Iñigo Alonso
November 15th, 2017

Machine Learning Engineer Nanodegree

Capstone Project

I. Definition

Project Overview

Back in 2013, DeepMind Technologies, a British artificial intelligence company, published a paper showing how an AI agent was able to learn to play Atari only by having the screen raw pixels as input and with no prior information about playing games. [1] Later on, this company was purchased by Google and became famous after develop an AI agent capable of wining 4 out of 5 games at Go to the 18-time world champion Lee Sedol. [2] Finally, on October 18th 2017, announced Alpha Go Zero, a set of algorithms capable of achieve superhuman performance with no human input. This means that a model could be trained to solve challenging problems from scratch leaving any kind of human bias behind.
While this is the cutting edge of research on this topic, self learning agents will become essential in order to aid people to solve complex problems even the ones were humans tend to get stuck in.

In this project I have developed a machine learning agent capable of solving the Open AI gym's public contest for CartPole game. This contest consists on balancing a pole inside a virtual environment and getting an average score of 195 over 100 consecutive trials. In order to achieve this, the agent has learn to react to every situation through experience.

Problem Statement

This project is focused on developing an agent capable of learning from experience to keep a pole inside a virtual environment steady, improving over time the amount of time it keeps it without falling. The experience will be composed by all the actions this agent takes at each stage of every game it plays. The learning performance will be measured by the amount of time it takes before the pole falls. The more time, the better the agent will perform.

This agent is developed using an OpenAI gym's game called CartPole as the environment. In this game a pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every time-step that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center [3].

This agent has been developed using a reinforcement learning technique called deep Q-learning. This agent uses a neural network (NN) to learn the main state-action policy which gives it the ability to decide which action should it take at every state of the environment in order to keep the pole steady. This NN will be tuned through the experience gained during the learning process of this agent were its performance improvement is measured by the increase on the amount of time it takes before the game meets the requirements to end.

Metrics

The main metric used to measure the performance of this agent is **the average score over 100 consecutive action-steps**. This score is given by the amount of time-steps taken before a game met the requirements to end. This will provide information on how close the actual state-action policy is to the optimal policy (were every action taken leads to keep the pole steady and the cart within ±2.4 units). However, metrics such as the **number of training episodes needed for the model to meet the benchmark** gives information on how well the neural network architecture is defined. Considering a good NN architecture able to beat the benchmark in a lower amount of training episodes.

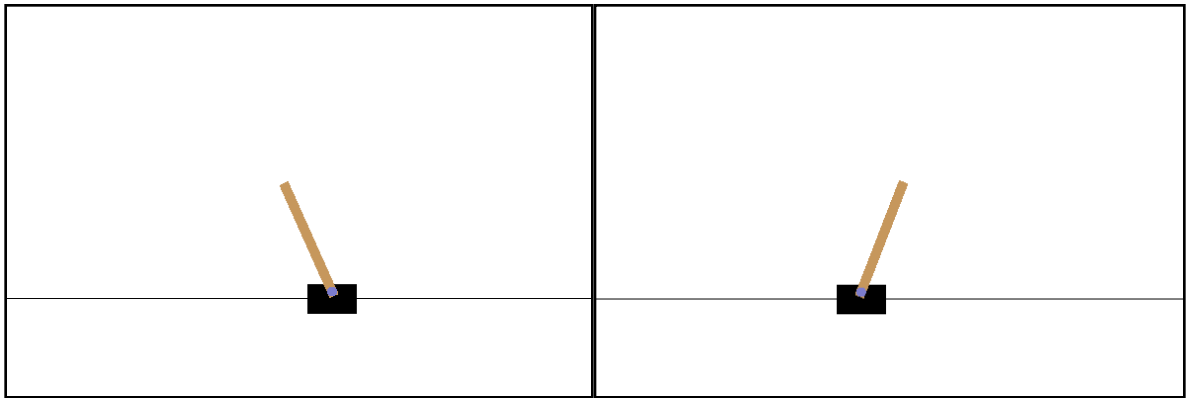
II. Analysis

Data Exploration

As this is a Reinforcement Learning problem no dataset will be provided to the model. Instead, the different states of the agent are determined by the pole position values provided by the environment (gym framework) after each action step. These values consist on the angle of the pole, and position of the cart.

Specifically, these values are:
Position of the cart: Represents the distance between the cart and the center of the environment in the horizontal axis. The range of this value goes from 2.4 to -2.4
Velocity of the cart: This value shows the speed of the cart. It can be negative if the cart moves backwards and it gets increased over consecutive movements in the same direction.
Angle of the pole: Represents the angle of the pole in radians and it can be found between the values 0.26 and -0.26
Rotation rate of the pole: This last value shows the speed in which the pole rotates. This can also be negative if the pole rotates in a negative angle.

