Guest Post (Part I): Demystifying Deep Reinforcement Learning

Two years ago, a small company in London called DeepMind uploaded their pioneering paper "Playing Atari with Deep Reinforcement Learning" to Arxiv. In this paper they demonstrated how a computer learned to play Atari 2600 video games by observing just the screen pixels and receiving a reward when the game score increased. The result was remarkable, because the games and the goals in every game were very different and designed to be challenging for humans. The same model architecture, without any change, was used to learn seven different games, and in three of them the algorithm performed even better than a human!

It has been hailed since then as the first step towards general artificial intelligence – an AI that can survive in a variety of environments, instead of being confined to strict realms such as playing chess. No wonder DeepMind was immediately bought by Google and has been on the forefront of deep learning research ever since. In February 2015 their paper "Human-level control through deep reinforcement learning" was featured on the cover of Nature, one of the most prestigious journals in science. In this paper they applied the same model to 49 different games and achieved superhuman performance in half of them.

Still, while deep models for supervised and unsupervised learning have seen widespread adoption in the community, deep reinforcement learning has remained a bit of a mystery. In this blog post I will be trying to demystify this technique and understand the rationale behind it. The intended audience is someone who already has background in machine learning and possibly in neural networks, but hasn't had time to delve into reinforcement learning yet.

The roadmap ahead:

1. **What are the main challenges in reinforcement learning?** We will cover the credit assignment problem and the exploration-exploitation dilemma here.
2. **How to formalize reinforcement learning in mathematical terms?** We will define Markov Decision Process and use it for reasoning about reinforcement learning.

3. **How do we form long-term strategies?** We define “discounted future reward”, that forms the main basis for the algorithms in the next sections.
4. **How can we estimate or approximate the future reward?** Simple table-based Q-learning algorithm is defined and explained here.
5. **What if our state space is too big?** Here we see how Q-table can be replaced with a (deep) neural network.
6. **What do we need to make it actually work?** Experience replay technique will be discussed here, that stabilizes the learning with neural networks.
7. **Are we done yet?** Finally we will consider some simple solutions to the exploration-exploitation problem.

Reinforcement Learning

Consider the game Breakout. In this game you control a paddle at the bottom of the screen and have to bounce the ball back to clear all the bricks in the upper half of the screen. Each time you hit a brick, it disappears and your score increases – you get a reward.

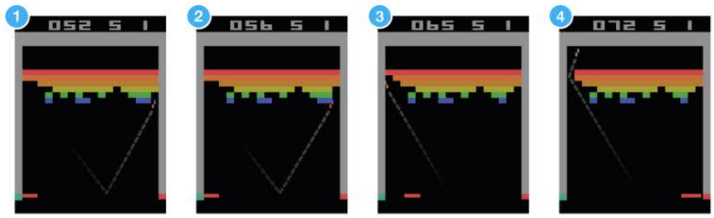

Figure 1: Atari Breakout game. Image credit: DeepMind.

Suppose you want to teach a neural network to play this game. Input to your network would be screen images, and output would be three actions: left, right or fire (to launch the ball). It would make sense to treat it as a classification problem – for each game screen you have to decide, whether you should move left, right or press fire. Sounds straightforward? Sure, but then you need training examples, and a lots of them. Of course you could go and record game sessions using expert players, but that’s not really how we learn. We don’t need somebody to tell us a million times which move to choose at each screen. We just need occasional feedback that we did the right thing and can then figure out everything else ourselves.

This is the task **reinforcement learning** tries to solve. Reinforcement learning lies somewhere in between supervised and unsupervised learning. Whereas in supervised learning one has a target label for each training example and in unsupervised learning one has no labels at all, in reinforcement learning one has sparse and time-delayed labels – the rewards. Based only on those rewards the agent has to learn to behave in the environment.

While the idea is quite intuitive, in practice there are numerous challenges. For example when you hit a brick and score a reward in the Breakout game, it often has nothing to do with the actions (paddle movements) you did just before getting the reward. All the hard work was already done, when you positioned the paddle correctly and bounced the ball back. This is called the **credit assignment problem** – i.e., which of the preceding actions was responsible for getting the reward and to what extent.

Once you have figured out a strategy to collect a certain number of rewards, should you stick with it or experiment with something that could result in even bigger rewards? In the above Breakout game a simple strategy is to move to the left edge and wait there. When launched, the ball tends to fly left more often than right and you will easily score about 10 points before you die. Will you be satisfied with this or do you want more? This is called the **explore-exploit dilemma** – should you exploit the known working strategy or explore other, possibly better strategies.

