**MSML 612 – Deep Learning**

***Final Project Writeup***

Prepared By: Ricardo Zambrano

UMD id: 115811614

**I. Introduction**

The intuitive idea that humans learn from interacting with their environment is at the core of some of the methodologies used in the field of machine learning. This project focuses on the machine learning subfield of reinforcement learning, which is founded on the idea that learning is a process by which agents learn from interaction with their environment. This approach to learning holds that by experimenting with their environment, agents produce information about cause and effect, about the consequences of actions, and about what to do in order to achieve goals.

Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics—trial-and-error search and delayed reward—are the two most important distinguishing features of reinforcement learning.

The subfield of reinforcement learning is much more focused on goal-directed learning from interaction than are other approaches to machine learning.

A reinforcement learning method draws ideas from dynamical systems theory, specifically, as the optimal control of incompletely-known Markov decision processes. These processes intend to include three aspects:

* A learning agent must be able to sense the state of its environment to some extent
* Must be able to take actions that affect the state
* The agent must have a goal or goals relating to the state of the environment

It is worth taking a moment to explore the differences between reinforcement learning and other approaches to machine learning.

*Supervised Learning*: It comprises learning from a training set of labeled examples. Each example is a description of a situation together with a specification—the label—of the correct action the system should take in that situation, which is often to identify a category to which the situation belongs. The object of this kind of learning is for the system to extrapolate, or generalize, its responses so that it acts correctly in situations not present in the training set. This is an important kind of learning, but alone it is not adequate for learning from interaction. In interactive problems it is often impractical to obtain examples of desired behavior that are both correct and representative of all the situations in which the agent has to act. In uncharted territory—where one would expect learning to be most beneficial—an agent must be able to learn from its own experience.

*Unsupervised Learning:* Typically, it is about finding structure hidden in collections of unlabeled data. Although one might be tempted to think of reinforcement learning as a kind of unsupervised learning

because it does not rely on examples of correct behavior, reinforcement learning is trying to maximize a reward signal instead of trying to find hidden structure. Uncovering structure in an agent’s experience can certainly be useful in reinforcement learning, but by itself does not address the reinforcement learning problem of maximizing a reward signal. Therefore, reinforcement learning is considered to be a third machine learning paradigm, alongside supervised learning and unsupervised learning and perhaps other paradigms.

One of the most exciting aspects of modern reinforcement learning is its substantive and fruitful interactions with other engineering and scientific disciplines. Recently, reinforcement learning has been used in a wide range of practical applications such as: self-driving cars, energy-saving applications for data centers (leading to 40% of energy spending at Google data centers), robotics, solving the board game of Go (AlphaGo and AlphaZero), predicting a protein's 3D structure from its amino acid sequence (AlphaFold), and magnetic control of tokamak plasmas in nuclear fusion reactors.

**II. Problem Statement**

**2.1 From Decision-Making in Games to Real-World Applications**

It has been proven that reinforcement learning can be used to solve practical applications once believed to be intractable due to size of the search space, as well as the difficulty of evaluating and selecting actions. This includes decision-making problems that can be abstracted as a game.

In the social sciences many problems of interest are related to decision making. Many of these problems involve the following: making decisions under uncertainty, experiments cannot be repeated, and the number of variables can be either intractable, incomplete, or both.

Luckily, a subset of these problems can be modeled as a simpler version of the problem, variables can be operationalized in ingenious ways—including the identification of instrumental variables, and it is possible to abstract them as games by restating the problem as a set of features, such as: rules, points, legal moves, length (infinite versus finite game).

The field of modern economics is based on rational *agents* (individuals, interest groups, and firms), trying to maximize their utility by their consumption *decisions* within an *environment* constrained by scarce resources and where only incomplete/asymmetric information is available.

It is the belief of this author that by combining modeling, simulation, and reinforcement learning it would be possible to solve problems in an array of fields in a more satisfactory way. These fields would include economics, as well as related areas of study such as business, finance, public policy, and sociology.

