Literature review

This task is to search the google deep mind, apply its method, deep reinforcement learning to a game of my choice. Deep mind company has already made an agent that can learn to play games through high sensory input, meaning that the agent only receives the screen pixel and the score, such that the agent can play cross wide range of games with previous knowledge.

Google DeepMind has done many research on Artificial intelligence and leading the world in this field. It has developed programs that can learn to play classic Atari 2600 games from high-dimensional sensory inputs (e.g. pixels in the screen and score, same as human player would experience). The algorithm behind it uses the deep reinforcement learning method that makes a system learn from rewards or punishments. After each training, data will be processed in the neural network and it will make some improvement to the system. That is basically how the agent learns. As in practical, Demis Hassabis, the co-founder and CEO of DeepMind, said that he was surprised that after the algorithm played the Space Invaders about 8 hrs, it was able to make a predict shot to kill the last alien, and after it played the Breakout 500 times, it discovered the optimal strategy which is to dig a tunnel round the left-hand and then send the ball round the back. Another famous example of this algorithm is Alpha Go, who beat the human professional Go player with 9 dan (highest rank in Go) Lee Sedol with the score of 4-1. AlphaGo was initially given 100,000 games to mimic human player, after that it was ordered to play the games itself 30,000,000 times, which made the Alpha Go stronger than human. Those examples show that deep reinforcement learning algorithm that DeepMind has been using is pretty strong. Demis also emphasised that the machine learns automatically from raw inputs, i.e. without pre-programmed, and thus that the same system can operate across a wide range of tasks. This reflects the ‘deep’ in ‘deep reinforcement learning’.

In this project, I will use the deep reinforcement learning algorithm to learn to play the game of my choice, and compare this algorithm with other different algorithms. The game I have decided is called StarCraft2. It is a real-time strategy video game developed by Blizzard Entertainment. There are many different units with different abilities and other elements in the game, these elements make the game colourful, complex and has many strategic possibilities. Here are the reasons that I chose the game: Firstly, it is one of my favourite game, I used to play a lot of this games and I have plenty knowledge of this game including its culture. Secondly, this game is really fun to play due to its complexity, unlike Go or any Atari 2600 game which has much simpler control. Hence Starcraft2 is more challenging for the algorithm to learn. And Finally, DeepMind has the source of Starcraft2 learning environment called PySC2, and it is open for everyone, which is really a good opportunity for me doing this project. On the other side, there are already some AI in this game, but those AI are not strong enough comparing to the master human player and they do not learn. Therefore, after I finish my code, I will compare it with the existing AI in order to see the difference between these algorithms and my algorithm. In summary, the problem is to apply the deep reinforcement learning to play the Starcraft2 and compare with the other algorithms in the game. The technique is the reinforcement learning algorithm, and the tool is the Starcraft2 learning environment PySC2.

The game of Pong is an excellent example of a simple RL task. In the ATARI 2600 version we’ll use you play as one of the paddles (the other is controlled by a decent AI) and you have to bounce the ball past the other player (I don’t really have to explain Pong, right?). On the low level the game works as follows: we receive an image frame (a 210x160x3 byte array (integers from 0 to 255 giving pixel values)) and we get to decide if we want to move the paddle UP or DOWN (i.e. a binary choice). After every single choice the game simulator executes the action and gives us a reward: Either a +1 reward if the ball went past the opponent, a -1 reward if we missed the ball, or 0 otherwise. And of course, our goal is to move the paddle so that we get lots of reward.

First, we’re going to define a policy network that implements our player (or “agent”). This network will take the state of the game and decide what we should do (move UP or DOWN). As our favorite simple block of compute we’ll use a 2-layer neural network that takes the raw image pixels (100,800 numbers total (210\*160\*3)), and produces a single number indicating the probability of going UP. Note that it is standard to use a stochastic policy, meaning that we only produce a probability of moving UP. Every iteration we will sample from this distribution (i.e. toss a biased coin) to get the actual move. The reason for this will become more clear once we talk about training.

where in this snippet W1 and W2 are two matrices that we initialize randomly. We’re not using biases because meh. Notice that we use the sigmoid non-linearity at the end, which squashes the output probability to the range [0,1]. Intuitively, the neurons in the hidden layer (which have their weights arranged along the rows of W1) can detect various game scenarios (e.g. the ball is in the top, and our paddle is in the middle), and the weights in W2 can then decide if in each case we should be going UP or DOWN. Now, the initial random W1 and W2 will of course cause the player to spasm on spot. So the only problem now is to find W1 and W2 that lead to expert play of Pong!

