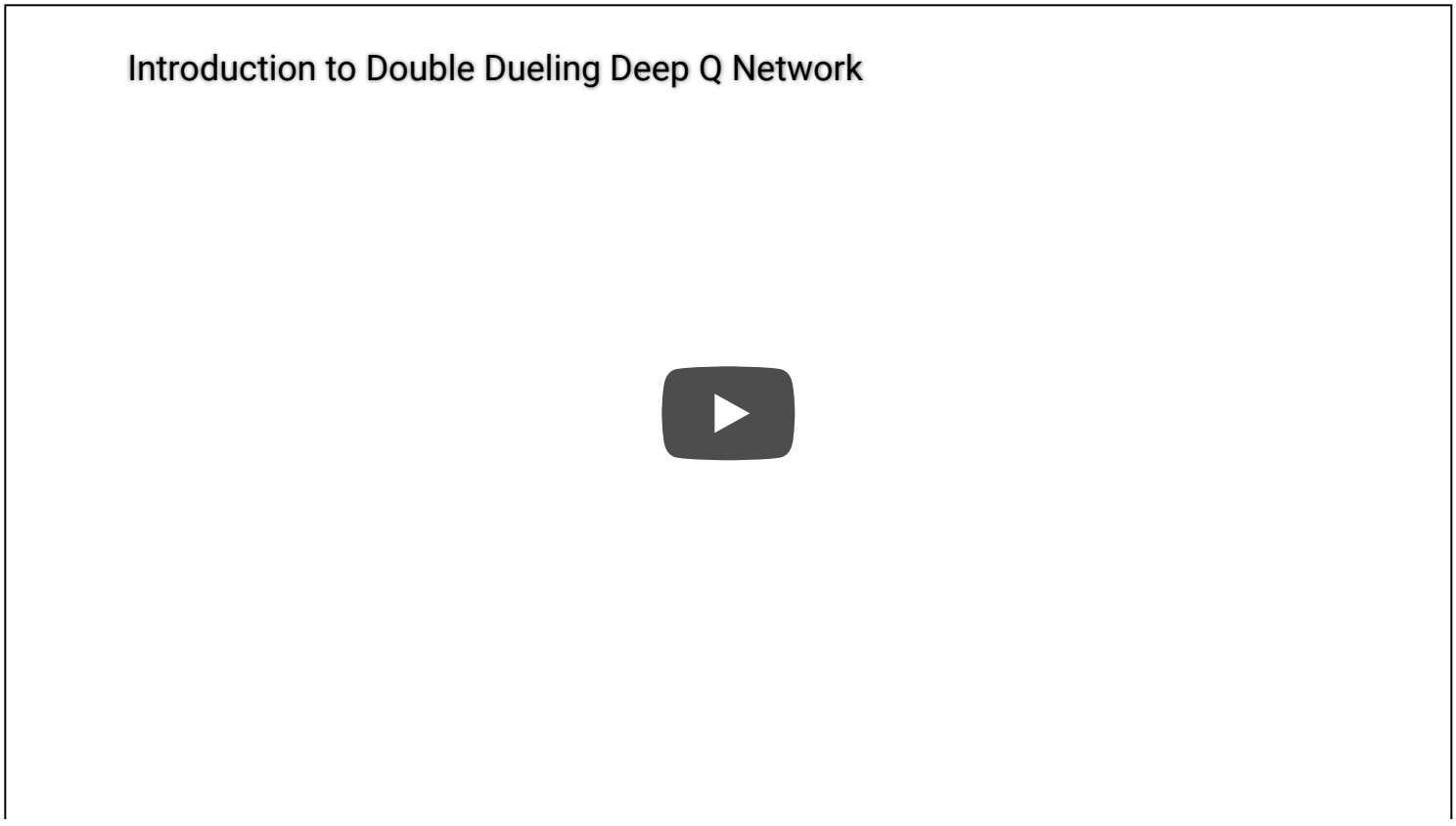
Introduction to Dueling Double Deep Q Network (D3QN)