The following examples shows how each pole position relates to each value set:



pole_01

Position of the cart: 0.27216694
Velocity of the cart: 2.36708376
Angle of the pole: -0.42092962
Rotation rate of the pole: -3.88865888

pole_02

Position of the cart: -0.23797936
Velocity of the cart: -2.1831041
Angle of the pole: 0.39488636
Rotation rate of the pole: 3.64952598

These four values contain all the spatial information needed to represent a unique state of this environment. However, in a future iteration of this project these values will be replaced by image frames of the simulator screen. This change will be essential in order to make a more realistic and real-world oriented approach. However, due to time and hardware limitations the scope of this project will be limited just to simulator provided values.

Algorithms and Techniques

Solving this problem will require the use of reinforcement learning, in particular deep Q-learning.

Why Q-learning?
Q-learning is a model-free reinforcement learning technique. It can be used to find an optimal action-selection policy for any given (finite) Markov decision process (MDP). [4] For any finite MDP, Q-learning is able to find an optimal policy, in the sense that the expected value of the total reward return over all successive steps, starting from the current state, is the maximum achievable. [4] This becomes very useful for this problem because usually, in games, actions don't have a direct reward. Instead, often the only reward comes after a long chain of actions. In this case, the agent must know that the expected future reward will be higher if it takes a set of actions that leads to a higher score of the game as the pole stays balanced longer.

The Neural Network
Considering that pole's position values would become in an infinite number of states, we can't use a Q-learner with an action-state mapped table. In order to estimate the utility at every state, the Q-learning table must be replaced by a neural network (NN) capable of learning and becoming the policy function. This technique is called Deep Q-learning. Given some state, the NN must output the Q value of every action in that state. In this case right or left.

Benchmark

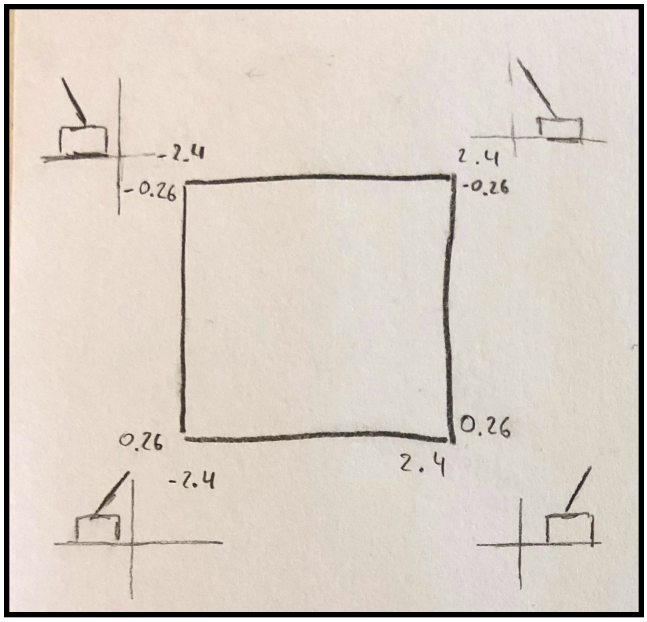
Open AI gym offers a public contest for everyone who wants to develop an AI agent capable of balancing the pole in CartPole game. They consider the game "solved" as getting an average reward of 195.0 over 100 consecutive trials. The score is defined by the amount of stages the game runs before the pole falls. At every stage the environment stops and waits for the agent to take an action, once the agent takes an action a new stage of the game is generated and the environment provides the agent with the consequences of the action taken, in this case the position of the pole after such action. I believe this is a strong benchmark as it sets the bar to consider an agent capable of learning how to move a track in order to balance the pole that stands over it. Therefore, I will consider this as a benchmark for this project.

III. Methodology

Data Processing and Visualization

As it was exposed in the Data Exploration chapter, this reinforcement learning problem has no dataset, therefore it will be no preprocessing. However, this section will address an important aspect of the data exploration of this project, the q-table visualization. As I mentioned previously, deep q-learning technique consists on replacing the state-action q-table with a neural network capable of learn and generalize a game policy. This function will be able to provide an approximate value of q for every action given a state. To see how this function evolves over time and see what the policy looks like, I have developed a function that creates a greyscale image representing the different q values for each action at every state. Every state is contained in a four dimensional space defined by the position of the cart, velocity of the cart, angle of the pole and rotation rate of the pole. As an image can only represent two dimensional matrixes, I chose what I think are the two most representative/crucial values of a state, position of the cart and angle of the pole, to define how the function represents its decision boundaries between actions.

