Deep Q Networks analysis: divergence and problem solving.

By Angelo Groot, Jordan Earle, Melle Vessies and Reitze Jansen.

Video 1) - *Example of a successful application of a Deep Q-network applied to the Atari game "breakout"*

Deep Q Networks (DQN) is a family of deep Reinforcement Learning (RL) algorithms which leverage the flexibility of neural networks to handle reinforcement learning tasks which would be too difficult or impossible to solve with tabular methods due to their large state space. Instead of trying to learn a tabular Q-function, a DQN tries to learn a continuous mapping from the state to an estimate of the value for each of the actions from that state. DQN is a popular architecture that can be used to solve a wide variety of tasks, however, they are far from perfect. A problem which DQN's suffer from is a phenomenon known in RL as the "deadly triad"[1],which refers to reinforcement learning approaches where **off-policy learning**, **function approximation** and **bootstrapping** are used in tandem. The simultaneous appearance of the methods from the deadly triad generally results in a highly unstable, volatile learning process caused by divergence of Q-value estimates. To counter the issues caused by the deadly triad many "tricks" have been constructed to stabilize the learning process such that we can still profit from the benefits of DQN's.

In this blogpost we try to understand how some of the common tricks can be used to stabilize training. For this we have created a set of experiments to examine the performance and divergence of the DQN in different environments. We then visualize the results of our experiments in such a way that we can clearly show when each method diverges and what the effect of this divergence is.

The importance of Divergence and Stability

We ideally want to have algorithms that do not only learn, but also learn consistently. This means we want an algorithm to be as independent as possible from factors such as hyper parameters and environments. We thus want stability in addition to just performance. The ultimate goal of the analysis done in this blog post is to understand one of the main factors that influences stability, divergence. Formally we define divergence as the continuously worsening estimation of the value of a state action pair or, in other words, the inverse of learning. This means that even though it is only one of the factors, it is an important one, as it will cause the performance of our algorithm to get increasingly worse.

Q-learning

Before we get into Deep Q Networks it's important to give a brief overview of Q-learning. Q learning is an off policy method for learning the Q-value function (Sutton and Barto 2020)[2] . An off-policy method means that we have a behavioural policy to sample data from which is different from the target policy we are trying to learn. In the case of Q learning, we are trying to learn a greedy policy. In off policy learning, we require a coverage assumption to be met for the behavioural policy. This means that the behavioural policy must have a non zero probability to visit at least all the states that the target would. Since at the start we do not know what the target policy should look like, we must have some policy which is able to visit all states.

Central to Q learning is the Q function. The Q function maps every state-action pair to a “quality” value, which indicates the expected (discounted) return as a result of performing that action in that state under the current policy. The Q-value of a state-action pair under a policy π is defined as the expected (discounted) return when performing the action in that state:

$$Q_\pi(S_t, A_t) = \mathbb{E}[G_t | S_t = s, A_t = a] = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a]$$

Where $Q_\pi(s, a)$ is the Q-value of state-action pair s and a, G_t is the (discounted) return at timestep t, R_t is the observed reward at timestep t and γ is the discount factor.

Q learning (Watkins, 1989[2]) is a Reinforcement Learning algorithm from the Temporal Difference family of algorithms. In Temporal Difference (TD) algorithms we use a bootstrapping method where, instead of the return from an episode, the update is based on the difference in rewards between n steps (in our case $n = 1$). This allows for training with single transitions rather than needing an entire episode.

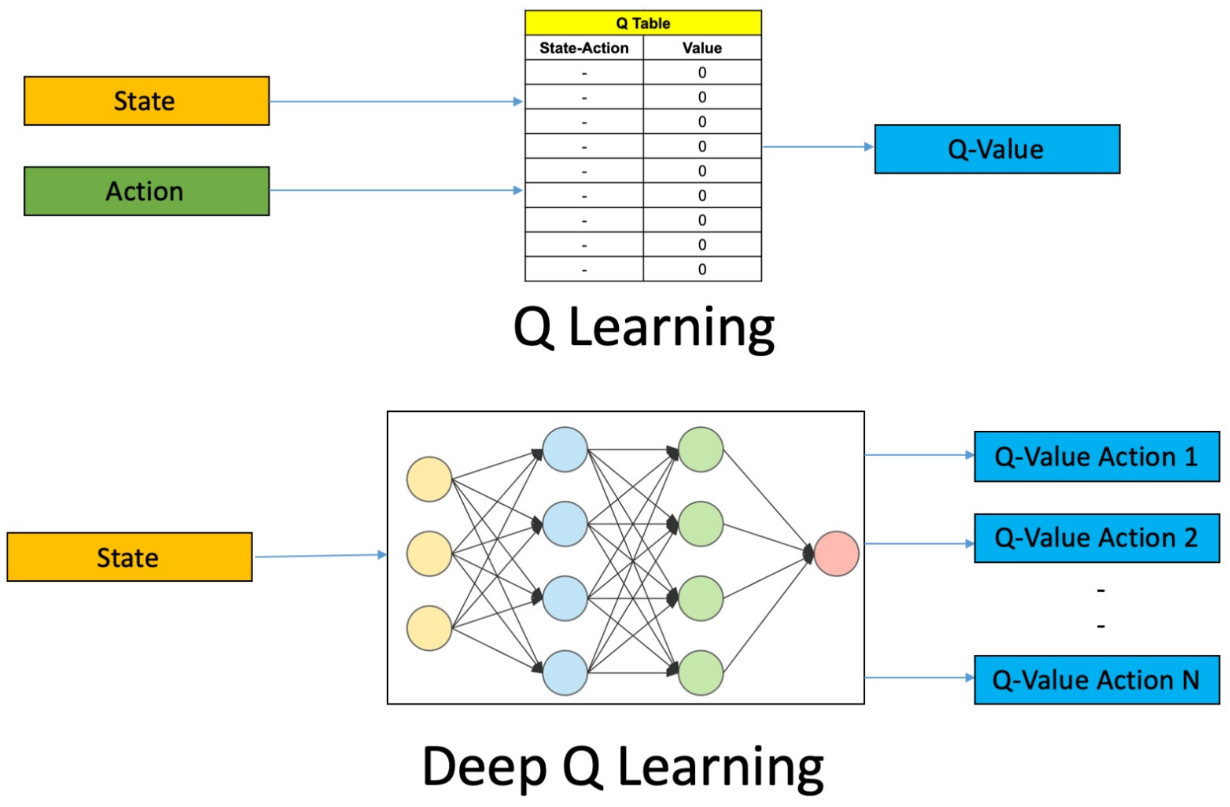
If our estimate of the expected return is correct, we would expect that the only difference between our current estimate and the estimate for the next step is the reward we would obtain. The TD error is a quantification of how much the estimated value differs from the true return for this timestep, in other words, how good our prediction is. Formally, the TD error is given by:

$$\delta_t = R_t + \gamma max_a Q_\pi(S_{t+1}, a) - Q_\pi(S_t, A_t)$$

Where $max_a Q(S_{t+1}, a)$ is the estimate of the expected return that we get after taking the estimated optimal action from state S_t under a certain policy π. If we then minimize this TD error we are thus effectively making our estimate more accurate. This use of the max operator causes a maximization bias which means the network will systematically overestimate the value of state-action pairs. As can be seen in the defintion, the TD error is a measurement of the predicted reward against the actual reward. In Q-learning these TD-errors are used to update the Q value estimates. By taking a step in the direction of the difference between the real value of the reward and the estimated, we can closer approximate the true expected return. This can also be seen in the Q-learning update equation, since the difference between the return G_t and γG_{t+1} should only be R_{t+1}.

