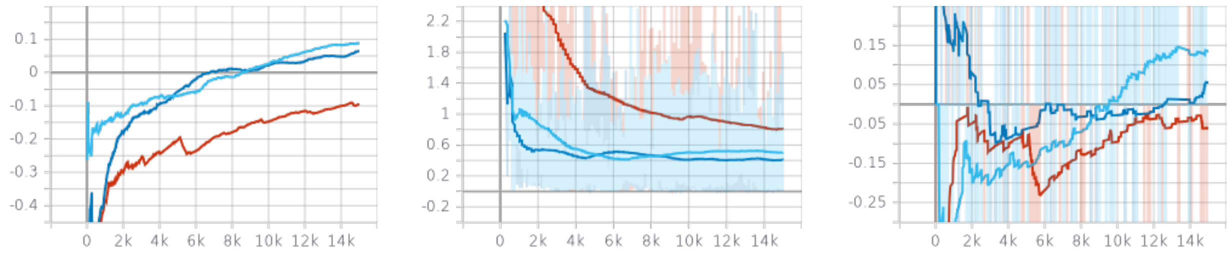
In **Figures 3** we can how different values of γ effect on the obtained reward of the model.



(a) Average Reward (b) Training Loss (c) Reward

Figure 3: Different Gamma factor, γ=0.3 (gray), γ=0.6 (orange), γ=0.8 (blue)

In **Figures 4** we can how different values of learning rated effect on the obtained reward of the model. for our model while the learning rate is smaller we get better result.



(a) Average Reward (b) Training Loss (c) Reward

Figure 4: Different learning rates, lr=0.0001 (light blue), epsilon=0.001 (blue), epsilon=0.01 (red)

2.2 Double Q-Learning

2.2.1 Motivation and pseudo-code

In some stochastic environments the DQN algorithm (which we implemented in the first part) performs very poorly [2]. The traditional DQN tends to over-estimate action-values, in which might lead to unstable training and low quality policy.

The over-estimations are result of a positive bias that is created in the approximation of the maximum expected value, since the Q-Learning algorithm uses the maximum action-value for it.

The Double Q-Learning (DDQN) algorithm is an alternative way to approximate the maximum expected value. The DDQN algorithm uses a separated Q-value estimator (the target model), which is updated after every `N` optimization steps with the weights of the primary model. With these independent estimators, we can less-biasly Q-value estimates of the actions selected using the opposite estimator.

```
Algorithm 1 : Double Q-learning (Hasselt et al., 2015)
  Initialize primary network $Q_\theta$, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $\tau << 1$
  for each iteration do
      for each environment step do
          Observe state $s_t$ and select $a_t \sim \pi(a_t, s_t)$
          Execute $a_t$ and observe next state $s_{t+1}$ and reward $r_t = R(s_t, a_t)$
          Store $(s_t, a_t, r_t, s_{t+1})$ in replay buffer $\mathcal{D}$
      for each update step do
          sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
          Compute target Q value:
              $Q^*(s_t, a_t) \approx r_t + \gamma\, Q_\theta(s_{t+1}, argmax_{a'} Q_{\theta'}(s_{t+1}, a'))$
          Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
          Update target network parameters:
              $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$
```

Figure 5: Pseudo-code of Double Q-learning algorithm [2].

