Deep RL is relevant even if you're not into gaming. Just check out the sheer variety of functions currently using Deep RL for research:

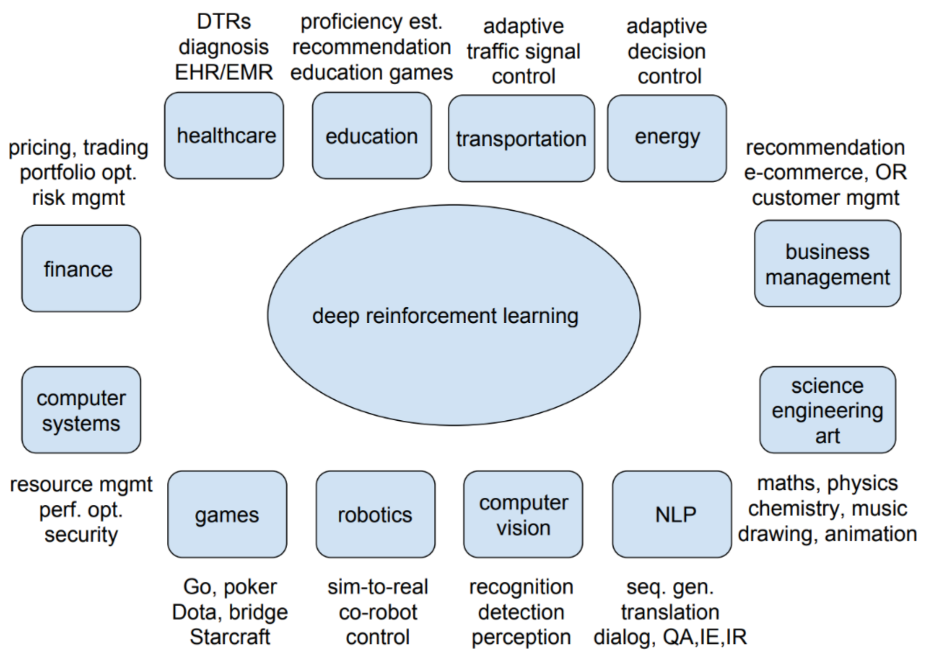


Figure 4: Deep Reinforcement Learning Applications

What about industry-ready applications? Well, here are two of the most commonly cited Deep RL use cases:

- Google's Cloud AutoML
- Facebook's Horizon Platform

The scope of Deep RL is IMMENSE. This is a great time to enter into this field and make a career out of it.

In this article, I aim to help you take your first steps into the world of deep reinforcement learning. We'll use one of the most popular algorithms in RL, deep Q-learning, to understand how deep RL works. And the icing on the cake? We will implement all our learning in an awesome case study using Python.

Table of Contents

5. Implementing Deep Q-Learning in Python using Keras & Gym

The Road to Q-Learning

There are certain concepts you should be aware of before wading into the depths of deep reinforcement learning. Don't worry, I've got you covered.

I have previously written various articles on the nuts and bolts of reinforcement learning to introduce concepts like multi-armed bandit, dynamic programming, Monte Carlo learning and temporal differencing. I recommend going through these guides in the below sequence:

- **Nuts & Bolts of Reinforcement Learning: Model-Based Planning using Dynamic Programming**
- **Reinforcement Learning Guide: Solving the Multi-Armed Bandit Problem from Scratch in Python**
- **Reinforcement Learning: Introduction to Monte Carlo Learning using the OpenAI Gym Toolkit**
- **Introduction to Monte Carlo Tree Search: The Game-Changing Algorithm behind DeepMind's AlphaGo**
- **Nuts and Bolts of Reinforcement Learning: Introduction to Temporal Difference (TD) Learning**