By following a behaviour policy which is also based on the Q-function, we can learn the Q-values iteratively while exploring in the environment. We will use epsilon-greedy in this blogpost, which has a probability of epsilon to do a random action rather than the predicted optimal action, and anneals linearly over the first 10 episodes. In this way the DQN can learn what happens when performing the best actions according to the Q-function, while also still exploring different options and therefore satisfying the coverage assumption. By following this policy we can update our Q-values using the TD error as described above which gives rise to the following update rule:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$



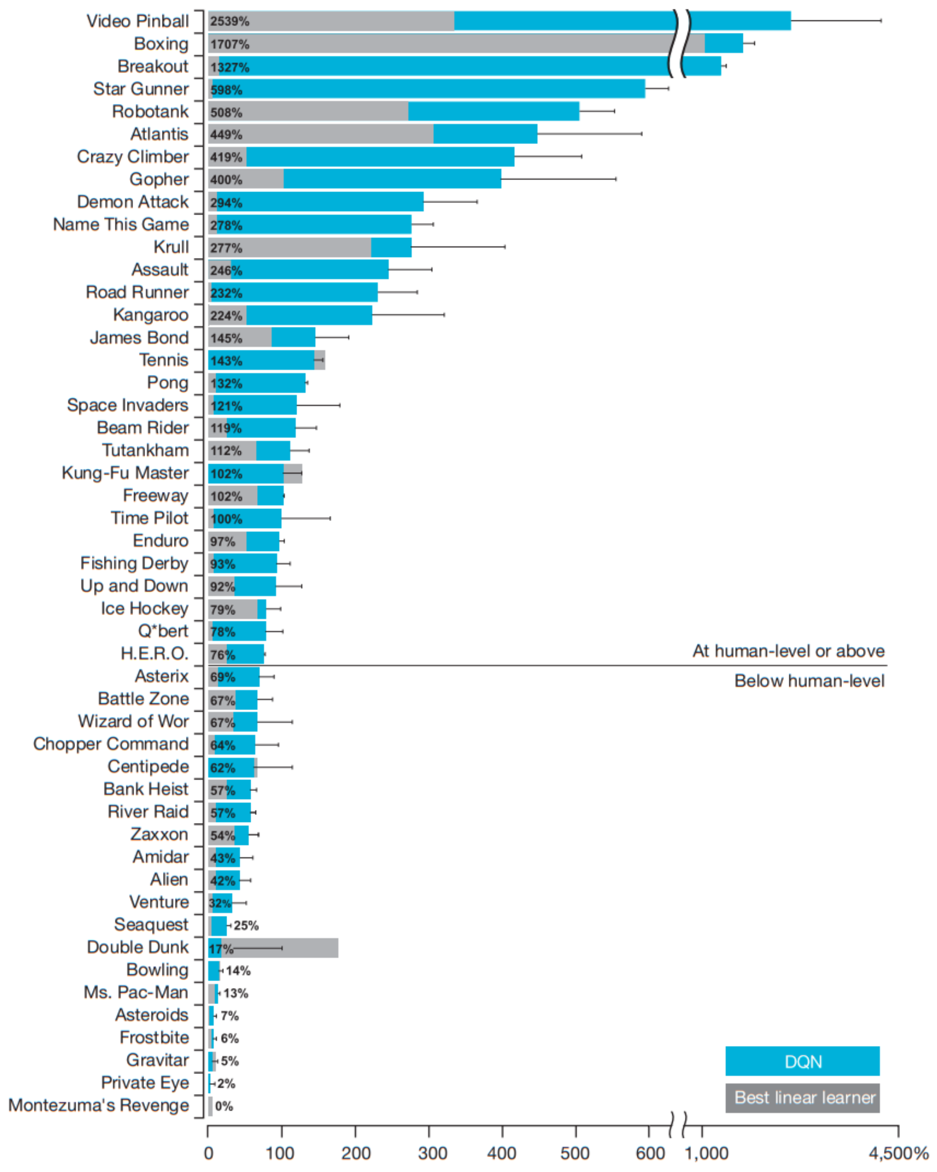
Picture 1) - *Schematic of tabular Q-learning versus deep Q-learning by Ankit Choudhary*

Deep Q-learning

A Deep Q Network (DQN) uses the same principle as Q-learning. The behaviour policy takes the action with the currently optimal value according to the current Q value function with some given epsilon chance of doing a random action, and the rewards from this action are used to update the Q value function. In the case of DQN however, the Q value is no longer represented as a table (a list of values corresponding to a state and action pair), but as a deep neural network function with the state as input and the output being a vector with the state-action values of possible actions. This allows us to have a much higher state and action space, since the values are represented by the network, which requires less space for the weights than would be required for storing all of the state action pairs for complex problems.

The difference between the two can be seen in the image from Ankit Choudhary [10], showing that the network input is just the state where the Q-Learning input would be the state action pair, and the output will be a list a Q-Values corresponding to the actions, where the Q-learning would have the output be the specific Q-Value for the state action pair input.

In neural networks, updates to the weights are done through taking the gradient of the loss with respect to the weights. In DQN, the loss is the TD error previously described, but due to the target in the DQN being the bootstrapped approximation for the Q function, taking the full gradient would cause problems due to the circular dependency. Therefore, in practice we use the semi-gradient approach in which the gradient is simplified, using just part of the true gradient. More information on the semi gradient approach can be seen in the book by Sutton & Barto [2] and the semi gradient update rule for the weights can be written as:



Picture 2) - *Results of deep Q-networks on various atari games as reported by Mnih et al.*

$$w_{t+1} = w_t + \alpha[R_{t+1} + \max_{a_{t+1}}(Q(s_{t+1}, a_{t+1}, w_t)) - Q(s_t, a_t, w_t)]\nabla_w Q(s_t, a_t, w_t)$$

Where w is the weights, t represents the current time steps, s and a represents the state and action, and Q is the state-action value function. by taking a small step in the direction which minimizes the TD-error , we will have a more accurate representation of the Q function. The power of the DQN can be seen in the paper by Mnih et al. [2015], where they train DQN's to complete many of the old Atari games. It was impressive to see it be able to discover longer term strategies and sequences, such as in breakout where it discovered the side tunnel strategy. The performance of the DQN compared to the best linear learner can be seen in the figure shown from the paper.

