Capstone Project

Playing Space Invaders with Deep Reinforcement Learning

Machine Learning Engineer Nanodegree

Pablo López Santori

August 20th, 2019

Definition

Project Overview

In the past few years, reinforcement learning has seen great progress thanks to advancements in Deep Learning, in areas such as computer vision. These breakthroughs have allowed reinforcement learning algorithms to successfully use neural networks as function approximators, which provide better results than previous techniques when working with continuous spaces.

Atari games have been used as a benchmark for Reinforcement Learning since the introduction of the Deep Q-Learning algorithm (Minh et al 2013). The motivation for this project is to recreate the algorithm for the Space Invaders game where an agent can learn to play from scratch. A successful implementation of a Reinforcement Learning algorithm that learns to play the game successfully, leads us to more meaningful implementations where, for example, an agent learns to drive a car without any human interaction.

Problem Statement

This problem can be framed into a Markov Decision Process (Sutton and Barto, 2018), where the agent is the learner making the decisions, and the environment is the Atari simulator. At each time step t the agent will select an action from a discrete space of actions $A = \{1, ..., K\}$. One time step later, as a consequence of that action, the agent will receive a new state $s_t \in S$ from the emulator as an image, as well as a reward R_{t+1} based on the current score after taking that action. This finite MDP process gets repeated rising to a sequence trajectory like $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_{,...}$.

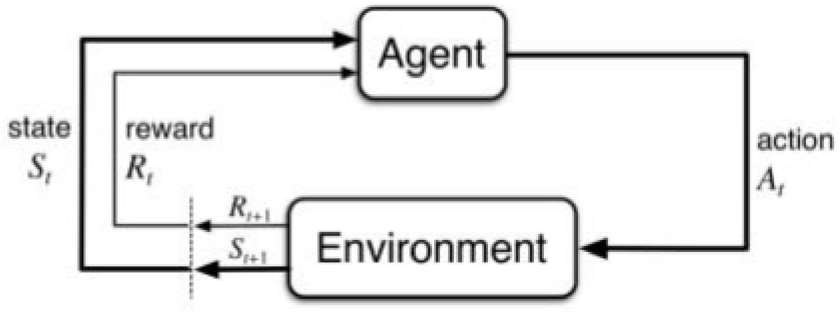

Figure 1 | The Markov Decision Process

The goal of the agent is to select actions that maximize the expected return. The agent has to take into account both immediate and future rewards, thus, we assume that the future rewards are discounted by a factor of γ per time step. If $\gamma = 0$ it means that the agent only cares about immediate rewards, and as γ gets closer to 1, the agent looks more into the future . The expected return can be defined as $\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$.

Metrics

In a supervised learning scenario, one can easily separate the dataset into train, test and validation. In reinforcement learning, evaluating the model's performance is not as straightforward, as we can only observe how the agent has learned over time from its own *experience*. The agent will use the game's score to evaluate its performance.

The metrics considered to evaluate this project is the average performance of an agent over 500 episodes. Comparing these results over the benchmarks will show whether the agent has learned how to play the game or not.

Analysis

Data Exploration

The goal of the agent in a reinforcement learning scenario, is to learn from *what it sees* just as a human would. It would be easy to create an agent that can play Space Invaders perfectly if it had access to the underlying game state, however that is not the case. The agent will only receive a screenshot image of the screen for every time step t, of size 210×160. Based on these images and its current score the agent will make an informed decision on what action it should take next.

Exploratory Visualization

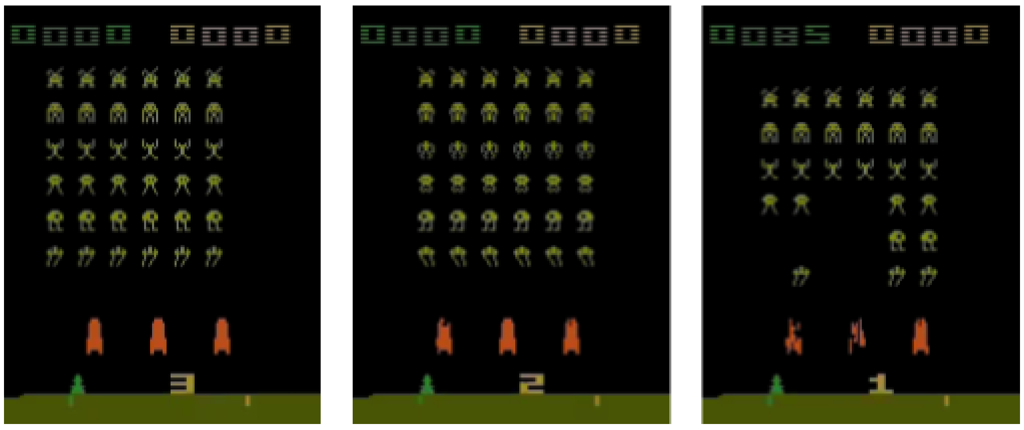

Figure 2 | Input data. Also shows the start position of the agent at the beginning of each game.