Rokas Balsys Follow
Jan 14 · 13 min read ★

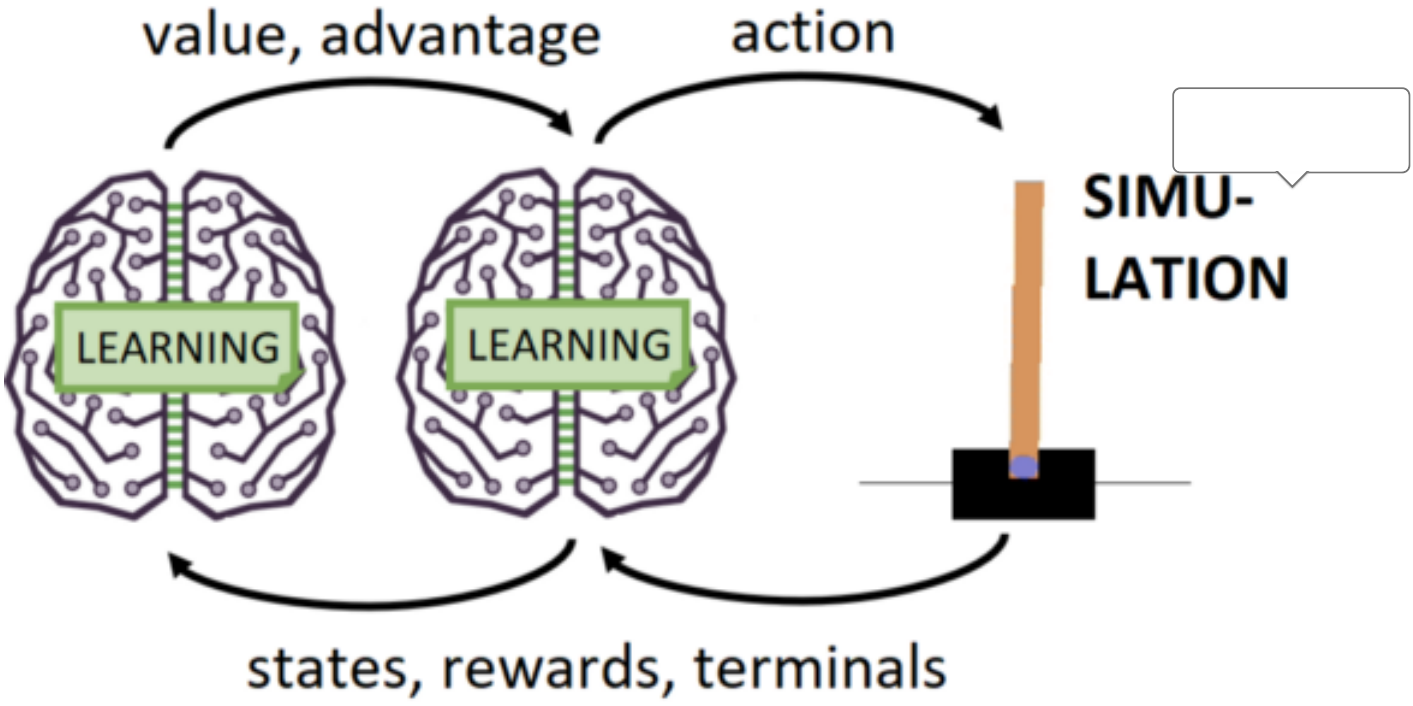


D3QN video tutorial

Dueling DQN introduction

In this post, we'll be covering Dueling DQN Networks for reinforcement learning. This reinforcement learning architecture is an improvement on our previous tutorial (Double

DQN) architecture, so before reading this tutorial, I recommend you to read my previous tutorials. In this tutorial, I'll introduce the Dueling Deep Q Network architecture (D3QN), it's advantages and how to build one in Keras. We'll be running the code on the same Open AI gym's CartPole environment so that everyone could train and test the network quickly and easily.



In future tutorials, I'll be testing this code on environments which are more complicated games which will need Convolutional Neural Networks. For an introduction to reinforcement learning, check out my previous tutorials. All the code for this and previous tutorials can be found on this site's GitHub repo.

Double DQN introduction

To get deeper understanding of Dueling Network, we should recap some points from Double Q learning. As discussed in previous DDQN tutorial, vanilla deep Q learning has some problems. These problems can be boiled down to two main issues:

- The bias problem: vanilla deep Q Network tend to overestimate rewards in noisy environments, leading to non-optimal training outcomes

- The moving target problem: because the same network is responsible for both the choosing of actions and the evaluation of actions, this leads to training instability

With regards to 1st point — say we have a state with two possible actions, each giving noisy rewards. Action a returns a random reward based on a normal distribution with a mean of 2 and a standard deviation of 1 — N(2, 1). Action b returns a random reward from a normal distribution of N(1, 4). On average, action a is the optimal action to take in this state — however, because of the argmax function in deep Q learning, action b will tend to be favored because of the higher standard deviation / higher random rewards.