This author believes that after selecting a model of the world, social scientists can explore potential solutions to specific social challenges by: creating a simulation that correctly abstracts the behavior of economic/social agents, modeling an environment for the agent to interact with, and using reinforcement learning to explore and discover the optimal policies that would maximize a given goal, such as: economic social welfare, profits, reducing production costs, building more infrastructure, among others . Hopefully, this best policy will either teach us some unforeseen solutions to real applications or find the optimum value for complex problems with either an intractable number of variables, or, where we have not reached a theoretical model to use the variables available at hand as inputs.

At the beginning of the semester, this author attempted to create a simulation of a free market involving consumers and producers, the latter being the agents learning from the environment. However, given the timeframe available for this project, creating the simulation proved to be a too complex problem, worthy of being a final project for a semester-long course on simulation.

Due to the aforementioned time limitation, this project will leave the problem of building a simulation for a future expansion of this idea and will focus on solving a game using reinforcement learning with deep neural networks. The hope is that the skills the author learns and develops during the process of solving a game will be transferrable to solving real-world decision-making problems at a later time.

**2.1. Selected Environment**

The selected game environment was VizDoom, which is an open-source python interface for the Doom Engine. This environment allows for the developing AI bots that play DOOM using visual information (the screen buffer). It is primarily intended for research in machine visual learning, and deep reinforcement learning in particular.



***Figure 0.*** Screen caption of the VizDoom environment in the “Deadly Corridor” scenario

The VizDoom library was created in 2016 by Marek Wydmuch, Michal Kempka at the Institute of Computing Science, Poznan University of Technology, Poland.

As mentioned above, VizDoom enables the training of agents directly from the screen pixels in a number of scenarios. There are a couple of technical details implied by this setup:

* Training: the information from the environment are frames because the agents are trained directly from pixels. Providing one frame at a time will lead to a problem known as “temporal limitation”, which is that one frame does not have enough information. For example, in Figure 0: is the enemy moving towards the agent or away from the agent? To overcome this problem, four frames are stacked and passed to the neural network as input.
* VizDoom provides several scenarios. These are small maps with a variety of tasks and goals for the reinforcement learning agents. In this project the focus was on the following scenarios:
  + Basic: In the basic scenario the agent needs to shoot an enemy that is stationed in one point in space.
    - The agent’s actions are: move left, move right, or attack
    - The reward function is: -1 for living, -6 for shooting and missing, +100 for killing
  + Health Gathering Supreme: In this scenario the agent is on an “acid pool” map. With each tick the agent loses health. The goal is to pick up health packs to stay alive.
    - The agent’s actions are: turn left, turn right, or move forward
    - The reward function is: +1 for living, -100 for dying
  + My Way Home: The agent needs to solve a maze in this scenario. The episode finishes when the agent finds an armor.
    - The agent’s actions are: move left, move right, turn left, turn right, or move forward
    - The reward function is: -0.0001 for living

Because the ViZDoom environment is based on a game that was created in the 1990s, it can be run on modern hardware at accelerated speeds, allowing us to learn complex AI behaviors fairly quickly. The screen caption below captures the three scenarios referenced above.

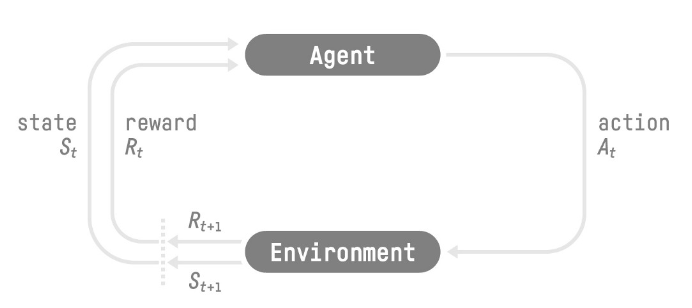


***Screen Caption 1.*** A selection of VizDoom Scenarios: Basic, Health Gathering Supreme, My Way Home

**III. Technical Background**

**3.1. Reinforcement Learning**

This is a reinforcement learning project, thereby, it is necessary to lay out the reinforcement learning methodology and define a few technical terms. Figure 1 depicts the reinforcement learning process.



***Figure 1.*** The reinforcement learning process: a loop of state, action, reward, and next state.

In the reinforcement learning framework, the agent receives a state S0 from the environment (i.e. the first frame, a stack of frames, or the position of pieces in a board game). Based on the state S0 the agent takes an action A0. Then the environment transitions to S1 and the agent receives a reward R1.