DQN is a powerful technique but it does require some tricks to work effectively since the method has all the components of the deadly triad:

- **Bootstrapping**: Using the estimation of the value function in order to calculate the expected return for a single timestep rather than using the received return for the entire episode. This can be seen in the update rule we are using.
- **Off Policy Learning**: Learning from data generated by a different policy than the one you are learning. In this case we gather data using ϵ-greedy to learn a greedy policy.
- **Function Approximation**: generalizing the statespace to some approximate domain, rather than learning (action) values for each distinct state. This is the case for DQN as it does Q-learning where it represents the Q-values for all state action pairs in a single neural network

Mnih et al. (2015) [12] combined the DQN with some tricks to prevent divergence and allow it to complete the complicated tasks. In this blog we'll look at the concrete effect of using "experience replay" and a "target networks".We further refer to either the target policy or behaviour policy acting in the environment as the agent. Now before looking at the tricks, lets first dive into the problems we want to investigate this issue on!

Environments

Environments in reinforcement learning refer to any space where agents interact to either get a task completed or to continually make good actions. This could be something as simple as learning which slot machine will give a good payoff, to putting a box from one conveyor belt to another or even as complex as a game of go. Since each of these environments can differ greatly and they each bring their own difficulties. We hypothesize that the reward landscape plays a large role in the chance of divergence for the DQN. In order to fully test what kind of problems can be seen when training a DQN, we chose 3 environments with different reward landscapes to see how they would affect the convergence of the network. For this, we have chosen 3 environments with very different reward landscapes: the Cartpole-v0, MountainCar-v0, and LunarLander-v2 environments from the OpenAI Gym Environment. OpenAI Gym [5]

CartPole-v1

In the CartPole problem, a pole is attached to an unarticulated joint on a cart, moving along a frictionless path. The objective is to keep a pole upright, between a certain angle for as long as possible. A reward of +1 is provided at every time step the pole remains upright. The episode ends when the pole falls below a 15 degree angle from the vertical axis, when the cart moves 2.4 units from the center, or when 500 steps have passed. The state the agent receives is (cart position, cart velocity, pole angle, pole velocity at tip). The actions the agent has are applying a force to either push the car to the left or the right to the kart to keep the pole upright.

Video 2) - *Example result of an episode of the CartPole-v1 environment rendered using the OpenAI Gym library*

This is an episodic environment as it does have an end of episode state, but if the agent is performing well, it could keep the pole upright for an indefinite amount of time. This would cause the reward to keep growing. The reward landscape here is uniform positive and the longer the cart keeps the pole upright, the more reward can be earned. Tends towards short episodes at the start as the agent is taking random moves and has not learned how to keep the pole upright yet and it fails after a short number of steps.

MountainCar-v0

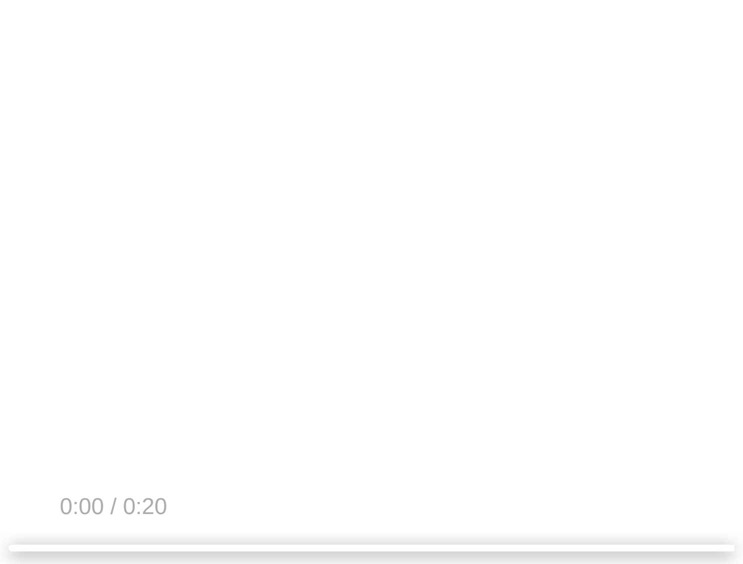
In the MountainCar problem, a car starts in between 2 mountains on a 1d track with the goal to the right. It is impossible for the car to get to the top by just heading towards the goal, so it must first learn to go backwards and then go to the right. The reward for the problem is -1 for each time step, and the episode ends when the cart reaches the goal or when 200 steps have passed. The states the agent receives are the (position, velocity) of the cart. The actions the agent has is to either push the kart to the left, no push, or push the cart to the right.

Video 3) - *Example result of an episode of the MountainCar-v0 environment rendered using the OpenAI Gym library*

This environment is also episodic as there is a clear end state. The problem in this environment is the need for exploration at the start. Since the kart has to go backwards first, it needs to remember that those backward steps were important. The reward landscape for MountainCart is the inverse of Cartpole as it has a uniform negative landscape where every time step we haven't solved the problem we get an additional negative reward of -1. This environment tends towards longer episodes at the start as the agent has not learned to first go backwards to gain momentum to go towards the goal, so it mostly times out.

LunarLander-v2

In the lunar lander problem, an agent is attempting to land on a landing pad with 0 speed after being spawned at the top of the screen with changing landscapes. Landing on the landing pad, between the flags, gives a reward of +100 and doing so safely gives the agent an additional reward of +100. Crashing into the ground gives a negative reward (-100). The agent also gets a bonus for landing safely, with +10 for each leg touching the ground and a quadratic penalty for its speed. Landing (anywhere) or crashing both end the episode. If the agent does not crash or land in 500 steps the epsiode is also terminated. The agent has 4 possible actions to its disposal to achieve this.



Video 4) - *Example result of an episode of the LunarLander-v2 environment rendered using the OpenAI Gym library*

Doing nothing, firing one of its side engines or firing the main engine, applying force from the bottom of the agent upwards (whatever may be 'up' from the perspective of the agent). Firing the engines has a cost however, although the engine has infinite fuel, firing the main engine has a cost of 0.3 and firing any of the side engines a cost of 0.03. The environment is considered solved when the agent reaches a return of 200 which happens when landing with low velicity on the pad while spending little fuel.

