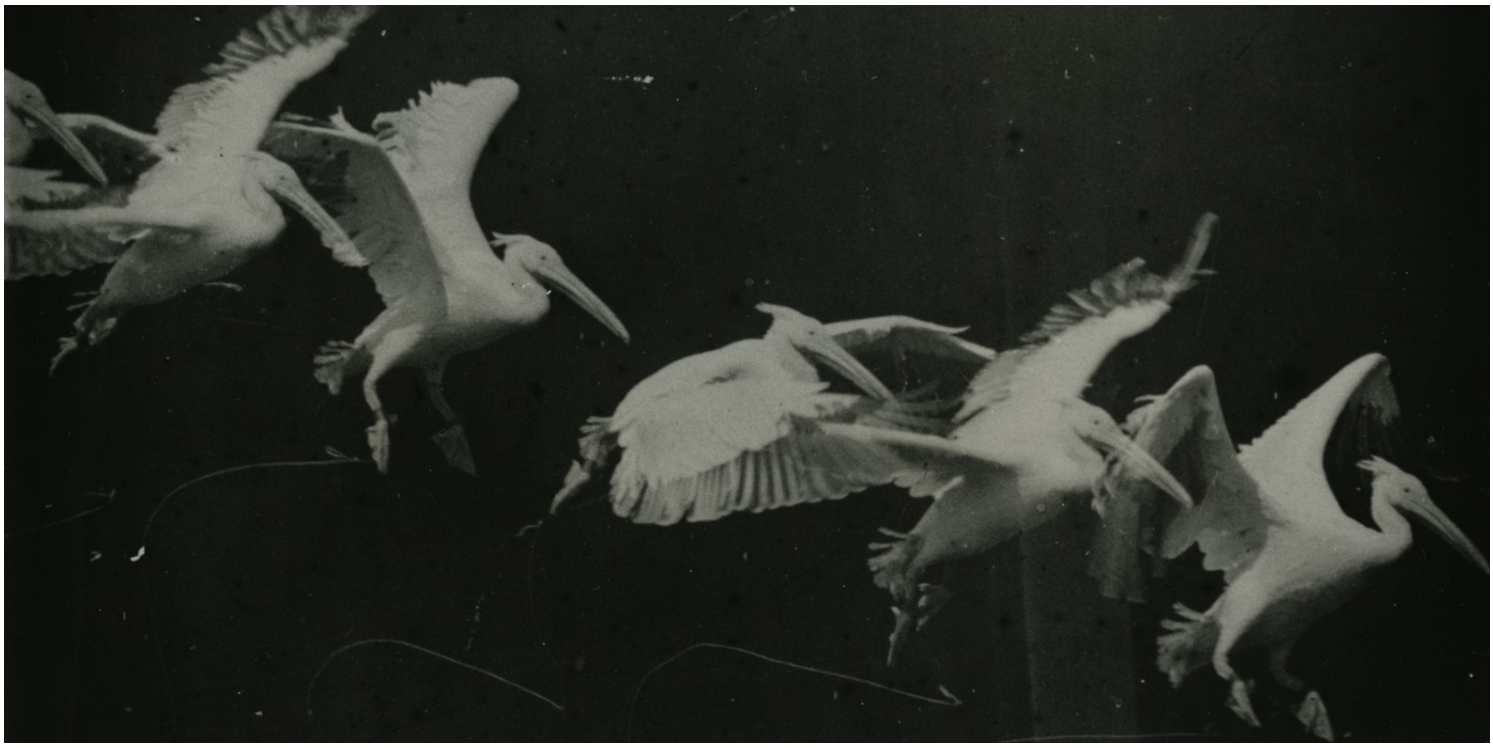
Oct 8, 2016 · 9 min read

Simple Reinforcement Learning with Tensorflow Part 6: Partial Observability and Deep Recurrent Q-Networks



