A summary paper on the most powerful and popular AI models that exist today.

ARTIFICIAL INTELLIGENCE A-Z

Learn How To Build An A.I

BY: Hadelin de Ponteves
and Kirill Eremenko

SUPER DATASCIENCE
MAKING THE COMPLEX SIMPLE

Table of contents

1 Introduction

This handbook is an additional resource for the Artificial Intelligence A-Z online course that contains all the theory used to implement the models. It will be useful for anyone who wants to have all the course maths gathered in one structured handbook and also those who want to go deeper in the mathematical details lying behind the AI equations. It assumes prior knowledge in neural networks.

2 Markov Decision Processes

To begin our journey of AI, we must start with Markov Decision Processes.

A Markov Decision Process is a tuple (S, A, T, R) where:

- S is the set of the different states. A state can be a vector of encoded values or a 3D input image. Let's take the Breakout game for example. What would be a state in that case ? If the state has the form of a vector of encoded values then this vector would contain all the informations that can describe what is going on at time t, that is: the coordinates of the position of the ball, the coordinates of the vector of direction where the ball is going, and maybe some binary values for each of the bricks being 1 if the brick is still there and 0 if not. All these values in a vector could encode a state of the environment. This vector is then going to be the input of a neural network where the output is going to be the action to play. And if the state was a 3D input image, then the input would be exactly a screenshot of the game screen at each time t. We will study that case in the section on Deep Convolutional Q-Learning.
- A is the set of the different actions that can be played at each time t. For Breakout, there are six possible actions: three levels of intensity to go to the left, and three levels of intensity to go to the right. For Doom, there are many more actions: move forward, move backward, left, right, run, shoot...
- T is called the transition rule:

  $$T : (a_t \in A, s_t \in S, s_{t+1} \in S) \mapsto \mathbb{P}(s_{t+1}|s_t, a_t)$$

  where $\mathbb{P}(s_{t+1}|s_t, a_t)$ is the probability that the future state is s_{t+1} given that the current state is s_t and the action played is a_t. Therefore T is the probability distribution of the future states at time $t + 1$ given the current state and the action played at time t. Accordingly, we can predict the future state s_{t+1} by taking of random draw from that distribution T:

  $$s_{t+1} \sim T(a_t, s_t, .)$$

- R is the reward function:

  $$R : (a_t \in A, s_t \in S) \mapsto r_t \in \mathbb{R}$$

  where r_t is the reward obtained after playing the action a_t in the state s_t.

After defining the MDP, it is important to remind that it relies on the following assumption: the probability of the future state s_{t+1} only depends on the current state s_t and action a_t, and doesn't depend on any of the previous states and actions. That is:

$$\mathbb{P}(s_{t+1}|s_0, a_0, s_1, a_1, ..., s_t, a_t) = \mathbb{P}(s_{t+1}|s_t, a_t)$$

Now let's recap what is going on in terms of MDPs. At every time t:

1. The AI observes the current state s_t.
2. The AI plays the action a_t.
3. The AI receives reward $r_t = R(a_t, s_t)$.
4. The AI enters the following state s_{t+1}.

Now the question is:

How does the AI know which action to play at each time t ?

To answer this question, we need to introduce the policy function. The policy function π is exactly the function that, given a state s_t, returns the action a_t:

$$\pi : s_t \in S \mapsto a_t \in A$$

Let's denote by Π the set of all the policy functions. Then the choice of the best actions to play becomes an optimization problem. Indeed, it comes down to finding the optimal policy π^* that maximizes the accumulated reward:

$$\pi^* = \underset{\pi \in \Pi}{\operatorname{argmax}} \sum_{t \geq 0} R(\pi(s_t), s_t)$$

So now of course the question becomes:

How to find this optimal policy π^* ?

That is where Q-Learning comes into play. But before getting to it, we need to understand the concept of future reward.

3 Future Cumulative Reward

Let's consider one episode of the game, that is from $t = 0$ (Game Start) to $t = n$ (Game Over). When playing games, most of the time the goal is to reach the highest score. Therefore to do this, we should not consider the reward only at time t, but the future cumulative reward until the end of the game (from time t to time n). Therefore, we need to consider, not $r_t = R(a_t, s_t)$, but instead:

$$R_t = R(a_t, s_t) + R(a_{t+1}, s_{t+1}) + ... + R(a_n, s_n) = r_t + r_{t+1} + ... + r_n$$

However we can still improve the model. The elements r_t, r_{t+1}, ..., and r_n are values we are trying to estimate with the reward function R. Therefore when we are at time t, the further we look into the future the less certain we are about the reward at this future time. In other words, the larger is t', the larger is the variance of the estimated reward $r_{t+t'}$. Hence, to fix that effect, we need to discount each of the future single rewards, and the further we are in the future, the more we discount it because the less certain we are about it. A way to do this is to take the discounted some of rewards:

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

where $\gamma \in [0, 1]$. That way the higher is t', the smaller is $\gamma^{t'}$, and therefore the more $r_{t+t'}$ is discounted.

