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

David Kleingeld

April 22, 2020

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

Contents

Introduction

Here

issue: speed and debugging, fast solution needed to be able to debug what is going on. issue: internet resources actually wrong, see: https://towardsdatascience.com/reinforcement-learning-w-keras-openai-dqns-1eed3a5338c (line 103 at the bottem. he quits if it succeeds once which could be by chance)

usefull: https://towardsdatascience.com/qrash-course-ii-from-q-learning-to-gradient-policy-actor-critic-in-12-minutes-8e8b47129c8c

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[2]: $\max_a Q(s_{t+1}, a)$ We additionally weigh this by the discount factor γ which determines how importend future rewards are.

[2] 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[3]. 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.

[3] 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 $Q\left(s_{t+1}, a\right)$ and $Q\left(s_t, a\right)$ 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[4]. 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[5] 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.

[4] 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

[5] 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.

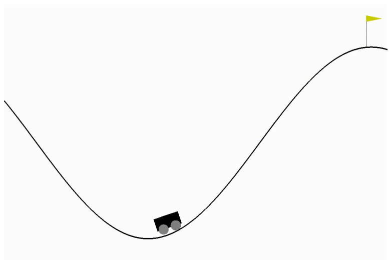
Figure 1: The mountain car problem after the agent has taken an action to the right

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.

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[6], 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[7]. 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

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

[7] 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[8] of the score. The moving average will help recognise medium term unstable learning.

[8] 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.

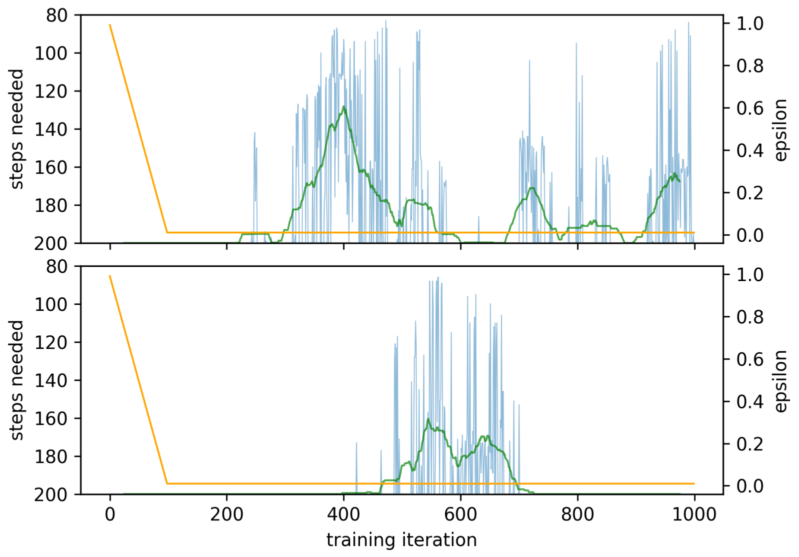

Figure 2: Two seperate runs of training the mountain car agent with a replay buffer which remembers the last 20000 timesteps. On the y-axis the number of steps needed to finish, 200 means the algorithm failed to reach the goal in time. On the x-axis the iteration number. The orange line is epsilon, the 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

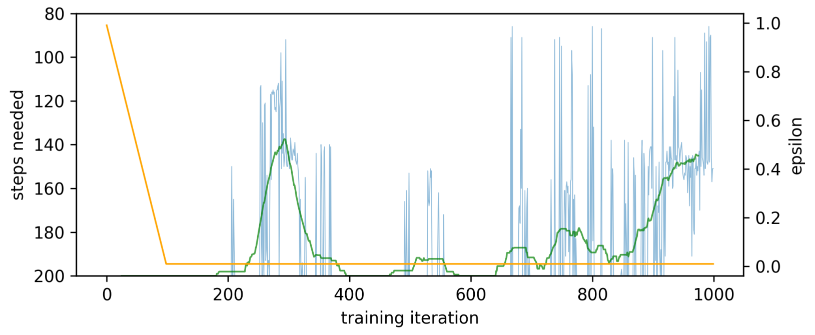

Figure 3: One run of training the mountain car agent with a replay buffer which remembers the last 20000 timesteps. On the y-axis the number of steps needed to finish, 200 means the algorithm failed to reach the goal in time. On the x-axis the iteration number. The orange line is epsilon, the blue line 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.

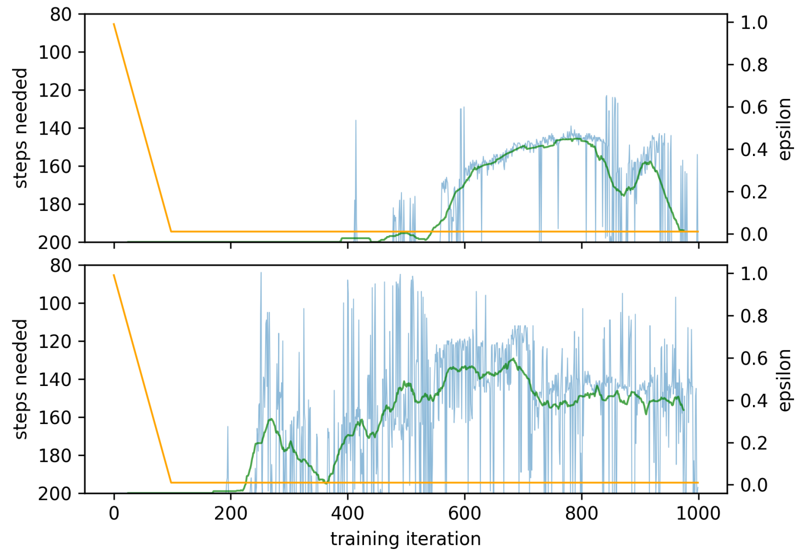

Figure 4: Two seperate runs of training the mountain car agent with a replay buffer which remembers the last 10000 timesteps. On the y-axis the number of steps needed to finish, 200 means the algorithm failed to reach the goal in time. On the x-axis the iteration number. The orange line is epsilon, the 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 figure Figure 4 wes see the agent grow stabily become better between episode 600 and 800 before collapsing. In **??** I see the agent 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.

