Adrià Ciurana Lanau

Udacity:

Project 1: Navigation

Project description:

In this project it is proposed to use an autonomous agent for the capture of wild bananas. Unfortunately, there are two types of bananas: yellow (perfect for consumption), blue (very toxic).

Our objective is to try to capture the "good" bananas while avoiding the “bad” ones.

In the first instance, we should try to formalize the problem proposed in one of RL.

Variable name	Description
state	A set of 37 sensors that describe the information of the current state.
actions	A set of 4 possible actions: move forward (0), move backward (1), turn left (2), turn right (3).
reward	+1 get a good banana, -1 get a bad one.

Once the problem is defined, we must remember that an RL problem is simply a **maximization problem**. Where we want to maximize the final reward based on all the actions taken. In addition, the problem we face is a problem in a **episodic** and **continuous state-space**.

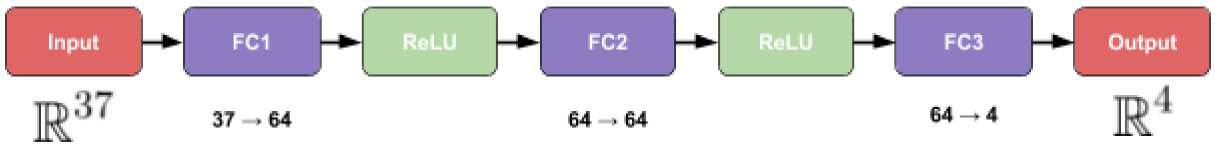
We will use the **state tuple as input** to a function that will return the **4 possible actions as output** and their approximate reward.

$$f(st\vec{a}te) = acti\vec{o}ns$$

This **mapping** can be treated as a function approximation problem. So we can use any algorithm that allows us to obtain a continuous output from a multivariable input of 37 values.

In this case, it has been obtained using a MLP neural network for its simplicity when dealing with this problem. But we could have used any other type of algorithm, or a vanilla gradient descent with our own custom function.

In this case I have decided to preserve the architecture used in previous projects for its simplicity. Keep in mind that RL learning has a high cost that is not directly involved in learning (rendering, environment, etc).



That means that using a very complex network can make learning very slow (more time per forward, more time in backpropagation and more time to convergence).

Therefore in this case, **more is not better.**

The problem of maximization is defined by the bellman optimization equations (specifically the action-value).

Loss definition:

Any optimization problem has a loss function associated with it. In this case, we have defined a loss based on the improvements of **Double DQN** together with **Prioritized Experience Replay**.

$$\mathcal{L}(s,a) = \sum_i \left(\frac{1}{N}\cdot\frac{1}{P(i)}\right)^{\beta}\left[r + \gamma \cdot max\ q(s', arg\max_a q(s',a,w), w^-) - \bar{q}(s,a,w)\right]^2$$

where $P(i)$ is the probability of the sample i, w^- are the "fixed weights" and w are the "normal weights".

With this we ensure several things:

- A less turbulent descent because the choice of the best action does not depend on the current weights.
- A better selection of experiences, prioritizing experiences that have big errors.

We are trying to make our current state-action function equivalent to that of the next state plus the associated rewards. By minimizing this difference we will get the agent to take the best actions to benefit in the near future (depends on the gamma parameter).

The derivative of the previous loss is quite simple (consider q(s', a, w) inside of TD target as constant w.r.t w).

$$\nabla_w\mathcal{L}(s,a) = \sum_i \left(\frac{1}{N}\cdot\frac{1}{P(i)}\right)^{\beta}\left[r + \gamma \cdot max\ q(s', arg\max_a q(s',a,w), w^-) - \bar{q}(s,a,w)\right]\nabla_w\bar{q}(s,a,w)$$

Agent step:

In each of the steps of our agent:

- We obtain the status, evaluate its absolute error using the following expression (network in evaluate state):

  $$e_i = abs\left(r_i + \gamma \cdot max\ q(s_i', arg\max_a q(s_i', a, w), w^-) - \bar{q}(s_i, a_i, w)\right)$$

  And register it in the buffer of experiences.
- Every X steps, we do a learning of the network.

In the **learning process**, it gets a bit complicated ...
The errors of each of the experiences stored in the buffer are converted into a probability distribution. Using a random variable, we select N samples corresponding to our batch size.