γ is called the discount factor. The closer γ is to 0, the more the AI will try to optimize the current reward r_t. The closer γ is to 1, the more the AI will aim to optimize the future reward. This discount factor is what we will keep for our Q-Learning algorithm.

4 Q-Learning

4.1 The Q-value

To each couple of action and state (a, s), we are going to associate a numeric value Q(a,s):

$$Q : (a_t \in A, s_t \in S) \mapsto Q(a_t, s_t) \in \mathbb{R}$$

We will say that $Q(a, s)$ is "the Q-value of the action a played in the state s".

To understand this purpose, we need to introduce the temporal difference.

4.2 The Temporal Difference

At the beginning $t = 0$, all the Q-values are initialized to 0.

Now let's suppose we are at time t, in a certain state s_t. We play the action a_t and we get the reward r_t.

Then we take a random draw from the $T(a_t, s_t, .)$ distribution, which leads us to the next state s_{t+1}:

$$s_{t+1} \sim T(a_t, s_t, .)$$

We can now introduce the temporal difference, which is at the heart of Q-Learning. The temporal difference at time t, denoted by $TD_t(a_t, s_t)$, is the difference between:

- $r_t + \gamma \max_a(Q(a, s_{t+1}))$, $\gamma \in [0, 1]$, that is the reward r_t obtained by playing the action a_t in the state s_t, plus a percentage (which is our previous discount factor γ) of the Q-value of the best action played in the future state s_{t+1},
- and $Q(a_t, s_t)$, that is the Q-value of the action a_t played in the state s_t,

thus leading to:

$$TD_t(a_t, s_t) = r_t + \gamma \max_a(Q(a, s_{t+1})) - Q(a_t, s_t)$$

But what exactly is the purpose of this temporal difference $TD_t(a_t, s_t)$?

$TD_t(a_t, s_t)$ is like an intrinsic reward. The AI will learn the Q-values in such a way that:

- If $TD_t(a_t, s_t)$ is high, the AI gets a "good surprise".
- If $TD_t(a_t, s_t)$ is small, the AI gets a "frustration".

Then, in the final next step of the Q-Learning algorithm, we use the temporal difference to reinforce the couples (action, state) from time $t - 1$ to time t, according to the following equation:

$$Q_t(a_t, s_t) = Q_{t-1}(a_t, s_t) + \alpha TD_t(a_t, s_t)$$

With this point of view, the Q-values measure the cumulation of surprise or frustration associated with the couple of action and state (a_t, s_t). In the surprise case, the AI is reinforced, and in the frustration case, the AI is weakened. Hence we want to learn the Q-values that will give the AI the maximum "good surprise".

Accordingly, the decision of which action to play mostly depends on the Q-value $Q(a_t, s_t)$. If the action a_t played in the state s_t is associated with a high Q-value $Q(a, s_t)$, the AI will have a higher tendency to choose a. On the other hand if the action a played in the state s is associated with a small Q-value $Q(a_t, s_t)$, the AI will have a smaller tendency to choose a_t.

There are several ways of choosing the best action to play. First, when being in a certain state s_t, we could simply take a with which we have the maximum of $Q(a, s_t)$:

$$a = \underset{a}{\text{argmax}}(Q(a, s))$$

But experience has shown that this is not the best option. A better solution is the softmax method.

The softmax method consists of considering for each state s the following distribution:

$$W_s : a \in A \mapsto \frac{\exp(Q(s, a))^\tau}{\sum_{a'} \exp(Q(s, a'))^\tau} \text{ with } \tau \geq 0$$

Then we choose which action a to play by taking a random draw from that distribution:

$$a \sim W_s(.)$$

4.3 The whole Q-Learning process

Let's summarize the different steps of the whole Q-Learning process:

Initialization:

For all couples of actions a and states s, the Q-values are initialized to 0:

$$\forall a \in A, s \in S, Q_0(a, s) = 0$$

We start in the initial state s_0. We play a random action and we reach the first state s_1.