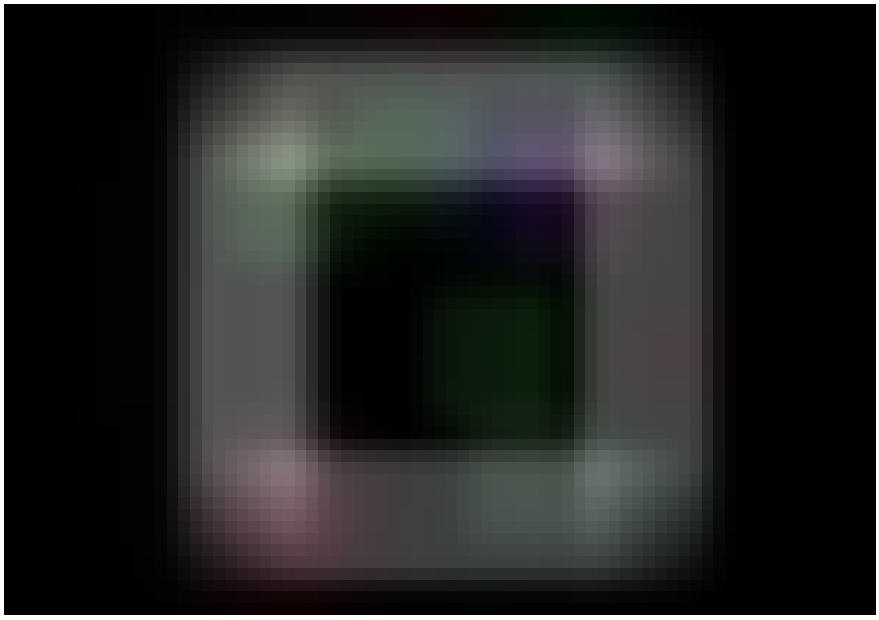
In this installment of my Simple RL series, I want to introduce the concept of *Partial Observability* and demonstrate how to design neural agents which can successfully deal with it. As always, in addition to a theoretical discussion of the problem, I have included a Tensorflow implementation of a neural agent which can solve this class of problems.

For us humans, having access to a limited and changing world is a universal aspect of our shared experience. Despite our partial access to the world, we are able to solve all sorts of challenging problems in the course of going about our daily lives. In contrast, the neural agents we

have discussed so far in this tutorial series are ill equipped to handle partial observability.

The Problem of Partial Observability

When we think about the kinds of environments used until this point to train our networks, the agent has had access to all the information about the environment it might need in order to take an optimal action. Take for example the Gridworld used in Tutorials 4 & 5 of this series:



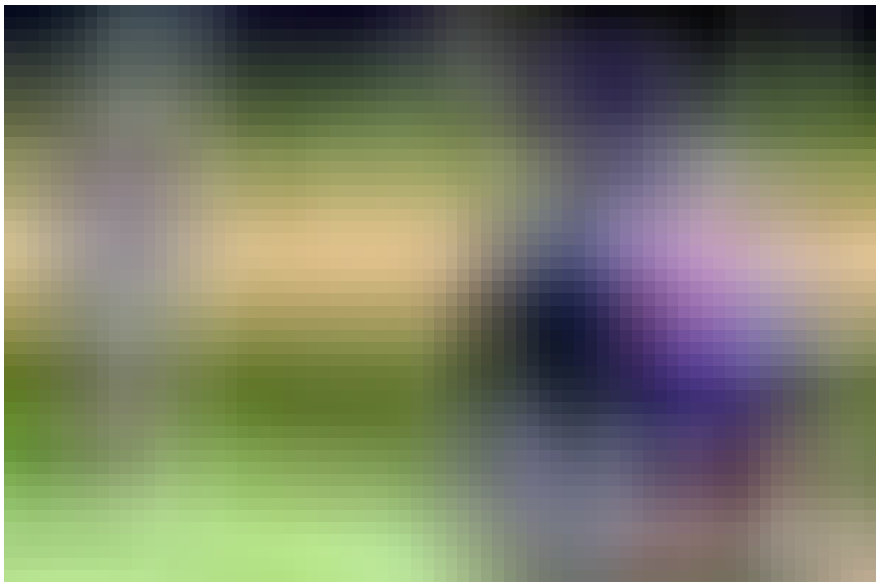
A fully observable MDP. The goal of the game is to move the blue block to as many green blocks as possible in 50 steps while avoiding red blocks. When the blue block moves to a green or red block, that other block is moved to a new random place in the environment. Green blocks provide +1 reward, while Red blocks provide -1 reward.

Because the entire world is visible at any moment (and nothing moves aside from the agent), a single frame of this environment gives the agent all it needs to know in order to maximize its reward. Environments which follow a structure where a given state conveys everything the agent needs to act optimally are called *Markov Decision Processes (MDPs).*

While MDPs provide a nice formalism, almost all real world problems fail to meet this standard. Take for example your field of view at this very moment. Can you see what is behind you? This limited perspective on the visual world is almost always the default for humans and other animals. Even if we were to have 360 degree vision, we may still not know what is on the other side of a wall just beyond us. Information

outside our view is often essential to making decisions regarding the world.

