Deep-Q-Network implementation on Atari 2600 Space Invaders

Abstract

In this report we apply the theory of deep neural networks to reinforcement learning. We focus on the improvement of the Q-learning algorithm using deep neural network, the DQN(Deep Q-Network),with two different kind of inputs(pixel input or raw input) with respectively two different neural networks typologies.Then we use RAW inputs to perform experiments on Space Invaders environment and we plot the final results.

Introduction

Video games provide an ideal testbed for artificial intelligence methods and algorithms. In particular, programming intelligent agents that learn how to play a game with human-level skills is a difficult and challenging task. Reinforcement learning has been widely used to solve this problem traditionally. The goal of reinforcement learning is to learn good policies for sequential decision problems by maximizing a cumulative future reward. The reinforcement learning agent must learn an optimal policy for the problem, without being explicitly told if its actions are good or bad.
Reinforcement learning approaches rely on features created manually by using domain knowledge (e.g. the researcher might study game playing patterns of an expert gamer and see which actions lead to the player winning the game, and then construct features from these insights). However, the current deep learning revolution has made it possible to learn feature representations from high-dimensional raw input data such as images and videos, leading to breakthroughs in computer vision tasks such as recognition and segmentation. Mnih et al. 2013 apply these advances in deep learning to the domain of reinforcement learning. They present a convolutional neural network (CNN) architecture that can successfully learn policies from raw image frame data in high dimensional reinforcement learning environments. They train the CNN using a variant of the Q-learning and call this network a Deep Q-Networks(DQN).
In this project, we train an agent to play Space Invaders, a popular Atari game using Deep Reinforcement Learning(DQN), with different architectures and parameters so to show how vary the final results

State of the art

Reinforcement learning (RL) provides a neuropsychological and cognitive science perspective to animal behavior and sequential decision making.
The recently introduced Deep Q-Networks (DQN) algorithm has gained attention as one of the first successful combinations of deep neural networks and reinforcement learning. It achieved dramatically better results than earlier approaches, showing that its ability to learn good representations is quite robust and general. Another important development in the field of RL has been indirectly borrowed from enormous successes of deep convolutional neural networks(CNN) in image feature extraction. A direct implication of CNNs in reinforcement learning was the use of image pixels as states instead of hand-crafted parameters, which was widely in practice in RL landscape. Use of such an expressive parametrization also enabled learning of value function and policies that were previously deemed complicated. We use it to represent and then optimize using stochastic gradient descent value function, policy and model.
Furthermore, it was shown that combining model-free reinforcement learning algorithms such as Qlearning with non-linear function approximators, or indeed with off-policy learning could cause the Q-network to diverge.
Subsequently, the majority of work in reinforcement learning focused on linear function approximators with better convergence guarantees. In fact TD-Gammon approximated the state value function V (s) rather than the action-value function Q(s,a), and learnt on-policy directly from the self-play games
More recently, there has been a revival of interest in combining deep learning with reinforcement learning. In addition, the divergence issues with Q-learning have been partially addressed by gradient temporal-difference methods. These methods are proven to converge when evaluating a fixed policy with a nonlinear function approximator; or when learning a control policy with linear function approximation using a restricted variant of Q-learning. In 2015, van Hasselt et al. applied *double q-learning* to DQN, a modification that improved performance and convergence on some games. Notice that we use the target network both to calculate the q values for each action in the next state and to select which of those actions we will want to take (the highest one). It turns out this can lead to some overestimation problems, particularly when the discrepancy between the target network and online network cause them to recommend totally different actions given the same state (as opposed to the same action with slightly different q values). To solve this, we take away the target network's responsibility to determine the best action; it simply generates the Q values, and we let the online model decide which to use.
In 2016, Wang et al. published the *dueling network architecture*, which splits the neural network into two streams that share a convolutional base, one for estimating each function.
Up until now, our agent has been sampling from its replay buffer uniformly at random. But what if it could learn from the experiences where it is the most wrong? Schaul et al.'s 2016 paper proposes a solution, known as *prioritized experience replay* (PER). In PER, we use an additional data structure that keeps a record of the priority of each transition. Experiences are then sampled proportional to

