Acing simple games, an exploration of neural networks for reinforcement learning[1]

David Kleingeld

April 24, 2020

[1] A report for the Leiden Univesity Reinfocement Learning course

Contents

Introduction

The field of reinforcement learning (RL) has long relied on heuristics, sampling and hand written feature recognition. To learn end to end, directly from high level input, has been a significant challenge. Utilizing (convolutional) neural networks the field of deep learning has succeeded to recognise features from high level input. Here we take a look at Deep Q networks (DQN) a successful approach to end-to-end reinforcement learning. The Q-table from classic Q-learning is replaced by a neural network where the training of the network mimics the Q-learning update formula.

There are a number of significant challenges. Reinforcement learning generates its own data, measures must be taken to prevent feedback loops. In deep learning the network immidiately gets feedback weather it was correct or not. In Reinforcement Learning there might be hunderds of not thousands of steps between an action and the reward. An RL agent must be able to deal with delayed rewards.

Here we implement a DQN agent to solve the tasks mountain car and breakout. Mountain car is a simple control based task where one challange is that the reward is only given after the task is completed. Breakout is one of the classic Atari 2600 games sucessfully learned in the famouse "Human-level control through deep reinforcement learning"[2] paper. I will use this to adapt the implementation for mountain car to work on breakout. I will use two of the learning techniques introduced there: experiance replay and infrequent weight updates. For the mountain car problem we will experiment with agents with various memory sizes for experiance replay and with and without infrequent weight updates. The breakout problem requires carefull tuning, together with lots of training, which did not leave time to experiment.

In the next section I discuss the theory of DQN, the challanges introduced above and how these are partially solved with experiance replay and infrequent weight updates. Then I will discuss my implementation of DQN for mountain car. This is followed by a section on the performance while experimenting with the agent. Next I shift the focus to breakout, detailing how the implementation for mountain car needed to be adjusted to get the breakout agent to learn. After this we take an in depth look at the training of the agent and how it performed. And in the final section of this report we conclude weather our agents have succeeded and how they could be improved. In the appendix at the end we detail how our agents can be tested.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015

Theory

Here we will give a short theoretical introduction to deep Q-networks (DQN). A DQN is capable of learning a diverse array of challanging tasks without hand coded domain specific knowledge. Recieving only the state of its envirement it returns one of the given possible actions, it then gets feedback in the form of a reward.

DQN is based on Q-learning or action-value learning. In Q-learning the Q function Equation 1 calculates the quality of a state-action combination. The outcome is then stored in the Q table and can be used to create more Q values. When an action is required the action with the highest Q value in the table is performed.

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \Big(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of future value}} \Big) \tag{1}$$

Here Q^{new} is the new Q-value for the state s_t and action a_t. We calculate it by combining the old Q value with new information we learned as we transitioned to state s_{t+1} by action a_t. We weigh the combination with the learning rate α, the higher the learning rate the higher the impact of the new information. The new information consists of the reward r_t and Q value of the best action we can take from here[3]: $\max_a Q(s_{t+1}, a)$ We additionally weigh this by the discount factor γ which determines how importend future rewards are.

[3] as the best action is the action with the highest Q value we can add the maximum Q value corrosponding to the new state s_{t+1} we are now in.

Initially the Q table is empty and it will need training. The only way to fill the table is to performce random actions, using the reward to find the Q-value. A good strategy is to let the rate of random actions ϵ decrease throughout the training process. Starting at $\epsilon = 1$ and ending at $\epsilon = 0.1$.

This table based way of learning has only limited applications. For example if we where to apply Q-learning to the mountain car problem section we would immidiately run into a problem. The needed Q-table would need to be infinite[4]. While this can be solved by discretizing the input before the Q-learning algorithm there are many more complex problem where this is not feasable. One of these is our second problem, the atari game breakout, see section . For this we turn to DQN.

[4] ignoring the limited resolution of the `doubles` used to represent the state of the car

In DQN the optimal action-value is approximated by a neural network. When we need to perform an action we ask the neural network to make predictions for the Q-values for all the different possible actions. We then pick the action with the highest predicted Q-value.

Initially the neural network will behave like the empty table from Q-learning, not knowing the right Q value. However as we train the

network it wil converge to the Q value. The network is trained with as input the state and action and as label the correct Q-value as calculated from Equation 1, getting $Q(s_{t+1}, a)$ and $Q(s_t, a)$ by asking the network to make a prediction for input $[s_{t+1}, a]$ and $[s_t, a]$ respectively.