In addition to being spatially limited, information available at a given moment is also often temporally limited. When looking at a photo of a ball being thrown between two people, the lack of motion may make us unable to determine the direction and speed of the ball. In games like Pong, not only the position of the ball, but also it's direction and speed are essential to making the correct decisions.



Is the ball going from the child to the baseball player, or the other way around?

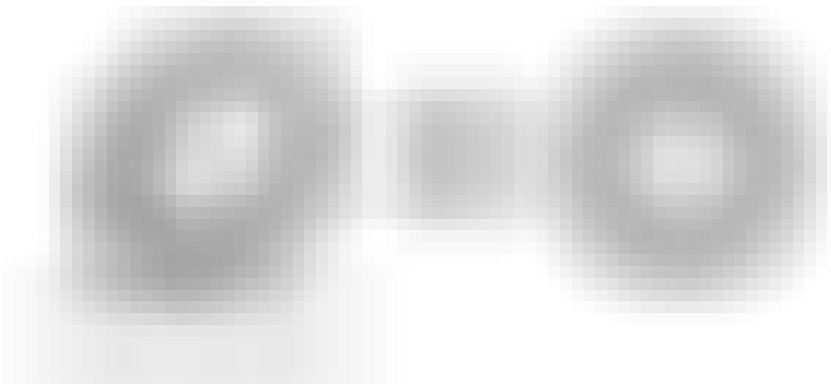
Environments which present themselves in a limited way to the agent are referred to as *Partially Observable Markov Decision Processes (POMDPs)*. While they are trickier to solve than their fully observable counterparts, understanding them is essential to solving most realistic tasks.

Making sense of a limited, changing world

How can we build a neural agent which still functions well in a partially observable world? The key is to give the agent a capacity for *temporal integration of observations*. The intuition behind this is simple: if information at a single moment isn't enough to make a good decision, then enough varying information over time probably is. Revisiting the photo example of the thrown ball A single image of a ball in motion tells us nothing about its movements, but two images in sequence

allows us to discern the direction of movement. A longer sequence might even allow us to make sense of the speed of the ball. The same principle can be applied to problems where there is a limited field of view. If you can't see behind you, by turning around you can integrate the forward and backward views over time and get a complete picture of the world with which to act upon.

Within the context of Reinforcement Learning, there are a number of possible ways to accomplish this temporal integration. The solution taken by DeepMind in their original paper on Deep Q-Networks was to stack the frames from the Atari simulator. Instead of feeding the network a single frame at a time, they used an external frame buffer which kept the last four frames of the game in memory and fed this to the neural network. This approach worked relatively well for the simple games they employed, but it isn't ideal for a number of reasons. The first is that it isn't necessarily biologically plausible. When light hits our retinas, it does it at a single moment. There is no way for light to be stored up and passed all at once to an eye. Secondly, by using blocks of 4 frames as their state, the experience buffer used needed to be much larger to accommodate the larger stored states. This makes the training process require a larger amount of potentially unnecessary memory. Lastly, we may simply need to keep things in mind that happened much earlier than would be feasible to capture with stacking frames. Sometimes an event hundreds of frames earlier might be essential to deciding what to do at the current moment. We need a way for our agent to keep events in mind more robustly.

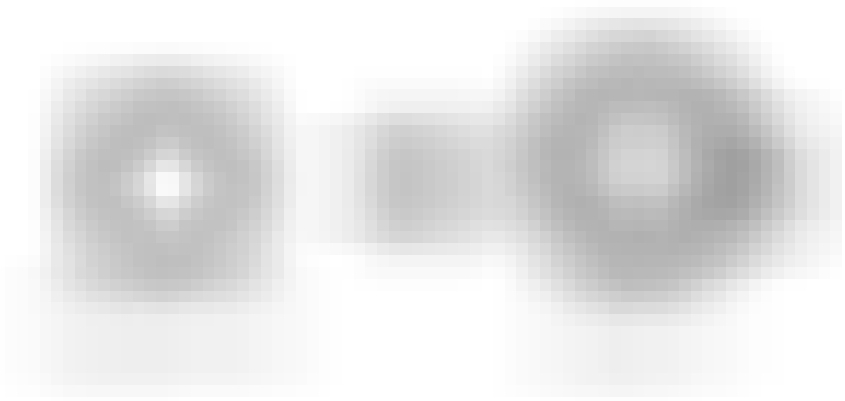


The approach used by DeepMind for dealing with partial observability.