Deep-Q-Network implementation on Atari 2600 Space Invaders

their priorities:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

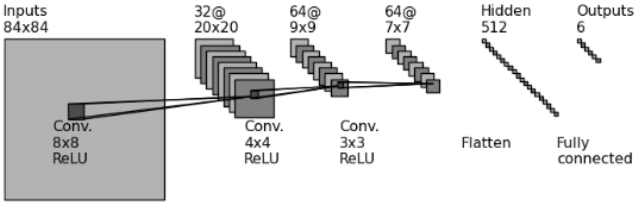
Where *alpha* is a hyperparameter that indicates how aggressive we want this sampling bias to be. The priorities themselves are just the agent's temporal difference error the last time it trained on that experience. In this way, our agent is more likely to learn from its most inaccurate predictions, smoothing out its trouble spots and generally being much more sample efficient.

Reinforcement Learning: Reinforcement learning is an area of machine learning concerned with how an agent should act in an environment so as to maximize some cumulative reward. Q-learning is a model-free technique for reinforcement learning which can be used to learn an optimal action-selection policy for any finite MDP. Q-learning was first introduced by Watkins and Dayan 1992. However, it has been shown that Q-learning with non-linear function approximation is not guaranteed to converge (Tsitsiklis and Van Roy 1997)

Convolutional Neural Networks: Recent advances in deep learning, especially the use of convolutional neural networks (CNNs), have made it possible to automatically learn features representation from high-dimensional raw input data such as images and videos. Now, CNN-based models produce state-of-the-art performance in various computer vision tasks, including image classification (Krizhevsky, Sutskever, and G. Hinton 2012, K. He et al. 2015) and object detection (K. Heetal.2014).

Deep Q-Learning: Mnih et al.2013 present a convolutional neural network(CNN)architecture that can successfully learn policies from raw image frame data in high dimensional reinforcement learning environments. They train the CNN using a variant of the Q-learning, hence the name Deep Q-Networks (DQN).

To extract features manually from high dimensional and numerical data, like images from game screen it's very difficult and inefficient. We can use Deep Q-Networks which uses a convolutional neural network to approximate the Q-function. If the state is n dimensional and the number of actions is m then the CNN is a mapping from R^n to R^m.



Van Hasselt, Guez, and Silver 2015 recently extended this work by incorporating Double Q-Learning which addresses the overestimation problem (Q-learning algorithm is known to overestimate action values since the same set of parameter weights are used to select an optimal action as well as evaluate the Q-value resulting from this selection). More recently, DeepMind has achieved a breakthrough in using deep reinforcement learning, combined with tree search, to master the game of Go (Silver et al. 2016).

It still remains difficult to apply these algorithms to real world settings such as data centers, autonomous vehicles , helicopters. Typically these algorithms learn good control policies only after many millions of steps of very poor performance in simulation. This situation is acceptable when there is a perfectly accurate simulator; however, many real world problems do not come with such a simulator.

Approach

Task definition : We consider the task of learning to play Space Invaders in the Atari2600 game simulator Gym, a popular Atari console game. The game play is discretized into time-steps and at each time step, the agent chooses an action at from the set of possible actions for each state A = {1,2,...L},in our case 6 actions. The emulator applies the action in the current state, and brings the game to a new state.. The game score is updated the a reward is returned to the agent.

We formalize this problem as follows:

• *State* s : A sequence of observations, where an observation, depending on the state space we are operating in, is a matrix representing the image frame or a vector representing the emulator's RAM state. In our experiment we use RAM state for computationally efficiency.

• *Action* a : We have a total of 6 actions and they are: FIRE(shoot without moving), RIGHT (move right), LEFT (move left), RIGHTFIRE (shoot and move right), LEFTFIRE (shoot and move left), NOOP (no operation).

• *Reward* r : Reward returned by the environment. Our task is to learn a policy for the agent to enable the agent to make 'good' choice of action for each state.