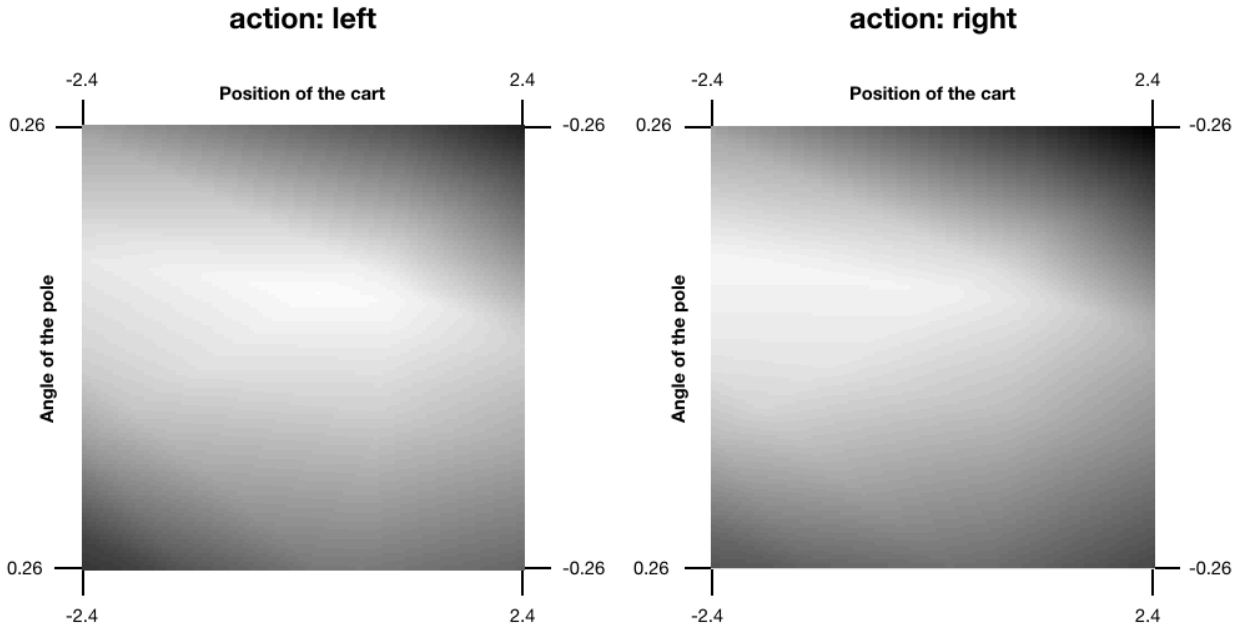
These grayscale image representations of the q-table rely on how the q-learning function works in order to propagate future rewards through the table. As the function provided by the neural network is aimed to provide the same functionality as the q-table, lets assume for now on (for a simpler explanation) that this function works like a table. This table's axis will be determined by discretized values inside the occurrence boundary of two values of the state. For example in this case, X will be the position of the cart with a range of values between -2.4 and 2.4 and Y will be the angle of the pole defined by a range of values between -0.26 and 0.26.



Q-table corner value relation representation

Every cell will contain the q value of both "right" and "left" actions. Let's separate these action values in two tables. Now, if we represent high q values in a color close to white and low values close to black (being white the highest value of the table, and black the lowest). We will see during the training process how the black color spreads though the table emanating from the points where the combination of x and y leads to a negative reward (in this case the

premature end of the game). By contrast, the combination of x and y far from failure points gets brighter over time, as we reward actions with a positive reward when they don't lead to the end of the game. This way, by the time when training process is finished, we end up with images similar to the following ones.



This way we can clearly see which action is prone to be taken at each value of these two dimensions of the state.

Implementation

In this section I will speak about the final refined implementation. Notice that this is a slightly different model form the one planed on the section "Algorithms and Techniques" as that model failed to generalize and learn properly a good game policy preventing it from meeting the benchmark. The process of fixing the main issue of the implementation is extensively described in the next section "Refinement".

The main asset of this implementation is, of course, the agent that takes actions and learns over time. I believe that the easiest way to learn and explain how the agent was implemented and works is by describing every function within it.

Agent parameters:
All the parameters of the agent are set during the initialization. While these parameters have a fixed value in the final implementation, they were variables during the development and refinement process in order to find the ones providing the highest performance. Although some new values were tested, most of them were finally set by the ones used by convection.

The learning rate: Rate in which the loss function corrects old values to the new ones during the training step of the neural network. This rate is set as 0.001 in the final solution.

Epsilon: Defines initial exploitation/exploration ratio. This value (1.0 in the final implementation) will be decreased by a rate also defined in the parameters (0.005 in the implementation) until it reaches a minimum value of 0.01
Gamma: Also known as discount rate, defines the rate by which future q values propagate across previous states. I set it as 0.9 after several sets of trial and error.
Memory: A queue where a list of experiences in the form of (state, action, reward, next_state, done) are stored in order to provide the agent with a long-term memory. The amount of experiences taken during the learning process are also part of the agent parameters and were finally set as 64 for the final implementation.

Agent's Neural Network:
The neural network of the agent allows the agent to learn and generalize the policy that will let it know the approximate q value of every action at each estate of the game in an infinite state environment. The NN of this agent was build using Keras framework and it has the following architecture:
First one input layer with four nodes, one for each state value [position of cart, velocity of cart, angle of pole, rotation rate of pole]. This layer is fully connected to two 16 node fully connected hidden layers with Relu activation functions. For the activation functions I tested both hyperbolic tangent and relu. The results were similar, so I ended up choosing relu because theoretically improves the training without sacrificing much accuracy and resolves vanishing gradient problem [5]. Finally, the last hidden layer fully connects with the two node output layer with a linear activation function. This function converts the 0 or bigger values from the hidden layers into the proper q values for each action.