The agent will receive an 210×160 pixel image with a 124 colour palette. Each screenshot contains the current score, the state of the game at a timestep t, and the remaining lives.

The agent starts on the bottom left corner at the beginning of the game, or any time it loses a life. At each timestep the agent has to decide what action is going to take, from a fan of 6 different actions, with the goal of achieving the maximum score possible. To get a high score, it must kill as many *space invaders* as possible. The score is displayed on the top left corner.

The *space invaders* move from left to right, and shoot back at the agent. The agent has to avoid getting hit to not lose any lives. There are also three shields that the agent can use to protect itself from incoming projectiles.

Algorithms and techniques

In a reinforcement learning scenario, the agent is not given full knowledge of how the environment operates, instead it must learn from interaction. The goal of the agent is to select actions which will maximize the expected return (future rewards). The agent chooses actions based on a policy π, which is a mapping from a state s to an action a. The action-value function can be defined as a function that yields the expected return if the agent starts in state s, takes action a, and then follows the policy for all future time steps.

Bellman's expectation equation $Q^{*}(s,a) = \max_{\pi} E\left[G_t \mid S_t = s,\ A_t = a,\ \pi\right]$ gives us a way to express the action-value of any state in terms of the values of the states that could potentially follow. In a temporal difference problem, this equation $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\,(R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t))$ can be used to create an update rule for every time step and find the optimal action-value function. In order to keep a balance between exploration and exploitation, I'll use a ϵ-greedy policy.

Using a temporal difference approach to solve this problem, I could use an algorithm such as Q-learning (Watkins and Dayan, 1992). This algorithm is able to learn on every step of the way, without having to wait for an episode to end. However, Q-learning uses a Q-table to map every state to an action. This works fine when the state space is discrete, but it can get out of hand in a continuous space, as it would be impossible to keep that many values in a table. A solution to this problem is to use function approximation to map states to actions.

Function approximation consists of using function methods to find out the best action to take, given a state. An example of function approximation would be using a linear method to map states to actions, and update the function weights using gradient descent. On a trivial problem, this function should be able to approximate to the true Q-value function, allowing it to map a state to the optimal action. The prime problem with linear approximation is that we can only represent linear relationships between inputs and outputs. Most times the underlying value function will have a more complex non-linear shape, which is why we need to look at non-linear functions.

Artificial neural networks are mostly used in reinforcement learning these days. This is because they have non-linear activation functions that are able to capture the complexity of the underlying value function. The implementation I've used in this project is a Deep Q-Network (Minh et al, 2013), which is an improvement of the Q-learning algorithm (Watkins and Dayan, 1992) that uses a neural network instead of a Q-table.

I'll use a Convolutional Neural Network as the function approximator because it will work best with image inputs. The update rule will need to be changed so that the neural network can compute the loss function and update w using gradient descent. Same as before, I can use the estimated return in the next time step as a target to compute the loss function and create our new update rule:
$\Delta w = \alpha \left(R_{t+1} + \gamma max_a Q(S_{t+1}, a, w^-) - Q(S_t, A_t, w)\right) \nabla_w Q(S_t, A_t, w)$, where α is the importance of newer memories and w^- are the fixed function parameters that don't change during the learning step.

In order to improve performance, a series of improvements are implemented, such as Experience Replay, Fixed Q-Targets and Dueling networks, which will be fully discussed later on.

Benchmark

Three benchmarks have been used to determine if the trained agent has learned or not. Each benchmark model yielded the average performance over a 500 episodes, which will be used to compare with the average performance of the trained agent. The human player however was evaluated over a 100 games. The benchmarks used are:

- The performance of a human player.
- The performance of an agent acting randomly.
- The performance of an un-trained agent initialized with random weights.