Experimental Setup

DQN Base Model

Divergence can be seen when a value of a state action pair is grossly overestimated and its estimation keeps growing over time. This can cause instability in learning. In order to examine the stability of the learning and the divergence, we selected the above environments for their diverse reward landscapes. In this blogpost, we will be using a DQN network consisting of 2 hidden layers with size 128 and relu activation (with no activation on the last layer). During training we use a batch size of 64 and a learning rate of 0.001.

Experiment Design

To determine the stability of the models and determine if there is divergence we look at both the relative performance (Return, G) of the models in the different environments as well as the growth in Mean Squared TD errors (Γ_δ). The performance is examined as network divergence causes instability in the learning of the networks which causes poor performance in some cases. In order to measure divergence rate, we define Γ to be the order of growth of the mean squared TD-error (MSTD) between the first and the final training episode, measured in factors of 10 (thus, a growth factor of $\Gamma = 4$ means that the MSTD has grown by a factor of 10^4). In this way, Γ will show if the MSTD shrinks or grows instead, indicating if the Q-value estimates are converging or diverging.

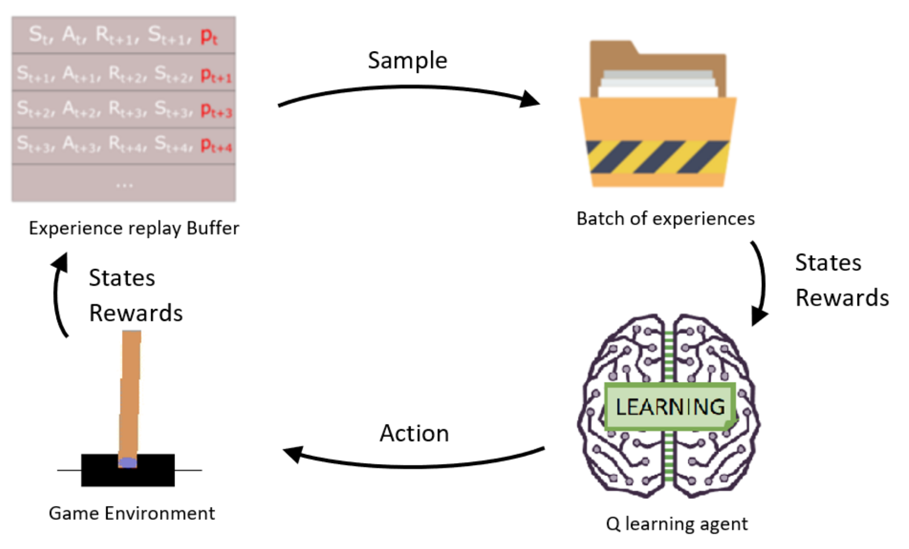
Comparing these methods should be done in a way that has an equal playing field. Since the tricks change how the agent learns and what it uses to learn, it might be the case that one setting of hyperparameters favours the performance of some methods over others. As Henderson et. al [8] find, different algorithms have different hyperparameters where they perform well. As such to make an at least somewhat fair comparison where the model hyperparameters are not the (sole) reason for the divergence, we will be performing a grid search over the discount factor γ and the exploration probability (ϵ). By doing this we will investigate how the algorithms behave across the different environments with different hyperparameters. In order to more robustly evaluate the effect of hyperparameters, we train the models on 5 different random seeds (1-5) and evaluate the target policy from each trained model on 10 different test seeds (100-109).

DQN Tricks

Now that we have selected environments with diverse landscapes, we will now introduce 2 common tricks used to combat divergence and to increase stability. These tricks are the experience replay and the target network.

Experience Replay

Now, let us say the agent tries to become good at whatever task is given by changing its actions according to what it encounters in its ventures, using the above update rule to update the parameters that determine its estimated Q-values. . It could do so whenever it gets a new reward (whenever it gets enough data to perform a batch update) but this means the agent continually only is adjusting to what it has encountered most recently. This problem was - according to Barto and Sutton [2] - first described by Long-Ji Lin in 1992 [4] . He made the observation that it might be wasteful to throw away experiences as some might be rare or costly to obtain in situations with physical agents.



Picture 3) - *Cartoon about the function of experience replay by MCC AI*

As a result he proposed to reuse experiences, which he defines as a pairing of state, the action taken, the reinforcement (reward) obtained and the following state. He calls reusing past experiences *Experience Replay*. In the original paper by Lin the 100 most recent episodes (lessons) are sampled randomly, with recent experiences being exponentially more likely. However, more recently the approach has been to either sample experiences uniformly with recency governed by the size of a replay buffer (see this paper for example [6]) or by using prioritization [7] to determine which experiences should be replayed. In this blog we will look at whether standard (uniform sampling) experience replay can improve the performance of DQNs through reducing divergence.

Target Network Fixing

In order to help with the problem of the target (TD error) constantly changing with each update (making it hard to converge towards anything), the network parameters are saved to be used in computing the target for the TD error during the next 2 episodes during sampling. This fixed network is called the target network and allows for a more stable update by decreasing the correlation between the network estimates and the target. The target network, in combination with the experience replay, is part of what made the DQN in the paper by Mnih et. al. (2015)[12] perform well on the Arati games. The target network in our implantation starts at 0 updates and then every 2 episodes. This is because we want the network to be dependent on more recent samples than those seen more than 2 episodes prior.

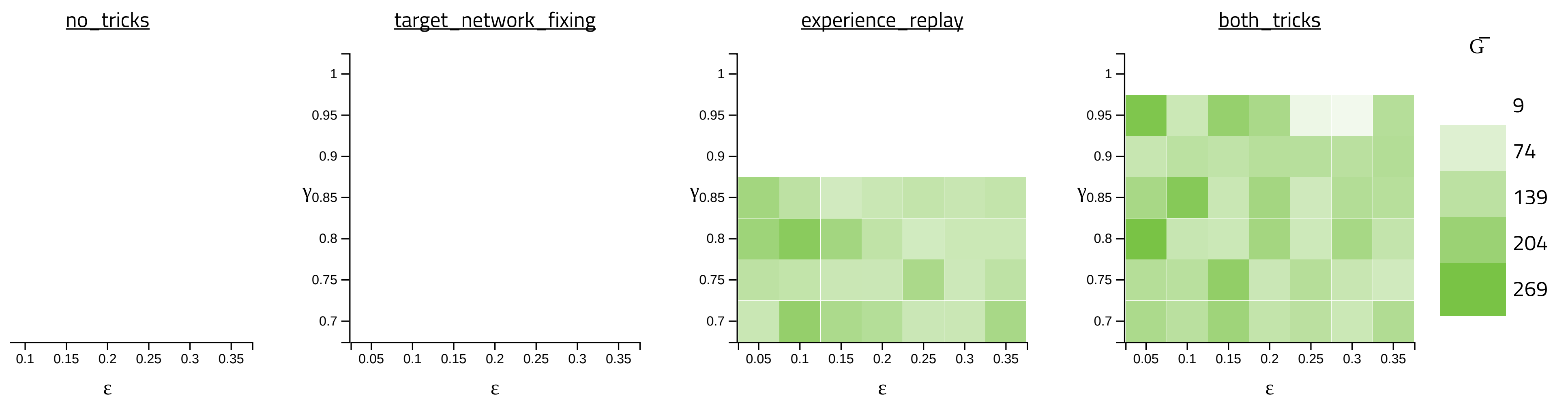
Given this setup, in order to examine the stability of training and the divergence we will be examining 3 things. The first is the average return of the final trained model for each set of hyperparameters over the 10 test seeds in a heatmap, which will give an indication of the stability and performance of the networks. This will be useful in determining if a network is diverging or just not performing well due to poor training. The second is the average growth magnitude Γ of the Mean Squared TD errors during training over the 5 training seeds, which will show when our estimates are diverging. The final is the average performance for each of the trick settings over episodes during training and 0.2 standard deviations in order to see how much spread there is and to get more insight into the training behaviour