During the design process I thought 16 node hidden layers would be enough in order to define the policy function. However, after I developed the script to generate a grey-scale image representing the q-table function, I decided to add 8 more nodes to the layers as I saw 16 ones struggle to get the complex decision boundary needed to implement correctly this function.

The loss function for this network is MSE (mean squared error). This was chosen because it allows the optimizer to penalize more large error values and treat the negative values same as the positive ones.

Finally, as the optimizer of this network I chose the adam optimizer. This optimizer lets to the learning rate which is the last part of the Q-learning function we needed to implement the q-learning table functionality on a NN. In the same way as most of the parameters of this solution, other optimizer was tested (adagrad) which due to its lower performance was not selected for the final version.

Agent taking actions:
This is a simple function were the action for a given state is returned. It balances between exploration and exploitation using the current value of epsilon. The higher epsilon is, more probabilities will be of taking a random action. Epsilon gets decreased over time until it gets a value were most of the actions taken are form the agent knowledge. Which means it will make the NN predicts the q values of the two possible actions for the current state and will return the one with the highest value.

Memory:
In order to assist during the learning process, the agent stores all the experiences it has gone through. An experience consist on a set of state, action, reward and next state. Storing passed experiences became crucial in order to make the agent remember in a long tern. Without this memory, the agent would be overfitted to the latest experiences making it more unreliable,

inconsistent and unable to generalize and converge to the optimal policy. This was the main problem I had to address in order to meet the benchmark therefore it deserves a deep analysis.

The initial design made the model learn after every experience. This made the agent perform well during the first 200 games but beyond that, it suddenly started to get fail. Its behavior became unreliable as it had 4 to 10 consecutive good games with scores between 150 and 350 followed by another 4 to 10 consecutive bad games with really poor scores between 12 and 20.

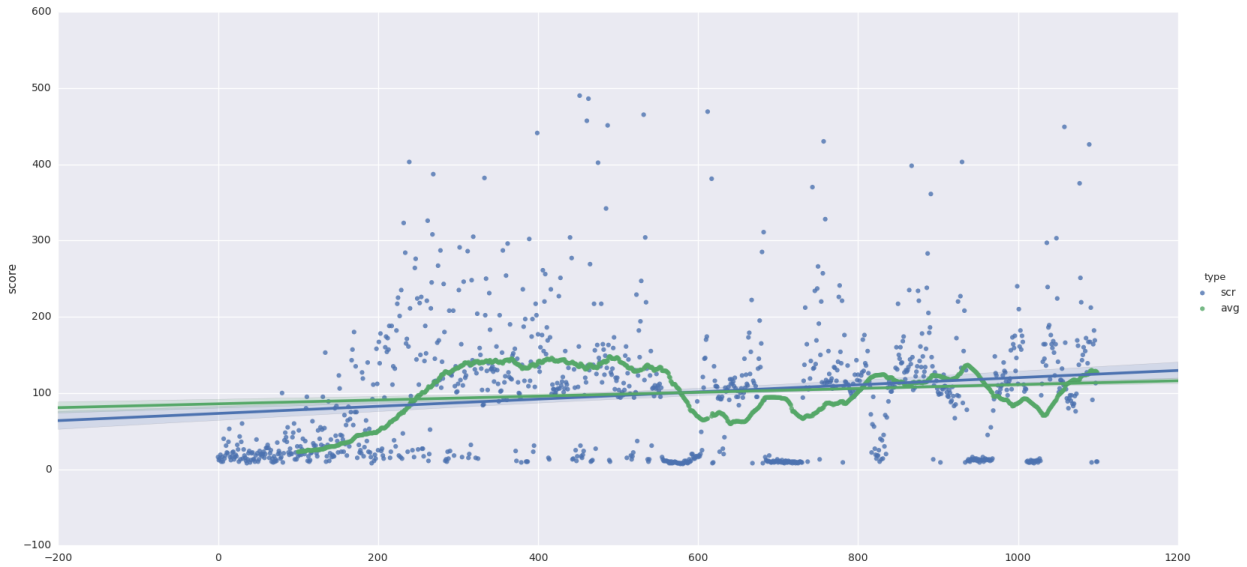


Chart 1. Agent learning progress with no long-term memory

This chart shows a common behavior of an agent with no long-term memory. The X axis represents each game and the Y axis represents that game's score. The green dots shows the average score of the 100 previous games. As we can see, how the agent reaches its peak score average in the 300th game followed by a set of high and low score streaks. Furthermore, the behavior of a model with the same parameters was inconsistent between executions as each execution featured different learning curves and average scores. This issue prevent me of seeing the consequence of parameter changes as the difference between executions of a model with the same parameters was higher than the difference between agents with other parameters. Some of the executions met the benchmark at some period of its learning process. I saved the weights of the model at those periods so I could export that novel behavior before it gets lost again but the agents with these weights couldn't get a score higher than 40 in different executions. This was neither a weight initialization problem as I tried to solve it changing its initialization to zeros and random uniform generated weights. Finally, I turned to the solutions of the winers of the OpenAI's CartPole challenge, in particular n1try's algorithm [6], were I found the idea of storing experiences in order to make the model generalize and not get stack to the recent ones. With this technique the model was trained with a random set of experiences after each game making the agent relearn experiences not related with the previous game instead of learning each experience after experiencing it. This was a breakthrough in how the agent behave making it more consistent over time.