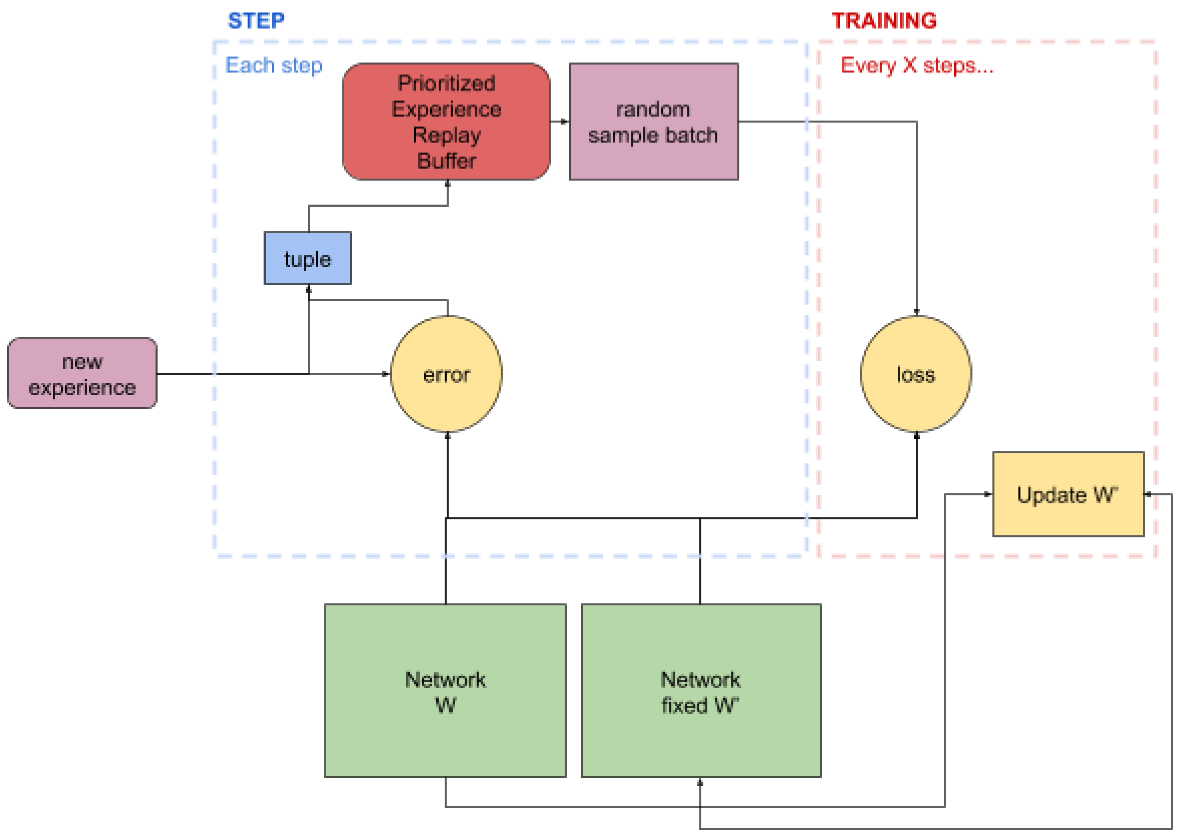
These samples will be responsible for trying to reduce the loss described above.

Our agent really always learns from a “previous step”, that is to say: we consider state as previous state, the next_state as current one. In that way we know the associated reward, the action and if it has finished (perfect to be learned through our network).

All these experiences that are evaluated are updated their errors in order to correct the buffer distribution.

On the other hand, the weights fixed in the other network are also updated every so often (using a soft average between new weights and old fixed weights).

The following figure shows the process from when a new experience is glued to learning the network:



Hyperparameters:

The exploration of hyperparameters has not been very exhausted, which is why it is probably a point to consider in the future.

A **learning rate of 0.001** has been used together with an optimizer **Adam** so that he will be in charge of adjusting it based on the gradients.