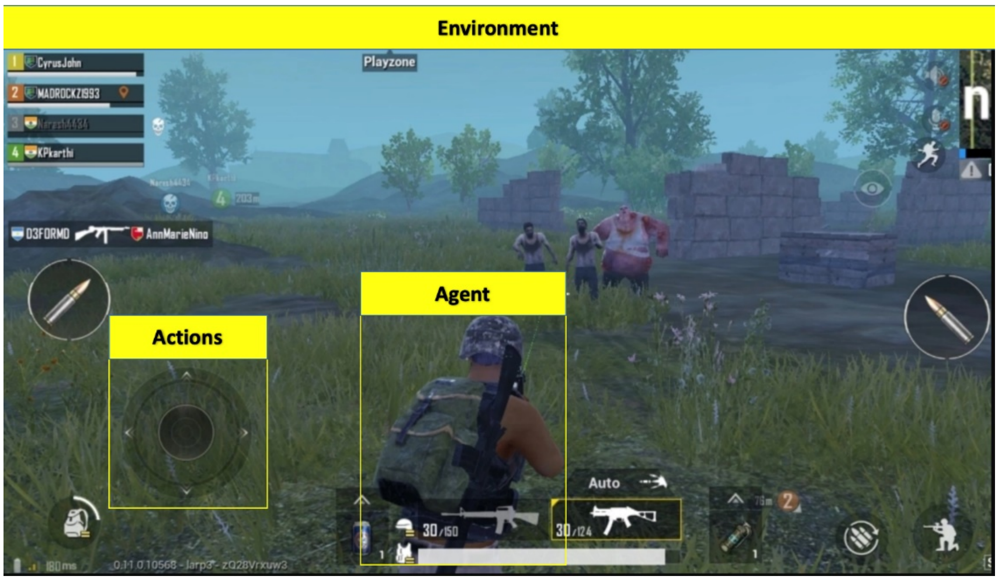
These articles are good enough for getting a detailed overview of basic RL from the beginning.

However, note that the articles linked above are in no way prerequisites for the reader to understand Deep Q-Learning. We will do a quick recap of the basic RL concepts before exploring what is deep Q-Learning and its implementation details.

RL Agent-Environment

A reinforcement learning task is about training an **agent** which interacts with its **environment**. The agent arrives at different scenarios known as **states** by performing **actions**. Actions lead to rewards which could be positive and negative.

The agent has only one purpose here – to maximize its total reward across an **episode.** This episode is anything and everything that happens between the first state and the last or terminal state within the environment. We reinforce the agent to learn to perform the best actions by experience. This is the strategy or **policy**.



Let’s take an example of the ultra-popular PubG game:

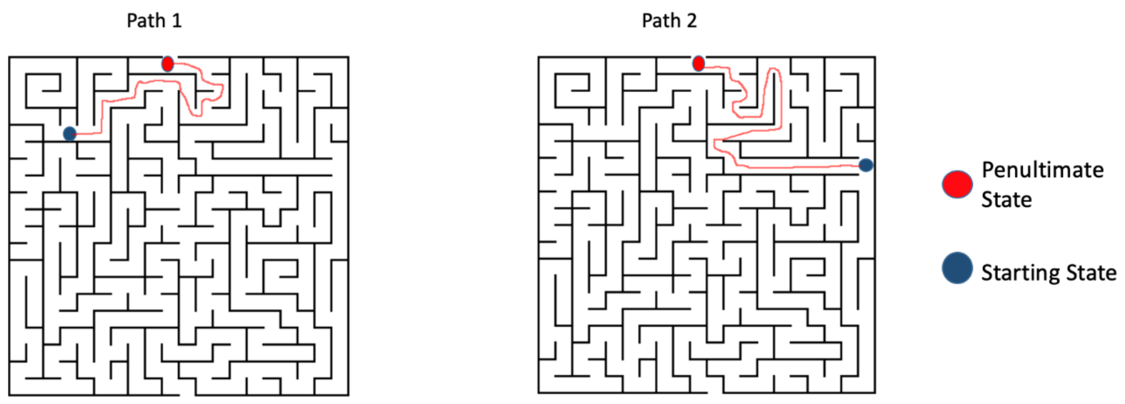
- The soldier is the agent here interacting with the environment
- The states are exactly what we see on the screen
- An episode is a complete game
- The actions are moving forward, backward, left, right, jump, duck, shoot, etc.
- Rewards are defined on the basis of the outcome of these actions. If the soldier is able to kill an enemy, that calls for a positive reward while getting shot by an enemy is a negative reward

Now, in order to kill that enemy or get a positive reward, there is a sequence of actions required. This is where the concept of delayed or postponed reward comes into play. The crux of RL is learning to perform these sequences and maximizing the reward.