Results and Discussion

In this section we will be looking at the results during the grid-search, which show the average target-policy returns $\bar{G}$ and the divergence measure $\bar{\Gamma}$. We also look at the average performance for each of the trick settings over episodes during training.

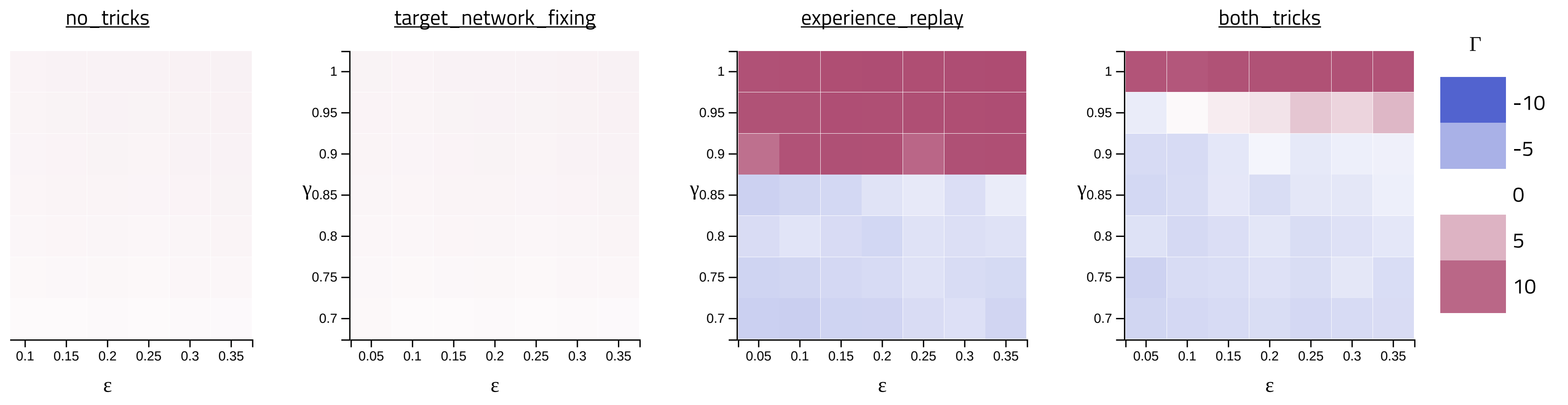
Cartpole-v1

We will start by examining the Cartpole environment. As previously mentioned, the environment will tend to have shorter episodes at the start while agents are learning. It has a positive uniform landscape, as each step the pole stays upright gives +1 reward. The heatmaps for the vanilla DQN, the two tricks and the combination of tricks can be seen in the figures below.



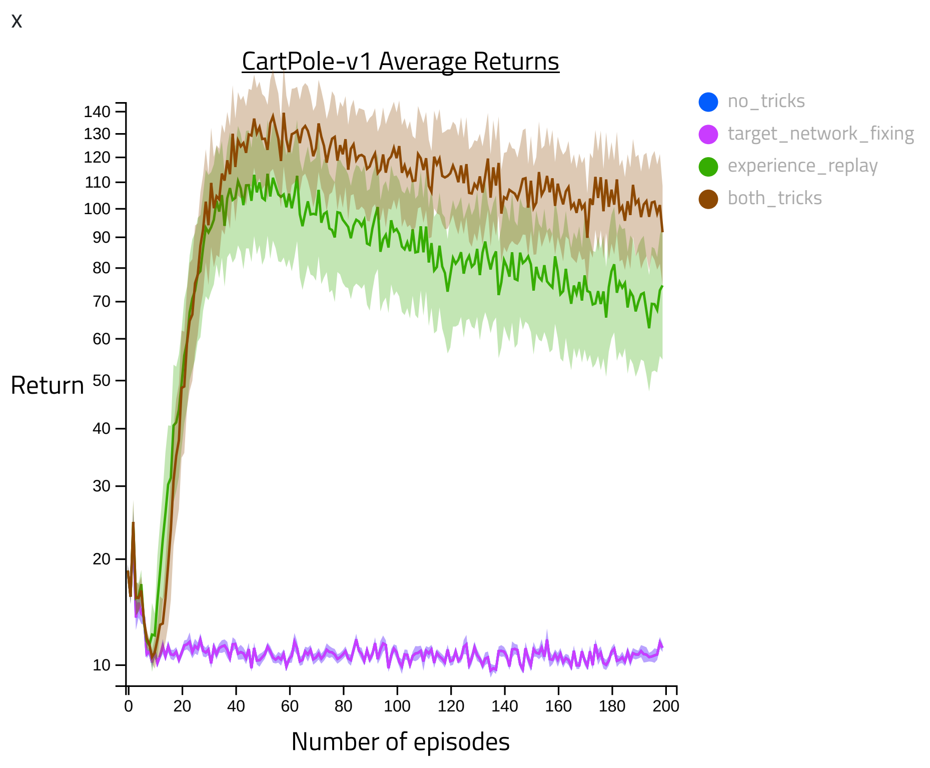
Graph section 1) - *Heatmaps of the returns of the gridsearch of the different trick combinations on the CartPole-v1 environment.* ***Left)*** *No tricks -* ***Center-left)*** *Target network fixing -* ***Center-right)*** *Experience replay -* ***Right)*** *Both tricks*

Divergence:



Graph section 2) - *Heatmaps of the* Γ *of the gridsearch of the different trick combinations on the CartPole-v1 environment.* ***Left)*** *No tricks -* ***Center-left)*** *Target network fixing -* ***Center-right)*** *Experience replay -* ***Right)*** *Both tricks*

From these heatmaps we can see that to solve the Cartpole problem it is very important to include experience replay, as without it (in the left and center-left heatmaps) the agent is not able to perform the task well for any hyperparameter setting. It is notable that this lack of performance does not appear to be caused by a high divergence rate but rather the inability to learn at all. This may be caused by the uniform reward landscape of the CartPole-v1 problem, where it is very hard to distinguish between good and bad actions on a short term.