**It sounds kind of impossible**. At this point I’d like you to appreciate just how difficult the RL problem is. We get 100,800 numbers (210\*160\*3) and forward our policy network (which easily involves on order of a million parameters in W1 and W2). Suppose that we decide to go UP. The game might respond that we get 0 reward this time step and gives us another 100,800 numbers for the next frame. We could repeat this process for hundred timesteps before we get any non-zero reward! E.g. suppose we finally get a +1. That’s great, but how can we tell what made that happen? Was it something we did just now? Or maybe 76 frames ago? Or maybe it had something to do with frame 10 and then frame 90? And how do we figure out which of the million knobs to change and how, in order to do better in the future? We call this the credit assignment problem. In the specific case of Pong we know that we get a +1 if the ball makes it past the opponent. The true cause is that we happened to bounce the ball on a good trajectory, but in fact we did so many frames ago - e.g. maybe about 20 in case of Pong, and every single action we did afterwards had zero effect on whether or not we end up getting the reward. In other words we’re faced with a very difficult problem and things are looking quite bleak.

**Supervised Learning**. Before we dive into the Policy Gradients solution I’d like to remind you briefly about supervised learning because, as we’ll see, RL is very similar. Refer to the diagram below. In ordinary supervised learning we would feed an image to the network and get some probabilities, e.g. for two classes UP and DOWN. I’m showing log probabilities (-1.2, -0.36) for UP and DOWN instead of the raw probabilities (30% and 70% in this case) because we always optimize the log probability of the correct label (this makes math nicer, and is equivalent to optimizing the raw probability because log is monotonic). Now, in supervised learning we would have access to a label. For example, we might be told that the correct thing to do right now is to go UP (label 0). In an implementation we would enter gradient of 1.0 on the log probability of UP and run backprop to compute the gradient vector

. This gradient would tell us how we should change every one of our million parameters to make the network slightly more likely to predict UP. For example, one of the million parameters in the network might have a gradient of -2.1, which means that if we were to increase that parameter by a small positive amount (e.g. 0.001), the log probability of UP would decrease by 2.1 \* 0.001 (decrease due to the negative sign). If we then did a parameter update then, yay, our network would now be slightly more likely to predict UP when it sees a very similar image in the future.

**Policy Gradients**. Okay, but what do we do if we do not have the correct label in the Reinforcement Learning setting? Here is the Policy Gradients solution (again refer to diagram below). Our policy network calculated probability of going UP as 30% (logprob -1.2) and DOWN as 70% (logprob -0.36). We will now sample an action from this distribution; E.g. suppose we sample DOWN, and we will execute it in the game. At this point notice one interesting fact: We could immediately fill in a gradient of 1.0 for DOWN as we did in supervised learning, and find the gradient vector that would encourage the network to be slightly more likely to do the DOWN action in the future. So we can immediately evaluate this gradient and that’s great, but the problem is that at least for now we do not yet know if going DOWN is good. But the critical point is that that’s okay, because we can simply wait a bit and see! For example in Pong we could wait until the end of the game, then take the reward we get (either +1 if we won or -1 if we lost), and enter that scalar as the gradient for the action we have taken (DOWN in this case). In the example below, going DOWN ended up to us losing the game (-1 reward). So if we fill in -1 for log probability of DOWN and do backprop we will find a gradient that discourages the network to take the DOWN action for that input in the future (and rightly so, since taking that action led to us losing the game).

And that’s it: we have a stochastic policy that samples actions and then actions that happen to eventually lead to good outcomes get encouraged in the future, and actions taken that lead to bad outcomes get discouraged. Also, the reward does not even need to be +1 or -1 if we win the game eventually. It can be an arbitrary measure of some kind of eventual quality. For example if things turn out really well it could be 10.0, which we would then enter as the gradient instead of -1 to start off backprop. That’s the beauty of neural nets; Using them can feel like cheating: You’re allowed to have 1 million parameters embedded in 1 teraflop of compute and you can make it do arbitrary things with SGD. It shouldn’t work, but amusingly we live in a universe where it does.

**Training protocol.** So here is how the training will work in detail. We will initialize the policy network with some W1, W2 and play 100 games of Pong (we call these policy “rollouts”). Lets assume that each game is made up of 200 frames so in total we’ve made 20,000 decisions for going UP or DOWN and for each one of these we know the parameter gradient, which tells us how we should change the parameters if we wanted to encourage that decision in that state in the future. All that remains now is to label every decision we’ve made as good or bad. For example suppose we won 12 games and lost 88. We’ll take all 200\*12 = 2400 decisions we made in the winning games and do a positive update (filling in a +1.0 in the gradient for the sampled action, doing backprop, and parameter update encouraging the actions we picked in all those states). And we’ll take the other 200\*88 = 17600 decisions we made in the losing games and do a negative update (discouraging whatever we did). And… that’s it. The network will now become slightly more likely to repeat actions that worked, and slightly less likely to repeat actions that didn’t work. Now we play another 100 games with our new, slightly improved policy and rinse and repeat.

Starcraft2 poses new grand challenge for RL.

* Multi-agent with problem with multiple players interacting.
* Imperfect information due to partially observed map.
* Large actions space involving select and control 100s units.
* Delayed credit assignment requiring long-term strategies over 1000s steps

Tools:

* Mini maps are provided for focusing on different elements.
* Dataset of game play from expert players.