Methodology

Data Preprocessing

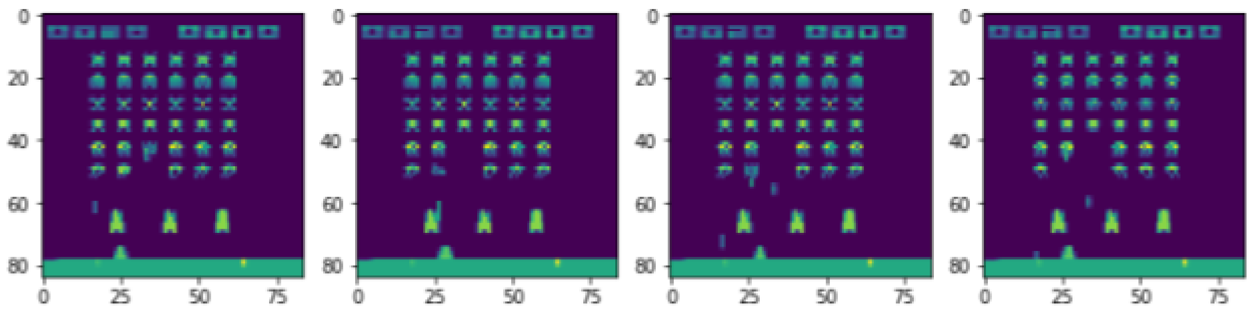

Figure 3 | Data preprocessing: each time step the agent will receive an stack of 4 images, rescaled, and converted to grayscale.

Atari games are displayed at a resolution of 210 by 160 pixels, with 128 different possible colors for each pixel. To reduce this complexity a number of preprocessing steps will be performed to the original raw images.

First, the images will be converted to grayscale, as the colours don't provide the agent with any useful information. Then, the images will be converted to a square of 84×84 pixels. Square images allow to perform more optimized neural network operations on GPUs. Smaller images will also be faster to pre-process.

In order to give the agent access to a sequence of frames, 4 images will be stacked together for every state input, creating a state space of $n \times n \times 4$. This preprocessing and stacking operation is denoted by Φ. Stacking M images as a single time step gives the agent spatial information, helping it to make better decisions (Figure 3).

Each image shows a lot of information about the game that the agent doesn't necessarily need to know at each time step, like for example the score (the agent already receives the current score for each time step), or the ground at the bottom. Getting rid of such information will help speed up the training process, as the agent doesn't get confused with information not useful for learning.

Implementation

The algorithm has been trained using an EC2 p2.xlarge instance from Amazon Web Services, intended for Deep Learning purposes. This instance has 1 GPU NVIDIA K80, 4 vCPU and 61 GiB RAM.

Model Architecture. On the output side, unlike the traditional reinforcement learning setup where only one Q-value is produced at a time, the deep Q-Network will produce a

Q-value for every possible action in a single forward pass. By doing it this way the agent can choose the action with the maximum value without having to run the network one time per possible action.

The first layers of the network will be convolutional layers, which will be in charge of finding spatial relationships. Also, since M frames are stacked to create a single input, the convolutional layers will be able to extract some temporal properties across these frames. These layers will be followed by a hidden fully connected layer with RELU activation function and an output layers with as many outputs as possible actions.