With regards to 2nd point — let's consider another state, state 1, with three possible actions a, b, and c. Let's say we know that b is the optimal action. However, when we first initialize the Neural Network, in state 1, action a tends to be chosen. When we're training our Network, the loss function will drive the weights of the network towards choosing action b. However, next time we are in state 1, the parameters of the network have changed to such a degree that now action c is chosen. Ideally, we would have liked the network to consistently chose action in state 1 until it was gradually trained to choose action b. But now the goal posts have shifted, and we are trying to move the network from c to b instead of a to b — this gives rise to instability in training. This is the problem that arises when we have the same network both choosing actions and evaluating the worth of actions.

To overcome this problem, Double Q learning proposed the following way of determining the target Q value:

$$Q_{target} = r_{t+1} + \gamma Q(s_{t+1}, argmax Q(s_{t+1}, a; \theta_t); \theta_t^-)$$

Here θt refers to the primary network parameters (weights) at time t, and θ−t refers to something called the target network parameters at time t. This target network is a kind of delayed copy of the primary network. As can be observed, the optimal action in state t + 1 is chosen from the primary network (θt) but the evaluation or estimate of the Q value of this action is determined from the target network (θ−t). This can be shown more clearly by the following equations:

$$a* = argmax Q(s_{t+1}, a; \theta_t)$$

$$Q_{target} = r_{t+1} + \gamma Q(s_{t+1}, a*; \theta_t^-)$$

By doing these two things occur. First, different networks are used to choose the actions and evaluate the actions. This breaks the moving target problem mentioned earlier. Second, the primary network and the target network have essentially been trained on different samples from the memory bank of states and actions (the target network is "trained" on older samples than the primary network). Because of this, any bias due to environmental randomness should be "smoothed out". As was shown in my previous tutorial on Double Q learning, there is a significant improvement in using Double Q learning instead of vanilla deep Q learning. However, a further improvement can be made on the Double Q idea — the Dueling Q architecture, which will be covered next.

Dueling DQN theory

The Dueling DQN architecture trades on the idea that the evaluation of the Q function implicitly calculates two quantities:

- V(s) — the value of being in state s
- A(s, a) — the advantage of taking action a in state s

These values, along with the Q function Q(s, a), are very important to understand, so we will do a deep dive of these concepts. Let's first examine the generalized formula for the value function V(s):

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{i=1}^{T} \gamma^{i-1} r_i\right]$$

This above formula means that the value function at state s, operating under a policy, is the summation of future discounted rewards starting from state s. In other words, if an agent starts at s, it is the sum of all the rewards the agent collects operating under a given policy π. The E is the expectation operator. For example, let's assume an agent is playing a game with a set number of turns. In the second-to-last turn, the agent is in state s. From this state, it has 3 possible actions, with a reward of 10, 50 and 100 respectively. Let's say that the policy for this agent is a simple random selection. Because

this is the last set of actions and rewards in the game, due to the game finishing next turn, there are no discounted future rewards. The value for this state and the random action policy is:

$$V^{\pi}(s) = \mathbb{E}\left[random\left(10, 50, 100\right)\right] = 53.333$$

Clearly this policy is not going to produce optimum outcomes. However, we know that for the optimum policy, the best value of this state would be:

$$V^{*}(s) = \max(10, 50, 100) = 100$$

If we recall, from Q learning theory, the optimal action in this state is:

$$a* = argmaxQ(s_{t+1}, a)$$

and the optimal Q value from this action in this state would be:

$$Q(s, a^{*}) = \max(10, 50, 100) = 100$$

Therefore, under the optimal (deterministic) policy we have:

$$Q(s, a^{*}) = V(s)$$

But what if we aren't operating under the optimal policy (yet)? Let's get back to the case where our policy is simple random action selection. In such a case, the Q function at state s could be described as (remember there are no future discounted rewards, and V(s) = 53.333):

