Scalable Machine Learning and Deep Learning: Project Report

Vittorio Maria Enrico Denti, Francesco Staccone

January 12, 2020

1 Problem description

The game of Pong is an example of a Reinforcement Learning (RL) problem, a subfield of Machine Learning where an agent takes actions in an environment in order to maximize its cumulative reward. The advantage of RL is that it is not needed to know a priori the information of dynamics but it learns the information from experience. When the agent directly learns from its current experience the learning task can be classified as on-policy learning while when the agent learns from experiences coming from the past and/or from other policies it is an off-policy learning task.
Pong is an Atari game in which you play as one of the paddles and you have to bounce the ball past the other player, as you can see in Figure 1. It is fundamental to remember that the agent is learning from the pixels coming from the game environment so its state is described by images.

Figure 1

On the low level the game works as follows: we receive an image frame and we get to decide if we want to move the paddle UP or DOWN. 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. Our goal is to move the paddle so that we maximize the reward over time.

2 Tools and requirements

The main tools used to address the problem and implement our solution are the following libraries:

- Python 3
- Tensorflow
- Keras
- Gym and Atari dependencies
- Numpy, SciPy
- Other minor Python libraries

The software has been developed using Git for version control and Google Colab as AI platform.

3 Data

Given the described scenario and the Reinforcement Learning nature of the problem, there is no need of initial data. Moreover, since we analyzed the context of on-policy learning, the agent will be trained from scratch since it was not be able to learn from past episodes but only from its experience. Of course, in a real setting it could not be an optimal choice because it increases the time needed for training the agent but since we wanted to experience on-policy learning, the project was developed under this initial hypotheses.

4 Methodology and algorithms

We implemented a Reinforcement Learning algorithm to solve this task. In particular implemented the Deep Q-learning approach proposed by Google Deep Mind and the Monte Carlo Policy Gradients approach. Since the input of the network are images, we preprocessed them in order to extract only useful features. Also, we removed useless information such as the score and the data about the colors of the playground. After these first steps, we focused on the implementation of the Reinforcement Learning algorithms and the analysis of the training performance in order to optimize the policy function, better manage the exploration-exploitation trade-off, and achieve reasonable training times.

5 Algorithms

5.1 Policy gradient

Theoretical background:
Policy gradient is a method that trains the policy functions by gradients. Basi-

cally in each training, we push the policy to take good actions more probably and bad actions less frequently.
Here we use a Neural Network as policy and a Policy gradient algorithm called REINFORCE. The agent collects a trajectory τ of one episode using its current policy, and uses it to update the policy parameter. Since one full trajectory must be completed to construct a sample space, REINFORCE is updated in an off-policy way.
The macro steps of the REINFORCE algorithm are shown in Figure 2:

```
function REINFORCE
    Initialise $\theta$ arbitrarily
    for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ do
        for $t = 1$ to $T - 1$ do
            $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
        end for
    end for
    return $\theta$
end function
```

Figure 2

The flow of the algorithm can be described as follows:

- Perform a trajectory roll-out using the current policy
- Store log probabilities (of policy) and reward values at each step
- Calculate discounted cumulative future reward at each step
- Compute policy gradient and update policy parameter
- Repeat 1–4

Given the formula $\theta = \theta + \alpha \nabla_\theta log(\pi) G$ and the policy $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$:

- π is the probability of taking the action that we took when playing.
- θ are the Neural Network parameters in last layer.
- G is the reward from the state we are updating.
- α is the learning rate.
- S0 is the initial state, A0 is the action that agent selected after observing S0, and R1 is the reward given to the agent by the environment after taking action A0.

The agent continues to play until it wins, loses or the game ends at time T, and we save the states and actions and rewards for each state.
It is noticeable that the formula mentioned above is similar to Gradient Descent with $-log(\pi)G$ as loss: $\theta = \theta - \alpha \nabla_\theta(-log(\pi)G)$
. The learning rate is multiplied by G, so if we get a high reward, we move much more in the direction of gradient descent, while if we are punished and got a

minus reward we move in opposite direction. In the learning phase the policy outputs the probability of each action and we sample from that probability. In the playing phase we pick the most probable action or just sample like the training phase.