At each time $t \geq 1$:

1. We play the action a_t, where a_t is a random draw from the W_s distribution:

$$a_t \sim W_{s_t}(.) = \frac{\exp(Q(s_t, .))^\tau}{\sum_{a'} \exp(Q(s_t, a'))^\tau}, \text{ with } \tau \geq 0$$

2. We get the reward $r_t = R(a_t, s_t)$
3. We get into the next state s_{t+1}, where s_{t+1} is a random draw from the $T(a_t, s_t, .)$ distribution:

$$s_{t+1} \sim T(a_t, s_t, .)$$

4. We compute the temporal difference:

$$TD_t(a_t, s_t) = r_t + \gamma \underset{a}{\max}(Q(a, s_{t+1})) - Q(a_t, s_t)$$

5. We update the Q-value:

$$Q_t(a_t, s_t) = Q_{t-1}(a_t, s_t) + \alpha TD_t(a_t, s_t)$$

5 Deep Q-Learning

5.1 Q-Learning into Deep Learning

Deep Q-Learning consists of combining Q-Learning to an Artificial Neural Network. Inputs are encoded vectors, each one defining a state of the environment. These inputs go to an Artificial Neural Network, where the output is the action to play. More precisely, let's say the game has n possible actions, the output layer of the neural network is comprised of n output neurons, each one corresponding to the Q-values of each action played in the current state. Then the action played is the one associated with the output neuron that has the highest Q-value (argmax), or the one returned by the softmax method. And since Q-values are real numbers, that makes our neural network an ANN for Regression.

Hence, in each state s_t of the game:

- the prediction is the Q-value $Q(a_t, s_t)$ where a is chosen by argmax or softmax
- the target is $r_t + \gamma \max_a(Q(a, s_{t+1}))$
- the loss error is the squared of the temporal difference:

$$\text{Loss} = \frac{1}{2}\left(r_t + \gamma \max_a(Q(a, s_{t+1})) - Q(a_t, s_t)\right)^2 = \frac{1}{2}TD_t(a_t, s_t)^2$$

Then this loss error is backpropagated into the network, and the weights are updated according to how much they contributed to the error.

5.2 Experience Replay

We notice that so far we have only considered transitions from one state s_t to the next state s_{t+1}. The problem with this is that s_t is most of the time very correlated with s_{t+1}. Therefore the network is not learning much. This could be way improved if, instead of considering only this one previous transition, we considered the last m transitions where m is a large number. This pack of the last m transitions is what is called the Experience Replay. Then from this Experience Replay we take some random batches of transitions to make our updates.

5.3 The whole Deep Q-Learning Process

Let's summarize the different steps of the whole Deep Q-Learning process:

Initialization:

For all couples of actions a and states s, the Q-values are initialized to 0:

$$\forall a \in A, s \in S, Q_0(a, s) = 0$$

The Experience Replay is initialized to an empty list M.

We start in the initial state s_0. We play a random action and we reach the first state s_1.

At each time $t \geq 1$:

1. We play the action a_t, where a_t is a random draw from the W_s distribution:

$$a_t \sim W_{s_t}(.) = \frac{\exp(Q(s_t, .))^\tau}{\sum_{a'} \exp(Q(s_t, a'))^\tau}, \text{ with } \tau \geq 0$$

2. We get the reward $r_t = R(a_t, s_t)$

3. We get into the next state s_{t+1}, where s_{t+1} is a random draw from the $T(a_t, s_t, .)$ distribution:

$$s_{t+1} \sim T(a_t, s_t, .)$$

4. We append the transition (s_t, a_t, r_t, s_{t+1}) in M.

5. We take a random batch $B \subset M$ of transitions. For each transition $(s_{t_B}, a_{t_B}, r_{t_B}, s_{t_B+1})$ of the random batch B:

   - We get the prediction:

$$Q(s_{t_B}, a_{t_B})$$

   - We get the target:

$$r_{t_B} + \gamma \max_a(Q(a, s_{t_B+1}))$$

   - We get the loss:

$$\text{Loss} = \frac{1}{2}\left(r_t + \gamma \max_a(Q(a, s_{t+1})) - Q(a_t, s_t)\right)^2 = \frac{1}{2} TD_t(a_t, s_t)^2$$

   - We backpropagate this loss error and update the weights according to how much they contributed to the error.