There is a danger in using a neural network. The reinforcement learning can become unstable[5]. This is attributed to the *deadly triad*: function approximation, bootstrapping and off policy learning. The Q-function is approximated by the neural network based on features recognized in states states that should have a different Q-values may share the same features leading to loops in behavior. Bootstrapping, or basing the new Q-values partially on the old may exaggerate the problems introduced by the imperfect approximation. Finally DQN uses off policy learning it converges less well then on policy learning. Mnih, Koray Kavukcuoglu, and Silver[6] adress these instabilities using a replay buffer. By storing the states, actions and corrosponding reward the network can learn on results out of order. This breaks feedback loops solving the unstability described above. Another significant improvement made is infrequent weight updates. Here a seperate network is used for the predictions. This prevents the network from returning values based on newly learned behaviour while training causing those behaviours to be reinforced entering a feedback loop. Once in a while the network used for prediction is replaced with the current training network.

[5] John N. Tsitsiklis and Benjamin Van Roy. Analysis of temporal-difference learning with function approximation. In *Proceedings of the 9th International Conference on Neural Information Processing Systems*, NIPS'96, page 1075–1081, Cambridge, MA, USA, 1996. MIT Press

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015

Mountain Car

In the mountain car problem we start at the bottom of a valley the goal is to reach the top of the mountain on the right (see Figure 1). On the left side is a wall that simply blocks the agent. For each timestep an agent can choose to give the car some impulse to the left, to the right or do nothing. This task seems trivial however the car is underpowered, simply going right all the time will not bring you to the top. As a human you would quickly figure out that you need to swing between the walls higher and higher to build up momentum just as if you where on a swing in a childrens playground. However the agent will get far less information.

The agent gets a reward *only* when the task has been completed. It gets punished for every timestep the task is not completed. The only thing it gets next to this feedback is the current state of the car, its location and its velocity. Here I use DQN to get the car to perform the task. In the beginning it will perform random actions learning features of the envirement, once it succeeds it will try to replicate that behaviour eventually learning how to optimally reach the goal.

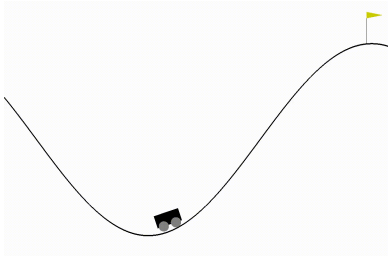

Figure 1: The mountain car problem after the agent has taken an action to the right

Implementation

This implementation of DQN implements both the essential replay buffer and infrequent weight updates for the neural network. The agent gets 1000 attempts to perform the task, we shut an attempt down after 200 timesteps. Each session starts by requisting an action from the network or taking a random step, this is governed by the rate of random actions ϵ which we decrease during training. The action is then performed on the envirement, we use the implementation of mountain car provided by the *openAI gym* project, specifically we use `MountainCar-v0`. The envirment returns the state after the action has been performed, the reward and weather the task is done. We add the before and after state together with the reward and weather the task is done to memory. Then if the memory has enough experiance (400 samples) we train the network before starting over for the next step. This continues until the task is completed or the maximum number of steps is reached. After each training step we copy the weights from the training network to the predicting network as motivated in Equation .

The memory, network and random step rate epsilon are implemented as python classes: `Memory`, `Predictor` and `Epsilon`.

The `Memory` class uses a deque to store events that consist of the *before* and *after* state together with the *score* and if the *game is over*. It has a member function to add such an event, a sample method to return a random sample of events and a length member which is used to determine weather the memory has enough experience to start training the network. The memory forgets old events keeping track of only the last 20000 events.

The `Predictor` wraps around a simple three layer neural network implemented using Keras[7], it provides a subset of the Keras api: `predict()`, `fit()` as well as methods for infrequent weight updates: `copy_weights_from(self, other)` and `clone(self)`. The first two functions help reshape the data using numpy[8]. The network consists of two hidden layers, an input of size 2 encoding the position and velocity as floats, and an output layer of size 3 that encodes the Q-score for each action, of which there are 3 (left, right or nothing). The first hidden layer has 24 fully connected nodes and the second 48. Both hidden layers use the *rectified linear unit* as activation function however the output layer uses a linear activation function. The network uses the *mean square error* as loss and optimises using the *Adam* optimiser as it is suitable for noisy and sparse gradiens.

The `Epsilon` class provides a decaying random action ratio. It starts with value 1 which translates as only take random actions. Every time the `decay` method is called the epsilon value is decreased slightly (by 5^{-5}) until a minimal value of 0.01 is reached which means

[7] François Chollet. keras. `https://github.com/fchollet/keras`, 2015

[8] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011

the agent takes a random action for only 1% of steps. As we call `decay` every time we take an action ϵ linearly decays from 1 to 0.01.

The network trains using experience replay by taking 32 samples from the replay buffer and training on them. For each sample the input for the neural network is the state before an action was applied, the *before state*. The labels are however not the Q-values. The learning rate from the classic Q-formula (Equation 1 in section) is applied by the learning of the network. The old Q-value is encoded in the existing network and incorporated in that way. The labels are then the part of the Q-formula not already implemented in the training: the reward r_t plus the discount factor γ and the estimate of the future Q value $max_a Q(s_{t+1}, a)$. Gamma is 0.95 for this implementation. The estimate is given by the output of the network for the after state and the reward is the score. When the game has ended it does no longer make sense to incorporate future results in the Q-value, in that case we let the label be the current score and do not add γ times the Q-value for the *after* state. As the output of the network is the Q-value of all possible actions and we only know how to update the Q-value for the action that was taken we use the current predicted Q-values for the other actions.