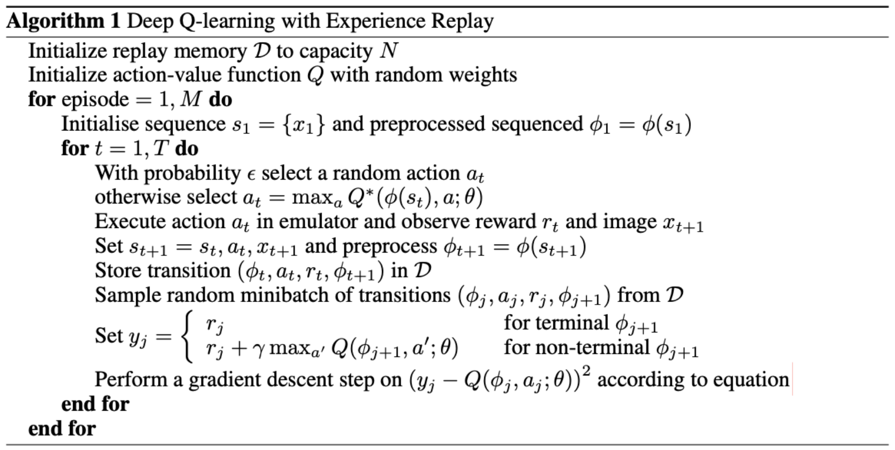
Implementation choices:

- In order to use a neural network as policy, we need to convey enough information so it can learn where the ball is moving. In the input images, it is not clear whether the ball is moving left or right. As simple solution to this problem, we subtract two consecutive frames and feed the resulting image to the network as input.
- Bottom and top of screen are cropped because they did not give any information, as well as the colours that are removed. Moreover, pixel values are scaled between -1 and +1.
- We defined a network with an architecture that inputs a 80x80 image and outputs the probability of going up. It is composed of 2 layers, 200 hidden units in the first layer and 1 sigmoid output.
- Keras loss function has format: def loss(y_true,y_pred): ... . Since we needed to include the reward in loss, we created m_loss function as a tool to input episode_reward as input.
- We noticed that only the action that we took to hit the ball was important to get the reward +1. Everything before hitting the ball was irrelevant to our win. A similar argument could be discussed about losing: the action near when we got score of -1 is more important than actions taken many steps earlier.
  So we set the reward of actions taken before each reward, similar to the reward obtained. For example if we got reward +1 at time 100, we say that reward of time 99 is +0.99, reward of time 98 is +0.98 and so on.
  We also normalized the rewards subtracting by mean and dividing by standard deviation because setting a positive score at the beginning of losing sequences could be justified by the fact that we approve actions where we caught the ball and did not get -1 reward immediately.

5.2 Deep Q learning

Theoretical background:
Deep Q learning is a Reinforcement Learning algorithm that implements Q learning through a Deep Neural Network. Traditional Q learning is based on value iteration and tables to store the state-action Q value, an iterative approach that evaluates the goodness of a state-action pair by looking at the maximum reward that I can achieve once that state-action pair brings the agent to the next state. The goal of the Neural Network is to predict the Q value associated to each action given the state, so the traditional tabular approach of Q learning

is replaced by the Neural Network that approximates the action-value function. The goal of the agent is to select actions in a way that maximizes the cumulative future reward and it is done by following the algorithm described as follows:



Algorithm 1 Deep Q-learning with Experience Replay

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation
    end for
end for
```

During training the target is $y_j = r_j + \gamma \max'_a Q(\phi_{j+1}, a')$ and it is always computed using the target network while the prediction is $Q(\phi_j, a)$ and it is the Q value predicted at this step by the prediction network. It is possible to notice that in this way the agent is trained in order to minimize $((r_j + \gamma \max'_a Q(\phi_{j+1}, a') - Q(\phi_j, a))^2$ that allows to train the Neural Network to estimate the Q value of the current state-action pair depending on the maximum Q value that could be achieved after such action has been performed. The roles of the prediction network and the target network are explained in the next section, however it is possible to notice that the target network transforms this problem into a supervised learning problem and that the target network works as an oracle to define what the prediction network must learn.

Implementation choices:
Deep Q learning requires to manage Deep Neural Networks in the context of Reinforcement Learning so it is fundamental to prevent the risk of overfitting and maintain the input samples independents. The architectures was based on two convolutional layers and two classification layers, different combinations of neurons and parameters were evaluated while training and the final architecture can be found in the submitted Jupyter notebook.

- Frame preprocessing: each input frame is preprocessed to remove the score in the header of the image and cut the image to only maintain the Pong court, the paddles and the ball. Moreover, each frame was down sampled to scale 80x80 and it was converted to black and white colors since RGB brings no information to solve this learning problem.
- Frame merging: Reinforcement Learning is based on the Markov assumption that states that the future is independent from the past given the

current state. Hence, it was necessary to capture the concept of movement in the state and it was done by stacking each 3 consecutive frames.

- Prediction and target network: as the problem of Deep Q-learning is defined, the risk is that the target value changes each time the model is updated. This causes the training to be unstable and was solved by using two Neural Networks with the same structure. A prediction network learns the parameters and predict the next action during training and a target network predicts the optimal value of the sequence at the next time step according to the Bellman equation. After each 4 episodes the weights are loaded from the prediction network to the target network so as to transfer the knowledge gained.

- Experience replay: Neural Networks are models that require independent input samples but in this context sequential samples have a strong dependency. The problem was solved by implementing an experience buffer: past experiences are stored in a buffer and each time the prediction network is fed with 32 samples randomly chosen from the buffer. The advantage is that the network is fed with independent inputs and past experiences are not forgotten reducing the risk of overfitting.

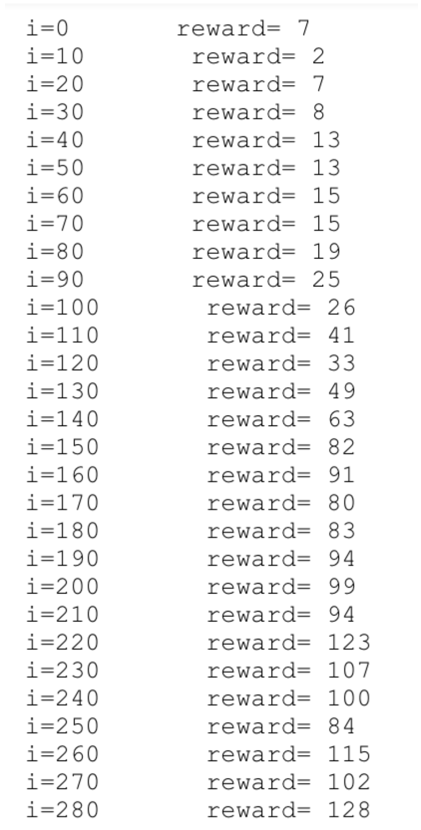
- Exploration and exploitation: at the beginning of the history there is the need to learn as much as possible and it is done by observing as many states as possible. Hence, an ϵ-greedy policy chooses with probability ϵ a random action and with probability 1-ϵ the action that the prediction network suggests as the best. In this way the agent explores many different actions but as training goes on the probability ϵ is decreased in order to leverage the knowledge gained during time.

6 Results and discussion

6.1 Policy gradient

The results obtained by the Policy gradient approach are meaningful and clearly show that the agent is learning how to score more and more goals to the opponent as the training proceeds, reaching also the victory.
In Figure 3 we report a sample of the results obtained during the last training we performed. In particular, **i** stands for the number of epochs performed until that moment (1 epoch = 1 match), while **reward** stands for the number of total positive rewards (1 positive reward= 1 goal scored) between **i** and **i+10** epochs. E.g. We obtained 123 positive rewards between match 220 and 230, meaning that in each of those 10 matches we obtained an average of 12,3 goals scored.
As noticeable from the reward amount every 10 epochs, it keeps increasing meaning that the agent keeps learning more and more as the training proceeds. We could have obtained even more interesting results going ahead with the training, but due to hardware limitations and Google Colaboratory timeouts we were not

able to do better. Anyway, the results we obtained are very satisfying and some simulations are linked in the next paragraph.



```
i=0       reward= 7
i=10       reward= 2
i=20       reward= 7
i=30       reward= 8
i=40       reward= 13
i=50       reward= 13
i=60       reward= 15
i=70       reward= 15
i=80       reward= 19
i=90       reward= 25
i=100       reward= 26
i=110       reward= 41
i=120       reward= 33
i=130       reward= 49
i=140       reward= 63
i=150       reward= 82
i=160       reward= 91
i=170       reward= 80
i=180       reward= 83
i=190       reward= 94
i=200       reward= 99
i=210       reward= 94
i=220       reward= 123
i=230       reward= 107
i=240       reward= 100
i=250       reward= 84
i=260       reward= 115
i=270       reward= 102
i=280       reward= 128
```

Figure 3

We report below some match simulations recorded with three different models obtained after 170, 220 and 280 training epochs respectively. As noticeable the agent keeps improving and it also reaches the victory in the last simulation.
Regarding the limitations of the agent, we could state that it still suffers from lack of experience in managing the case in which the ball reaches the upper and lower corner of its scope, while it is very good ad managing the central section case. This is due to more experience accumulated in the central part than the upper and lower corners, therefore with a longer training it will hopefully do better in this regard.

1. Click here to visualize a match simulation obtained after 170 training epochs.

2. Click here to visualize a match simulation obtained after 220 training

epochs.

3. Click here to visualize a match simulation obtained after 280 training epochs.

As future work, we could evaluate the introduction of a CNN layer to process the incoming frames and some other modifications to the network (e.g. more layers or neurons), tuning them until reaching better performance. Moreover, we could remove the limitation to Up and Down actions for the agent, including also the Stop one, so that during the simulation it can behave in a smoother way maintaining the same position in case of need.

6.2 Deep Q learning

Due to limited computing resources the training is still going on. However, the code can be found in the submitted notebook.