Foundations of Machine Learning II

Bastien BRIER - Andrei CONSTANTINESCU
bastien.brier@student.ecp.fr - andrei.constantinescu@essec.edu

March 27, 2017

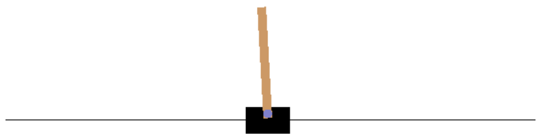
Final Project - Report

Our objective in this final project is to apply different reinforcement learning algorithms that would generalize quite well in discrete spaces and actions. First, we decided to focus on the cartpole environment to see the performance of different algorithms and then to generalize on a more complex environment, the lunar lander.

1 Cartpole

The principle is the following : a pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. There are two possible actions : apply a force of +1 to the left or to the right. The objective is to prevent the pendulum from falling during 200 consecutive timesteps (and 500 for the harder version). The environment is considered solved when obtaining a mean of 195 in 100 consecutive episodes. We used the implementation of Open AI Gym of this exercise.
More concretely, the action space is then divided in two values, each one corresponding to a +1 or -1 force. The state space is a four-variable vector. The first two dimensions correspond to the position of the pole, and its speed, the last two correspond to the angle of the pole and the angular speed of the pole. The first value is comprised in [-2.4; 2.4] and the third in [-0.20943951; 0.20943951]. The second and fourth ones are unbounded in this implementation. For practical purposes, we bounded empirically the the values of these two dimensions.
In the different algorithms below, we will try different implementations of the Q-Learning algorithm with its TD(λ) generalization.

Figure 1: Cartpole



1.1 Kernel Method

First, after trial and error, we realized that the two first variables of the input space(position and speed of the pole) did not have much impact on the output of the model, and only slowed convergence. So we decided to focus on the two last variables and discard the two first.
We then implemented the kernel method. The principle is to map the points (state[2], state[3]) in high dimension and to calculate the distance of the state to the points of a predefined grid. The different points of the grid are defined as:

$$s^{i,j} = (low\_bound(state[2]) + i * \frac{up\_bound(state[2])}{nb\_intervals}, low\_bound(state[3]) + j * \frac{up\_bound(state[3])}{nb\_intervals})$$

In our implementation, we decided to set :

- nb_intervals i = 25
- nb_intervals i = 12

And we already know that :

- [low_bound i, up_bound i] = [-0.20943951, 0.20943951]
- [low_bound j, up_bound j] = [-0.8726646, 0.8726646]

We then compute the distance of the state to the points. An important factor to keep in mind here is the standard deviation in this gaussian Kernel. If it is too high then it means all points will be considered close to the reference points in the grid, which is good for generalization but also makes the algorithm consider al points as the same, and if it is too small, then all the points will be too far apart from all the reference points, and all the distances will be equally 0. Stricking the right balance in this trade-off is essential in ensuring that the algorithm converges and performs well. The formula for computing distances is given below:

$$\phi^{i,j} = e^{\frac{-(state[2]-s_1^{i,j})^2}{\sigma_1}} e^{\frac{-(state[3]-s_2^{i,j})^2}{\sigma_2}}$$

The Q-function approximation is then computed as :

$$Q((state[2], state[3]), action) = \sum_{i,j} W_{i,j}^{action} \phi^{i,j}$$

We calculate the delta and update the weight matrix theta (alpha being the learning rate):

$$\delta = reward - Q[action] * \gamma * max(Q(next\_state))$$

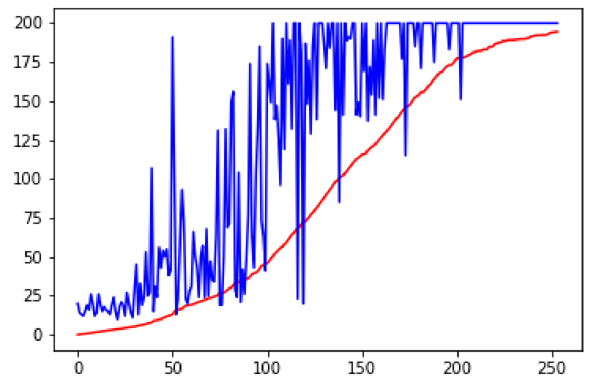
$$\theta[:, action] = \alpha * \delta * \phi$$