Approach: This emulator provides two possible kind of state to play space invaders . The First state is represented by a raw RGB image of the screen which has a size (210,160,3) (SpaceInvaders-v0), the second state is represented by a 128 byte array (SpaceInvaders-ram-v0). As mentioned previous, in this work we 're going to use RAM representation,more faster and lighter then full image representation. We describe the two different architecture for both the state and in the end we 're going to discuss the result obtained in the RAM situation. As mentioned in the previous section, our task is to learn a policy for the agent so that the agent can play the game. In order to accomplish this task,we use reinforcement learning. However,learning from high dimensional representation of states such as images is difficult for traditional reinforcement learning techniques with handcrafted features.In the case of images it used a Convolutional Neural Networks (CNN),to approximate and learn the Q function (which we will introduce later) ; in our case we use a Fully Connected Neural Network.

Deep-Q-Network implementation on Atari 2600 Space Invaders

Q-Learning : Given a sequence of state, actions, rewards s1,a1,r1,s2,a2,r2.., we want to learn the optimal strategy to play the game. The goal of the agent is to learn a strategy to maximize future rewards. We assume that the rewards are discounted by a factor of γ (which say us how many important are the future steps) at every time step and define the future discounted return as :

$$R_T = \Sigma_{t=0}^{t=T} \gamma^t r_t$$

We also define the optimal action value function $Q_{opt}(s,a)$ to be the reward we get on following the optimal policy and starting in state s and playing action a. Given a Markov Decision Process with transition probabilities t(s,a,s0) and rewards r(s,a,s0), the Q-function satisfies a recurrence known as the Bellman equation:

$$Q_{opt}(s,a) = \sum_{s'} T(s,a,s')(r(s,a,s') + \gamma V_{opt}(s'))$$

However in the setting of reinforcement learning, we do not know the transition probabilities and rewards for each transition. We try to learn the optimal strategy by minimizing the squared loss function:

$$\sum_{s,a,r,s'} (Q_{opt}(s,a) - (r + \gamma V_{opt}(s')))^2$$

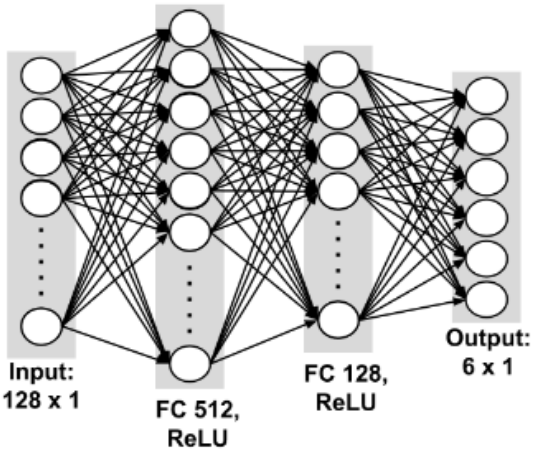
In practice, this equation does not generalize to unseen states and actions, so we use a function approximation to estimate the Q-function. It is common to use a linear approximator in traditional RL approaches where we estimate $Q_{opt}(s,a,w) = w.\phi(s,a)$. We still minimize the same loss function but now use gradient descent to learn the weights-w The update equations are:

$$w \leftarrow w - \eta(Q_{opt}(s,a) - (r + \gamma V_{opt}(s')))\phi(s,a)$$

Deep Q-Networks : To extract features manually from high dimensional and numerical data, like images from game screen it's very difficult and inefficient.
We refer to a neural network function approximator with weights θi as a Q-network. The objective is to now learn the weights θi Q-network.

$$\theta_i \leftarrow \theta_i - \eta(Q_{opt}(s,a,\theta_i) - (r + \gamma V_{opt}(s')))\nabla_{\theta_i} Q_{opt}(s,a,\theta_i)$$

We can also use the same approach of using a neural network as a function approximator for the Q function when we use the 128 byte representation of the state. While it is not evident a-priori to us what each byte represents, we can nonetheless use a fully connected neural network to represent the Q-function. Since the state is not an image this time,we do not use a CNN for the function approximator. We feel it is necessary to use a fully connected neural network to approximate the Q-function as there needs to be connections between every two bytes representing the state. Moreover, we can consider the 128 byte representation as a feature vector, which allows us to drastically simplify our network architecture.