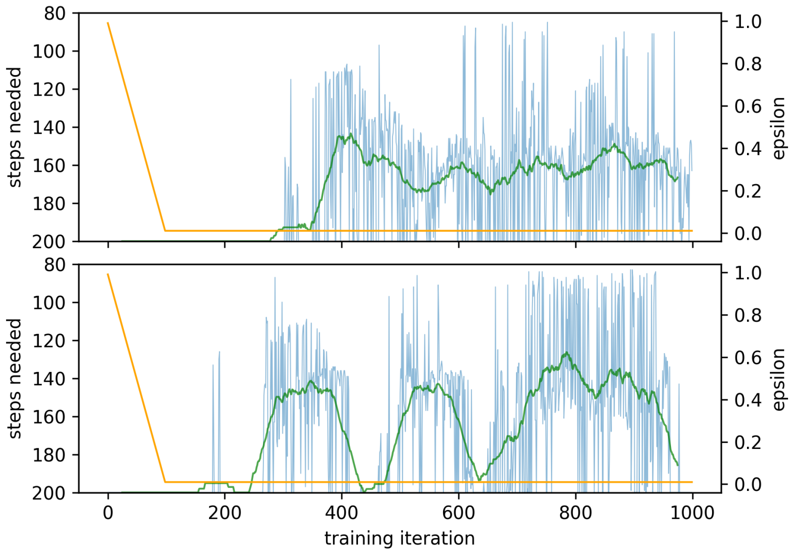

Figure 5: Two seperate runs of training the mountain car agent with a replay buffer which remembers the last 40000 timesteps. On the y-axis the number of steps needed to finish, 200 means the algorithm failed to reach the goal in time. On the x-axis the iteration number. The orange line is epsilon, the 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[9]. To do this I do two more runs at 40000 however the agent trains the same neural network network as it uses for predictions[10]. I expect this to cause the agent to destabilize more.

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

[10] 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

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

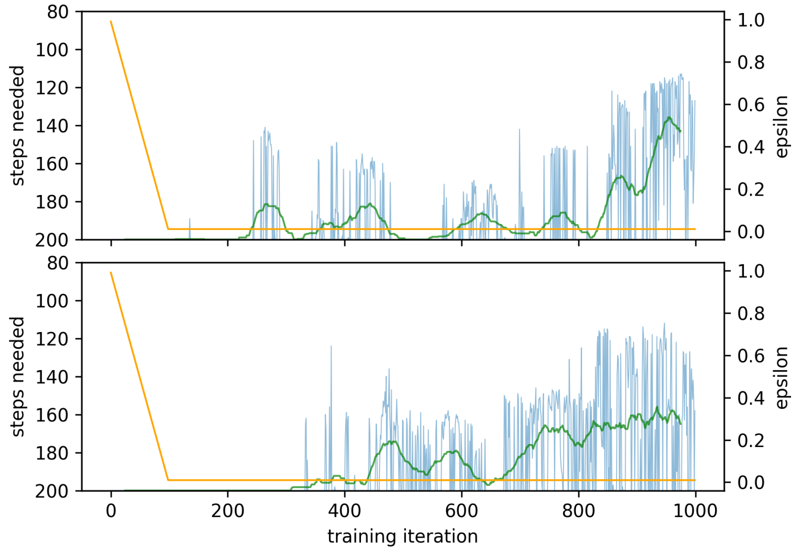

Figure 6: Two seperate runs of training the mountain car agent with a replay buffer which remembers the last 40000 timesteps. On the y-axis the number of steps needed to finish, 200 means the algorithm failed to reach the goal in time. On the x-axis the iteration number. The orange line is epsilon, the 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

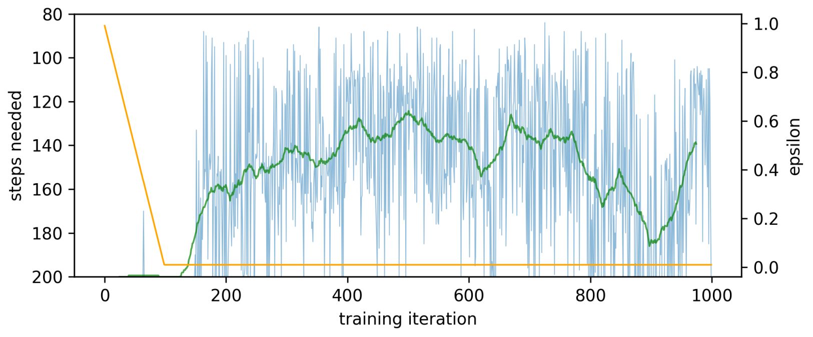

Figure 7: Two seperate runs of training the mountain car agent with a replay buffer which remembers the last 40000 timesteps. On the y-axis the number of steps needed to finish, 200 means the algorithm failed to reach the goal in time. On the x-axis the iteration number. The orange line is epsilon, the blue line follows the scores of the training iteration and the green line is the moving average of the score. The agent solves the problem after less then 200 training iterations

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

Breakout

Conclusion

Appendices