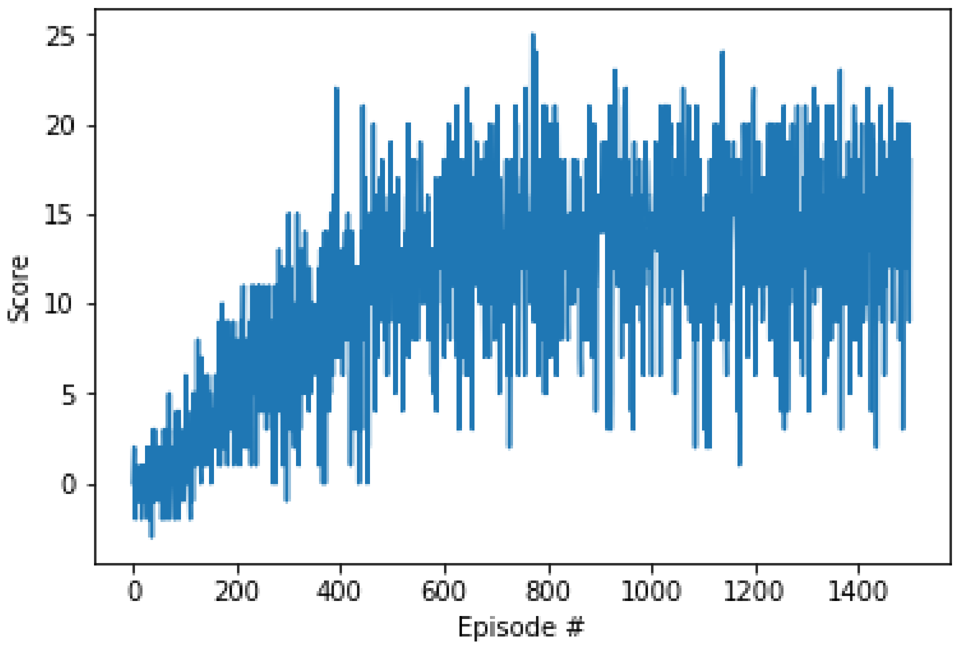
It has been considered a **fairly high gamma** to force to consider "further decisions".

The selected **batch size was 8**, it seems to be enough to make a compromise between speed / step learned against the generalizability of the model.

Finally, the parameters for the Prioritized Experience Replay have been alpha=0.7 and beta=0.5. Alpha is responsible for finding a compromise between uniform distributions and distributions based on the error. While beta we have selected a value and it has not been decided to move during the learning due to lack of time.

Results:

Below are the results obtained during the episodes. Learning has been limited to 1500 episodes, being satisfactory from obtaining +13 in contiguous 100 episodes.



Results:

```
Episode 100 Average Score: 0.58
Episode 200 Average Score: 3.80
Episode 300 Average Score: 6.72
Episode 400 Average Score: 8.51
Episode 500 Average Score: 10.98
Episode 600 Average Score: 12.12
Episode 700 Average Score: 13.38
Episode 800 Average Score: 13.55
Episode 900 Average Score: 13.10
Episode 1000      Average Score: 14.03
Episode 1100      Average Score: 13.86
Episode 1200      Average Score: 13.57
Episode 1300      Average Score: 13.37
Episode 1400      Average Score: 13.88
Episode 1500      Average Score: 14.82
```

We can observe how the learning stabilizes in 13-14. Possibly it is due to the limitations of the architecture.

PixelBananasNet:

In this case, the only change is in the set of states. Instead of having a sensor input (37), we have an RGB image of 84x84.

To carry out this process we need to work with a CNN (Convolutional neural network). For my part, I think that the best option to increase the speed of the process is to use a pre-trained network.

Much of the first layers will be responsible only for recombining patterns that have little to do with the precious bananas :(

But will be patterns that will be responsible for detecting contours, changes in contrast, simple geometry, etc. So why not use a network that already has a great advantage over one without any type of training?

In this case a **Mobilenet_v2** has been chosen, the advantage of this network is that it is fully-convolutional. This allows us to be almost independent of the input size because all the layers are convolutions until we reach an **AvgPooling** or **MaxPooling** (only multiplicity restrictions).

For our input (84x84) be accepted by a mobilenet, it must be multiple of $32 \cdot \alpha$ **(**where α is kind of of mobilenet selected).

$$32 \cdot \alpha \cdot \left\lceil \frac{input}{32 \cdot \alpha} \right\rceil$$

Unfortunately the closest number to this multiple resolution in both $\alpha = 1$ is 96 (or reducing dimensions to 64)**.** Same behaviour is observed with $\alpha = 0.75$ and $\alpha = 0.5$.

To use mobilenet_v2 on torch, I downloaded this repository and the pre-trained weights of alpha = 0.5. Although it has been included in this same project.

Changes with respect previous network:

I have decided to keep the same buffer size but this depends directly on the available RAM in the computer where it is made. I have also expanded the batch size to 256 to introduce more experiences during the training step because the state-space and the number of parameters are much larger than the previous problem.

Another alternative would be to store the parameters to perform the rendering but is not included in this project.

Why this architecture? Why not use your own? For many reasons:

- It has pre-weighed weights to speed up the process. An RL learning is already noisy by itself, pre-entered weights can help stabilize it.
- It is an architecture widely used by the industry and verified.
- It is optimized using Point-wise convolution and Depth-wise convolution.