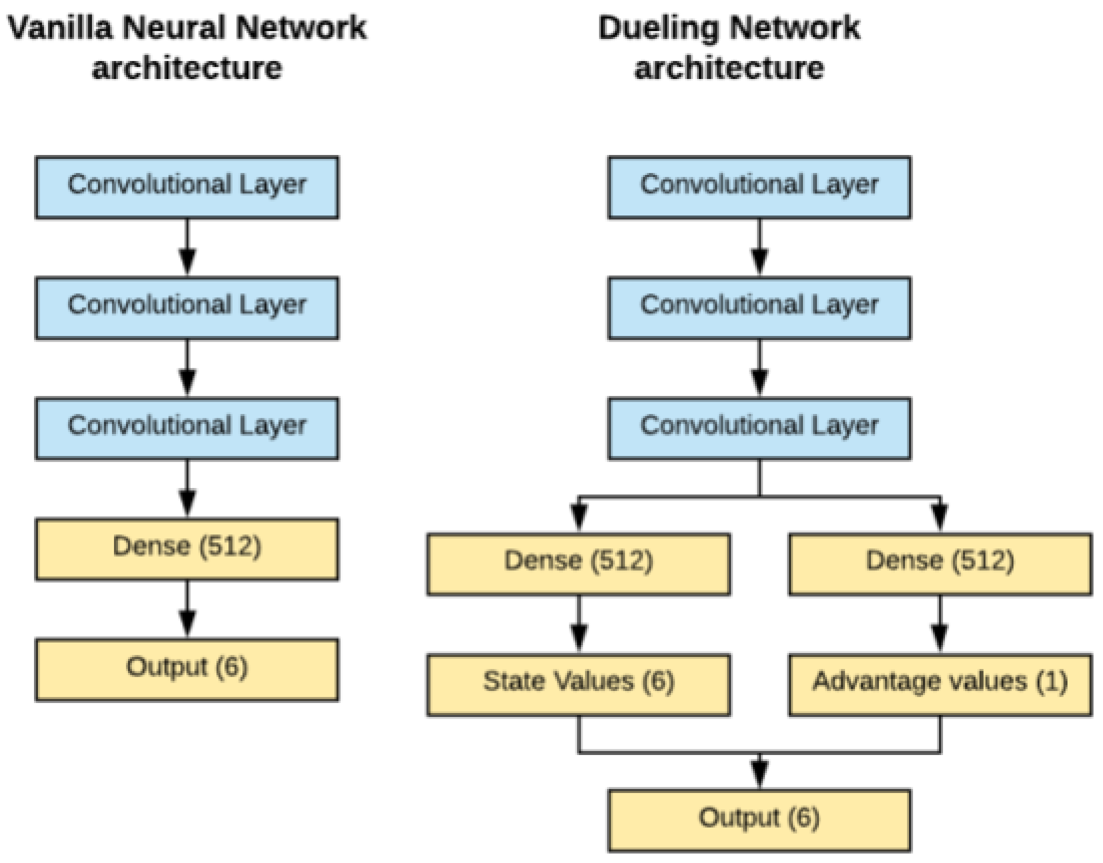

Figure 4 | Vanilla network vs a Dueling Network architecture.

In order to improve the overall performance, I'll also train the model using a Dueling Network architecture (Wang et al. 2015), an enhancement over the vanilla network introduced in 2015. This improvement consists of a network that uses 2 streams: one to estimate the state values, and one estimate the advantage values. Both streams get combined at the end to get the desired Q-values (Figure 4). The following image shows what it looks like when put into python code:

```python
class DuelingNet(ModelVanilla):

   def build_model(self):
       model = Sequential()
       input = Input((self.state_size))
       conv1 = Conv2D(filters=32, kernel_size=8,
                      padding='valid', activation='relu',
                      strides=(4,4), input_shape=self.state_size)(input)

       conv2 = Conv2D(filters=64, kernel_size=4,
                      padding='valid', activation='relu',
                      strides=(2,2))(conv1)

       conv3 = Conv2D(filters=64, kernel_size=3,
                      padding='same', activation='relu',
                      strides=(1,1))(conv2)

       flatten = Flatten()(conv3)

       fc1 = Dense(512, activation='relu')(flatten)

       # Build the dueling net
       x = Dense(self.action_size + 1, activation='linear')(fc1)

       policy = Lambda(lambda i: K.expand_dims(i[:,0],-1) + i[:,1:] \
                        - K.mean(i[:,1:],   keepdims=True),
                        output_shape=(self.action_size,))(x)

       model = Model(input, policy)

       # compile the model
       model.compile(loss='mean_squared_error',
                     optimizer=RMSprop(lr=self.learning_rate,
                                       rho=0.95,
                                       epsilon=0.01),
                     metrics=['accuracy'])

       return model
```

Training the neural network requires a lot of data. There are situations in which the network weights can diverge due to the high correlation between actions and states, which can result in a very unstable and inefficient policy. To overcome these challenges, two techniques are used: Experience Replay and Fixed Q-Targets.

Experience Replay consists on saving the *state, reward, done* tuples in a replay buffer and re-use them later to train the agent. These memories can be sampled into batches and feed them to the network at once. Experience Replay can help by recalling some unusual states as well as preventing the agent to learn from unwanted sequences by shuffling the batches.

```python
class ReplayBuffer:
 def __init__(self, buffer_size, batch_size):
   """
   Params
   ======
     buffer_size: maximum size of buffer
     batch_size: size of each training batch
   """
   self.memory = deque(maxlen=buffer_size)  # internal memory (deque)
   self.batch_size = batch_size
   self.experience = namedtuple("Experience", field_names=["state", "action",
"reward", "next_state", "done"])

 def add(self, state, action, reward, next_state, done):
   # Add a new experience to memory.
   e = self.experience(state, action, reward, next_state, done)
   self.memory.append(e)

 def sample(self, batch_size=64):
   # Randomly sample a batch of experiences from memory.
   return random.sample(self.memory, k=self.batch_size)

 def __len__(self):
   # Return the current size of internal memory.
   return len(self.memory)
```