Experience Replay: A technique called experience replay is used in Deep Q learning to help with convergence. At every time step we store the agents observations (s,a,r,s0) into a reply memory ***D***. We then randomly sample a minibatch observations from the replay memory D and use this minibatch to train the network (update weights with backpropagation). Learning directly from consecutive samples is inefficient as there could be strong correlations between samples. Randomizing the training by choosing random samples will help with this problem by smoothing the training distribution over many past behaviors.

Frame Skipping: During Training,the agent sees and chooses optimal actions on every k-th frame instead of every frame, and its last action is repeated on skipped frames. This works because the game state does not change much in every frame. Skipping frames allows the agent to play roughly k times more episodes for the same time/computational resources. We Experimented with k = 4 in our experiments and compare the results with the case without frame skipping.

Exploration Policy: In this work, we use two different kind of epsilon to allow the system to work better. First we use a value for epsilon which start from 1 to 0.05 linearly, allowing so to the game first exploring the states and then doing actions more based on the Q-function learned step by step.The second kind of epsilon trend value is founded on the idea that the game is sequential : so it is both difficult and slowly that the game learn temporal advanced states with the same method of first states (first exploration then exploitation). So we implement in the program after a fixed number of iteration, an increase in the epsilon value after each step, to allow to reproduce this behavior exploration-exploitation also in advanced episodes: in first state we use more value based on the huge past experience(so low value of epsilon),then we perform more random actions instead of only actions based on poor Q-values learned with few batches which had reaches that states and so more inclined to have bad values.During the testing phase we use a value of = .05.

Deep-Q-Network implementation on Atari 2600 Space Invaders

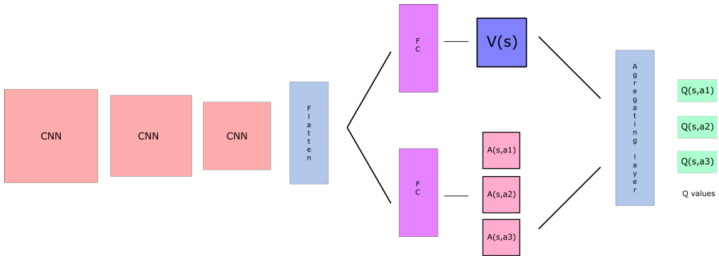
Duel Deep Q-Network: The Q-values correspond to how good it is to be at that state and taking an action at that state Q(s,a).So we can decompose Q(s,a) as the sum of:

- V(s): the value of being at that state
- A(s,a): the advantage of taking that action at that state (how much better is to take this action versus all other possible actions at that state).

$$Q(s,a) = A(s,a) + V(s)$$

With DDQN, we want to separate the estimator of these two elements, using two new streams:

- one that estimates the state value V(s)
- one that estimates the advantage for each action A(s,a)



And then we combine these two streams through a special aggregation layer to get an estimate of Q(s,a).
This is particularly useful for states where their actions do not affect the environment in a relevant way.For no fall inside identifiability problem (given Q(s,a) we're unable to find A(s,a) and V(s), and not being able to find V(s) and A(s,a) given Q(s,a) will be a problem for our back propagation), we can force our advantage function estimator to have 0 advantage at the chosen action.
To do that, we subtract the average advantage of all actions possible of the state.

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \frac{1}{\mathcal{A}}\sum_{a'} A(s,a';\theta,\alpha)\right)$$

Common network parameters
Value stream parameters
Advantage stream parameters
Average advantage

Therefore, this architecture helps us accelerate the training. We can calculate the value of a state without calculating the Q(s,a) for each action at that state. And it can help us find much more reliable Q values for each action by decoupling the estimation between two streams.

Results and experiment

The reference paper propose four different architecture for the network with RAM states. This compact representation is supposed to contains all information describing the game current game state and so a fully-connected architecture allow to relate the features each other. Unlike CNN, with RAM states we need no preprocessing step.

Linear model

A simple linear model, which approximates the Q-function as a linear combination of the 128
features and has 774 parameters.

2-layer fully connected network

A fully-connected neural network with 1 hidden layer of 512 units. This model has about 69K parameters.