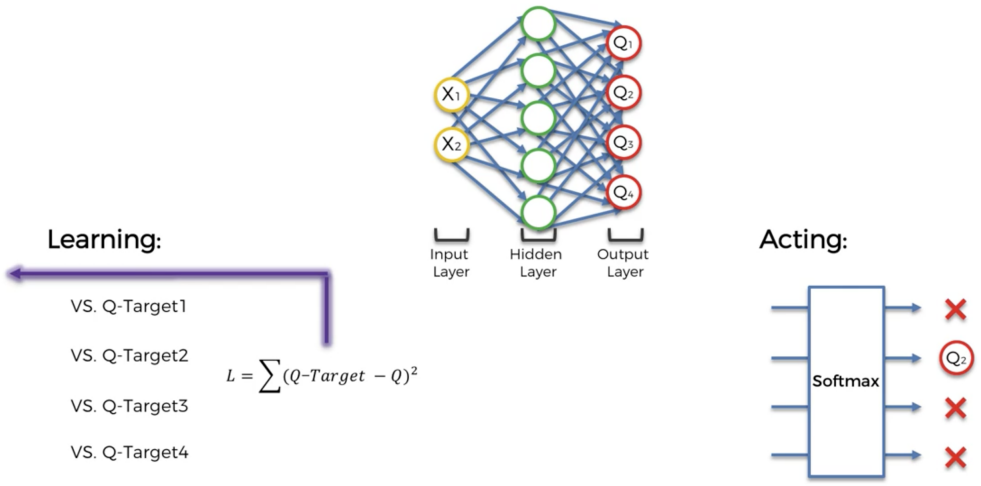

Figure 1: Deep Q-Learning

6 Deep Convolutional Q-Learning

In the previous section, our inputs were vectors encoded values defining the states of the environment. But since an encoded vector doesn't preserve the spatial structure of an image, this is not the best form to describe a state. The spatial structure is indeed important because it gives us more information to predict the next state, and predicting the next state is of course essential for our AI to know what is the right next move. Therefore we need to preserve the spatial structure and to do that, our inputs must be 3D images (2D for the array of pixels plus one additional dimension for the colors). In that case, the inputs are simply the images of the screen itself, exactly like what a human sees when playing the game. Following this analogy, the AI acts like a human: it observes the input images of the screen when playing the game, the input images go into a convolutional neural network (the brain for a human) which will detect the state in each image. However, this convolutional neural network doesn't contain pooling layers, because they would loose the location of the objects inside the image, and of course the AI need to keep track of the objects. Therefore we only keep the convolutional layers, and then by flattening them into a 1-dimensional vector, we get the input of our previous Deep Q-Learning network. Then the same process is being ran.

Therefore in summary, Deep Convolutional Q-Learning is the same as Deep Q-Learning, with the only difference that the inputs are now images, and a Convolutional Neural Network is added at the beginning of the fully-connected Deep Q-Learning network to detect the states (or simply the objects) of the images.

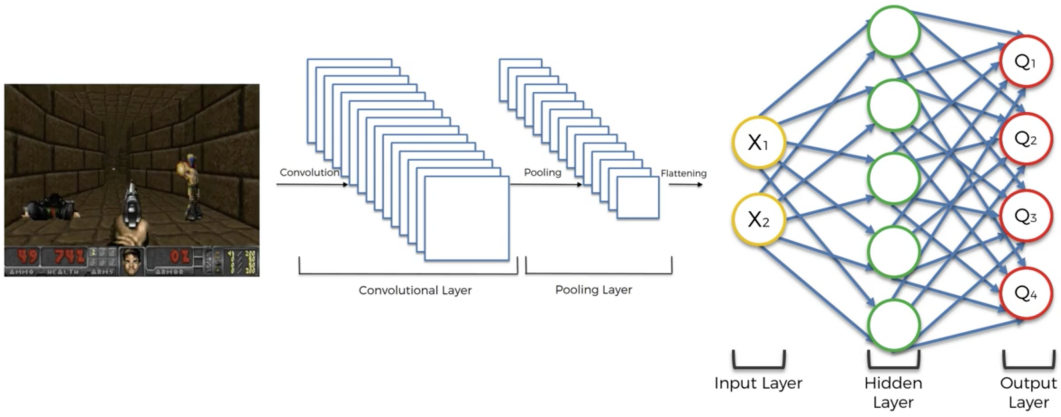

Figure 2: Deep Convolutional Q-Learning

7 Asynchronous Actor-Critic Agents (A3C)

7.1 A3C Intuition

So far, the action played at each time has been the output of one neural network, as if only one agent was deciding the strategy to play the game. This will no longer be the case with A3C. This time, we are going to have several agents, each one interacting with its own copy of the environment. Let's say there are n agents A_1, A_2,..., A_n.

Each agent is sharing two networks: the actor and the critic. The critic evaluates the present states, while the actor evaluates the possible values in the present state. The actor is used to make decisions. At each epoch time of training for on agent, it takes the last version of the shared networks and uses the actor during n steps in order to make a decision. Over the n steps, it collects all the observed new states, the values of these new states, the rewards, etc... After the n steps, the agent uses the collected observations in order to update the shared models. The times of epoch, and therefore the times of updates of the shared network by the agent are not synchronous, hence the name.