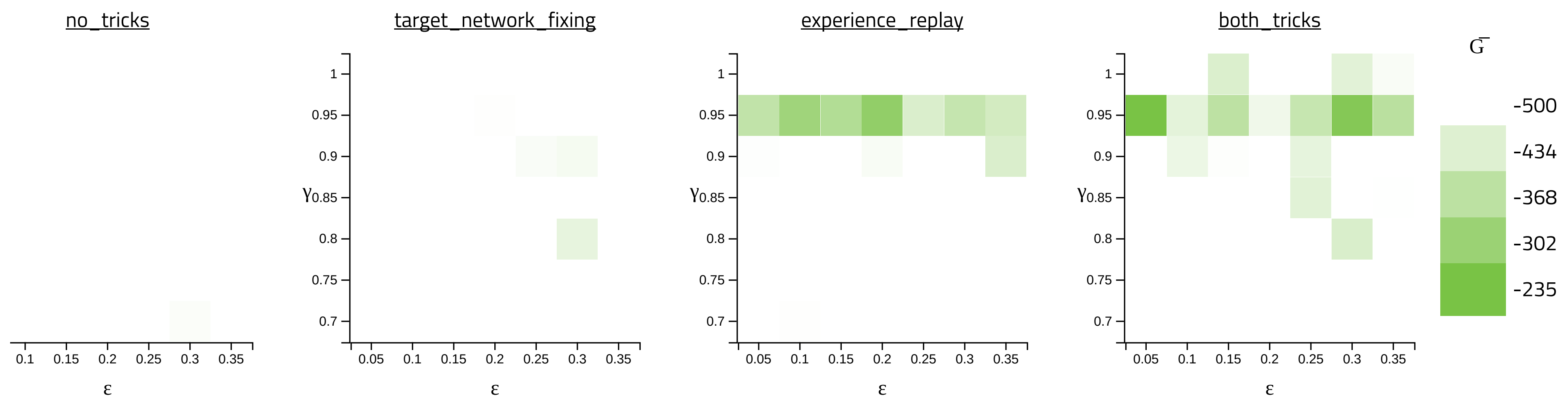
Linegraph 1) - *Line graph of the average returns over all hyper parameters for each trick (combination) on the CartPole-v1 environment. Standard deviations shown are 1 fifth of the original. Hover over lines or trick names in the legend to highlight parts of the graph.*

It can be seen from the returns heatmap that the network begins to train once the experience replay is added, and trains better when the target network is added to that. When taking a look at the divergence heatmap , it can be seen that when the model is failing it is because of divergence, and that when the target network is added it improves the training and reduces the divergence. This can be interpreted as the trick helping, but only when the DQN is learning useful estimates to begin with. This may be attributed to target network fixing mitigating the estimate-target correlation (while experience replay also allows for using more data during training).

When we take a further look and see how each of the trick applications performs over time we notice that the baseline without tricks and using the tarket network immediately decrease in performance, even to below their random initialization.

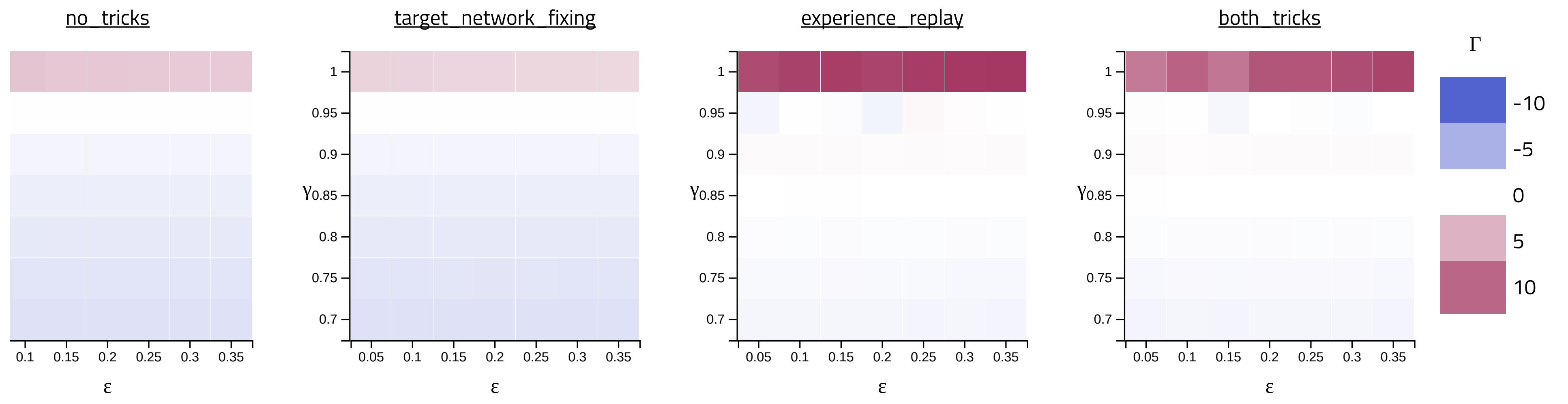
Mountaincar-v0

Next we take a look at the Mountaincar environment. This environment will tend to have longer episodes as the model needs to figure out that it has to go backwards first to gain momentum before going forwards. This environment is a negative uniform landscape, as each step the model hasn't reached the goal, it gains -1 reward.