To speed up the learning of the network I minimized the calls to *keras* functions by doing the predictions and learning for the entire batch. We first calculate the Q-values for all the *before states* and the *after states* in the sample. Then looping over the samples we use the just calculated Q-values to find the labels as described above. Finally we do the training of the network for the entire batch again.

Results

It is challenging to get the network to learn stable and well. Here I present the learning process for variouse sizes of the replay buffer and look at the effect of infrequent weight updating. I take a look at what these changes mean for learning stability. The paramaters in Table 1 are shared between all results. The results are presented in figures containing a pair of runs. Each run has the value of epsilon, the score for each iteration and the moving average[9] of the score. The moving average will help recognise medium term unstable learning.

[9] we use a window of 50 training iterations

Paramater	Value
discount factor γ	0.95
replay sample size	32
maximum steps	200
training iterations	1000
learning rate	0.001
epsilon start	5
epsilon decay	5^{-5}

Table 1: The paramaters and theire values shared between all mountain car results.

The default size of the replay buffer is 20000 timesteps or events. I start by looking at the learning behaviour of the agent with this size of buffer. Because reinforced learning can be unreproducible, especially when combined with the unreproducibility of neural networks I train the agent 4 times.

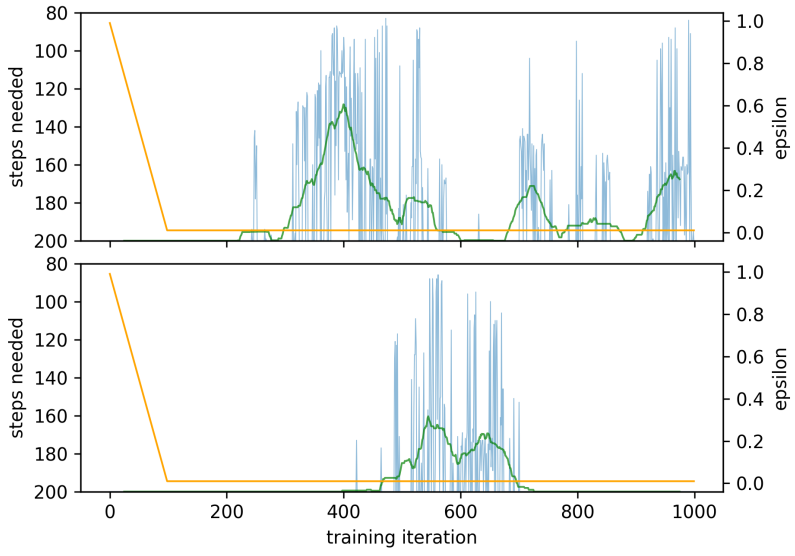

Figure 2: Two seperate runs of training the mountain car agent with a replay buffer which remembers the last 20000 timesteps. On the y-axis the number of steps needed to finish, 200 means the algorithm failed to reach the goal in time. On the x-axis the iteration number. The orange line is epsilon, the blue line follows the scores of the training iteration and the green line is the moving average of the score. Note how both attempts do not lead to convergence within 1000 steps

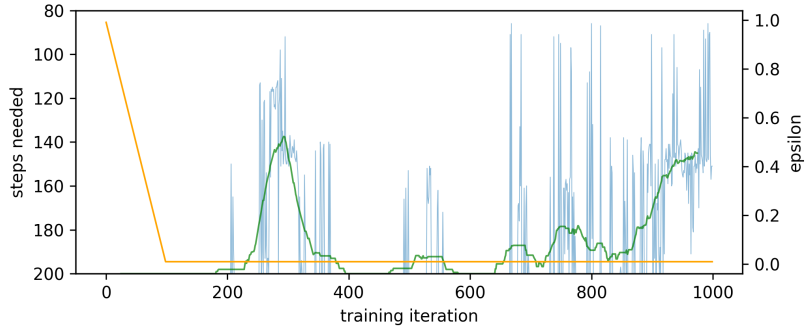

Figure 3: One run of training the mountain car agent with a replay buffer which remembers the last 20000 timesteps. On the y-axis the number of steps needed to finish, 200 means the algorithm failed to reach the goal in time. On the x-axis the iteration number. The orange line is epsilon, the blue line follows the scores of the training iteration and the green line is the moving average of the score. Note that agent seems to converge at around iteration 900.