Reinforcement learning is an important model of how we (and all animals in general) learn. Praise from our parents, grades in school, salary at work – these are all examples of rewards. Credit assignment problems and exploration-exploitation dilemmas come up every day both in business and in relationships. That's why it is important to study this problem, and games form a wonderful sandbox for trying out new approaches.

Markov Decision Process

Now the question is, how do you formalize a reinforcement learning problem, so that you can

reason about it? The most common method is to represent it as a Markov decision process.

Suppose you are an **agent**, situated in an **environment** (e.g. Breakout game). The environment is in a certain **state** (e.g. location of the paddle, location and direction of the ball, existence of every brick and so on). The agent can perform certain **actions** in the environment (e.g. move the paddle to the left or to the right). These actions sometimes result in a **reward** (e.g. increase in score). Actions transform the environment and lead to a new state, where the agent can perform another action, and so on. The rules for how you choose those actions are called **policy**. The environment in general is stochastic, which means the next state may be somewhat random (e.g. when you lose a ball and launch a new one, it goes towards a random direction).

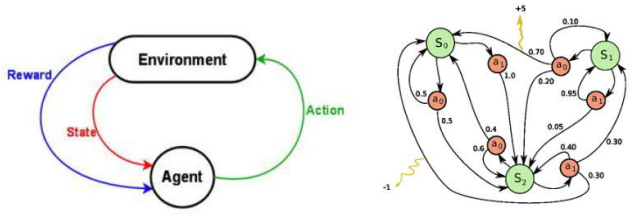

Figure 2: *Left:* reinforcement learning problem. *Right:*Markov decision process.

The set of states and actions, together with rules for transitioning from one state to another, make up a **Markov decision process**. One **episode** of this process (e.g. one game) forms a finite sequence of states, actions and rewards:

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, \ldots, s_{n-1}, a_{n-1}, r_n, s_n$$

Here s_i represents the state, a_i is the action and r_{i+1} is the reward after performing the action. The episode ends with **terminal** state s_n (e.g. "game over" screen). A Markov decision process relies on the Markov assumption, that the probability of the next state s_{i+1} depends only on current state $s_i$and action a_i, but not on preceding states or actions.

Discounted Future Reward

To perform well in the long-term, we need to take into account not only the immediate rewards, but also the future rewards we are going to get. How should we go about that?

Given one run of the Markov decision process, we can easily calculate the **total reward** for one episode:

$$R = r_1 + r_2 + r_3 + \ldots + r_n$$

Given that, the **total future reward** from time point *t* onward can be expressed as:

$$R_t = r_t + r_{t+1} + r_{t+2} + \ldots + r_n$$

But because our environment is stochastic, we can never be sure, if we will get the same rewards the next time we perform the same actions. The more into the future we go, the more it may diverge. For that reason it is common to use **discounted future reward** instead:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \ldots + \gamma^{n-t} r_n$$

Here γ is the discount factor between 0 and 1 – the more into the future the reward is, the less we take it into consideration. It is easy to see, that discounted future reward at time step *t* can be expressed in terms of the same thing at time step *t+1*:

$$R_t = r_t + \gamma(r_{t+1} + \gamma(r_{t+2} + \ldots)) = r_t + \gamma R_{t+1}$$

If we set the discount factor γ=0, then our strategy will be short-sighted and we rely only on the immediate rewards. If we want to balance between immediate and future rewards, we should set discount factor to something like γ=0.9. If our environment is deterministic and the same actions always result in same rewards, then we can set discount factor γ=1.

A good strategy for an agent would be to **always choose an action that maximizes the (discounted) future reward.**

Q-learning

In Q-learning we define a function *Q(s, a)* representing **the maximum discounted future reward when we perform action a in state s, and continue optimally from that point on.**

$$Q(s_t, a_t) = max\ R_{t+1}$$

The way to think about *Q(s, a)* is that it is "the best possible score at the end of the game after performing action a **in state s". It is called Q-function, because it represents the "quality" of a certain action in a given state.**

This may sound like quite a puzzling definition. How can we estimate the score at the end of game, if we know just the current state and action, and not the actions and rewards coming after that? We really can't. But as a theoretical construct we can assume existence of such a function. Just close your eyes and repeat to yourself five times: "*Q(s, a)* exists, *Q(s, a)* exists, …". Feel it?