Markov Decision Process (MDP)

An important point to note – each state within an environment is a consequence of its previous state which in turn is a result of its previous state. However, storing all this information, even for environments with short episodes, will become readily infeasible.

To resolve this, we assume that each state follows a Markov property, i.e., each state depends solely on the previous state and the transition from that state to the current state. Check out the below maze to better understand the intuition behind how this works:



Now, there are 2 scenarios with 2 different starting points and the agent traverses different paths to reach the same penultimate state. Now it doesn't matter what path the agent takes to reach the red state. The next step to exit the maze and reach the last state is by going right. Clearly, we only needed the information on the red/penultimate state to find out the next best action which is exactly what the Markov property implies.

Q Learning

Let's say we know the expected reward of each action at every step. This would essentially be like a cheat sheet for the agent! Our agent will know exactly which action to perform.

It will perform the sequence of actions that will eventually generate the maximum total reward. This total reward is also called the Q-value and we will formalise our strategy as:

$$Q(s,a) = r(s,a) + \gamma \max_a Q(s',a)$$

The above equation states that the Q-value yielded from being at state *s* and performing action *a* is the immediate reward r(s,a) plus the highest Q-value possible from the next state *s'*. Gamma here is the discount factor which controls the contribution of rewards further in the future.

Q(s',a) again depends on Q(s",a) which will then have a coefficient of gamma squared. So, the Q-value depends on Q-values of future states as shown here:

$$Q(s,a) \rightarrow \gamma Q(s',a) + \gamma^2 Q(s'',a) \ldots\ldots\ldots \gamma^n Q(s''^{\ldots n},a)$$

Adjusting the value of gamma will diminish or increase the contribution of future rewards.

Since this is a recursive equation, we can start with making arbitrary assumptions for all q-values. With experience, it will converge to the optimal policy. In practical situations, this is implemented as an update:

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1},a) - Q(S_t,A_t)\Big]$$

where alpha is the learning rate or step size. This simply determines to what extent newly acquired information overrides old information.

Why ‘Deep’ Q-Learning?

Q-learning is a simple yet quite powerful algorithm to create a cheat sheet for our agent. This helps the agent figure out exactly which action to perform.

But what if this cheatsheet is too long? Imagine an environment with 10,000 states and 1,000 actions per state. This would create a table of 10 million cells. Things will quickly get out of control!

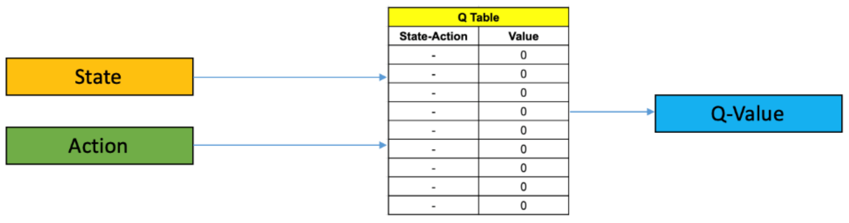
It is pretty clear that we can’t infer the Q-value of new states from already explored states. This presents two problems:

- First, the amount of memory required to save and update that table would increase as the number of states increases
- Second, the amount of time required to explore each state to create the required Q-table would be unrealistic