One of the times the agents dit not solve the problem once using the maximum of 200 steps each iteration, the three succesful results are in Figure 2 and Figure 3. The agent is unstable it can start producing good results before failing at the task again 200 training iterations later as seen in the bottem plot of Figure 2

Now I look at the learning stability of the agent if I decrease the buffer size, I expect the stability to decrease as the famouse deadly triad becomes a larger problem and the agent starts biting in its own tail more. In Figure 4 and **??** we see the learning behaviour of the agens with a replay buffer of 10000 timesteps.

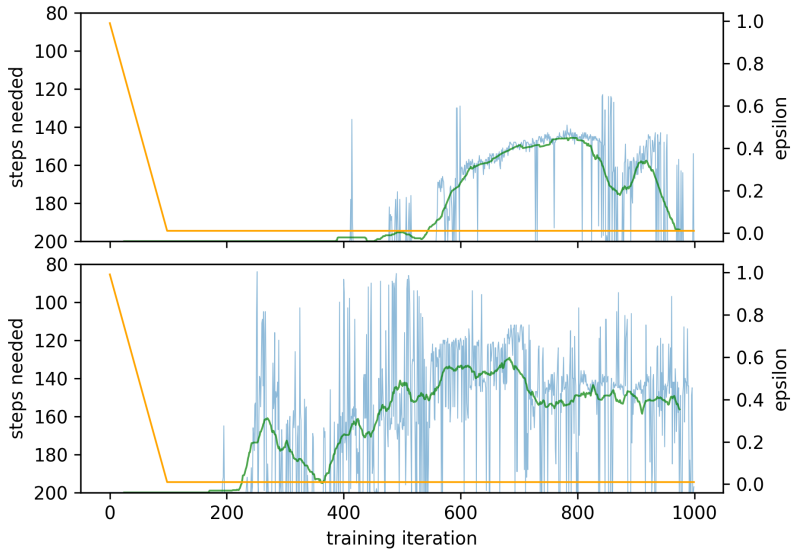

Figure 4: Two seperate runs of training the mountain car agent with a replay buffer which remembers the last 10000 timesteps. On the y-axis the number of steps needed to finish, 200 means the algorithm failed to reach the goal in time. On the x-axis the iteration number. The orange line is epsilon, the blue line follows the scores of the training iteration and the green line is the moving average of the score. The first run in the top part of this plot shows stability improving as the training iterations continue. Note in the second run (bottem part of figure) the seemingly stable behaviour improving slowly before collapsing. The overal performance is worse then that of any of the other successful runs.

In the upper part of figure Figure 4 wes see the agent grow stabily become better between episode 600 and 800 before collapsing. In

the lower part of that figure the agent is behaving more stable as the training iterations go on. In half of the runs the agent failed to complete the task once within the available time (200 timesteps).

Doubling the replay buffer size from the default to 40000 timesteps gives rise to the results in Figure 5 and Figure 6. By increasing the replay buffer size I expect more stable behavior.

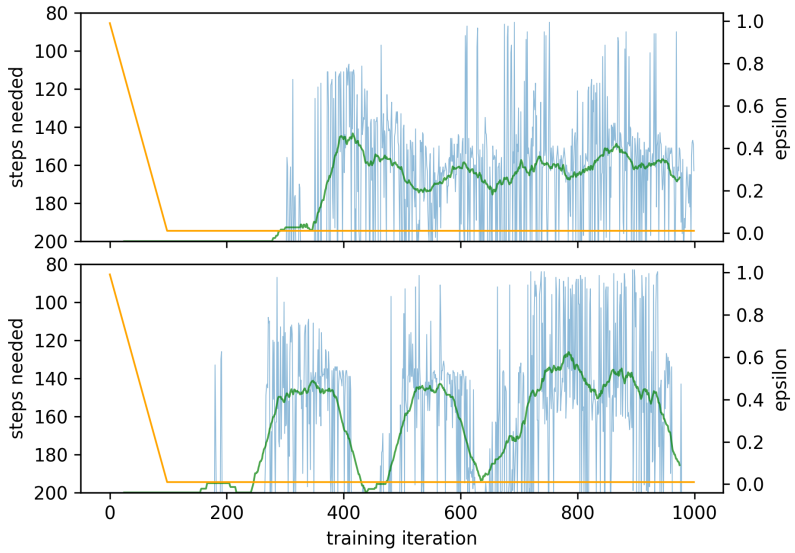

Figure 5: Two seperate runs of training the mountain car agent with a replay buffer which remembers the last 40000 timesteps. On the y-axis the number of steps needed to finish, 200 means the algorithm failed to reach the goal in time. On the x-axis the iteration number. The orange line is epsilon, the blue line follows the scores of the training iteration and the green line is the moving average of the score. Note the agent solves the problem in most training iterations

We see that with the larger replay buffer the agent performs okay in more scenarios it also was able to solve the task every time I trained the agent from scratch. The overall performance however is worse compared to the other successful training runs as seen best in Figure 6.