If you're still not convinced, then consider what the implications of having such a function would be. Suppose you are in state and pondering whether you should take action *a* or *b*. You want to select the action that results in the highest score at the end of game. Once you have the magical Q-function, the answer becomes really simple – pick the action with the highest Q-value!

$$\pi(s) = argmax_a\ Q(s, a)$$

Here π represents the policy, the rule how we choose an action in each state.

OK, how do we get that Q-function then? Let's focus on just one transition <*s, a, r, s'*>. Just like with discounted future rewards in the previous section, we can express the Q-value of state *s* and action *a* in terms of the Q-value of the next state *s'*.

$$Q(s, a) = r + \gamma max_{a'} Q(s', a')$$

This is called the **Bellman equation**. If you think about it, it is quite logical – maximum future reward

for this state and action is the immediate reward plus maximum future reward for the next state.

The main idea in Q-learning is that **we can iteratively approximate the Q-function using the Bellman equation**. In the simplest case the Q-function is implemented as a table, with states as rows and actions as columns. The gist of the Q-learning algorithm is as simple as the following[1]:

```
initialize Q[num_states,num_actions] arbitrarily
observe initial state s
repeat
      select and carry out an action a
      observe reward r and new state s'
      Q[s,a] = Q[s,a] + α(r + γ max_a' Q[s',a'] - Q[s,a])
      s = s'
until terminated
```

α in the algorithm is a learning rate that controls how much of the difference between previous Q-value and newly proposed Q-value is taken into account. In particular, when α=1, then two *Q[s,a]* cancel and the update is exactly the same as the Bellman equation.

The $\max_{a'} Q[s',a']$ that we use to update $Q[s,a]$ is only an approximation and in early stages of learning it may be completely wrong. However the approximation get more and more accurate with every iteration and it has been shown, that if we perform this update enough times, then the Q-function will converge and represent the true Q-value.

Deep Q Network

The state of the environment in the Breakout game can be defined by the location of the paddle, location and direction of the ball and the presence or absence of each individual brick. This intuitive representation however is game specific. Could we come up with something more universal, that would be suitable for all the games? The obvious choice is screen pixels – they implicitly contain all of the relevant information about the game situation, except for the speed and direction of the ball. Two consecutive screens would have these covered as well.

If we apply the same preprocessing to game screens as in the DeepMind paper – take the four

last screen images, resize them to 84×84 and convert to grayscale with 256 gray levels – we would have $256^{84x84x4} \approx 10^{67970}$ possible game states. This means 10^{67970} rows in our imaginary Q-table – more than the number of atoms in the known universe! One could argue that many pixel combinations (and therefore states) never occur – we could possibly represent it as a sparse table containing only visited states. Even so, most of the states are very rarely visited and it would take a lifetime of the universe for the Q-table to converge. Ideally, we would also like to have a good guess for Q-values for states we have never seen before.

This is the point where deep learning steps in. Neural networks are exceptionally good at coming up with good features for highly structured data. We could represent our Q-function with a neural network, that takes the state (four game screens) and action as input and outputs the corresponding Q-value. Alternatively we could take only game screens as input and output the Q-value for each possible action. This approach has the advantage, that if we want to perform a Q-value update or pick the action with the highest Q-value, we only have to do one forward pass through the network and have all Q-values for all actions available immediately.

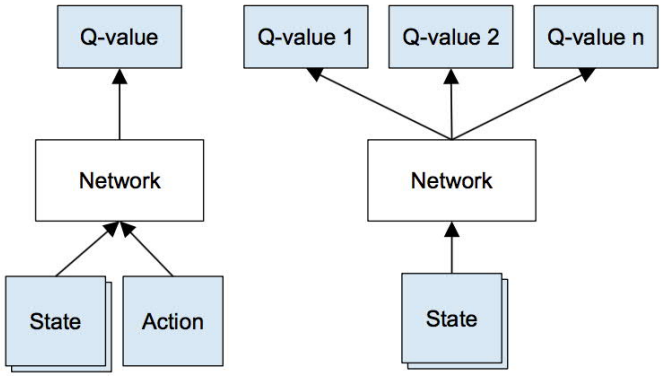

Figure 3: *Left:* Naive formulation of deep Q-network. *Right:*More optimized architecture of deep Q-network, used in DeepMind paper.