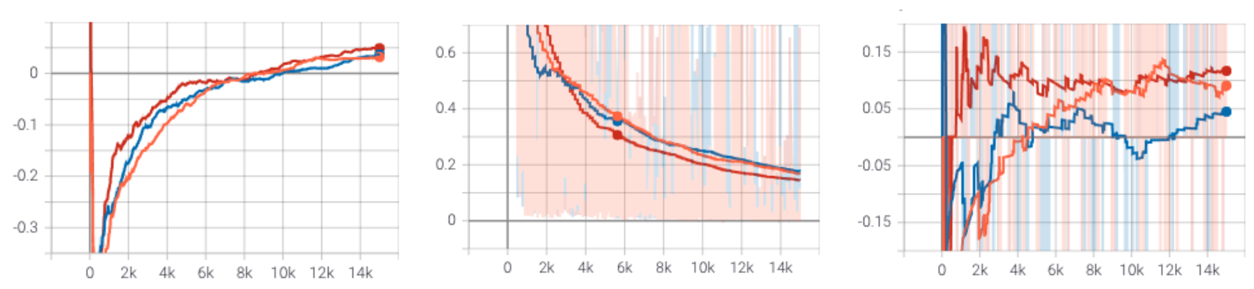
2.2.2 Training the models

Parameters The training parameters and model's hyper-parameters we used for each training of the Double Q-Learning models are:

Table 2: Training parameters for Double Q-Learning

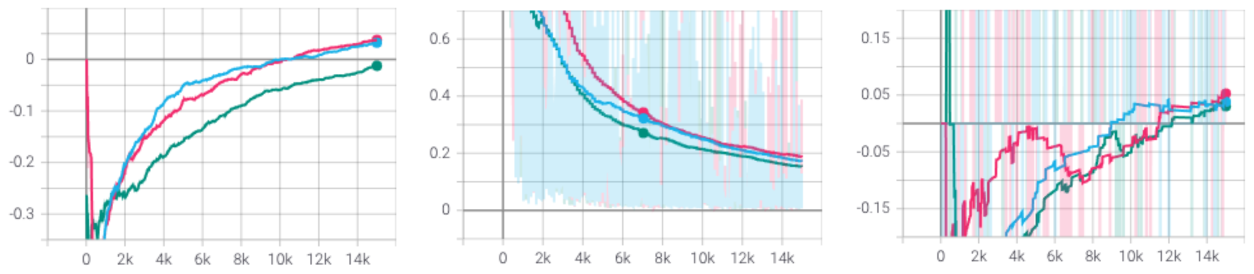
Parameter	Default value
Steps `N` steps	15,000
Optimize model every each `N` steps	128
Buffer size	1,024
Batch size	32
Learning rate	0.001
Epsilon	0.4
Gamma	0.6

2.2.3 Results



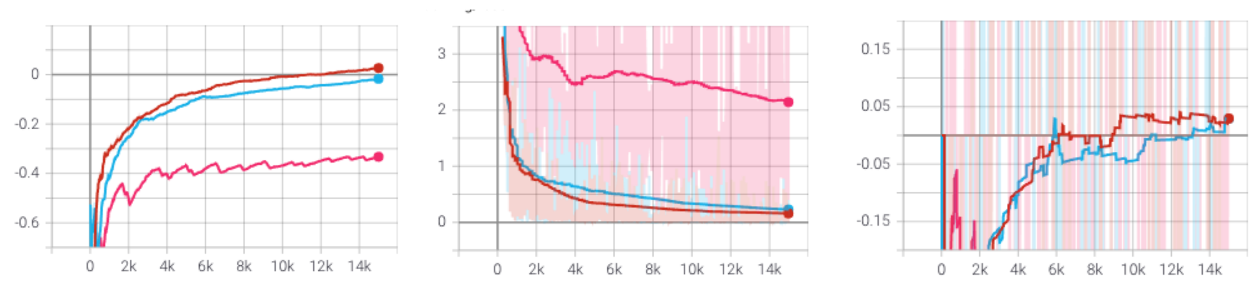
(a) Average Reward (b) Training Loss (c) Reward

Figure 6: Different Gamma factor, γ=0.7 (red), γ=0.5 (blue), γ=0.3 (orange)



(a) Average Reward (b) Training Loss (c) Reward

Figure 7: Different Epsilon, ϵ=0.7 (green), ϵ=0.5 (pink), ϵ=0.3 (light-blue)



(a) Average Reward (b) Training Loss (c) Reward

Figure 8: Different learning rates, lr=0.001 (red), lr=0.01 (light-blue), lr=0.1 (pink)

Observations Although all different variations of the Gamma and Epsilon factors performed pretty-bad (with almost zero average reward), we do see that big Gamma performs better from smaller Gamma and small Epsilon performs better than big Epsilon, therefore the exploration factor should be small enough and the Gamma factor should be big enough.