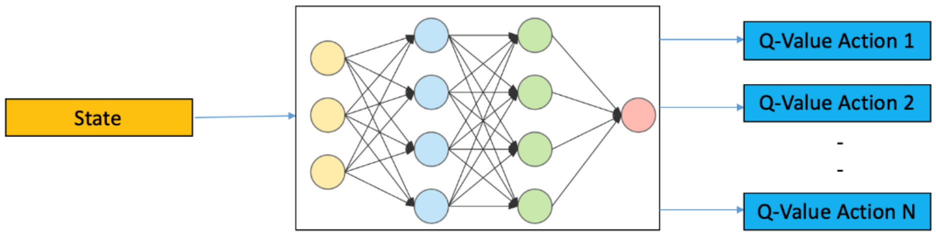
Here’s a thought – what if we approximate these Q-values with machine learning models such as a neural network? Well, this was the idea behind DeepMind’s algorithm that led to its acquisition by Google for 500 million dollars!

Deep Q-Networks

In deep Q-learning, we use a neural network to approximate the Q-value function. The state is given as the input and the Q-value of all possible actions is generated as the output. The comparison between Q-learning & deep Q-learning is wonderfully illustrated below:



Q Learning



Deep Q Learning

So, what are the steps involved in reinforcement learning using deep Q-learning networks (DQNs)?

1. All the past experience is stored by the user in memory

2. The next action is determined by the maximum output of the Q-network
3. The loss function here is mean squared error of the predicted Q-value and the target Q-value – Q*. This is basically a regression problem. However, we do not know the target or actual value here as we are dealing with a reinforcement learning problem. Going back to the Q-value update equation derived fromthe Bellman equation. we have:

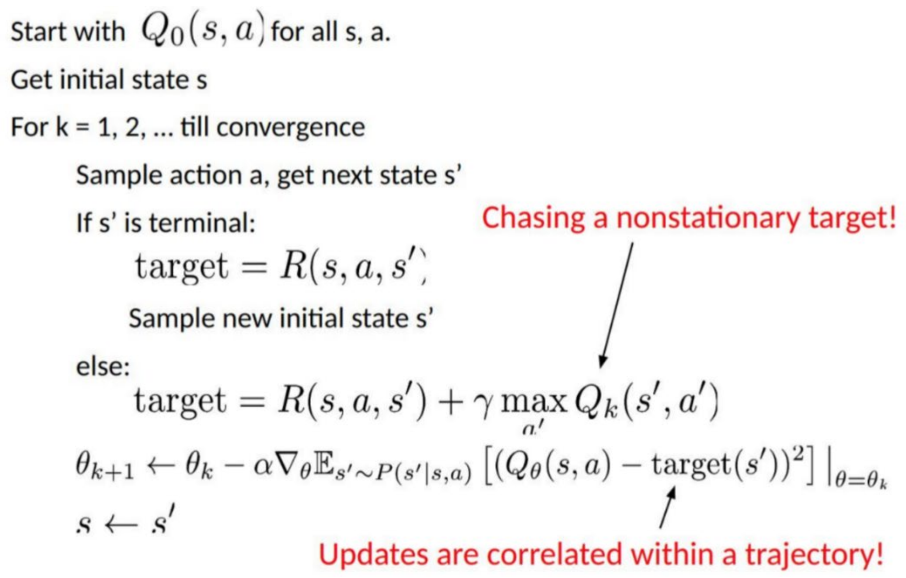
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[\boxed{R_{t+1} + \gamma \max_a Q(S_{t+1}, a)} - Q(S_t, A_t)\right]$$

The section in green represents the target. We can argue that it is predicting its own value, but since R is the unbiased true reward, the network is going to update its gradient using backpropagation to finally converge.