As mentioned earlier in this write up, the agent seeks to maximize the cumulative reward, called the expected return.

Just as in some language models, the reinforcement learning framework assumes the Markov property. The Markov property implies that the agent needs only the current state to decide what action to take and not the history of all the states and actions they took before.

Observations/States are the information the agent gets from the environment. For example, in the case of a video game, it can be a frame or a stack of frames. In the case of the trading agent, it can be the value of a certain stock plus a set of financial and economic variables (interest rate, inflation, GNI, sales in the last quarter, even weather forecasts).

The difference between an observation and a state is the following:

* *State* s: is a complete description of the state of the world (there is no hidden information). It is a fully observed environment.
* *Observation* o: is a partial description of the state. It is a partially observed environment.

The action space is the set of all possible actions in an environment. A trading agent will have a discrete action space: sell, buy, or do nothing. A delivery robot will have a continuous action space, given that the number of possible actions is infinite.

The reward is fundamental for the reinforcement learning framework, this is because it is the only feedback the agent can get. Thanks to the reward, the agent knows if the action taken was good or not. The cumulative reward at each time step t can be written as: R(τ) = rt+1 + rt+2 + …

An improvement to the reward model is to assume that rewards closer to time step ‘t’ are more likely to happen. Rewards in the long-term future are less predictable hence less likely. To model this time-dependent likelihood each reward at time step ‘t’ can be multiplied by a discount factor ‘γ’ -between 0 and 1. Then the resulting cumulative reward equation is: R(τ) = rt+1 + γ\* rt+2 + γ2\* rt+3 + …

With a larger discount factor the agent behavior is modified to seek long-term rewards. Conversely, lower discount rate values lead to a focus on short-term reward.

An instance of a Reinforcement Learning problem is called a task. A task can be episodic or continuing. Episodic tasks have a starting point and an ending point. The latter is called ‘terminal state’. When a task reaches its terminal state, an episode is completed, which consist of: a list of states, actions, rewards, and new states. Continuing tasks continue forever, no terminal state is reached. In this case, the agent must learn how to choose the best actions and simultaneously interact with the environment.

For some tasks, where the terminal state is reached only after a long sequence or for continuing tasks, it is better to work with a trajectory. A trajectory is just a state-action-reward sequence. It is a little bit more flexible than an episode because there are no restrictions on its length: it can correspond to a full episode or just a part of an episode. The length of a trajectory is represented with a capital H, where H stands for Horizon.

One of the challenges that arise in reinforcement learning, and not in other kinds of learning, is the trade-off between exploration and exploitation. To obtain a lot of reward, a reinforcement learning agent must prefer actions that it has tried in the past and found to be effective in producing reward. But to discover such actions, it has to try actions that it has not selected before. The agent has to exploit what it has already experienced in order to obtain reward, but it also has to explore in order to make better action selections in the future. The dilemma is that neither exploration nor exploitation can be pursued exclusively without failing at the task. The agent must try a variety of actions and progressively favor those that appear to be best. On a stochastic task, each action must be tried many times to gain a reliable estimate of its expected reward. The exploration–exploitation dilemma has been intensively studied by mathematicians for many decades, yet remains unresolved.

To solve a reinforcement learning problem a policy π is required. The policy π is the ‘brain’ of the agent. It’s the function that tells the agent what action to take given a state st. The policy defines the agent’s behavior at a given time step.

The goal pursued by researchers using the reinforcement learning framework is to learn the policy function. In particular, the goal is to find the optimal policy π\*, the policy that maximizes expected return when the agent acts according to it. The optimal policy π\* is found through training.

There are two approaches to train agents to find this optimal policy π\*:

* Directly: by teaching the agent to learn which action to take, given the current state. This approach is known as Policy-Based Methods.
* Indirectly: teach the agent to learn which state is more valuable and then take the action that leads to the more valuable states. This approach belongs to the group of Value-Based Methods.