Fixed Q-Targets consists of fixing the function parameters used to generate the target . It can be implemented by copying w into w^{-} , and use it to generate targets while changing w for a certain number of learning steps. Then, the value of w^{-} gets updated to the value of w, the agent learns for a number of steps, and so on. This technique helps decouple the parameters. In the following implementation, the fixed model updates its weights every c_steps.

```python
def update_model_f(self):
   """
   Updates fixed weights of the 'model_f' every C steps.
   """
   if self.c_steps_counter == self.c_steps:
     self.model_f.set_weights(self.model.get_weights())
     self.c_steps_counter = 0
   else:
     self.c_steps_counter += 1
```

Algorithm. There are two main processes that are interleaved in this algorithm:

The first one samples the environment by performing actions and store away the experienced tuples in a replay memory.

```
def act(self, state):
   """
   Choose action using greedy policy
   """
   if np.random.rand() <= self.epsilon:
     # select random action
     return np.random.choice(np.arange(self.action_size))

   # select greedy action
   act_values = self.model.predict(state)
   return np.argmax(act_values[0])
```

The other one is where a small batch of tuples is selected randomly from the memory and uses the gradient descent step to learn from that batch. These two processes are not directly dependent on each other.

```
def learn(self):
   """
   1. Obtain a batch of random samples
   2. Set target y = r + gamma*max q(s,a,w-)
   3. Update weights (forward pass -> Gradient descent)
   4. Update fixed w after C steps w- <- w
   """
   minibatch = self.memory.sample(self.batch_size)

   for state, action, reward, next_state, done in minibatch:
     target = reward
     if not done:
       target = reward + self.gamma * np.amax(self.model_f.predict(next_state)[0])

     # make the agent to approximately map
     # the current state to future discounted reward
     target_f = self.model.predict(state)
     target_f[0][action] = target

     self.model.fit(state, target_f, epochs=1, verbose=0)
     self.update_model_f()

   if self.epsilon > self.epsilon_min:
     self.epsilon *= self.epsilon_decay

 def save_weights(self, filename="best_model"):
   self.model.save_weights('saved_models/'+ filename + '.h5')

 def load_weights(self, filename):
   self.model.load_weights('saved_models/'+ filename + '.h5')
   self.model_f.load_weights('saved_models/'+ filename + '.h5')
   print("Loaded {}.h5".format(filename))
```

The rest of the algorithm is built to support the two steps. The algorithm starts by initializing a finite replay memory D with capacity N. Then, the parameters w for the action-value function $\hat{q}$ are initialized. To use the fixed Q-targets technique, w^- will be initialized as w, and will be periodically updated later on. Finally the necessary preprocessing for the input images will be implemented, followed by a stacking of M frames together to create a sequence.

Deep Q-learning with Experience Replay

```
Initialize replay memory $D$ 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 $\varphi_1 = \varphi(s_1)$
    for t = 1, $T$ do
        With probability $\varepsilon$ select a random action $a_t$
        otherwise select $a_t = max_a Q*(\varphi(s_t), a; \theta)$
        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 $\varphi_{t+1} = \varphi(s_{t+1})$
        Store transition $(\varphi_t, a_t, r_t, \varphi_{t+1})$ in $D$
        Sample random minibatch of transitions $(\varphi_j, a_j, r_j, \varphi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma max_{a'} Q(\phi_{j+1}, a'; \phi), & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\varphi_j, a_j; \theta))^2$ according to equation 3
    end for
end for
```

Figure 5 | DQN Algorithm

Refinement

The initial model was tested on OpenAI's *cartpole game*, to make sure it converged. The problem that atari games present is that it can take up to several days to train an agent that can actually play the game, that is why an initial version of the model was tested on the simpler *cartpole game*, which converged after 15.000 episodes.

The model was first trained on Space Invaders for over 20000 episodes. When looking at the performance over time (episodes), the agent seemed to have a steady learning curve. However, it showed that this learning curve was not flattening towards the end, which means that there was a possibility that it could still learn more if trained for longer. Finally the agent was trained for 40.000 episodes.

The first implementation was using a simple neural network that took the images and score as input, and gave one of six actions as an output. To improve the performance I implemented a Dueling Network which had better scores after training the algorithm for the same amount of time.