Now I compare two mountain car agent with both a replay buffer of size 40000[10]. To do this I do two more runs at 40000 however the agent trains the same neural network network as it uses for predictions[11]. I expect this to cause the agent to destabilize more.

After running the training 4 times the agent was able to reach the goal in only one run, in the other 3 the number of steps needed stayed at 200 for all 1000 training iterations. The succesful run is presented in Figure 7.

We see the agent learned really quickly, which is to be expected since the actions are immidiately based on newly learned behaviour. However this plot also seems to show slightly more stable performance, as it performs really well for most iterations this is counterintuitive. A possible explanation could be the network adapting quickly or this

[10] we pick this size over the default 20000 as it performs more reliably

[11] the copying of weights is also disabled as we are no longer using the seperate train network and copying its random weights would make training the network useless

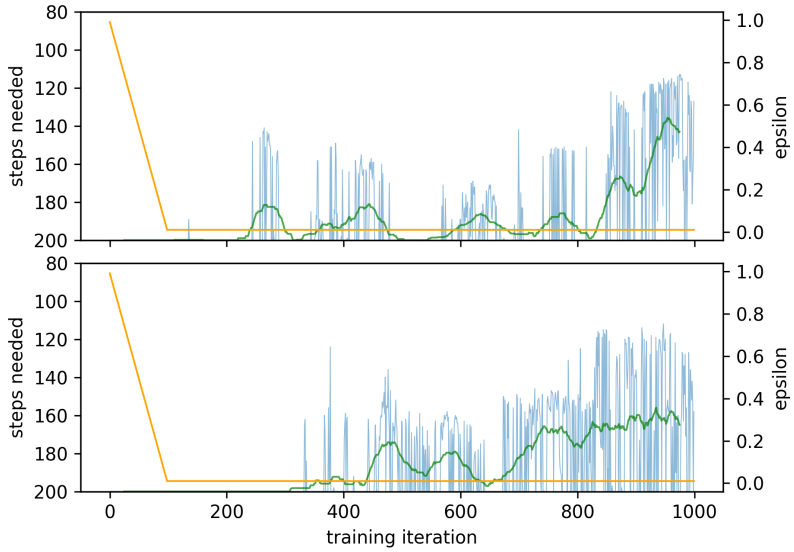

Figure 6: Two seperate runs of training the mountain car agent with a replay buffer which remembers the last 40000 timesteps. On the y-axis the number of steps needed to finish, 200 means the algorithm failed to reach the goal in time. On the x-axis the iteration number. The orange line is epsilon, the blue line follows the scores of the training iteration and the green line is the moving average of the score. Here the agent solves the problem often but it is very inefficient in doing so

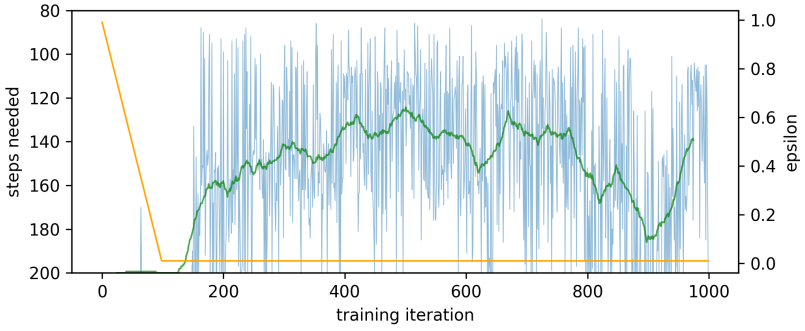

Figure 7: The training of the agent with a replay buffer (size: 40000 timesteps) but without infrequent weights updates. On the y-axis the number of steps needed to finish, 200 means the algorithm failed to reach the goal in time. On the x-axis the iteration number. The orange line is epsilon, the blue line follows the scores of the training iteration and the green line is the moving average of the score. The agent solves the problem after less then 200 training iterations

training run being a peticularly easy one due to random chance.

Breakout

Here the agent needs to master the atari game breakout, see Figure 8. The agent controls a small bar at the bottem of the screen with possible actions: move left, move right or stay. A ball moves with constant speed through the envirement. If the agent positions the small bar such that the ball hits the bar the ball inverts its vertical velocity bouncing upwards. Whenever the ball hits the bottem of the screen it disappears, the agent loses a life (it starts with 5) and a new ball is introduced with a random direction downwards. When the ball hits the left and right side of the screen the horizontal velocity is inverted making it bounce of the walls. Somewhat below the top of the screen is a stack of rows of blocks. Whenever the ball hits one of the blocks: the block disappears, the ball inverts its vertical velocity and the agent is rewarded with score one for the current timestep. If the ball hits the top of the screen its vertical velocity is inverted, bouncing it off the top.