To select the action, we keep an ε-greedy policy, which means that:
- with probability ε, we choose a random action
- else we select $max(Q(state[2], state[3]))$

For this policy to be more and more relevant, we also use a decaying epsilon formula. The more episodes we would have done, the lower the epsilon.

This algorithm had great results on the Cartpole-v0. It solved the environment in about 150 episodes depending on the runs. Here is a graph that shows the performance of our kernel method. In blue is the performance by episode, and in red the 100-episode mean. Our python coding of this method can be found in the file : simple_cartpole_kernel.py. Submitted on OpenAI (v0):
`https://gym.openai.com/evaluations/eval_VwdplxUzTVei6A7I4LYPrg`

Figure 2: Kernel method for the cartpole



1.2 Tile Learning

The principle of this model is to simplify the state space to feed as input to our Q-Learning algorithm. We discretize the input space in different tiles, and when we are given the original input, we discretize it, which means converting it to one of the tiles (the closest one). This enables the algorithm to converge faster.
As in the previous part, we considered only the last two variables : we divide up the space in 10 for the third variable and in 5 for the fourth one, and used the same bounds as before. We keep track of the performance in each tile in a dictionary, and we add a tile only if it appeared before. We update the values as follows (alpha being the learning rate):

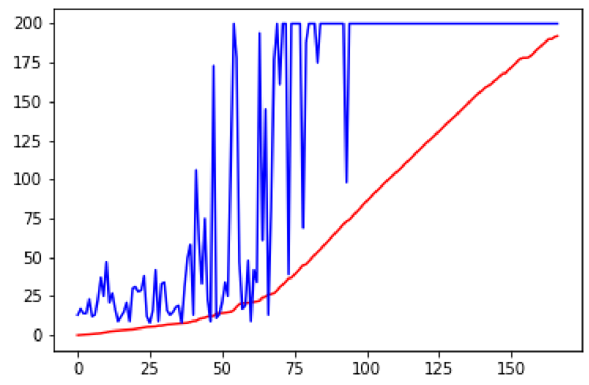
$$\delta = reward + \gamma * max_{action}(current\_tile) - previous\_tile[action]$$

$$previous\_tile[action] = \alpha * \delta$$

Concerning the action selection, we use the same ε-greedy policy as in the kernel method, and we also apply a decay to the ε:
- with probability ε, we choose a random action
- else we select $argmax_{action}(current\_tile)$

The performance is really great on the Cartpole. The environment was solved in 83 episodes. The code can be found in the file : tile_learning_cartpole.py.
Submitted on OpenAI (v0 and v1):
`https://gym.openai.com/evaluations/eval_BsWCOs8SCiumswSmqPInA`
`https://gym.openai.com/evaluations/eval_c7LZCdPQse1hVzhOcktmQ`

Figure 3: Tile learning for the cartpole



1.3 Deep Q-Learning and Experience Replay

Deep Learning is a machine learning framework that was made popular recently by the victory of AlexNet in the 2012 Image Net challenge. Since then Neural Networks have become obiquitous, and have managed to achieve state-of-the-art or near state-of-the-art performances in a wide range of applications, from Computer vision to NLP. One of the most notable successes of deep learning in the reinforcement learning framework was made by Google DeepMind who managed to create very good agents for 7 Atari Games using Deep Learning.