Also, as once can notice from the graphs, the Double Q-Learning model converge slower than the regular Q-Learning model. Moreover, in the Double Q-Learning we see a positive average reward after 10,000 steps of training, while in the Q-Learning model we have positive results after only around 5,000 steps.

2.2.4 Additional Notes

Conclusion Even though this algorithm makes more "sense" then the regular DQN algorithm, it has no better performance and even worse.

Exploring the algorithm We tried to explore the algorithm a bit more with different approaches that did not worked better than the existing one (even much worse), one example is we tried to optimize the model after each step and optimize the target model 32 optimizations. We also tried different combinations of hyper-parameters and still got pretty-small average reward.

2.3 Q Actor Critic

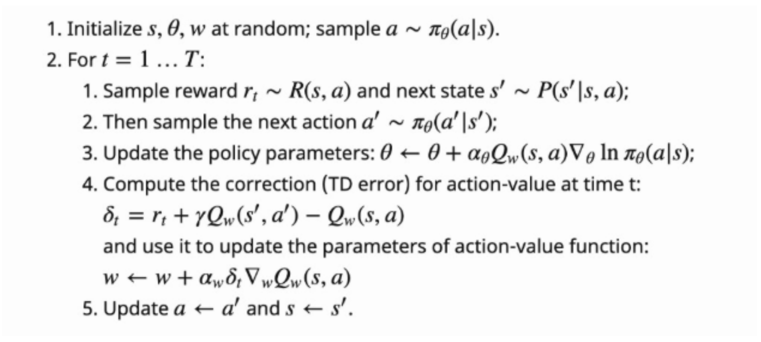
This model is trying to estimate two different functions:

- The Q function (same as in DQN algorithm)
- Function which maps from a state to a probability distribution (same as in PG algorithm)

Q Actor Critic's main principle is to split the model into two separate parts. The actor takes as input the state and outputs the best action. In addition, it controls how the agent behaves by learning the optimal policy. The critic evaluates the action by computing the Q values of the current state and action.
The actor and critic evolve in parallel. The actor evolves according to the critic, the optimization tweaks the probabilities for each action in a given state by maximizing the critic. The critic evolves according to the explored environment, the Q function is approximated by the Neural Network using the Bellman equation.

The policy is called the actor because it selects actions based on current actions and states, and the Q value function is called critic because it criticizes the actions made by the actor.
This is a policy gradient algorithm. However, compared to Monte-Carlo policy gradient, the algorithm regards not only the reward of a given trajectory, but also the Q values of each state and action in the trajectory.

2.3.1 Psuedo Code



```
1. Initialize $s, \theta, w$ at random; sample $a \sim \pi_\theta(a|s)$.
2. For $t = 1 \dots T$:
    1. Sample reward $r_t \sim R(s, a)$ and next state $s' \sim P(s'|s, a)$;
    2. Then sample the next action $a' \sim \pi_\theta(a'|s')$;
    3. Update the policy parameters: $\theta \leftarrow \theta + \alpha_\theta Q_w(s, a) \nabla_\theta \ln \pi_\theta(a|s)$;
    4. Compute the correction (TD error) for action-value at time t:
       $\delta_t = r_t + \gamma Q_w(s', a') - Q_w(s, a)$
       and use it to update the parameters of action-value function:
       $w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s, a)$
    5. Update $a \leftarrow a'$ and $s \leftarrow s'$.
```

Figure 9: Q Actor Critic Psuedo Code [1]