The network architecture that DeepMind used is as follows:

Layer	Input	Filter size	Stride	Num filters	Activation	Output
conv1	84x84x4	8x8	4	32	ReLU	20x20x32
conv2	20x20x32	4x4	2	64	ReLU	9x9x64
conv3	9x9x64	3x3	1	64	ReLU	7x7x64
fc4	7x7x64			512	ReLU	512
fc5	512			18	Linear	18

This is a classical convolutional neural network with three convolutional layers, followed by two fully connected layers. People familiar with object recognition networks may notice that there are no pooling layers. But if you really think about it, pooling layers buy you translation invariance – the network becomes insensitive to the location of an object in the image. That makes perfectly sense for a classification task like ImageNet, but for games the location of the ball is crucial in determining the potential reward and we wouldn't want to discard this information!

Input to the network are four 84×84 grayscale game screens. Outputs of the network are Q-values for each possible action (18 in Atari). Q-values can be any real values, which makes it a regression task, that can be optimized with simple squared error loss.

$$L = \frac{1}{2}[\underbrace{r + max_{a'}Q(s', a')}_{\text{target}} - \underbrace{Q(s, a)}_{\text{prediction}}]^2$$

Given a transition < *s, a, r, s'* >, the Q-table update rule in the previous algorithm must be replaced with the following:

1. Do a feedforward pass for the current state *s* to get predicted Q-values for all actions.
2. Do a feedforward pass for the next state *s'* and calculate maximum overall network outputs $max_{a'} Q(s', a')$.
3. Set Q-value target for action to $r + \gamma max_{a'} Q(s', a')$ (use the max calculated in step 2). For all other actions, set the Q-value target to the same as originally returned from step 1, making the error 0 for those outputs.
4. Update the weights using backpropagation.

Experience Replay

By now we have an idea how to estimate the future reward in each state using Q-learning and approximate the Q-function using a convolutional neural network. But it turns out that approximation of Q-values using non-linear functions is not very stable. There is a whole bag of tricks that you have to use to actually make it converge. And it takes a long time, almost a week on a single GPU.

The most important trick is **experience replay**. During gameplay all the experiences $< s, a, r, s' >$ are stored in a replay memory. When training the network, random minibatches from the replay memory are used instead of the most recent transition. This breaks the similarity of subsequent training samples, which otherwise might drive the network into a local minimum. Also experience replay makes the training task more similar to usual supervised learning, which simplifies debugging and testing the algorithm. One could actually collect all those experiences from human gameplay and then train network on these.

Exploration-Exploitation

Q-learning attempts to solve the credit assignment problem – it propagates rewards back in time, until it reaches the crucial decision point which was the actual cause for the obtained reward. But we haven't touched the exploration-exploitation dilemma yet…

Firstly observe, that when a Q-table or Q-network is initialized randomly, then its predictions are initially random as well. If we pick an action with the highest Q-value, the action will be random and the agent performs crude "exploration". As a Q-function converges, it returns more consistent Q-values and the amount of exploration decreases. So one could say, that Q-learning incorporates the exploration as part of the algorithm. But this exploration is "greedy", it settles with the first effective strategy it finds.

A simple and effective fix for the above problem is ***ε*-greedy exploration** – with probability ε choose a random action, otherwise go with the "greedy"

action with the highest Q-value. In their system DeepMind actually decreases ε over time from 1 to 0.1 – in the beginning the system makes completely random moves to explore the state space maximally, and then it settles down to a fixed exploration rate.

Deep Q-learning Algorithm

This gives us the final deep Q-learning algorithm with experience replay:

```
initialize replay memory D
initialize action-value function Q with random weights
observe initial state s
repeat
      select an action a
            with probability ε select a random action
            otherwise select a = argmax_a' Q(s,a')
      carry out action a
      observe reward r and new state s'
      store experience <s, a, r, s'> in replay memory D

      sample random transitions <ss, aa, rr, ss'> from replay memory D
      calculate target for each minibatch transition
            if ss' is terminal state then tt = rr
            otherwise tt = rr + γmax_a' Q(ss', aa')
      train the Q network using (tt - Q(ss, aa))^2 as loss

      s = s'
until terminated
```

There are many more tricks that DeepMind used to actually make it work – like target network, error clipping, reward clipping etc, but these are out of scope for this introduction.

The most amazing part of this algorithm is that it learns anything at all. Just think about it – because our Q-function is initialized randomly, it initially outputs complete garbage. And we are using this garbage (the maximum Q-value of the next state) as targets for the network, only occasionally folding in a tiny reward. That sounds insane, how could it learn anything meaningful at all? The fact is, that it does.

Final notes

Many improvements to deep Q-learning have been proposed since its first introduction – Double Q-learning, Prioritized Experience Replay, Dueling Network Architecture and extension to continuous action space to name a few. For latest