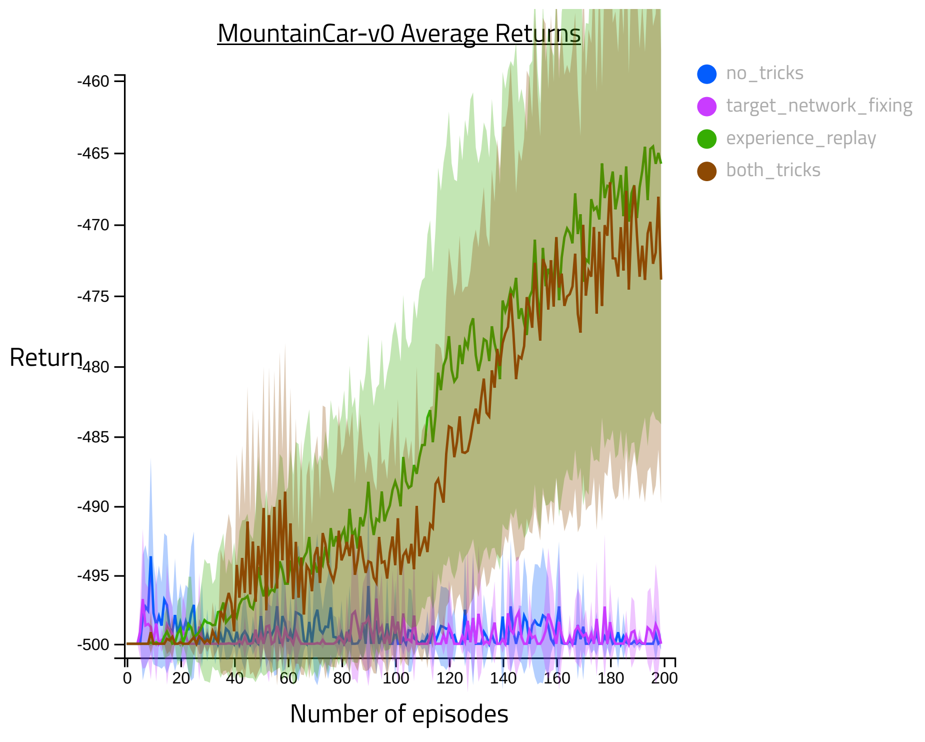


Graph section 3) - *Heatmaps of the returns of the gridsearch of the different trick combinations on the MountainCar-v0 environment.* ***Left)*** *No tricks -* ***Center-left)*** *Target network fixing -* ***Center-right)*** *Experience replay -* ***Right)*** *Both tricks*

Divergence:



Graph section 4) - *Heatmaps of the Γ of the gridsearch of the different trick combinations on the MountainCar-v0 environment.* ***Left)*** *No tricks -* ***Center-left)*** *Target network fixing -* ***Center-right)*** *Experience replay -* ***Right)*** *Both tricks*



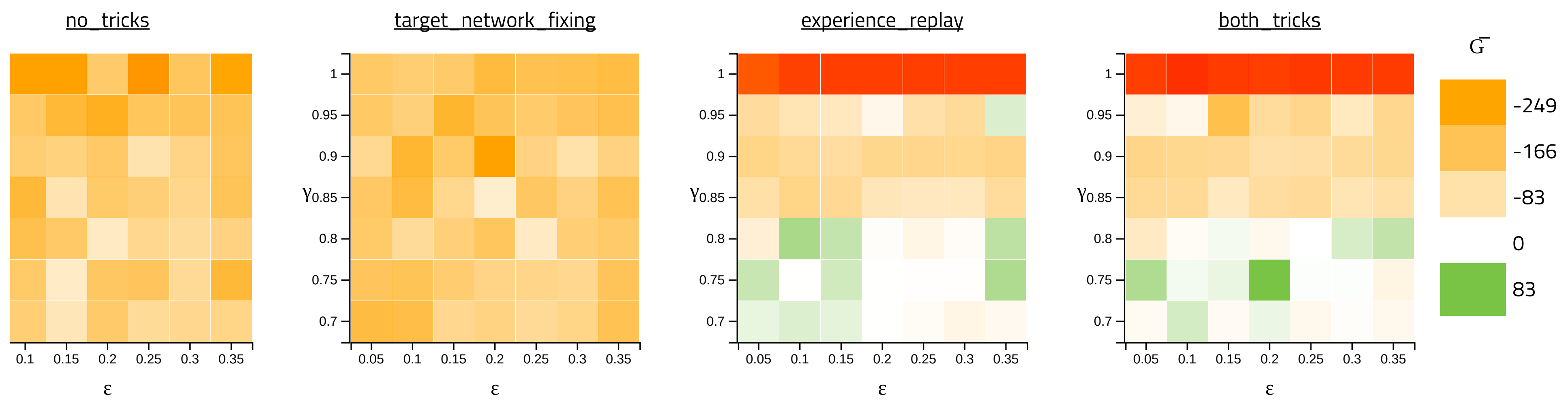
Linegraph 2) - *Line graph of the average returns over all hyper parameters for each trick (combination) on the MountainCar-v0 environment. Standard deviations shown are 1 fifth of the original. Hover over lines or trick names in the legend to highlight parts of the graph.*

The heatmaps for the model setups can be seen in the figures above and show similar results as the cartpole setting. Without the tricks and with only the target network it is unable to train, seemingly not due to divergence. Then once experience replay is added, the model is able to find solutions for some of the settings. There are however, three differences. Firstly there seems to be a more narrow range of hyperparameters for reliable good performance for this problem. Secondly, using a target network potentially does have *some* impact here, although its impact is not significant enough to draw any further conclusions about if it is helping to prevent divergence again as seen in the cartpole. Finally, using experience replay also seems to increase divergence rates for $\gamma = 1$, which can be explained by the network having more updates when using experience replay, hence also more steps in which the MSTD can grow. This appears to indicate that for this problem, the addition of the tricks does allow for more stable learning, but unlike in the previous environment, the target network seems to only mitigate divergence slightly.

When looking at the improvement over episodes we can see that on average convergence has not been reached yet. This means that, to further explore the divergence characteristics of this environment, we should in future experiments train longer.

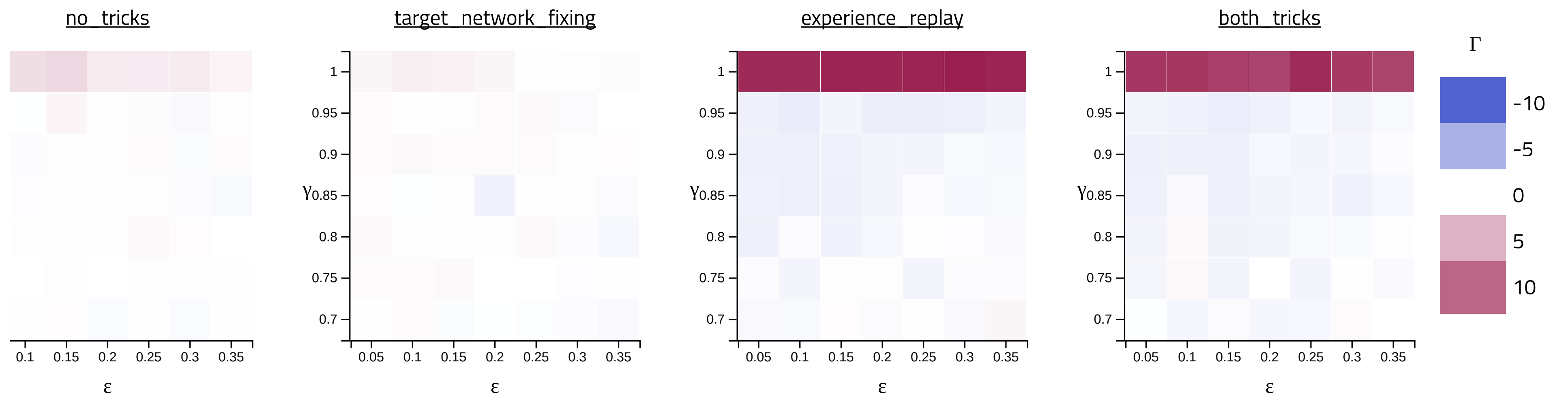
LunarLander-v2

Finally we take a look at the LunarLander environment. This environment will tend to have shorter episodes at the start as it tends to crash. It has a much more complicated rewards landscape, with negative reward for the actions it takes and any time it crashes, but positive rewards for different types of landings as previously mentioned.