But Deep Learning comes with its own challenges, especially when it comes to reinforcement learning. Deep Learning is used in Reinforcement Learning as a tool to obtain better policy and action approximations from the observations. CNNs were a ready fit for Atari games because of their wide success in Computer Vision, so there was a lot of hope that the CNN could capture meaningful and interesting features from the input image and use them to obtain better function approximations. In our problems, cartpole and lunar lander, we will not use CNN since the input is not an image, but we will rather use usual fully connected networks. These have the advantage of being robust, somewhat impervious to the dimension of the observation state, contrary to the tile and kernel techniques. We need to describe more in depth the specific techniques we implemented to make Deep Learning work in this reinforcement learning framework.

General framework:
It is the same as for tile learning, or kernel learning. We give as input to the network the state representation and we expect as a return the prediction of the sum of the discounted rewards for each action.
To update the network, we use as before a tuple consisting of state, action, reward, next state, and is_done(flagging if a state is terminal or not). The formula is given below:
If is_done:

$$target = reward$$

Else:

$$target = reward + \gamma * max(next\_state)$$

Experience replay:
The idea is to create a neural network able to evaluate the value of an action given an input state. Neural networks need a lot of data to train well, we cannot just feed it the last observation. Because in the reinforcement learning framework, learning data is rare, we need a new technique to better profit from our data. Deep Mind described the technique

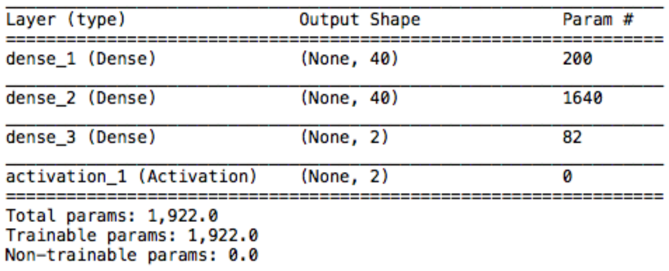
of experience replay which they use to solve their Atari games. The idea is to store at each step a tuple (state, action, reward, next state) in memory. Then take a sample of these tuples at random, and train the network on this random batch at each step. We perform this sampling because successive observation are correlated which leads to convergence issues. Nevertheless sampling uniformly is not an ideal solution, and some kind of prioritized sampling can improve the performance of the network. We did not have time to implement such a technique.

Target Network run in parallel:
The main network is used in determining the target to predict through the formula described earlier. Having a target that always changes poses obvious convergence issues. Therefore we made a copy of the neural network which we use to predict future values, and which we update by copying into it the weights of our main neural network every few steps. We found empirically that this helps greatly the network convergence.

Preliminary notes:
If Deep Learning is a powerful framework, it also comes with its own downfalls. First of all, there are a lot more hyperparameters to optimize, and we found that even if just one is slightly off, this can lead to the whole algorithm not converging and thus giving very poor results. Furthermore, because of the nature of the data in reinforcement learning(hihgly random, correlated, and scarce), we found that the performance of deep learning is highly volatile. It can get stuck pretty easily in a local minimum, depending on the sampling, on the data it is given, etc. For instance, in the cartpole example, if our random moves are highly biased towards an action(say left) then the neural network will hardly enter the exploration stage, even with a slowly decaying epsilon, and so it will give very poor performance even after 1000 iterations.

Application to Cartpole:
Because Deep Learning is not supposed to be influenced by the dimensionality of the input data, we decided to feed it the full 4-dimensional state of observation. We used keras to build our neural network which is a user-friendly deep-learning package.
The Network Architecture is as follows:

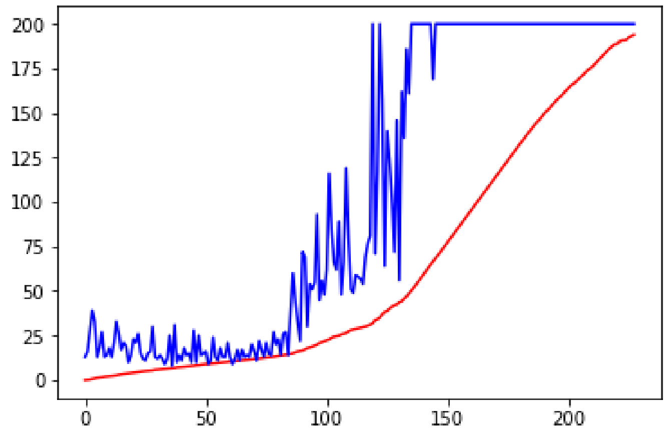
Figure 4: Network Architecture for cartpole



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 40)                200
_________________________________________________________________
dense_2 (Dense)              (None, 40)                1640
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 82
_________________________________________________________________
activation_1 (Activation)    (None, 2)                 0
=================================================================
Total params: 1,922.0
Trainable params: 1,922.0
Non-trainable params: 0.0
```

We use Adam as an optimization technique with the following hyperparameters : lr=0.0005, decay=1e-4, we found these values empirically by trial and error.
And we use MSE to compute the loss.
Below is a performance chosen at random, we run the algorithm 10 times and 9 time out of 10 it gives similar results. In blue the raw reward, in red the mean reward over the last 100 rewards. The algorithm finishes when the mean of the 100 last rewards is above 195.
Submitted on OpenAI (v0):
`https://gym.openai.com/evaluations/eval_QgDOGmQcSngb7J5R1c8w`

Figure 5: Deep Q-Learning for the cartpole

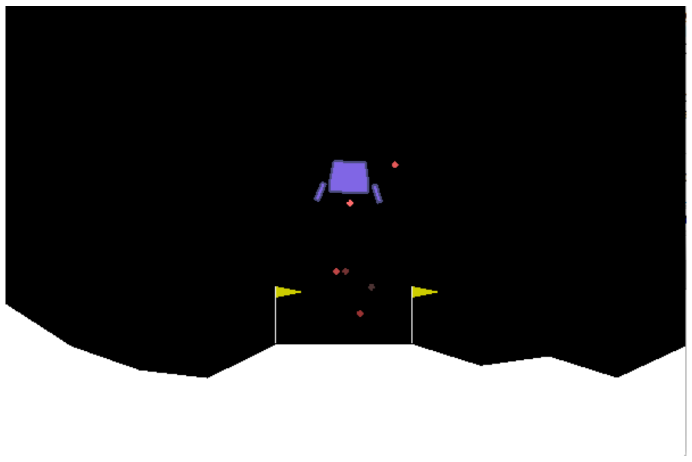


2 Lunar Lander

The objective of this environment is to make our spaceship safely land in the coordinates (0, 0). This environment is considerably more complicated than the cartpole. The state space is bigger, each state is comprised of 8 variables, each of which is unbounded. There are 4 different actions : do nothing, fire left orientation engine, fire main engine, fire right orientation engine.
The rewards are attributed with the following rules:
- if the lander crashes or comes to rest, -100 or 100 points are attributed
- if the spaceship lands outside the landing pad, it loses some part of the reward
- moving from top of the screen to the bottom brings between 100 and 140 points
- firing main engine is -0.3 point per frame
- each leg ground contact is +10 points

The environment is considered solved when obtaining a mean of 200 points in 100 episodes.

Figure 6: Lunar Lander



2.1 Deep Q-Learning

First, we tried to apply the same techniques as we implemented in the cartpole. But the state space in this new environment is much bigger (8 variables) and we have also 4 actions. This means that mapping these in high dimension or in a grid will create spaces that are too big and, thus, we could not reach convergence in reasonable time.

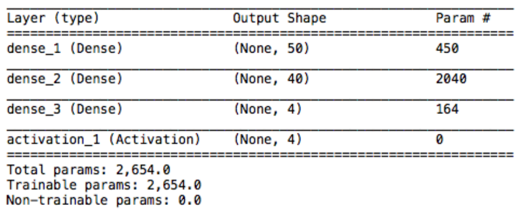
We then tried to focus on the Deep Q-Learning algorithm. The results were good but not convincing with a lot of variance. This is what encouraged us to seek a new loss function. Indeed because the rewards of this environment are quite spread out, ranging from -100 to +100, with most of them being of -0.3, mse leads to convergence issues, and high variance.

Hubert Loss:
We realized that the classical ways of computing errors are not very well adapted for the reinforcement learning framework. Indeed since we update the network with values that can be quite large(which is often the case for Q learning, as we need to take into account the discounted sum of all future rewards), the difference between the predicted and the target can be quite large as well. This is a problem because it can easily cause the network to diverge. The Hubert loss on the contrary has the advantage of caping the loss between -1 and 1 which helps the network converge.

Hubert loss has the advantage of caping the loss function which in this case helps a lot the algorithm to convergence. After that the main challenge was finding the right hyperparameters. This game is different than the last one where a lot of exploration was necessary in order to get good results. Here because rewards are skewed towards the left (the lunar landed crashes more ofthen than lands appropriately) exploration is in part assured by that. So we could diminish the value of the epsilon, and also the number of purely random episodes.

The Network Architecture is as follows on figure 7:

Figure 7: Network Architecture for Lunar Lander

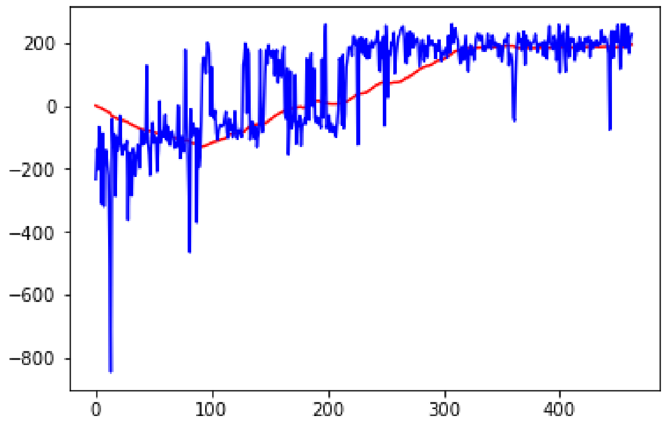