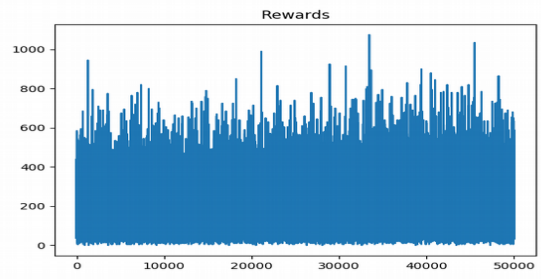
3-layer fully connected network

Two variations of a 3-layer neural network, one with hidden layers of 256 and 128 dimensions (69K parameters)and another with hidden layers of 512 and 128 dimensions(132K parameters).

I have tested all of this architectures each of them with two different kind of epsilon trend and for 50.000 episodes. In order to train the networks, we tried the Adam algorithm(variant of stochastic gradient descent more faster).Then we also compare standard algorithm which sees and select actions every frame with a variant which use frame-skipping, selecting actions on every 4 frames. We use typically minibatch of size 100 (except in last test where we use 32 minibatch size) and a discounting gamma = 0.9.
Our models are implemented using Keras, with Tensorflow backend and trained on CPU system i7-8600K. The reward for each singular match and the total cumulative rewards are similar for each different applied methodology, so in this section they are omitted,except for the first model which explain that plot. The reward plot is similar for each single experiment because of the intrinsic nature of the game(full of noise like explained in the paper), while the difference in the cumulative reward are more highlighted with the mean plot. The final plot of each single experiment will be showed in the appendix.

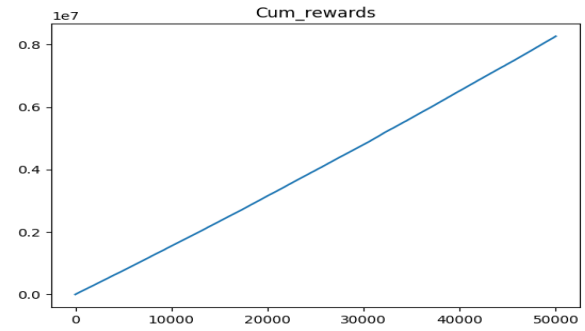
Linear model

Like in the paper also in our experiment we can witness the presence of high noise in the model. The basis of the game itself lead to this kind of behavior(e.g. the high pick at the start maybe correspond to the fact that the player hit the pink spaceship which give the higher reward). Moreover the fact that if the player shot many times, certainly hit some enemy in front of it, many depend in the initial position of both agent and enemies. So the most important feature that the agent must learn is to avoid to be hit by the enemy while shot.

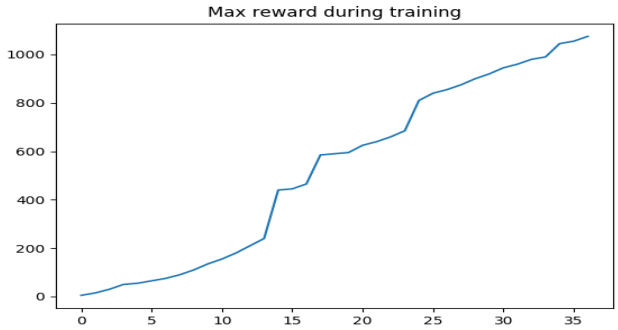


Deep-Q-Network implementation on Atari 2600 Space Invaders

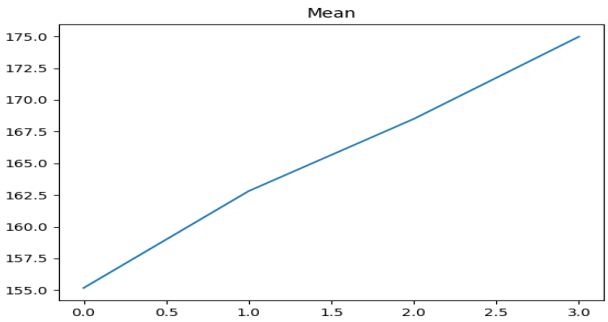
Here we can see the cumulative rewards almost grows linearly with a slight increase with the increase of episodes.



Linear model reach and exceeds the 1000 point for a single episode.