2.3.2 Model architecture

We chose to do it in two ways: one is with two-headed neural network and the other is with two separate networks. As for the double headed network, the architecture of both the policy network and the Q approximation network are the same therefore we unified the networks into one with two outputs: one is the q values, and the other is the probabilities vectors which is just a softmax activation on the q values. this way the loss is calculated as:

$$-p_{objective} + q_{objective} - \alpha * H(probabilities)$$

That is, we summed the loss of the two networks and try to minimize it using the gradient descent algorithms. Where the $p_{objective}$ is the policy network objective which is the mean of the logs of the probabilities given the actions. The $q_{objective}$ is equal to the q loss as in the dqn algorithm. The entropy is as in the REINFORCE ALGORITHM. Here is an illustration of network architecture:

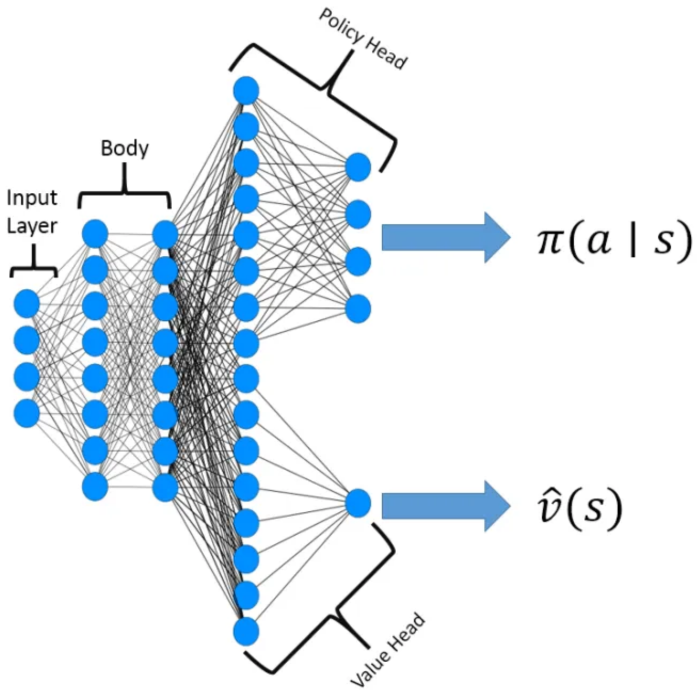
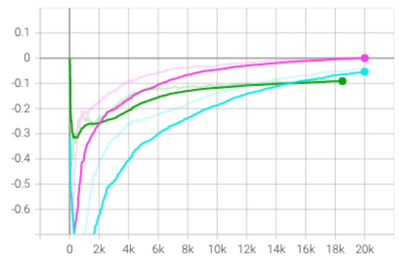
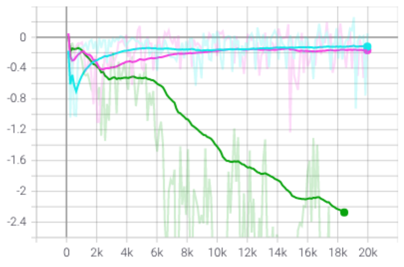


Figure 10: two headed architecture [1]

2.3.3 Results

Result of the separate networks architecture:


(a) Average Reward


(b) Training Loss

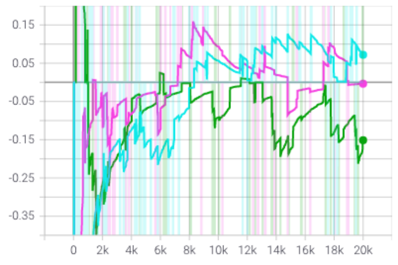
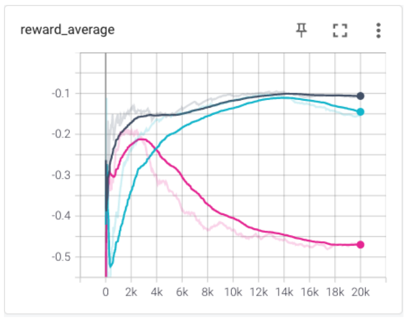
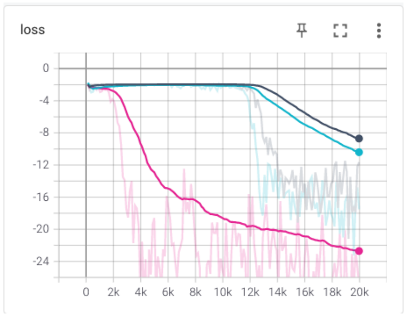