In this problem agent gets the same information as a human would, the pixels on the screen. Aditionally we give it the reward of the current timestep. Due to the computational complexity we do not want the agent to process the whole screen each timestep. We build our implementation on top of the `BreakoutDeterministic-v4` envirement provided by the *openAI gym* project. This envirement returns the game state only once every 4 frames lowering the computational requirements.

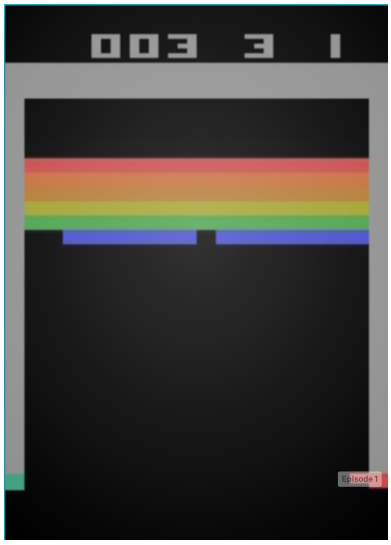

Figure 8: The atari breakout envirement

Implementation

Initially we used a slightly adapted version of the mountain car DQN implementation described in subsection . The simple multilayer dense neural network was replaced with a convolutional neural network consisting of 2 layers both with kernal size 3 followed by two dense layers. The frames from the envirment where changed to greyscale and cropped to 80 by 72 pixels. This leaves the game envirement intact but removes the score at the top and ornamental edges, leaving an images as in Figure 9 for the network.

This did not result in any learning by the agent. I tried multiple different networks before taking a look at related literature[12]. This lead to the conclusion that my agent was that my hyper paramaters where far from where they should be.

Taking the hyperparamaters from the famouse "Human-level control through deep reinforcement learning"[13] we start changing the implementation. The mountain car agent is limited in the number of training sessions, this does not make sense for the breakout agent as we want to limit the number of steps not training sessions. From the

[12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. 12 2013

[13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015

perspective of a human traning there is no difference between losing a life and the game restarting however the agent was not even punished for losing a life. That was changed so losing a life is treated as losing the game during training. Using only one frame as input to the network does not allow the agent to use the concept of direction or speed, essential to a human player. As in[14] we feed the network a stack of the last 4 frames, as given to us by the envirement, for prediction and during training.

These changes give a number of implementation problems. We can not take any action for the first 3 timesteps as we can not make a stack of 4 frames to feed the agent. We have removed the training in sessions however we should not use frames from another training session. For example is the games has ended in frame x we can not use the network until frame $x + 4$. To enforce this we use the function `reset`. It resets the envirement and forwards the game 3 timesteps without taking any action. We use a new class `State` to keep track of the 4 frames. I provides a method `push` that takes as arguments an before and after state together with the action taken, score and if the game is over. It then forgets the oldest before and after state adding the new state to its internal stack. This introduced a new problem, the stack of images drastically increased the memory requirements of the agent. Instead of single images the `event` inserted into the memory (see subsection) are a lot larger at 184kB per state[15]. If we want to have a replay buffer as in the literature[16] of 1 million states the agent would need at least 184 gigabytes. I solved this in two steps. Instead of storing the pixels in the default floating point format they are stored as bytes. This does not lead to significant data loss as the envirment uses one byte per color per pixel, since we are using grayscale we can use a single byte per pixel. The `State` class is modified to cast pixels to bytes when a new frame is pushed. The `Predictor` class (see subsection) casts the pixels back to floats before feeding them to the convolutional neural network. By lowering the replay buffer size from one million to 1/4 million. Later after the agent was run for 10 million steps the memory use was further reduced no longer saving 8 frames for each state but only two and recreating the full 8 frame states as they are sampled from the memory. The `Epsilon` class constuctor was rewritten, now the decay is not a constant but rather determined by the number of steps that epsilon should decay over from its starting value to its final value. Finally we took the network architecture from the deepmind paper[17] after our own would not let the agent learn. It consists of 2 convolutional layers, the first has 16 filters with square kernals size 8 using a stride of 4. The second convolutional layer has 32 filters with square kernals of size 4 and uses a stride of 2. There layers are followed by a fully connected layer and then the fully con-

[14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015

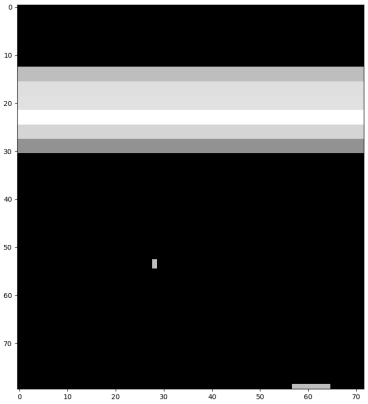

Figure 9: A frame returned by the atari breakout envirement after postprocessing for our agent, converting to color and cropping out the unneeded edges

[15] frame before and after an action each 4 images of $72 \cdot 80$ pixels, each pixel represented as a python floating point number of 8 bytes giving a total of 184kB per state. Multiplied by a million that becomes gigabytes

[16] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015

[17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control

nected output layer, the first with 256 nodes and the second 4, the number of actions for the environment. Except for the output layer all layers use the rectified linear unit as activation function.