Challenges in Deep RL as Compared to Deep Learning

So far, this all looks great. We understood how neural networks can help the agent learn the best actions. However, there is a challenge when we compare deep RL to deep learning (DL):

- **Non-stationary or unstable target:** Let us go back to the pseudocode for deep Q-learning:



```
Start with $Q_0(s, a)$ for all s, a.
Get initial state s
For k = 1, 2, … till convergence
    Sample action a, get next state s'
    If s' is terminal:
        $\text{target} = R(s, a, s')$
        Sample new initial state s'
    else:
        $\text{target} = R(s, a, s') + \gamma \max_{a'} Q_k(s', a')$
    $\theta_{k+1} \leftarrow \theta_k - \alpha \nabla_\theta \mathbb{E}_{s' \sim P(s'|s,a)} \left[(Q_\theta(s, a) - \text{target}(s'))^2\right] \big|_{\theta=\theta_k}$
    $s \leftarrow s'$
```

Chasing a nonstationary target!
Updates are correlated within a trajectory!

As you can see in the above code, the target is continuously changing with each iteration. In deep learning, the target variable does not change and hence the training is stable, which is just not true for RL.

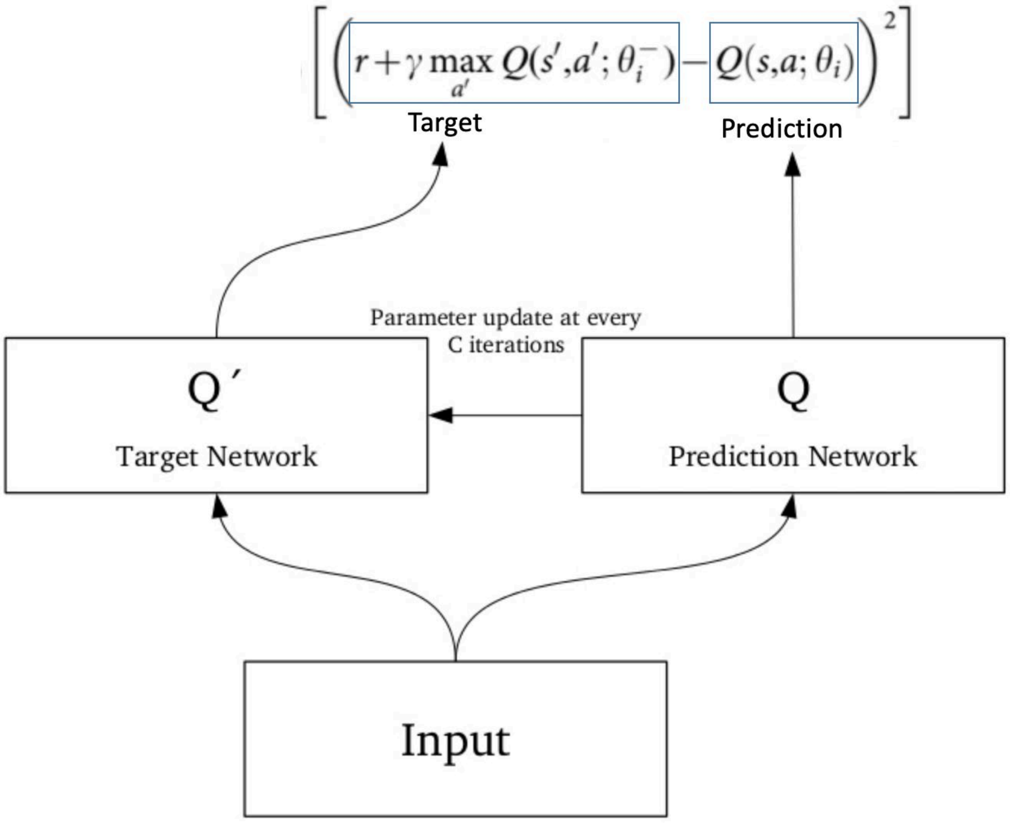
To summarise, we often depend on the policy or value functions in reinforcement learning to sample actions. However, this is frequently changing as we continuously learn what to explore. As we play out the game, we get to know more about the ground truth values of states and actions and hence, the output is also changing.

So, we try to learn to map for a constantly changing input and output. But then what is the solution?