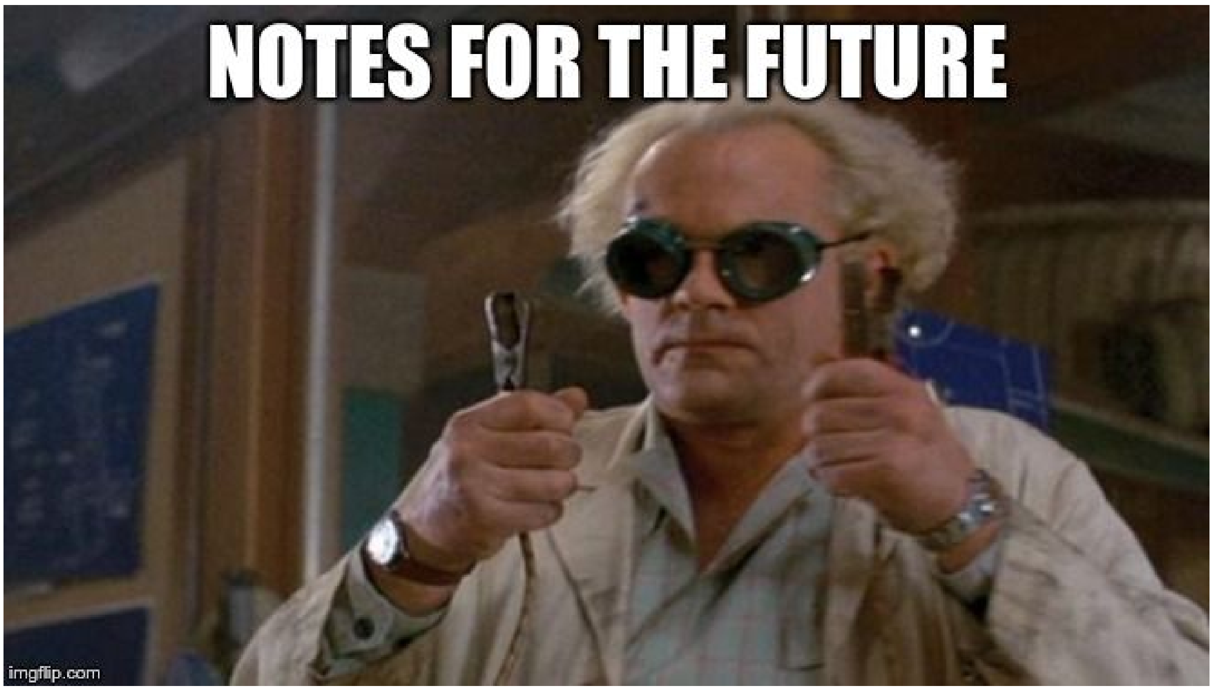
Results:

I have had several problems with the **Unity Environment** that closes to the window at 2000 episodes without indicating any error. And the process freezes waiting for indications of the environment.

So I have not been able to do a 100% satisfactory learning. Although learning seems to work because there is an improvement in the score.

Results:

```
Episode 100    Average Score: 0.28
Episode 200    Average Score: 0.06
Episode 300    Average Score: 0.14
Episode 400    Average Score: 0.29
Episode 500    Average Score: 0.76
Episode 600    Average Score: 0.88
Episode 700    Average Score: 0.88
Episode 800    Average Score: 1.17
Episode 900    Average Score: 1.52
Episode 1000   Average Score: 2.36
Episode 1100   Average Score: 2.82
Episode 1200   Average Score: 3.03
Episode 1300   Average Score: 3.66
Episode 1400   Average Score: 4.72
Episode 1500   Average Score: 4.37
Episode 1600   Average Score: 4.62
Episode 1700   Average Score: 4.91
Episode 1800   Average Score: 5.43
Episode 1900   Average Score: 5.12
Episode 2000   Average Score: 5.37
Episode 2014   Average Score: 5.59
```



In the future…

- Introduce stochastic processes such as Dropout to make new connections.
- Test new hyperparameters.
- It would have been interesting to try other network architectures, together with new RL ideas (for example: dueling networks, rainbow).
- In the case of the PixelBananasNet, try different CNN architectures (or different alphas of the mobile) together with smaller resolutions of the input image.
- Test the architecture proposed in the DeepMind paper, although personally I am not very "fan" of the big convolutions and the big strides. Large convolutions can be simulated with small convolutions unless you want to do "LocalFullyConnected". The big strides/maxpooling can make you lose a lot of information and you have to be very careful.