The term (-43.33, -3.33, 46.67) under such an analysis is called the Advantage function A(s, a). The Advantage function expresses the relative benefits of the various actions possible in state s. The Q function can therefore be expressed as:

Under the optimum policy we have $A(s,a*)=0$ and $V(s)=100$ and full result is:

But why do we want to decompose the Q function in this way? Because there is a difference between the value of a particular state s and the actions proceeding from that state. Consider a game where, from a given state s*, all actions lead to the agent dying and ending the game. This is an inherently low value state to be in, and who cares about the actions which one can take in such a state? It is pointless for the learning algorithm to waste training resources trying to find the best actions to take. In such a state, the Q values should be based solely on the value function V, and this state should be avoided. The converse case also holds — some states are just inherently valuable to be in, regardless of the effects of subsequent actions.

Consider below images taken from the original Dueling DQN paper — showing the value and advantage components of the Q value in the Atari game:

Enduro atari game

In the Atari Enduro game, the goal of the agent is to pass as many cars as possible. Crashing into a car slows the agent's car down and therefore reduces the number of cars which will be overtaken. In the images above, it can be observed that the value stream considers the road ahead and the score. However, the advantage stream, does not "pay attention" to anything much when there are no cars visible. It only begins to register when there are cars close by and an action is required to avoid them. This is a good outcome, as when no cars are in view, the network should not be trying to determine which actions to take as this is a waste of training resources. This is nice example of showing the benefit of splitting value and advantage functions.

If the ML engineer already knows that it is important to separate value and advantage values, why not build them into the architecture of the network and save the learning algorithm the hassle? That is essentially what the Dueling Q network architecture does. Consider the image below showing the original Dueling DQN architecture:

original Dueling DQN architecture

First, notice that the first part of architecture is common, with CNN input filters and a common Flatten layer (for more on convolutional neural networks, see this tutorial), we

will use this architecture later when we'll do something more serious than CardPole. After the flatten layer, the network divides to two separate densely connected layers. The first densely connected layer produces a single output corresponding to V(s). The second densely connected layer produces n outputs, where n is the number of available actions — and each of these outputs is the expression of the advantage function. These value and advantage functions are then aggregated in the Aggregation layer to produce Q values estimations for each possible action in state s. These Q values can then be trained to approach the target Q values, generated via the Double Q mechanism i.e.:

Through these separate value and advantage streams, the network will learn to produce accurate estimates of the values and advantages, improving learning performance. What goes on in the aggregation layer? One might think we could just add the V(s) and A(s, a) values together like so: $Q(s,a)=V(s)+A(s,a)$ However, there is an issue here and it's called the problem of identifiability. This problem in the current context can be stated as follows: given Q, there is no way to uniquely identify V or A. What does this mean? Say that the network is trying to learn some optimal Q value for action a. Given this Q value, can we uniquely learn a V(s) and A(s, a) value? Under the formulation above, the answer is no.

Let's say the "true" value of being in state s is 50 i.e. $V(s) = 50$. Let's also say the "true" advantage in state s for action a is 10. This will give a Q value, Q(s, a) of 60 for this state and action. However, we can also arrive at the same Q value for a learned V(s) of, say, 0, and an advantage function $A(s, a) = 60$. Or alternatively, a learned V(s) of -1000 and an advantage A(s, a) of 1060. In other words, there is no way to guarantee the "true" values of V(s) and A(s, a) are being learned separately and uniquely from each other. The commonly used solution to this problem is to instead perform the following aggregation function:

Here the advantage function value is normalized with respect to the mean of the advantage function values over all actions in state s.

So as usual we create a common network, consisting of introductory layers which act to process state inputs. Then, two separate streams are created using densely connected layers which learn the value and advantage estimates, respectively. These are then combined in a special aggregation layer which calculates the equation explained above to finally arrive at Q values. Once the network architecture is specified in accordance with the above description, the training proceeds in the same fashion as Double Q learning. The agent actions can be selected either directly from the output of the advantage function, or from the output Q values. Because the Q values differ from the advantage values only by the addition of the V(s) value (which is independent of the actions), the argmax-based selection of the best action will be the same regardless of whether it is extracted from the advantage or the Q values of the network.