1. Target Network

Since the same network is calculating the predicted value and the target value, there could be a lot of divergence between these two. So, instead of using 1one neural network for learning, we can use two.

We could use a separate network to estimate the target. This target network has the same architecture as the function approximator but with frozen parameters. For every C iterations (a hyperparameter), the parameters from the prediction network are copied to the target network. This leads to more stable training because it keeps the target function fixed (for a while):


$\left[\left(\boxed{r+\gamma \max_{a'} Q(s',a';\theta_i^-)}-\boxed{Q(s,a;\theta_i)}\right)^2\right]$
Target
Prediction
Parameter update at every C iterations
Q′
Target Network
Q
Prediction Network
Input

2. Experience Replay

To perform experience replay, we store the agent's experiences – et=(st,at,rt,st+1)

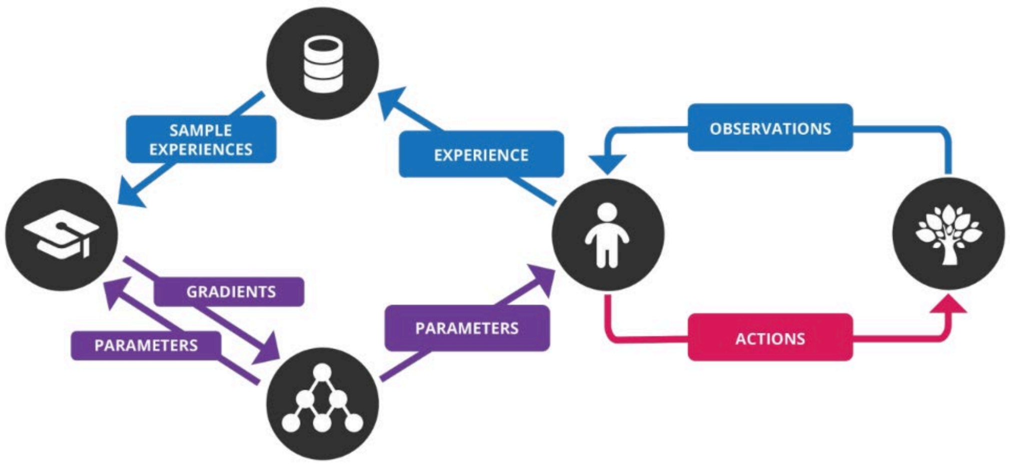
What does the above statement mean? Instead of running Q-learning on state/action pairs as they occur during simulation or the actual experience, the system stores the data discovered for [state, action, reward, next_state] – in a large table.

Let's understand this using an example.

Suppose we are trying to build a video game bot where each frame of the game represents a different state. During training, we could sample a random batch of 64 frames from the last 100,000 frames to train our network. This would get us a subset within which the correlation amongst the samples is low and will also provide better sampling efficiency.

Putting it all Together

The concepts we have learned so far? They all combine to make the deep Q-learning algorithm that was used to achive human-level level performance in Atari games (using just the video frames of the game).



I have listed the steps involved in a deep Q-network (DQN) below:

1. Preprocess and feed the game screen (state s) to our DQN, which will return the Q-values of all possible actions in the state
2. Select an action using the epsilon-greedy policy. With the probability epsilon, we select a random action *a* and with probability 1-epsilon, we select an action that has a maximum Q-value, such as a = argmax(Q(s,a,w))
3. Perform this action in a state *s* and move to a new state *s'* to receive a reward. This state s' is the preprocessed image of the next game screen. We store this transition in our replay buffer as <s,a,r,s'>
4. Next, sample some random batches of transitions from the replay buffer and calculate the loss
5. It is known that: $Loss = (r + \gamma max_{a'} Q(s', a'; \theta') - Q(s, a; \theta))^2$ which is just the squared difference between target Q and predicted Q
6. Perform gradient descent with respect to our actual network parameters in order to minimize this loss
7. After every C iterations, copy our actual network weights to the target network weights
8. Repeat these steps for *M* number of episodes