Graph section 5) - *Heatmaps of the returns of the gridsearch of the different trick combinations on the LunarLander-v2 environment.* ***Left)*** *No tricks -* ***Center-left)*** *Target network fixing -* ***Center-right)*** *Experience replay -* ***Right)*** *Both tricks*

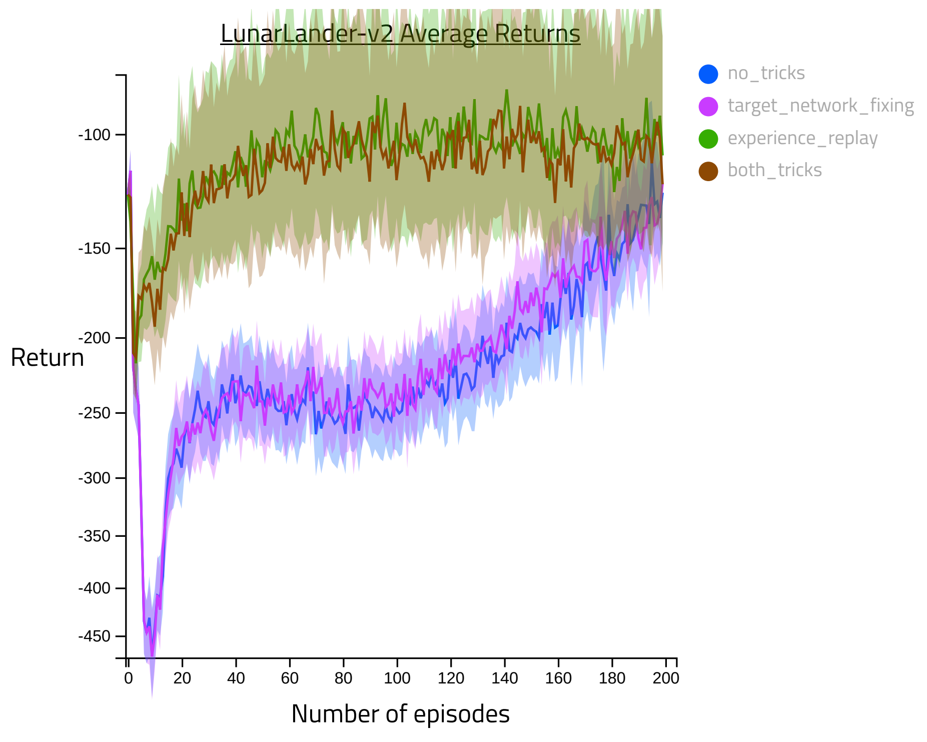
Divergence:



Graph section 6) - *Heatmaps of the Γ of the gridsearch of the different trick combinations on the LunarLander-v2 environment.* ***Left)*** *No tricks -* ***Center-left)*** *Target network fixing -* ***Center-right)*** *Experience replay -* ***Right)*** *Both tricks*

The heatmaps above seem to show relatively good performance for all hyperparameters, with an emphasis for lower gamma values when experience replay is used. It can be seen that there is a much higher variance in the test performance without tricks, which is to be expected for the more complex but also more information-dense reward landscape (even if the DQN cannot learn how to solve the problem of landing, it can still learn how to crash in a less expensive way).

In contrast with the CartPole-v1 and MountainCar-v0 experiments, target network fixing seems to only have a small influence on divergence, but only when the agent is not performing well. We do however see that experience replay is again the factor which gives a major improvement overall, but when we look at the line plot it appears to be mostly in how quickly the DQN learns what the good actions are.



Linegraph 3) - *Line graph of the average returns over all hyper parameters for each trick (combination) on the LunarLander-v2 environment. Standard deviations shown are 1 fifth of the original. Hover over lines or trick names in the legend to highlight parts of the graph.*

Conclusion

As the DQN approach suffers from highly unstable, volatile learning as a symptom of using the deadly triad, it is important to use methods to stabilize learning and mitigate the divergence of the estimates. We have looked at two common "tricks" that would tackle this problem: target network fixing and experience replay. We evaluated the overall influence these tricks have on the training process for environments with different reward landscapes and found that both experience replay and target network fixing appear to have general positive effects. Where experience replay appears to have a large effect on the quality of the learned policies and target network fixing aids in mitigating divergence in settings where the agent is able to start learning. From this, we can conclude that both tricks can help stabilize the training process, but how much they help appears to depend on the reward landscape of the environment.

For future experiments, it would be good to extend the hyperparameter search into the number of episodes between updates for the target network, as 2 episodes might not have been the optimal choice. It was fixed in order to lower the number of hyperparameters we were searching over as it was too time intensive to add. Similarly we used a replay-memory with a fixed size of 10000 samples which might be more or less effective depending on the environment. Another metric we would like to look at in the future would be the magnitudes of the parameters in the network. As they are surrogates for the Q values, they should also grow with time if the network is diverging. Finally, it would likely be useful to obtain the target-seed performance before training, so we can measure the difference in performance than the absolute values, as this would clarify results with environments that can have both positive and negative rewards/returns.

References

1. Van Hasselt, H., Doron, Y., Strub, F., Hessel, M., Sonnerat, N., & Modayil, J. (2018). Deep reinforcement learning and the deadly triad. arXiv preprint arXiv:1812.02648.
2. Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction. MIT press.
3. Watkins, C. J., & Dayan, P. (1992). Q-learning. Machine learning, 8(3-4), 279-292.
4. Lin, L. J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. Machine learning, 8(3-4), 293-321.
5. OpenAI Gym
6. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.
7. Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). Prioritized experience replay. arXiv preprint arXiv:1511.05952.
8. Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., & Meger, D. (2017). Deep reinforcement learning that matters. arXiv preprint arXiv:1709.06560.
9. https://github.com/pytorch
10. Ankit Choudhary's A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python
11. Deepminds Atari Breakout Video
12. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Petersen, S. (2015). Human-level control through deep reinforcement learning. nature, 518(7540), 529-533.