That way, if an unlucky agent starts to be stuck into a suboptimal but attractive policy, it will reach out that state – because other agents also updated the shared policy before the agent got stuck – and will continue effective exploration.

In order to explain the update rules of the actor and the critic, let us see the networks as functions that depend on vectors of parameters θ (for the actor) and θ_v (for the critic).

7.2 The whole A3C Process

The official A3C algorithm is the one of the Google DeepMind paper, "Asynchronous Methods for Deep Reinforcement Learning" (https://arxiv.org/pdf/1602.01783.pdf). In this paper you will find it in the following S3 algorithm:

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ and θ_v and global shared counter $T = 0$
// Assume thread-specific parameter vectors θ' and θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial \left(R - V(s_i;\theta'_v)\right)^2 / \partial \theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

Figure 3: A3C algorithm (https://arxiv.org/pdf/1602.01783.pdf)

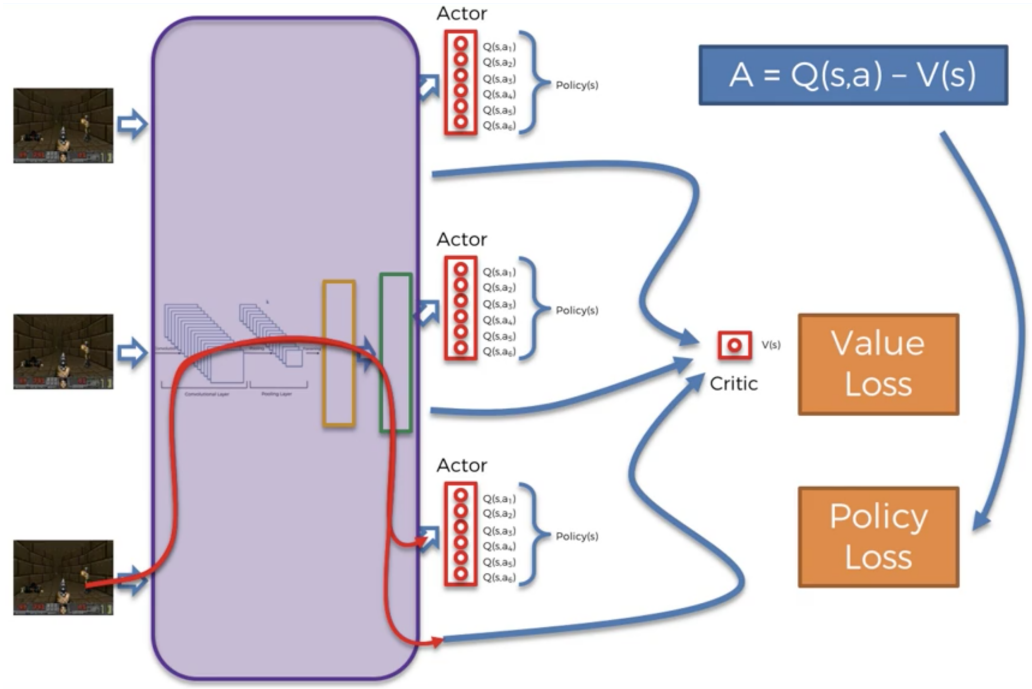

Figure 4: A3C

In this figure above we can clearly see the three As of the A3C:

- **Asynchronous**: There are several agents, each one having their own copy of the environment, and all asynchronised (playing the game at different times).
- **Advantage**: The advantage is the difference between the prediction of the actor, $Q(s, a)$, and the prediction of the critic, $V(s)$:

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

- **Actor-Critic**: Of course we can see the actor and the critic, that therefore generate two different losses: the policy loss and the value loss. The policy loss is the loss related to the predictions of the actor. The value loss is the loss related to the predictions of the critic. Over many epochs of the training, these two losses will be backpropagated into the neural network, then reduced with an optimizer through stochastic gradient descent.

8 Conclusion

Today the most powerful AI model is the A3C. But research is making great progress at a fast pace, so who knows if the A3C could already be outdated in a couple of years. And by 2029, this course might have to be called "Weak Artificial Intelligence A-Z", because if we believe Ray Kurzweil predictions (that so far have tended to be true), we will have human level AIs (with possibly a conscience) by the end of the 2020s. Hence congratulations for having taken on the challenge to learn AI, because one day you might have an important role to play.