Dueling DQN network model

In this part we will building a Dueling DQN architecture. However, the code will be written so that both Double Q and Dueling Q networks will be able to be constructed with the simple change of a Boolean identifier, so that we could easily compare results. While Dueling Q was originally designed for processing images, with its multiple CNN layers at the beginning of the model, in this example we will be replacing the CNN layers with simple dense connected layers, so our results may do not show any improvement. Because training reinforcement learning agents using images only (i.e. Atari RL environments) takes a long time, in this introductory post, only a simple environment is used for training the model. In future tutorials we will learn how to train RL model on games like Atari.

So, lets begin with creating a Keras model where we could easily switch between Dueling or not dueling DQN model:

Let’s go through the above code line by line. First, a number of parameters are passed to this model as part of its initialization — data input shape, the number of actions in the environment and finally a Boolean variable — dueling, to specify whether the network should be a Q network or a Dueling Q network. The first three model layers defined are simple Keras densely connected layers. These layers have ReLU activations and use the

He uniform weight initialization. If in fact the network is to be a Q network (i.e. dueling == False), then there will simply be a fourth densely connected layer followed by the output of third Q layer.

However, if the network is specified to be a Dueling Q network (i.e. dueling == True), then the value and advantage streams are created. Then a final, single dense layer is created to output the single state_value estimation (V(s)) for the given state. However, keeping with the Dueling Q terminology, the last dense layer associated with the action_advantage stream is simply another standard dense layer of size = action_space, each of these outputs will learn to estimate the advantage of all the actions in the given state (A(s, a)).

These layers specify the advantage and value streams respectively. Now the aggregation layer is created. This aggregation layer is created by using two Keras layers — a Lambda layer and an Add layer. The Lambda layer allows the developer to specify some user-defined operation to perform on the inputs to the layer. In this case, we want the layer to calculate the following:

This is calculated easily by using the lambda ***lambda a: a[:, :] — k.mean(a[:, :], keepdims=True)*** expression in the Lambda layer. Finally, we need a simple Keras addition layer to add this mean-normalized advantage function to the value estimation. This completes the explanation of our defined model function.

Full code for this tutorial can be found on GitHub link.

A comparison of the training progress of the agent in the CartPole environment under Double DQN, Dueling DQN, and Double dueling DQN architectures can be observed in the figures below, with the x-axis being the number of episodes:

1. First example is with Single dueling DQN:

Single dueling DQN example

2. Second example is with Double DQN:

Double DQN example

3. Third example is with Double dueling DQN:

Double dueling DQN example

From above graphs we can see that single dueling cartpole isn't better than single not dueling cartpole. Second example we made with DDQN architecture that we could get comparable results with Dueling DDQN architecture. Seeing the graph, we can notice that while using dueling we received much more maximum spikes, and if we would save the model on top of spikes, while testing, we would receive quite nice results.

Conclusion:

As can be observed, there is a slightly higher performance of the Double Q network with respect to the Dueling Q network. However, the performance difference is fairly marginal, and may be within the variation arising from the random weight initialization of the networks. There is also the issue of the Dueling Q network being slightly more complicated due to the additional value stream. As such, on a fairly simple environment like the CartPole environment, the benefits of Dueling Q over Double Q may not be realized. However, in more complex environments like Atari environments, it is likely that the Dueling Q architecture will be superior to Double Q (this is what the original Dueling Q paper has shown). So, in future tutorials we will demonstrate the Dueling DQN architecture in Atari or other more complex environments.

In the next tutorial we will try to implement Prioritized Experience Replay method and we'll see what we can get with it.

Reinforcement learning tutorial series:

1. Deep Q Learning tutorial (DQN)
2. Double Deep Q Learning tutorial (DDQN)

3. Dueling Double Deep Q Learning tutorial (D3QN) — Current tutorial
4. Epsilon Greedy Dueling Double Deep Q Learning tutorial (D3QN)
5. Prioritized Experience Replay (PER) D3QN tutorial
6. D3QN PER with Convolutional Neural Networks tutorial
7. A.I. learns to play Pong game from pixels with DQN
8. Introduction to RL Policy Gradient (PG or REINFORCE)
9. Introduction to Advanced Actor Critic algorithm (A2C)
10. Introduction to Asynchronous Advanced Actor Critic algorithm (A3C)
11. Introduction to Proximal Policy Optimization algorithm (PPO) — Current tutorial


Download on the
App Store


GET IT ON
Google Play