Recurrent Neural Networks

All of these issues can be solved by moving the temporal integration into the agent itself. This is accomplished by utilizing a recurrent block in our neural agent. You may have heard of recurrent neural networks, and their capacity to learn temporal dependencies. This has been used popularly for the purpose of text generation, where groups have trained RNNs to reproduce everything from Barack Obama speeches to freeform poetry. Andrej Karpathy has a great post outlining RNNs and their capacities, which I highly recommend. Thanks to the high-level nature of Tensorflow, we are free to treat the RNN as somewhat of a black-box that we simply plug into our existing Deep Q-Network.

By utilizing a recurrent block in our network, we can pass the agent single frames of the environment, and the network will be able to change its output depending on the temporal pattern of observations it receives. It does this by maintaining a hidden state that it computes at every time-step. The recurrent block can feed the hidden state back into itself, thus acting as an augmentation which tells the network what has come before. The class of agents which utilize this recurrent network are referred to as Deep Recurrent Q-Networks (DRQN).



The approach taken DRQN which allows for neural agent to learn temporal patterns.

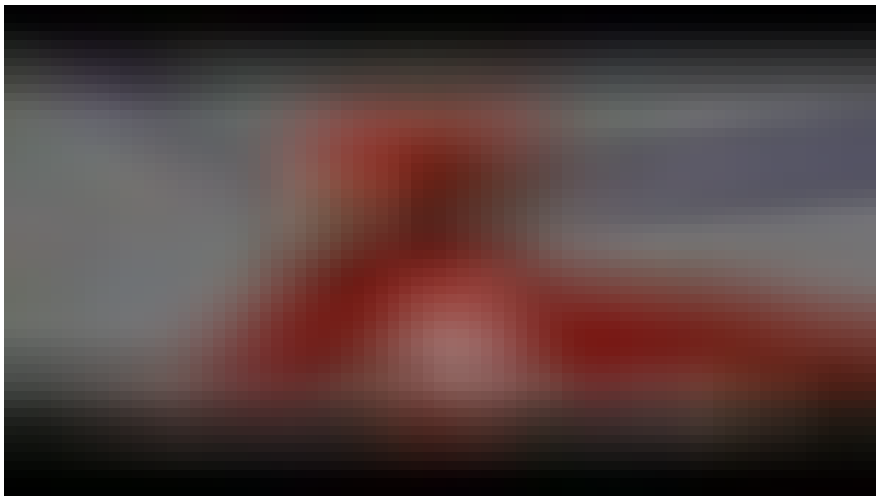
Implementing in Tensorflow

In order to implement a Deep Recurrent Q-Network (DRQN) architecture in Tensorflow, we need to make a few modifications to our DQN described in Part 4 (See below for full implementation, or follow link here).

- The first change is to the agent itself. We will insert a LSTM recurrent cell between the output of the last convolutional layer

and the input into the split between the Value and Advantage streams. We can do this by utilizing the *tf.nn.dynamic_rnn* function and defining a *tf.nn.rnn_cell.LSTMCell* which is feed to the rnn node. We also need to slightly alter the training process in order to send an empty hidden state to our recurrent cell at the beginning of each sequence.