Results

I was able to train the agent for the full length of 10 million steps only once, I did a second run with 1 million steps. During the training for each game the score achieved, timesteps taken and final epsilon value where recorded. Achieving a heigher score is the primary goal for this agent. Taking more timesteps means the agent survives longer and has more opportunities to accidentily hit a block and score a poiont. For both training sessions I show the value of epsilon, the score and the timesteps per game vs training progress. For the score and timesteps per game we also plot a moving average created with a window size of 50. We expect score and timesteps per game to be highly correlated.

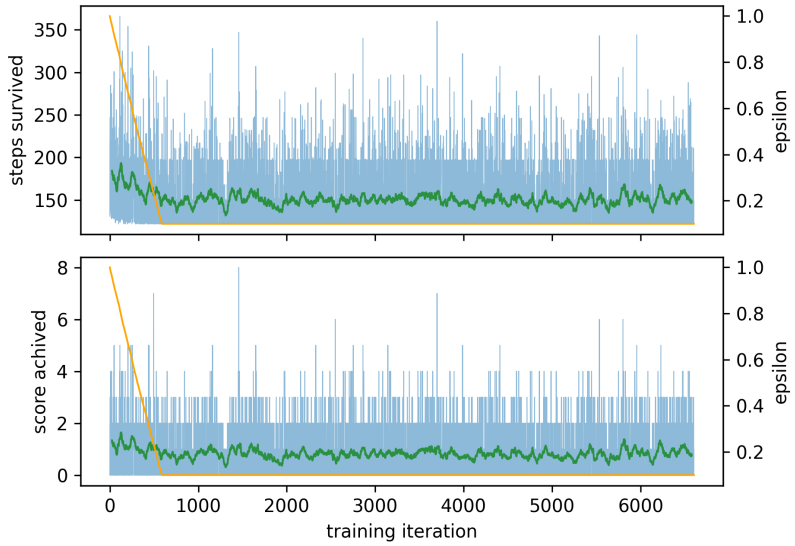

Figure 10: The performance of the agent when trained for 1 million steps. In orange the value of epsilon, in blue the steps per game for the top plot and score achived in the bottom plot. In green the moving average over the blue lines.

In Figure 10 we see how the agents performans during a training of 1 million steps. The behaviour of the agent is unstable it scores bad during the training not reaching howvering around a score of 1.2 which is the mean score for a random agent[18]. At around zero the agent seems take longer then later during training, I could not reproduce this thus attribute it to the randomness of the agent, especially as it mostly took random actions here due to the high epsilon value.

When we train longer for 10 million steps we see a number of hopfull jumps before the agent jumps in performance around game 36000,

[18] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. 12 2013

see Figure 11.

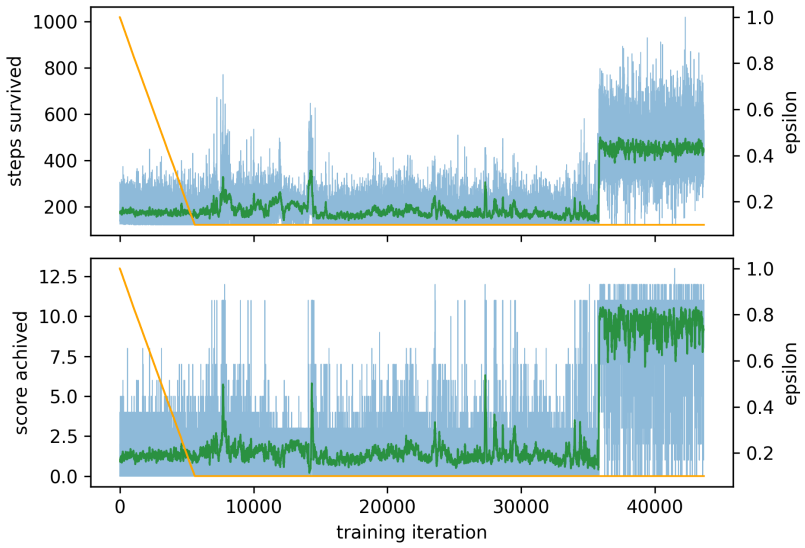

Figure 11: The performance of the agent when trained for 10 million steps. In orange the value of epsilon, in blue the steps per game for the top plot and score achived in the bottom plot. In green the moving average over the blue lines. For a closer look at the jump in performance around game 36000 see Figure 12

From the moving average (green) we clearly see the agent had two early peaks where it seemd to grasp the game for a short while before collapsing again. Looking closer at the later jump (Figure 12) note the improvement is quite a sudden, within 20 games the agent has grasped the basics of the game and is far outperforming its previous self only failing incidentilly. However we also see that save for one a single peak the agent never exceeds a score of 12.