In Policy-Based methods, the policy function is learned directly. The policy might be deterministic, where it will always return the same action at a given state: at = π(st). A stochastic policy outputs a probability distribution over actions: p(a|s)

In value-based methods, instead of learning a policy function, the framework learns a ‘value function’. A value function maps a given state *‘st’* to the expected value of being in that state. The value of a state is the expected discounted return the agent can get if it starts in that state, and then acts according to a given policy π thereafter. A value function *vπ(s)* can be defined as the expected discounted sum of future rewards if the agent follows policy π. The value function is expressed as: vπ(st) = Eπ[rt+1 + γ\* rt+2 + γ2\* rt+3 + … | S = st], where st is the starting state.

**3.2. Reinforcement Learning Algorithms**

There were four deep reinforcement learning algorithms that were explored in this project, namely:

* Deep Q-Learning
* Policy Gradient
* Actor-Critic
* Proximal Policy Optimization

*3.2.1. Deep Q-Learning*

Deep-Q Learning is an extension of Q-Learning. Like the classic Q-Learning, this is a value-based method. The latter means the goal is to learn a value function that maps a state to the expected value of being at that state. It is worthwhile emphasizing that in a value-based method we learn/approximate the optimal policy π\* indirectly, by training a value function that outputs the value of a state or a state-action pair. Given this value function, the policy will take an action.

In this family of reinforcement learning algorithms there is no policy function available after training. However, after training there is a function that outputs the value of every state action pair, if the goal were to choose the action leading to the biggest reward, it would be possible to obtain a policy by encoding a ‘greedy’ behavior. In this way, actions can be chosen and the agent would have -in practice- a policy.

Therefore, for value-based methods, there is no need to train the policy. In the case of the classis Q-Learning algorithms, the policy is a simple pre-specified function (i.e. a Greedy Policy, as state above), that uses the values given by the value-function to select its actions. The following equation presents formally the link between the value function and the optimal policy approximation in the context of classic Q-Learning:

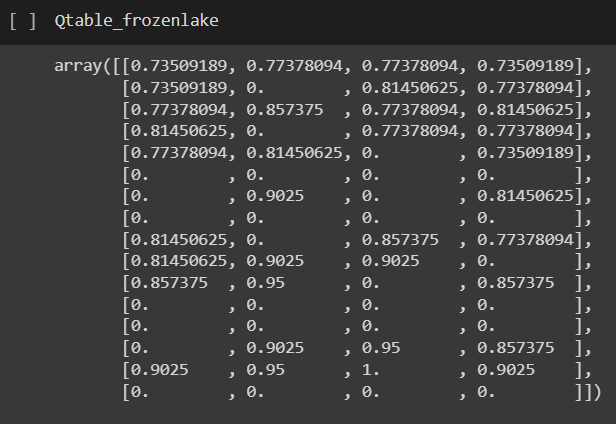
Qπ(s,a) stands for the value-action function. For each state-action pair, this functions outputs the expected return if the agent starts in that state, takes a given action, and then follows the policy forever after. Formally;

Where, Gt is the total cumulative reward from timestep t.

Before jumping into deep Q-learning, a definition of the classic q-learning method might be in order. Q-Learning is an off-policy value-based method that uses a temporal difference approach to train its action-value function.

* Off-policy refers to the fact that the policy for acting is different than the policy for training updating).
* The temporal difference approach is a strategy that uses only a step to learn. The alternative strategy is Monte Carlo rollouts, which uses an entire episode of experience to learn.

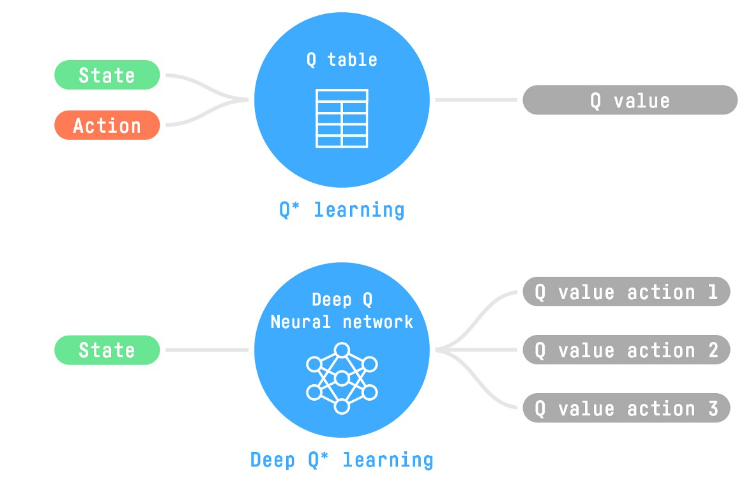
The Q-function is encoded by a Q-table, a table where each cell corresponds to a state-action pair value. For example, while researching for this project an agent was trained to play the “Frozen Lake” game using the classic Q-Learning algorithm. In this game there are 16 possible states (one per each position on the board) and 4 possible actions (go up, down, left, or right). Figure 2 depicts a screen caption of the game and the Q-Table used to solve the game. Note the table shape is 16 rows x 4 columns.