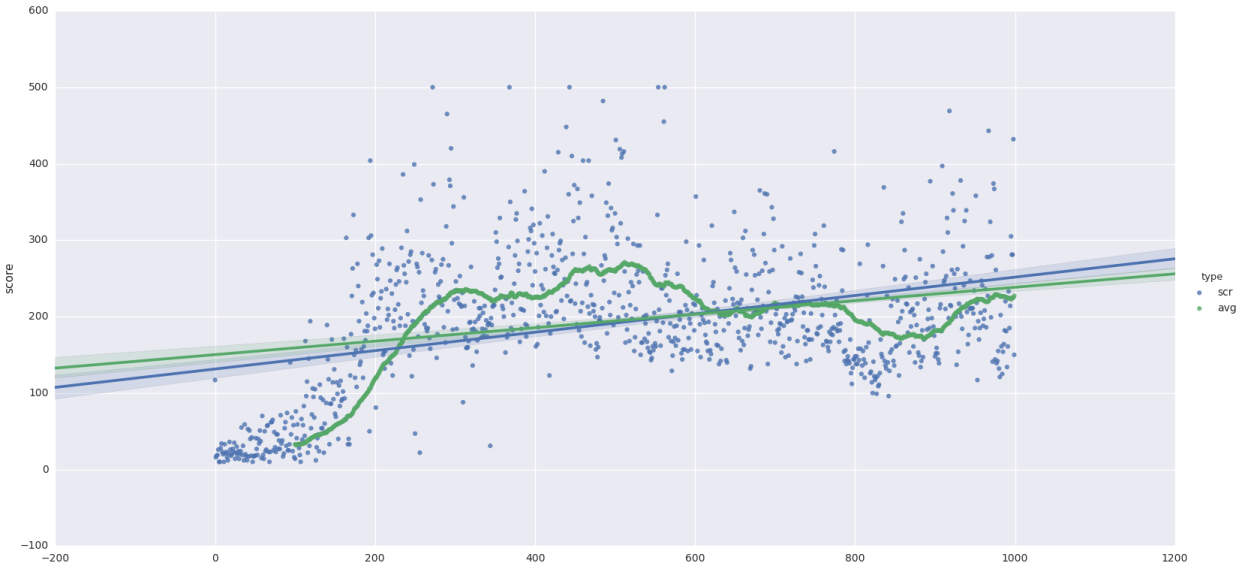


Chart 2. Agent learning progress with long-term memory

Learning process:
During the learning process, the agent learns from memory and trains the model to provide the agent with long-term memory. To do this, it gets a random set of experiences and for each one gets the target value T of the loss formula (in this case MSE) following the q-learning function.

loss = (T - P)^2
where:
P = the Q value predicted for the current state
T = the utility of the next state (max Q value for the next state)
T = reward + gamma * max(model.predict(next_state))

Therefore, after getting all the target values for each randomly selected state, the neural network is retrained in order to propagate the q value backwards through the state space of the q-function the NN is trying to replicate.

Executing the agent
Once the agent is implemented, making it play and learn is quite simple. First, OpenAI's gym environment CartPole is loaded and the first state is obtained. The agent provides with an action for that state and it is executed on the environment. Next, the environment provides the next state, the reward and a boolean variable'0' saying if the game met the requirements to end. This set of variables are stored in the agent's memory for learning after the game ends. This sequence is repeated until the game ends. Once a game ends, the learning function of the agent is executed, the epsilon variable is decreased and the environment reset to repeat this process in a new game.

Refinement

Besides the main improvement to the model related to the long term memory already addressed in the previous section, further improvements to the model were made in terms of parameter tuning. While the initial values were set according to theoretical knowledge in this topic and convention, the decision behind every change was supported by the performance improvement of such changes on the learning process and the final result.

Neural Network related parameters

Hidden layers: The initial implementation featured a network with two hidden layers. After measuring the performance improvements of using three layers I ended up setting 2. Three layers led to slower learning and a lack of performance in some executions.

Hidden layer size: With an initial size of 24 nodes for each hidden layer, executions with 16, 24 and 32 showed that 16 nodes were enough to define the q-learning function needed for this problem. Therefore 16 node hidden layers were used for the final solution.

Optimizer: Both initial and final solutions uses Adam optimizer on its neural networks. Adagrad optimizer was tested but struggled to generalize the desired function leading it to have an inconsistent behavior during the training process.

Reinforcement Learning related parameters

Epsilon decay rate: The initial rate in which the epsilon value was decreased was set to 0.001 but after several test with a set of rates ranging between 0.0008 and 0.01 I found a rate of 0.005 was the optimal rate for this solution.

Gamma: The initial value of gamma selected for the first implementation was 0.95 as I wanted to give a high level of future reward propagation. However after several tests with different gammas ranging between 0.98 and 0.8 I ended up choosing a gamma of 0.9 due to its performance.