At the beginning of the project I decided to implement my own pre processing techniques, by using libraries like OpenCV to crop, downscale and gray scale the input images. These transformations can be computationally expensive, and it was causing the computer to crash in the early stages of training. Finally I decided to use OpenAI's env wrappers for Atari games, which have been optimized for this purpose.

Results

Model Evaluation and Validation

During training, the scores were saved to see how the agent performed over time. The final architecture and hyperparameters were chosen because they performed best compared to other combinations:

- The first convolutional layer has 32 filters with a kernel size of 8x8, a stride of 4x4 and a ReLU activation function.
- The second convolutional layer has 64 filters with a kernel size of 4x4, a stride of 2x2 and a ReLU activation function.
- The third convolutional layer has 64 filters with a kernel size of 2x2, a stride of 1x1 and a ReLU activation function.
- The first dense layer has two streams (dueling network): both streams have an input size of 512 and a ReLU activation function.
    - The first stream has an output of 6, one for each possible action.
    - The second stream outputs the action value.
- Both streams get combined for a final dense layer, with an output of 6.
- The loss function used is the Mean Square Error
- The optimizer used is RMSprop with a learning rate of 0.00025, rho of 0.95 and epsilon of 0.01.

The hyperparameters for the agent have been chosen according to those found in research papers as well as those that have proven to work best during training:

- A buffer memory of 100.000.
- An update rate for the Q-target of 10.000 steps.
- Batch size of 64.
- Discount rate of 0.99.
- Minimum exploration rate of 0.1 with a decay of 0.995.

The model can be validated by observing the performance of the agent over time. A positive change in the learning curve proves that the algorithm is learning from experience. To visualize this, we can visualize the average performance of the agent every 10, 50 and 100 episodes.

Justification

Three different benchmarks were established at the beginning of this project:

- The first benchmark was the average performance of an agent acting randomly over 500 episodes. The average result was 135.
- The second benchmark was the average performance of an agent initialized with random weights and no training. The average result was 140. Very similar to the random agent.
- The final benchmark was the average of a human player over 100 games. The average score was 520.

After training the agent for 40.000 episodes, it has shown an average performance of 190, which is a slight improvement over the random agent, and yet very far from the results in the paper. The algorithm has shown that it can learn to play Space Invaders, however there is a lot of room for improvement.

Conclusion

Free-Form Visualization

After training the agent for 40.000 episodes we see that there is a learning curve up to the first 20.000 episodes but then it begins to decrease. If trained for longer the agent might be able to keep learning and get a new learning curve, or it might stay where it is.

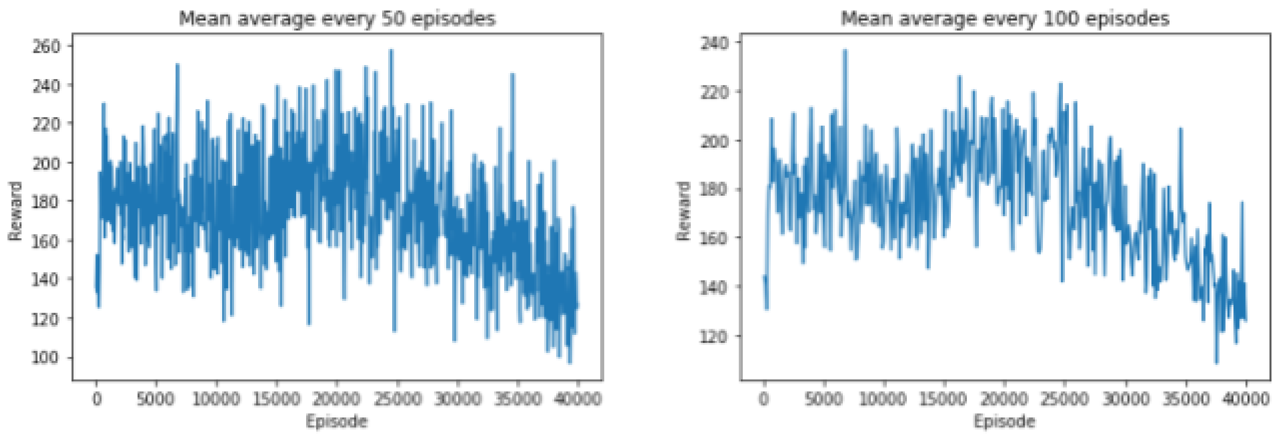

Figure 6 | Average rewards after training the agent for 40.000 episodes.

When evaluating the model at its best performance (around 20.000 steps), the average score is 190. This is barely 50 points higher than the benchmark where the agent acts randomly. However this is an indicator that the agent has the capacity to learn, and if improvements were made, the agent might be able to play better.

