Simple Reinforcement Learning with Tensorflow Part 4: Deep Q-Networks and Beyond

A smart game agent will learn to avoid dangerous holes in the ground.

Welcome to the latest installment of my Reinforcement Learning series. In this tutorial we will be walking through the creation of a Deep Q-Network. It will be built upon the simple one layer Q-network we created in Part 0, so I would recommend reading that first if you are new to reinforcement learning. While our ordinary Q-network was able to barely perform as well as the Q-Table in a simple game environment, Deep Q-Networks are much more capable. In order to transform an ordinary Q-Network into a DQN we will be making the following improvements:

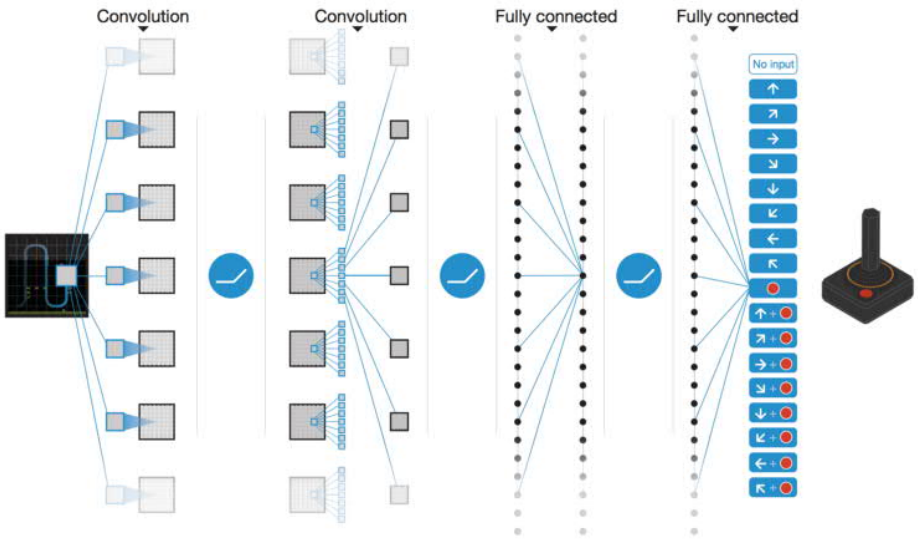
Going from a single-layer network to a multi-layer convolutional network.

Implementing Experience Replay, which will allow our network to train itself using stored memories from it's experience.

Utilizing a second "target" network, which we will use to compute target Q-values during our updates.

It was these three innovations that allowed the Google DeepMind team to achieve superhuman performance on dozens of Atari games using their DQN agent. We will be walking through each individual improvement, and showing how to implement it. We won't stop there though. The pace of Deep Learning research is extremely fast, and the DQN of 2014 is no longer the most advanced agent around anymore. I will discuss two simple additional improvements to the DQN architecture, Double DQN and Dueling DQN, that allow for improved performance, stability, and faster training time. In the end we will have a network that can tackle a number of challenging Atari games, and we will demonstrate how to train the DQN to learn a basic navigation task.

Getting from Q-Network to Deep Q-Network



Addition 1: Convolutional Layers

Since our agent is going to be learning to play video games, it has to be able to make sense of the game's screen output in a way that is at least similar to how humans or other intelligent animals are able to. Instead of considering each pixel independently, convolutional layers allow us to consider regions of an image, and maintain spatial relationships between the objects on the screen as we send information up to higher levels

of the network. In this way, they act similarly to human receptive fields. Indeed there is a body of research showing that convolutional neural network learn representations that are similar to those of the primate visual cortex. As such, they are ideal for the first few elements within our network.

In Tensorflow, we can utilize the tf.contrib.layers.convolution2d function to easily create a convolutional layer. We write for function as follows:

convolution_layer = tf.contrib.layers.convolution2d(inputs,num_outputs,kernel_size,stride,padding)

Here *num_outs* refers to how many filters we would like to apply to the previous layer. *kernel_size* refers to how large a window we would like to slide over the previous layer. *Stride*refers to how many pixels we want to skip as we slide the window across the layer. Finally, *padding* refers to whether we want our window to slide over just the bottom layer ("VALID") or add padding around it ("SAME") in order to ensure that the convolutional layer has the same dimensions as the previous layer. For more information, see the Tensorflow documentation.

Addition 2: Experience Replay

The second major addition to make DQNs work is Experience Replay. The basic idea is that by storing an agent's experiences, and then randomly drawing batches of them to train the network, we can more robustly learn to perform well in the task. By keeping the experiences we draw random, we prevent the network from only learning about what it is immediately doing in the environment, and allow it to learn from a more varied array of past experiences. Each of these experiences are stored as a tuple of <*state,action,reward,next state*>. The Experience Replay buffer stores a fixed number of recent memories, and as new ones come in, old ones are removed. When the time comes to train, we simply draw a uniform batch of random memories from the buffer, and train our network with them. For our DQN, we will build a simple class that handles storing and retrieving memories.