***Figure 2.*** Frozen Lake and Q-Table used to solve the game. The goal is crossing a frozen lake from start to goal (the gift package), without falling into any hole

One limitation one encounters with Q-Learning is that it is a tabular method. Therefore, Q-Learning works well with discrete action spaces. However, the tabular approach becomes a problem if the states and actions spaces are not small enough to be represented efficiently by arrays in a Q- Table.

For large state spaces an alternative approach is using a parametrized Q-function Qθ(s,a). This is where we replace the searchable Q-Table by a neural network (see Figure 3).



***Figure 3.*** From Q-Learning to Deep-Q-Learning

Whereas in Q-Learning the Q-value of each state-action pair are updated directly (in the table), in Deep Q-Learning, a loss function that compares the Q-value prediction and the Q-target is created. Then, gradient descent is used to update the weights of the Deep Q-Network to approximate more accurately the Q-values.

Another important difference of the Deep Q-Learning algorithm is that it has two phases:

* Sampling: actions are performed, then the observed experience tuples are stored in a replay memory.
* Training: during training a small batch of tuples is selected randomly, then the network learns from this batch using a gradient descent update step.

The replay memory is used to overcome instability issues that are found in deep Q-learning (this is not a problem in Q-Learning). With replay memory the agent does not have to learn to play the whole game in every training step. This is because the agent can learn from the same set of experiences multiple times.

Using a replay memory has other benefits: by randomly sampling the experiences, the correlation in the observation sequences is removed; it also helps avoid action values from oscillating or diverging catastrophically.

Other measures need to be taken to prevent instability. One salient issue is that at every step of training, both the Q-values and the target values shift. The Q-value prediction is getting closer to the target, but the target is also moving. This can lead to significant oscillation in training. The solution to this problem is having the Q-target network fixed. In practice, the Q-target network is updated after a given number of training steps.

The main methodology used in this project was deep Q-learning. The subsequent methods, presented in the next sections, were tested but led to no results or were used as an off-the-shelf solution. Thereby, the following algorithms will be discussed briefly.

*3.2.2. Policy Gradient*

Algorithms that learn the policy directly have some advantages over value-based methods, among them:

* The policy can be estimated directly without storing additional data (state-action values)
* Given that policy-gradient methods output a probability distribution over actions, they:
  + can learn a stochastic policy while value functions can’t. Thereby, there is no need to hardcode a solution to the exploration/exploitation trade-off
  + won’t get stuck in a given trajectory (value-based algorithms may get stuck in a given trajectory because they result in a quasi-deterministic policy)
* Policy-gradient methods are more effective in high-dimensional action spaces and continuous actions spaces

However, policy gradient methods often converge to a local minimum, they can take longer to train, and they can have high variance.

Policy-gradient methods aim to find parameters θ that maximize the expected return. As mentioned above, a neural network outputs a probability distribution over actions: πθ(ai|st). Then, the goal is to control the probability distribution of actions by tuning the policy such that good actions (that maximize the return) are sampled more frequently in the future.

In a high-level view, the training loop for policy gradient consist on the following:

* Collect an episode with the policy π
* Calculate the return of the episode by adding the rewards
* Update the weights of the policy π as follows:
  + If positive return 🡺 increase the probability of each state-action pairs taken during the episode
  + If negative return 🡺 decrease the probability of each state-action pairs taken during the episode

Gradient Ascent and the Policy-gradient Theorem are used to find the policy parameters θ that maximize the expected return. The objective function in policy gradient is equal to:

Where R(τ) is the return from an arbitrary trajectory.