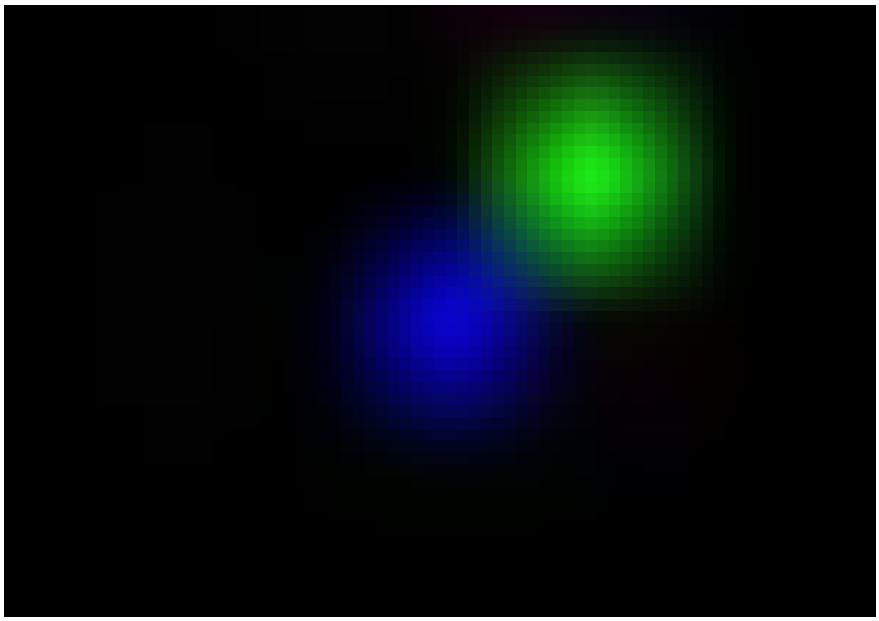
- The second main change needed will be to adjust the way our experience buffer stores memories. Since we want to train our network to understand temporal dependencies, we can't use random batches of experience. Instead we need to be able to draw traces of experience of a given length. In this implementation, our experience buffer will store entire episodes, and randomly draw traces of 8 steps from a random batch of episodes. By doing this we both retain our random sampling as well as ensure each trace of experiences actually follows from one another.
- Finally, we will be utilizing a technique developed by a group at Carnegie Mellon who recently used DRQN to train a neural network to play the first person shooter game Doom. Instead of sending all the gradients backwards when training their agent, they sent only the last half of the gradients for a given trace. We can do this by simply masking the loss for the first half of each trace in a batch. I suggest reading their paper for a more thorough explanation of this, but they found it improved performance by only sending more meaningful information through the network. In fact, it worked so well that the agent was able (with a few other tricks) to play Doom at a convincing level.



A DRQN playing Doom. For more information, see https://arxiv.org/abs/1609.05521

The Limited Gridworld

For this tutorial however I'd like to work with something a little less flashy, though hopefully more informative. Recall our Gridworld, where everything was visible to the agent at any moment. By simply limiting the agent's view of the environment, we can turn our MDP into a POMDP. In this new version of the GridWorld, the agent can only see a single block around it in any direction, whereas the environment itself contains 9x9 blocks. Additional changes are as follows: each episode is fixed at 50 steps, there are four green and two red squares, and when the agent moves to a red or green square, a new one is randomly placed in the environment to replace it. What are the consequences of this?



In the limited GridWorld, the agent can only see 3x3 area around it, whereas the environment has 9x9 blocks total (see image at beginning of article).

If we attempt to use our DQN as described in Parts 4 and 5 of this series, we find that it performs relatively poorly, never achieving more than an average of 2.3 cumulative reward after 10,000 training episodes.

Average reward over time for DQN agent in limited GridWorld.

The problem is that the agent has no way of remembering where it has been or what it has seen. If two areas look the same, then the agent can do nothing but react in exactly the same way to them, even if they are in different parts of the environment. Now let's look at how our DRQN does in the same limited environment over time.

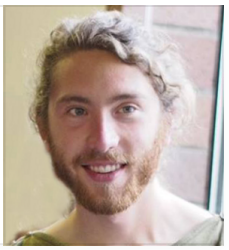
Average reward over time for DRQN agent in limited GridWorld.

By allowing for a temporal integration of information, the agent learns a sense of spatial location that is able to augment its observation at any moment, and allow the agent to receive a larger reward each episode. Below is the Ipython notebook where this DRQN agent is implemented. Feel free to replicate the results yourself, and play with the hyperparameters. Different settings for many of them may provide greater performance for your particular task.

awjuliani/DeepRL-Agents

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

github.com



With this code you have everything you need to train a DRQN that can go out into the messy world and solve problems with partial observability!

. . .

If this post has been valuable to you, please consider *donating* to help support future tutorials, articles, and implementations. Any contribution is greatly appreciated!

If you'd like to follow my work on Deep Learning, AI, and Cognitive Science, follow me on Medium @Arthur Juliani, or on twitter @awjliani.

. . .

More from my Simple Reinforcement Learning with Tensorflow series:

1. *Part 0—Q-Learning Agents*
2. *Part 1—Two-Armed Bandit*
3. *Part 1.5—Contextual Bandits*
4. *Part 2—Policy-Based Agents*
5. *Part 3—Model-Based RL*
6. *Part 4—Deep Q-Networks and Beyond*
7. *Part 5—Visualizing an Agent's Thoughts and Actions*
8. **Part 6—Partial Observability and Deep Recurrent Q-Networks**
9. *Part 7—Action-Selection Strategies for Exploration*

10. *Part 8—Asynchronous Actor-Critic Agents (A3C)*