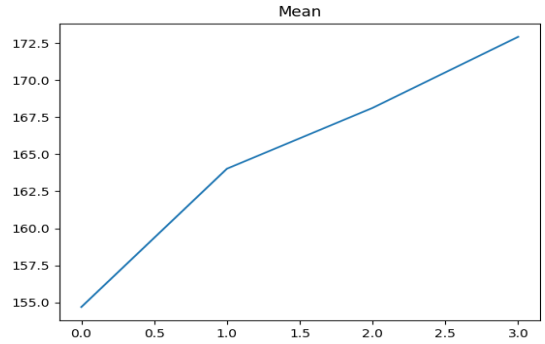


In this figure we can see the effectiveness of this model. We calculated four averages for every quarter of the training period : below it is showed the growth of the medium reward with growing up of the episodes and so with a largest number of actions take looking at the approximated Q function.



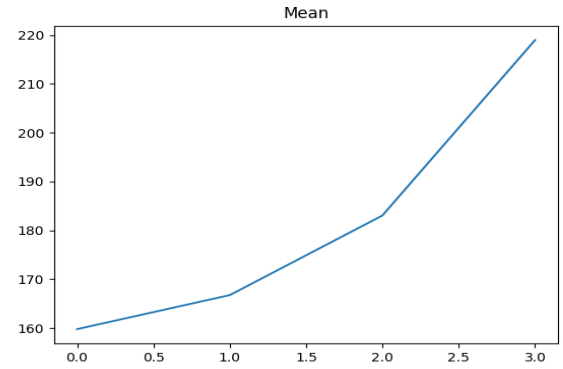
Linear model with greedy policy

Here below we can see the lighter difference between the two different policy methods : the previous method with further shuffle of epsilon in the final step show to reach a greater score respect to the greedy policy.



2-Fully-Connected layer model

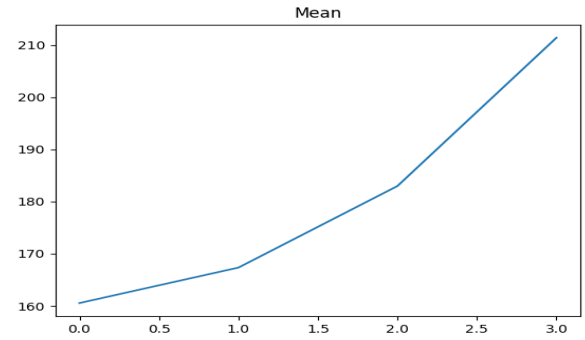
With two fully connected layers the numbers of parameters to estimate the Q function increase. This computationally cost it is justified by an increasing in the mean score. Indeed the final score increase from 175 of lineard model to 220 of the 2-FC layer model.
Meanwhile the maximum score reached during the training is similar to the previous model



2-Fully-Connected Layer Model with greedy policy

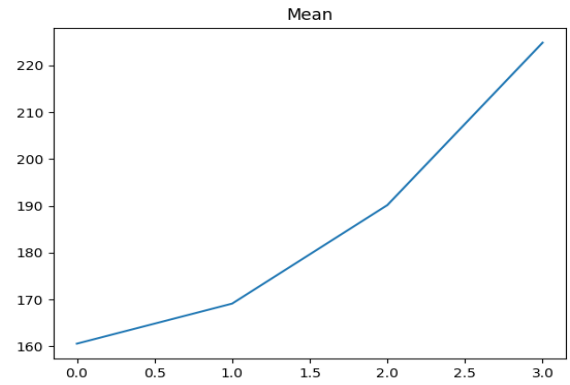
With this setup, working with greedy policy allow to reach in some episode a greater score, but the mean of the last period it's lower respect the previous 2-FC model with different epsilon policy. This maybe can show the compactness of the previous policy respect this one.

Deep-Q-Network implementation on Atari 2600 Space Invaders



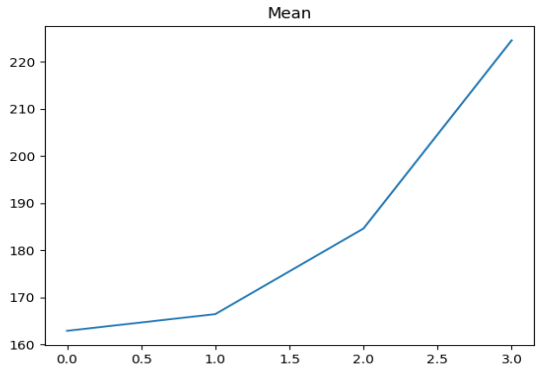
3-Fully-Connected Layer Model(a)

Below i show the results obtained from training a deep q network with a three fully connected layers with 256 nodes in the first layer and 128 in the other layer. After i show the other version of this architecture with a double number of nodes in the first layer, 512.