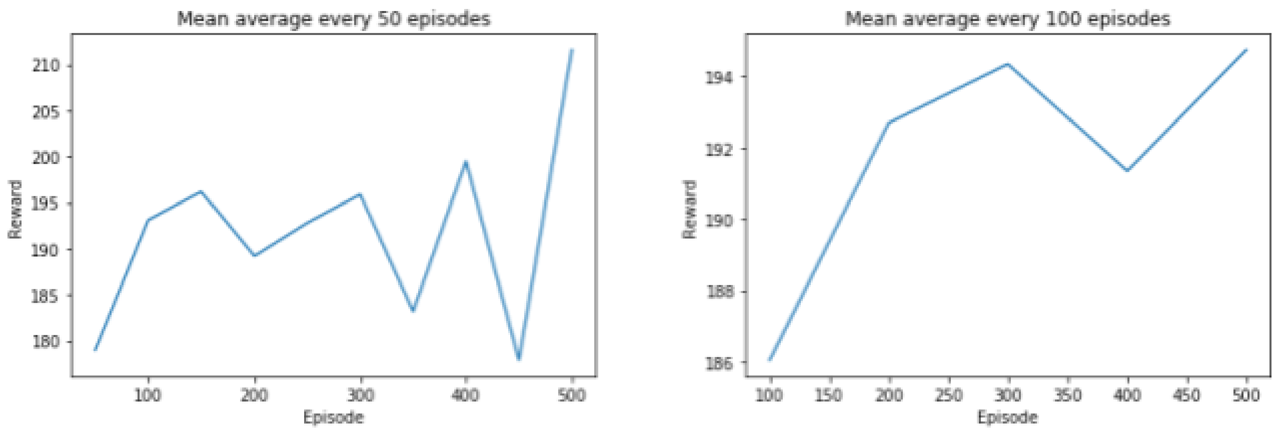

Figure 7 | Evaluating the trained model.

Even with such a low average score after training, we can see the agent making (or at least it seems that way) deliberate actions to avoid getting hit by the enemy. In figure 8 the agent takes cover to avoid getting hit by the martians. This type of behaviour wasn't present when observing the agent acting randomly.