Memory batch size: As I didn't have any intuition on which size the long term memory batch size for learning should have, I set a value of 64 according to the source where I found the idea of implementing this memory. However, after several tests I found out that 32 was a much proper value making the model learn from less experiences in order to get less overfitted.

Reward for a premature ending: As the environment didn't provide me with a punishment reward when the game was prematurely finished, I decided to set this as -10 in the first implementation but after some refinement I found -20 was more appropriate for a better performance of the model.

The difference on learning process results between the initial solution and the final one can be seen in the Chart 2 and Chart 3. Figure X shows the results of the firs working model before refinement and Figure Y shows the results with the final parameters.

IV. Results

Model Evaluation, Validation and Justification

With the described architecture and the refined parameters, the agent shows the following learning progress:

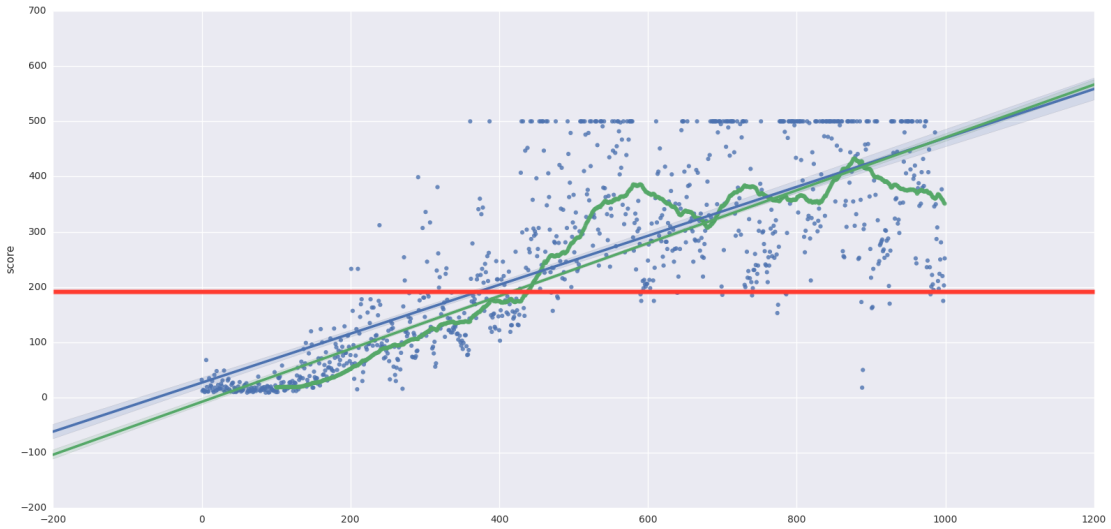


Chart 3. Learning progress of the final implementation

As it was mentioned before, the X axis represents each game and the Y axis represents that game's score. The green dots shows the average score of the 100 previous games. As we can see, the learning progress has a positive trend and it meets the benchmark from the game 440 on. The model keeps improving until it reaches its maximum peak at the game 890 with an average score of 443 over 500. This implementation features a consistent behavior over time obtaining an average score of twice the benchmark. However, models tend to behave differently, in most cases poorly, when are exported. Therefore, after exporting the weights of the neural network as they were at the game number 1000 and loaded on a non learning agent I obtained the following behavior:

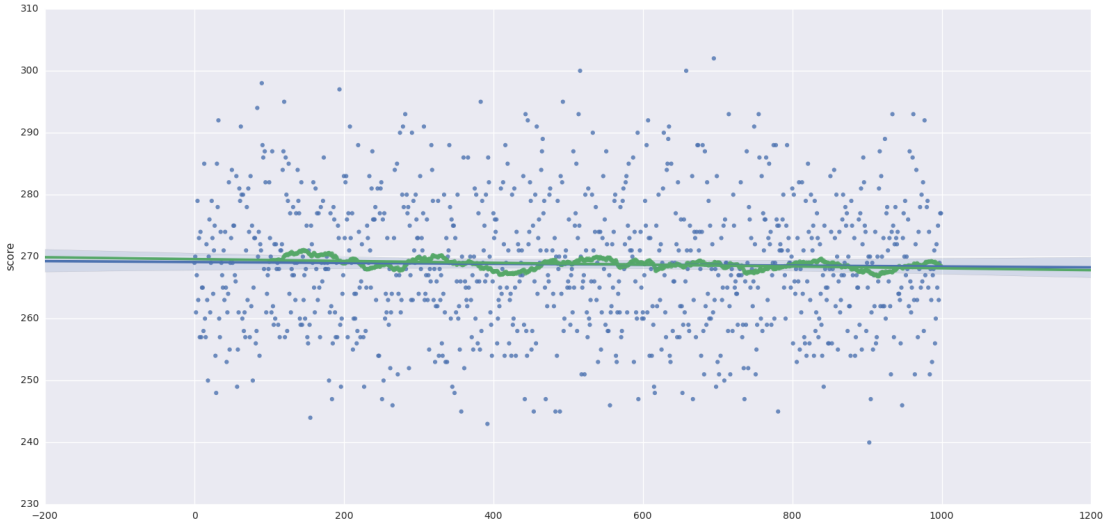
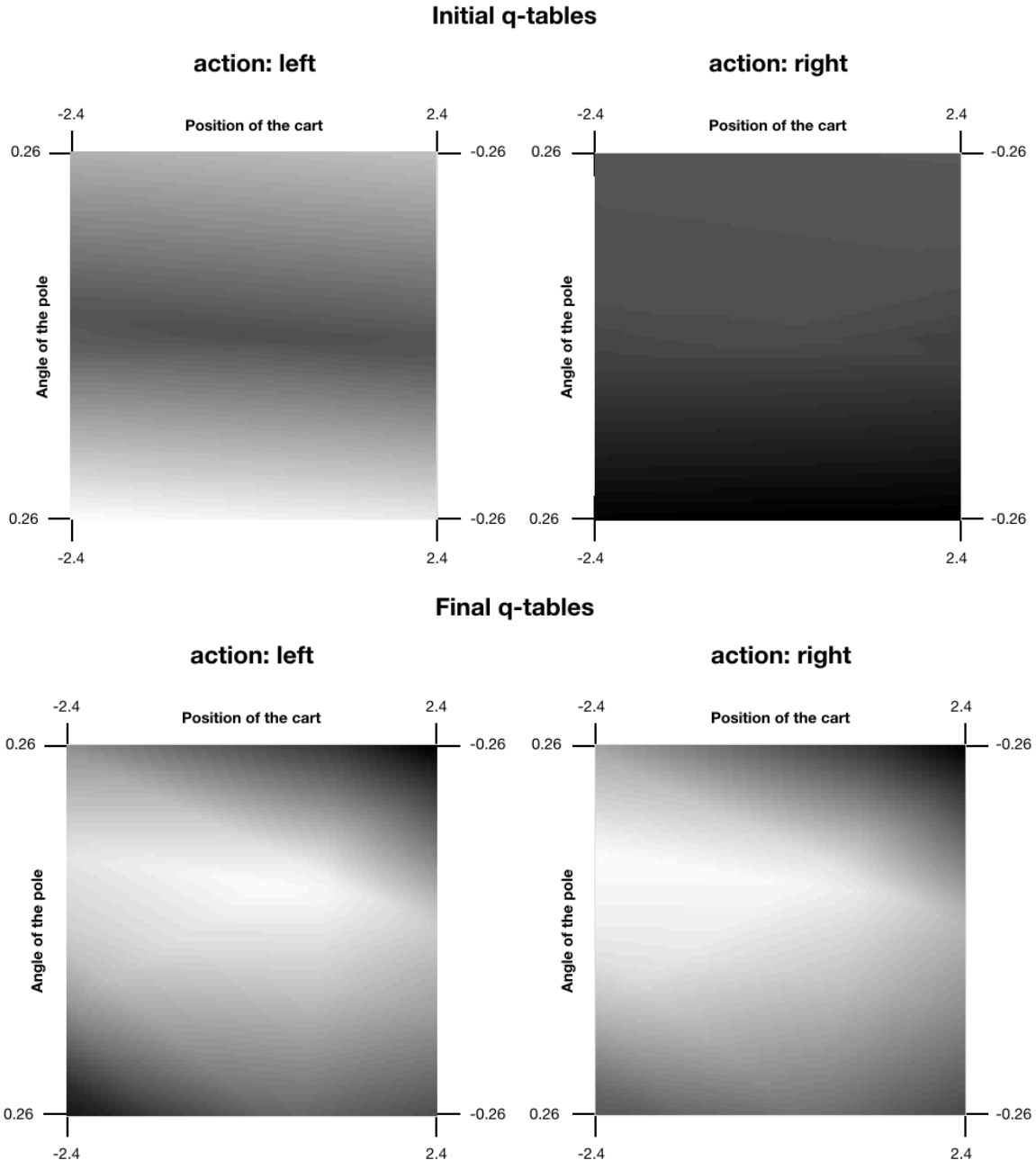


Chart 4. Non-learning agent behavior with exported weights

Notice that the Y axis was expanded in order to obtain a more detailed view of the variations between scores. As we can see, the behavior is consistent over time and its score varies within a range of 63 steps. This means that the exported non learning agent performs consistently with an average score of 270, 75 steps higher than the benchmark. This small range of scores over 1000 consecutive games shows how robust and reliable this agent is.

In addition to the previous results, I also obtained the representation of the q-learning function which shows how a discretized q-table of the policy would look like as a grey scale image. This kind of representation is widely explained in the Data Processing section.