Addition 3: Separate Target Network

The third major addition to the DQN that makes it unique is the utilization of a second network during the training procedure. This second network is used to generate the target-Q values that will be used to compute the loss for every action during training. Why not use just use one network for both estimations? The issue is that at every step of training, the Q-network's values shift, and if we are using a constantly shifting set of values to adjust our network values, then the value estimations can easily spiral out of control. The network can become destabilized by falling into feedback loops between the target and estimated Q-values. In order to mitigate that risk, the the target network's weights are fixed, and only periodically or slowly updated to the primary Q-networks values. In this way training can proceed in a more stable manner.

Instead of updating the target network periodically and all at once, we will be updating it frequently, but slowly. This technique was introduced in another DeepMind paper earlier this year, where they found that it stabilized the training process.

Going Beyond DQN

With the additions above, we have everything we need to replicate the DWN of 2014. But the world moves fast, and a number of improvements above and beyond the DQN architecture described by DeepMind, have allowed for even greater performance and stability. Before training your new DQN on your favorite ATARI game, I would suggest checking the newer additions out. I will provide a description and some code for two of them: Double DQN, and Dueling DQN. Both are simple to implement, and by combining both techniques, we can achieve better performance with faster training times.

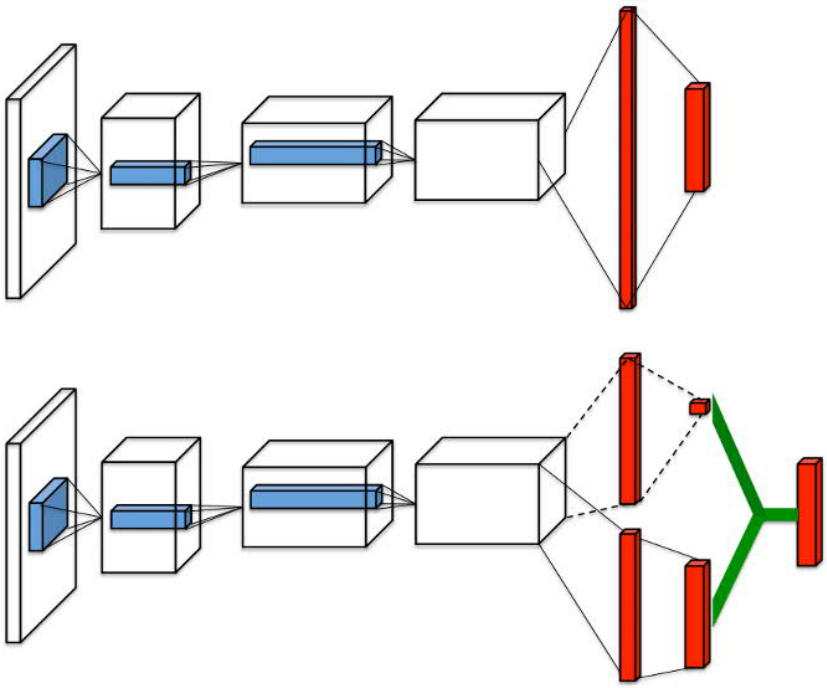
Double DQN

The main intuition behind Double DQN is that the regular DQN often overestimates the Q-values of the potential actions to take in a given state. While this would be fine if all actions were always overestimates equally, there was reason to believe this wasn't the case. You can easily imagine that if certain suboptimal actions regularly were given higher Q-values than optimal actions, the agent would have a hard time ever learning the ideal policy. In order to correct for this, the authors of DDQN paper propose a simple trick: instead of taking the max over Q-values when computing the target-Q value for our training step, we use our

primary network to chose an action, and our target network to generate the target Q-value for that action. By decoupling the action choice from the target Q-value generation, we are able to substantially reduce the overestimation, and train faster and more reliably. Below is the new DDQN equation for updating the target value.

$$Q\text{-}Target = r + \gamma Q(s', argmax(Q(s', a, \theta), \theta'))$$

Dueling DQN



Above: Regular DQN with a single stream for Q-values. Below: Dueling DQN where the value and advantage are calculated separately and then combined only at the final layer into a Q value.