(c) Reward

Figure 11: Different Gamma factor, γ=0.7 (green), γ=0.5 (pink), γ=0.3 (light-blue)

Results of the two-headed architecture:


(a) Average Reward


(b) Training Loss

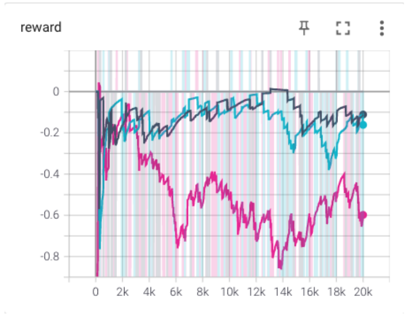

(c) Reward

Figure 12: Different Gamma factor, γ=0.7 (pink), γ=0.5 (light-blue), γ=0.3 (dark-blue)

From what we see the separate headed is performing better. we assume the reason for that is the loss of the two headed, since it is a sum of the two networks the gradient descent might change the q relevant parameters when the p objective is low and vice versa. That is, we mixed the gradient descent process of two different objectives and one network might be "punished" when the other is the culprit.

2.4 A2C

For A2C algorithm we introduce the advantage function:

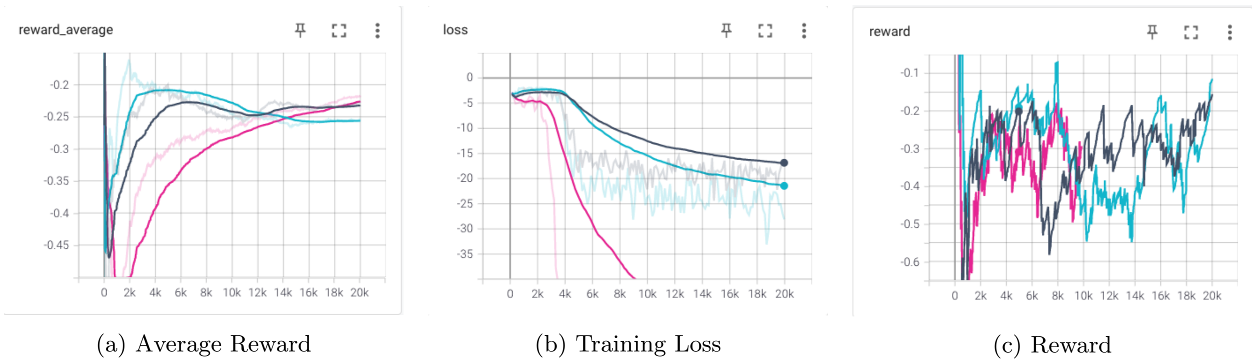
$$A(s_t, a_t) = Q_w(s_t, a_t) - Vv(s_t)$$

The advantage function's main goal is to compute how better an action is compared to the others at a specific state. This means how better is is to choose a specific action while comparing to the average action at a specific state.
For implementing this algorithm, we changed the Q Actor Critic to learn the advantage values instead of the Q values. In this way, the evaluation of a specific action is based on how much better the action can be for a specific state and not only on how good the action is (without comparison).
Therefore, the advantage function reduces the high variance of policy networks and stabilizes the model. This is the advantage of the advantage function :)

Regarding the psuedo code, it is the same as the Q Actor Critic, but instead of multiplying by $Q(s, a)$ we multiply be $Q(s, a) - V(a)$.

2.4.1 Results

As one can see from the graphs below, the A2C policy performed poorly bad, after few thousands of steps it also had sudden drop in the performance. We tried different models and played with the hyper-parameters but did not got any better results.