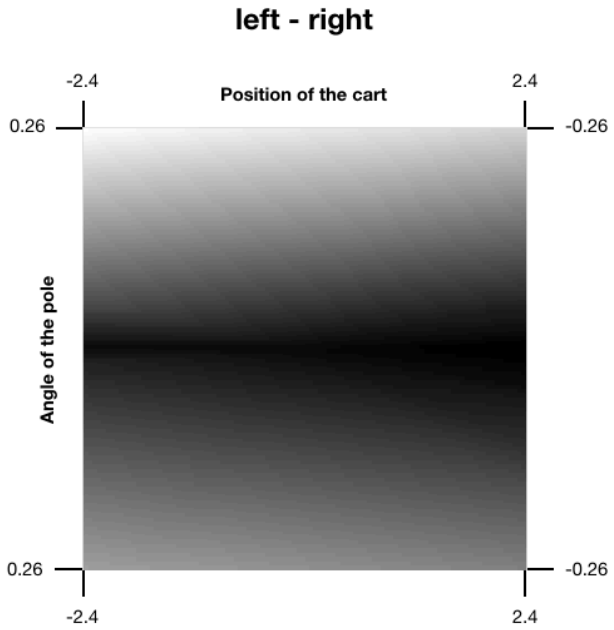
As we can see, the low values have spread across the table from the points that leads to failure. Both representations look similar, however, differences can be notice on the bottom left corner of action “right” were values are lower (darker) than the ones in the “left” action table. And by contrast, higher (brighter) values on the top left corner of action “left” can also be distinguished. While these tables are not completely correct, as they should have dark colors in opposite corners form one another, they provide a clear vision of were the model will be prompt to fail and should keep improving.

These results, specially the learning progress charts, shows how the implemented agent meets, with a significant margin, the requirements of this project. Furthermore, a “gif” image showing an execution of the implemented agent trying to balance the pole in real time is attached to the project submission. It can be seen how the cart executes the proper movements to keep the pole balanced as much time as possible. Together, these results lead me to consider this project as a success.

V. Conclusion

Free-Form Visualization

An interesting aspect of the implemented agent I would like to address before concluding this report is, once again, the representation of the q-learning function. We have seen how this grey-scale image represents the decision boundary between actions on different states that the agent follows during its decision taking process. This could also be interpreted as a visualization of which strategies the agent has figured out in order to get the higher score in the game. This strategies could make us understand better the problem the agent is trying to solve. For example, when we get the difference between the two matrixes for both actions, we can see the common areas were both actions are equally viable and those were the decision of which action to take is clear.



We can see how common areas (the brighter ones) are related to safe areas of the environment of both actions. Thus we can say that it clearly identifies were the safest areas of the environment are. While the problem this game addresses is quite simple and easy to explain without any sophisticated technique, more complex problems can be analyzed and explained thought this process.

Reflection

This project reports the development of a self learning agent capable of move a cart to keep balanced a pole inside a virtual environment using deep reinforcement learning techniques. To do this, first, I got familiarized with OpenAI's CartPole environment analyzing how its action-states were defined and related to the environment. Next I developed a deep reinforcement learning agent capable of learning from the environment and take actions that lead it to keep the pole without falling for at least an average of 195 action steps for 100 consecutive games.

This agent features a neural network built to be trained with states and q values in order to provide the function of a decision taking policy. The agent also uses a memory were it stores all the experiences it has gone though (a set of states, actions and consequences) in order randomly learn from them after every game during the training process. This particular feature becomes crucial in order to make the agent act in a consistent way as this allows the neural network to generalize a global policy that can make it take better actions at very different states of the game. This was the main problem I had to address in order to meet the benchmark as the first implementation of the agent made it learn after every action it took. This led it to forget the basic mechanics of the game leading it to behave inconsistently poorly after a few plays.

To assist in the implementation of this agent, a set of functions were also developed such as a q-learning function representation technique and a learning progress data visualization function. This functions provided the results that showed, after a parameter refinement process, what and how the final agent learned. This results showed also how the implemented solution was able to meet, with a significant margin, the requirements of this project.

However, meeting the requirements of the project does not make this solution suitable for a real world implementation as this was trained and executed in a very simple and controlled sandbox. A real world environment would be far more complex than the one used for this project. However, as I said in the beginning of this report, this project was not aimed to solve any real world project but to provide me with enough knowledge to develop self learning agents capable of solving complex problems such as games. Now I feel ready to improve this agent and to build new ones in order to solve real world problems.

Improvement

In the current implementation, the agent uses the state values provided by OpenAI's gym environment in order to define in which state is in. These four values contain all the spatial information needed to represent a unique state of this environment. These preprocessed values that gym provides are not easily obtainable in a real world application. This is why, in a future iteration of this project, these values will be replaced by image frames of the simulator screen. In order to do this we only should have to replace the actual neural network by a more complex convolutional neural network capable of detecting where the pole and the cart is, what is the position relative to each other and correlate this information with the reward it gets. This would lead to a more complex and time consuming learning process but also would become a more resilient and real world ready solution of this problem.

References

1. https://arxiv.org/abs/1312.5602
2. http://www.bbc.com/news/technology-35797102
3. https://gym.openai.com/envs/CartPole-v0/
4. http://users.isr.ist.utl.pt/~mtjspaan/readingGroup/ProofQlearning.pdf
5. http://www.cs.toronto.edu/~rgrosse/courses/csc321_2017/readings/L15%20Exploding%20and%20Vanishing%20Gradients.pdf
6. https://gym.openai.com/evaluations/eval_EIcM1ZBnQW2LBaFN6FY65g