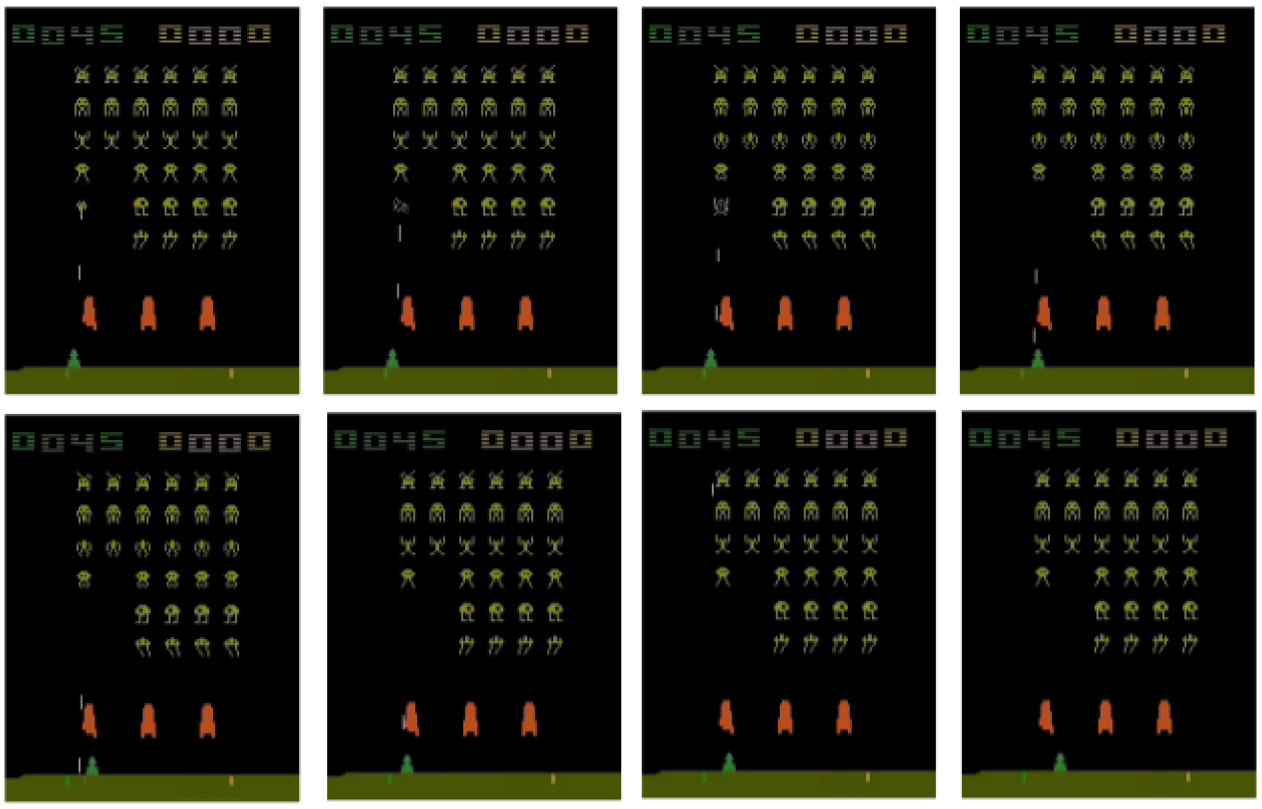

Figure 8 | From left to right: sequence of 8 frames where we see the agent taking cover to dodge an incoming bullet.

Reflection

The initial challenge was to write the program in an efficient way that was easy to read and reuse. I've decided to create different classes (agent, model, training...) for different tasks, trying to separate concerns thus making the code more intuitive and reusable.

The main challenge when working with a complex RL problem is that it takes a long time of training before it shows any results, thus making it very time consuming to make any changes/improvements in the algorithm. I started the project by using simple code that would resolve a simpler reinforcement learning problem (Open AI's GYM pole). Once the algorithm proved to converge to an optimal solution, I modified the code to tackle the current, and more complex, problem. This helped make sure that the algorithm was actually working without having to wait too long for it to converge.

The most difficult challenge has been training the algorithm. After doing some research, I came to realize that this algorithm can take up to 90 hours of training depending on the GPU. As I mentioned before this made making changes and improvements very difficult, especially on a local machine, which is why I ended up training the algorithm using and EC2 instance from Amazon Web services.

This has been a difficult project, however it has also been very rewarding. It is really fascinating to build first hand a reinforcement learning agent that can actually learn a task just as a human would. This has also been the first time that I use python in an object oriented way, as well as being the first time that I organize the code for a Deep Learning project this way, which has been a great experience.

Improvement

This project has been built using a DQN-Network, which is a value based method. This method tries to find the optimal value function, where the policy is implicitly defined by that value function. However the results could possibly be improved by using a policy based method like an actor-critic method, which tries to find the optimal policy directly without using a value function. These types of methods are especially useful when using stochastic policies.

Another feature that could have helped improve the algorithm is Prioritized Experience Replay. This was a feature that I tried implementing but I wasn't successful. By implementing this feature the agent would have kept the most important memories in the buffer replay and deleted the less relevant ones, making the learning process more efficient and better.

Finally, the last improvement that could be made to the project would be further training times. If I had access to more powerful GPUs, I could have tried to train the agent for longer to see if it converged to a more optimal score.

References

J.C.H Watkins, C. and Dayan, P. (1992). Q-Learning. citeseerx.ist.psu.edu. [online] Available at: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.80.7501&rep=rep1&type=pdf [accessed 21 Aug. 2019]

Mnih, V., Kavukcuoglu, K., Silver, D., A. Rusu, A., Veness, J., G. Bellemare, M., Graves, A., Riedmiller, M., K. Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, [online] (518). Available at: https://www.nature.com/articles/nature14236 [Accessed 6 Jun. 2019].

Minh, V., Kavukcuoglu, K., Silver D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller M. (2013). Playing Atari with deep reinforcement learning. *arXiv*. [online] Available at: https://arxiv.org/pdf/1312.5602v1.pdf [Accessed 5 Jun. 2019].

Schaul, T. Quan, J., Antonoglou, I. and Silver, D. (2015). Prioritized Experience Replay. *arXiv*. [online] Available at: https://arxiv.org/abs/1511.05952 [Accessed 5 Jun. 2019].

Sutton, R. and Barto, A. (2018). *Reinforcement learning*. 2nd ed. Cambridge, Massachusetts: The MIT Press, pp.47-68.

Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., and de Freitas, N. (2015). Dueling Network Architectures for Deep Reinforcement Learning. *arXiv*. [online] Available at: https://arxiv.org/abs/1511.06581 [Accessed 5 Jun. 2019].