```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 50)                450
_________________________________________________________________
dense_2 (Dense)              (None, 40)                2040
_________________________________________________________________
dense_3 (Dense)              (None, 4)                 164
_________________________________________________________________
activation_1 (Activation)    (None, 4)                 0
=================================================================
Total params: 2,654.0
Trainable params: 2,654.0
Non-trainable params: 0.0
```

We use Adam as an optimization technique with the following hyperparameters : lr=0.001, we found these values empirically by trial and error.
Let us note that the key parameter that enabled us to get such results here was memory size. Indeed at first we had a very big memory size thinking that more is better. Nevertheless we noticed that if we reduced it dramatically (from 500 000 to 10 000) then we got improved performances. Storing too old steps is detrimental and makes it harder for the neural network to converge since it has to take into account a wide series of states. Or at least this is our interpretation of these results.

Submitted a run on OpenAI:
`https://gym.openai.com/evaluations/eval_Bb2qSaXrRPe8TQQjU26fg`

Figure 8: Deep Q-Learning for the Lunar Lander



2.2 Conclusion

In this work we learned how to use the Q learning framework with different function approximations in order to solve reinforcement learning problems with continuous state spaces but discrete action spaces. If the tile and the kernel techniques give some impessive results, they are limited and do not scale well to states with high dimensions. The Deep Q learning framework on the other bridges this gap and seems fit to answer a wide variety of reinforcement learning problems. Nevertheless it has a lot of hyperparameters to optimize and requires great knowledge and intuition to get the good combination of hyperparameters.