(a) Average Reward (b) Training Loss (c) Reward

Figure 13: Different Gamma factor, γ=0.7 (pink), γ=0.5 (light-blue), γ=0.3 (dark-blue)

2.5 Comparing performance with and without reward normalization

When having both negative and positive rewards, they might cancel each other out when adding positive and negative reward values equally. In this scenario, nothing really changes.

The optimizer is trying to minimize the loss (i.e., maximize the reward), that means that the optimizer is only interested in the delta of the values, the gradients, and not on their absolute value or sign. When the agent performs rather badly, it receives much more bad rewards than good rewards. The normalization makes the gradient steeper for the good rewards and shallower for the bad rewards.

We compared the two models from the first section (DQN and Vanilla Reinforcement) with and without reward normalization.

2.5.1 Results

As we can notice from the results, adding the reward normalization significantly improves the performance (average reward).

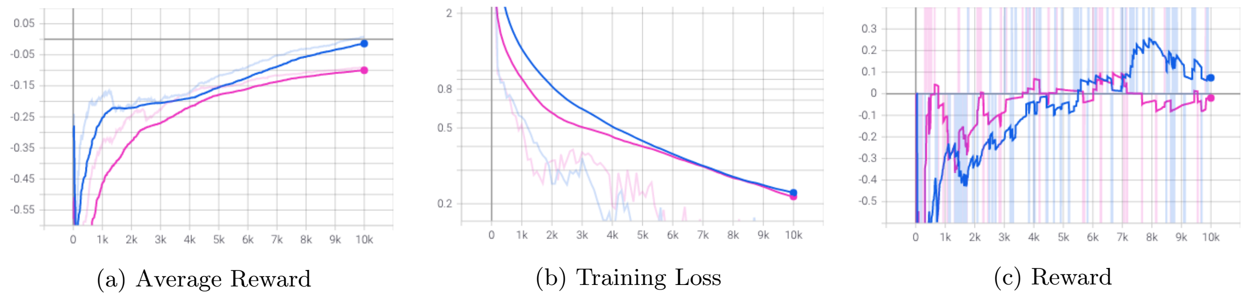
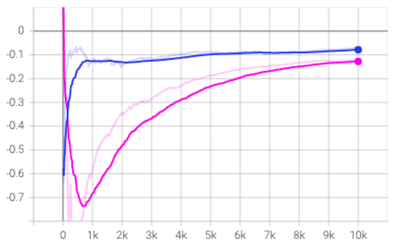
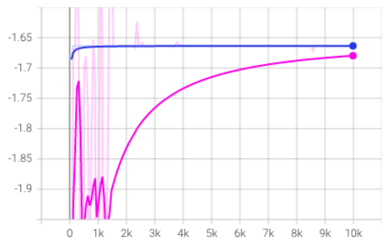

(a) Average Reward (b) Training Loss (c) Reward

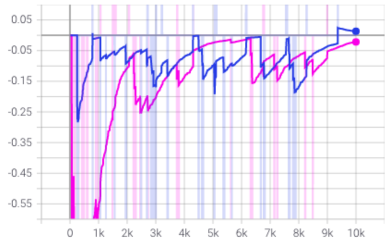
Figure 14: DQN, Without Reward Normalization (pink), With Reward Normalization (blue), trained over a simple model with small state



(a) Average Reward



(b) Training Loss



(c) Reward

Figure 15: Reinforcement Vanilla, Without Reward Normalization (pink), With Reward Normalization (blue), trained over a simple model with small state

References

[1] DataHubbs. *Two-Headed A2C Network*. URL: `https://www.datahubbs.com/two-headed-a2c-network-in-pytorch/`.

[2] Hado van Hasselt. *Double Q Learning*. URL: `https://arxiv.org/pdf/1509.06461.pdf`.

[3] Google Deep Mind. *Dueling Network Architectures for Deep Reinforcement Learning*. URL: `https://arxiv.org/pdf/1511.06581.pdf`.