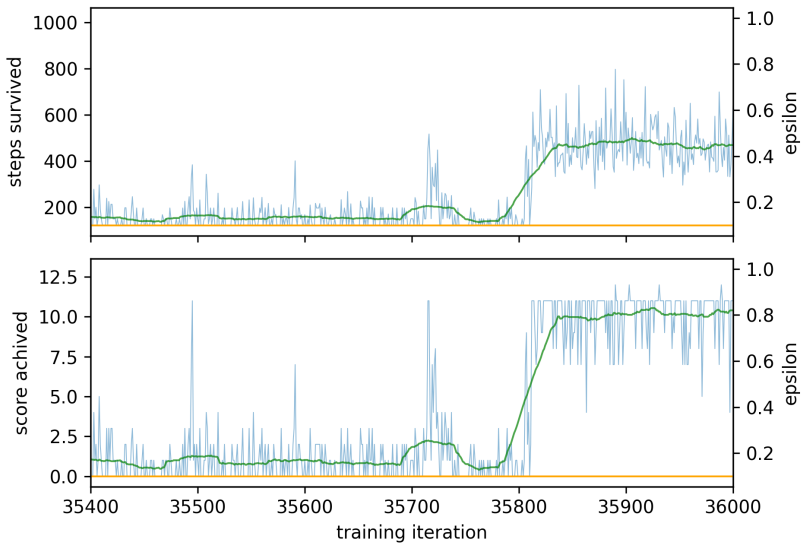

Figure 12: The performance of the agent when it jumps up in performance at around game 36000. In orange the value of epsilon, in blue the steps per game for the top plot and score achived in the bottom plot. In green the moving average over the blue lines.

Conclusion

Here I implemented a deep neural network agent for mountain car and breakout. I used DQN with experience replay and infrequent weight updates. I tried different sizes for the replay buffer and looked at the influence of weight updates on the algorithm. The goal was to find out how the learning stability is influenced by tweaking these parts. Due to the lengthy training times for the breakout agent we did this exclusively for the mountain car problem. I tried halving and doubling the replay buffer size and turning off and on the infrequent weight updates enhancement. I ran each variant four times to account for inherent variation in the training. The mountain car agents learning was unstable by default, 1 out of 4 times it would not learn at all and one run it forgot almost immediately what it learned, only perfoming well for a few hunderd iterations. When the replay buffer was halved only the number of times the agent would not learn at all doubled though when it started to learn it would seem more stable. Doubling the replay buffer size eliminated the times the agent would not learn at all. The agent was more stable. I conclude that a larger replay buffer helps training stability a lot, however it might destabilizes training that is going well introducing bad results from the past. Taking the best result, the doubled replay buffer, and disabling infrequent weight updates had two effects. Learning destabilized dramatically, only 1 of 4 agents succeeded during training. The agent that did succeed learned dramatically faster which can be explained by the decrease in delay between learning and acting.

Moving on to the breakout agent. We see that training for one million steps did nothing for the performance of the agent, not beating a random player. When we train longer we see 2 large peaks in performance in the beginning followed by a number of smaller shorter ones before the agent dramatically and consistently improves its performance at iteration 36000 about 75% through training. When we look closer we see the jump is as "instant" as it looked taking about 20 games. The breakout agent is promising but at an average score of 12 after playing still below human levels of performance. Investing reinforcement learning agents is inherently unstable itself. As there is a lot of random decision making involved that could make or break an agent multiple training runs are needed to see if results can be replicated. Though we were easily able to replicate the results for the mountain car agent (as presented) I could not do multiple runs for the breakout agent as training it took over 2 day and compute resources where limited[19]. It would be interesting to increase the size of the replay buffer for the breakout agent, support for this was added in the code and memory use could further be reduced by no longer storing

[19] taking 4 computers for over 2 days to train multiple runs when access to free computers was deemed "not nice"

the `after` action frames as the before after pairs can be recreated from the memory. A major flaw in the breakout agent is that it determines its next action on the before action frame. That means the agent has to think ahead one more frame than is necessary. This flaw was discovered when no more time was left for retraining the agent. The results above seem to indicate that infrequent weight updates somewhat inhibit learning, we could try disabling it when the agent is learning stabily.

Appendices

Run Instructions

Before running either agent the dependencies listed in `requirements.txt` should be installed[20].The mountain car agent can then be started by calling `src/mountain_car/main.py` with *Python 3*. The breakout agent is started by calling `src/breakout/main.py` again using *Python 3*. Both agents log their progress to standard out (stdout), to create the plots seen in the report redirect stdout to a file and place it in a directory `logs`. Then change `logs_to_graphs.py` in either `src/breakout/` or `src/mountain_car/`.

[20] on systems with both *Python 3* and *Python 2* installed the *pip* version installed with *Python 3* should used, often called with `pip3`