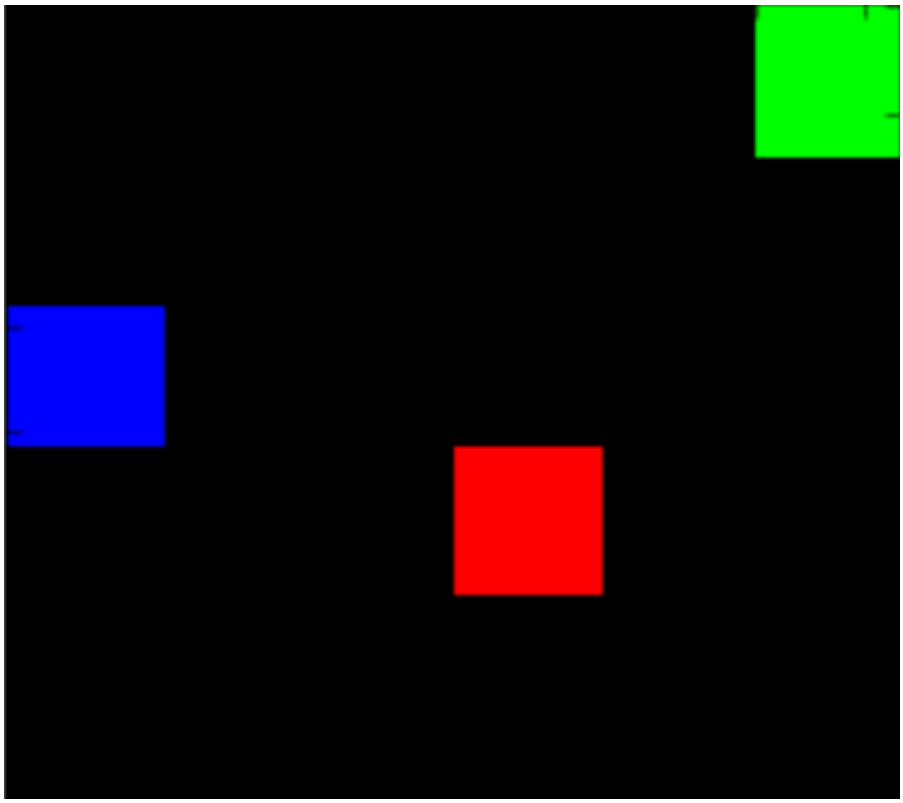
In order to explain the reasoning behind the architecture changes that Dueling DQN makes, we need to first explain some a few additional reinforcement learning terms. The Q-values that we have been discussing so far correspond to how good it is to take a certain action given a certain state. This can be written as Q(s,a). This action given state can actually be decomposed into two more fundamental notions of value. The first is the value function V(s), which says simple how good it is to be in any given state. The second is the advantage

function A(a), which tells how much better taking a certain action would be compared to the others. We can then think of Q as being the combination of V and A. More formally:

$$Q(s,a) = V(s) + A(a)$$

The goal of Dueling DQN is to have a network that separately computes the advantage and value functions, and combines them back into a single Q-function only at the final layer. It may seem somewhat pointless to do this at first glance. Why decompose a function that we will just put back together? The key to realizing the benefit is to appreciate that our reinforcement learning agent may not need to care about both value and advantage at any given time. For example: imagine sitting outside in a park watching the sunset. It is beautiful, and highly *rewarding* to be sitting there. No action needs to be taken, and it doesn't really make sense to think of the value of sitting there as being conditioned on anything beyond the environmental state you are in. We can achieve more robust estimates of state value by decoupling it from the necessity of being attached to specific actions.

Putting it all together



Simple block–world environment. The goal is to move the blue block to the green block while avoiding the red block.

Now that we have learned all the tricks to get the most out of our DQN, let's actually try it on a game environment! While the DQN we have described above could learn ATARI games with enough training, getting the network to perform well on those games takes at least a day of training on a powerful machine. For educational purposes, I have built a simple game environment which our DQN learns to master in a couple hours on a moderately powerful machine (I am using a GTX970). In the environment the agent controls a blue square, and the goal is to navigate to the green squares (reward +1) while avoiding the red squares (reward -1). At the start of each episode all squares are randomly placed within a 5x5 grid-world. The agent has 50 steps to achieve as large a reward as possible. Because they are randomly positioned, the agent needs to do more than simply learn a fixed path, as was the case in the FrozenLake environment from Tutorial 0. Instead the agent must learn a notion of spatial relationships between the blocks. And indeed, it is able to do just that!

awjuliani/DeepRL-Agents

DeepRL-Agents - A set of Deep Reinforcement Learning Agents implemented in Tensorflow.

github.com

The game environment outputs 84x84x3 color images, and uses function calls as similar to the OpenAI gym as possible. In doing so, it should be easy to modify this code to work on any of the OpenAI atari games. I encourage those with the time and computing resources necessary to try getting the agent to perform well in an ATARI game. The hyperparameters may need some tuning, but it is definitely possible. Good luck!