Respect the previous experiment, with a greater number of parameters, with same training episodes, we can reach higher performance, achieving the 220 mean score in the final episodes. The cost of this better result is more training time and a more computationally expensive execution.

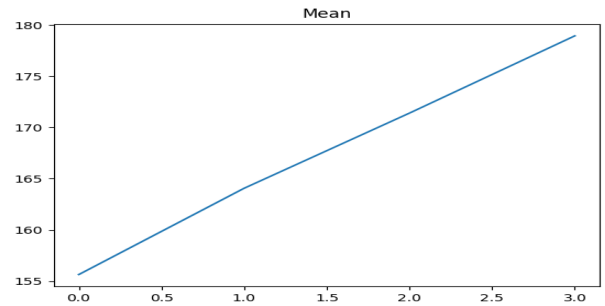
3-Fully-Connected Layer Model(b)



In this other version of the 3FC architecture,with a greater number of parameters we can see slow and slightly better results.

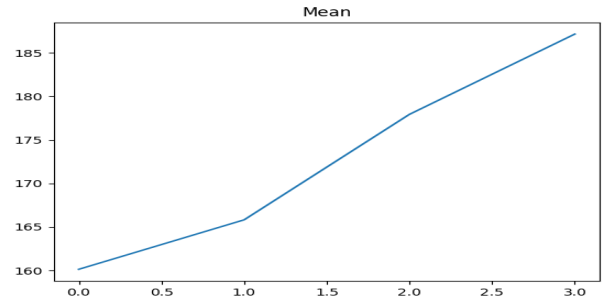
FrameSkip(4)

In this section i've tested the idea of frameskip.I wanted to analyze the effectiveness of this technique applied to the linear model : in a brief time of training
we have tested the program on a larger number of episodes achieving of course best results optimizing the process. Indeed, it is not mandatory processing all consecutive frame step since two or more consecutive frame show similar state and they are not very useful for the purpose of the learning process. Moreover skip some frames doesn't cut the performance of the network.



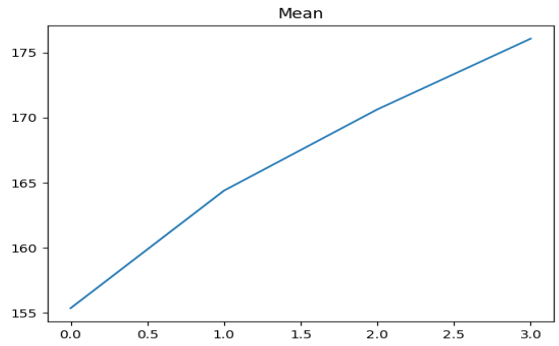
Duel Deep Q Network

Also in this case we've tested this improvement of DQN on allowing the execution of the program in less time.



Deep-Q-Network implementation on Atari 2600 Space Invaders

Batchsize varying(32)
Using a smaller dimension for the batchsize allow us to reach yet good performance decreasing computationally effort.



Conclusion

In this work i have showed the power and the efficiency to combine both reinforcement learning and deep neural networks. Starting from Q-learning we have tested different architectures for the networks varying also parameter during the training phase to seek the best performing implementation.We‘ve introduced improvement to face the convergence problem of DQNs and to improve performance. Nowadays this family of algorithms is considered one of the best methods to perform optimal results in the atari virtual environment and also in other fields.

References

- Nihit Desai et al. ‘Deep Reinforcement Learning to play Space Invaders’ (main)
- Volodymyr Mnih et al. ‘ Human-level control through deep reinforcement learning’
- Volodymyr Mnih et al . ‘Playing Atari with Deep Reinforcement Learning’
- Todd Hester et al. ‘’ Deep Q-Learning from Demonstrations’’