Using the policy gradient theorem, it can be shown that for any differentiable policy and for a policy objective function J(θ), the policy gradient is:

Then, to update the parameters θ the gradient ascent step is equal to:

This setup is known as the REINFORCE algorithm or Monte Carlo Reinforce.

*3.2.3. Actor-Critic*

In Policy-Based methods, the goal is to optimize the policy directly without using a value function. In the previous section the policy gradient method called Reinforce was briefly explored. Reinforce optimizes the policy directly by estimating the weights of the optimal policy using Gradient Ascent.

One disadvantage of Reinforce arises because the Monte-Carlo sampling to estimate return (it uses an entire episode to calculate the return). Although using an entire episode offers an unbiased return, it leads to high variance in policy gradient estimation. This in turn causes slower training since a lot of samples are needed to mitigate the variance.

The high variance is a result of the stochasticity of the environment (random events during an episode) and stochasticity of the policy, in where trajectories can lead to different returns. In other words, the return starting at the same state can vary significantly across episodes (thus the variance).

Actor-Critic methods are a hybrid architecture that combines value-based and Policy-Based methods. This helps to stabilize the training by reducing the variance, it uses:

* An “Actor” that controls how the agent behaves (Policy-Based method), and
* A “Critic” that measures how good the taken action is (Value-Based method)

*3.2.4. Proximal Policy Optimization*

Proximal Policy Optimization (PPO) is an architecture that improves training stability by avoiding policy updates that are too large. This architecture uses a ratio that indicates the difference between the current and old policy and clip this ratio to a specific range [1−ϵ,1+ϵ].

There are two reasons to avoid having too large of a policy update:

* Smaller policy updates during training are more likely to converge to an optimal solution
* A too-big step in a policy update can result in falling “off the cliff” (getting a bad policy) and taking a long time or even having no possibility to recover

For PPO the Sample Factory library was used. This is an off-the-shelf implementation focused on very efficient synchronous and asynchronous implementations of policy gradients (PPO).

**IV. Reinforcement Learning Agents**

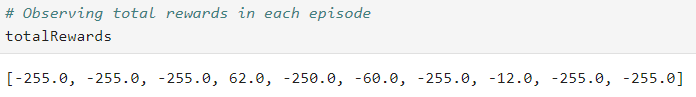
*4.1. Scenario: Basic – Agent: Baseline*

A baseline had to be established in order to have an idea of how effective were the agents trained using reinforcement learning. For the case of VizDoom the selected baseline was setting up an untrained agent that took actions randomly.

This was set up by registering the actions in a VizDoom game and then using the *random.choice()* function to pick an action at random. The following snippet of code summarizes this setup:

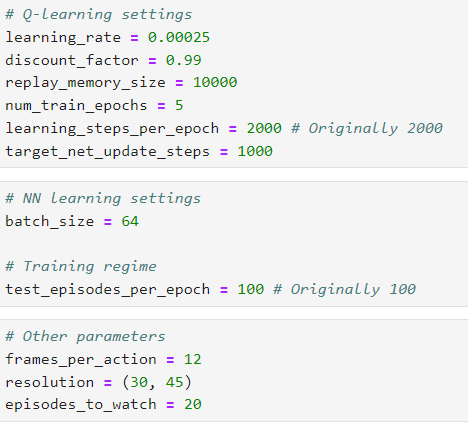


Using this random agent, the following rewards were obtained in the Basic scenario:

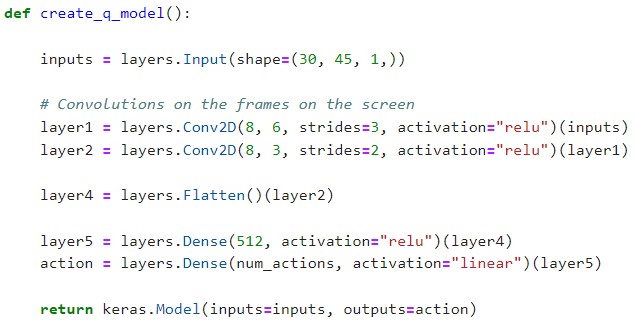


*4.2. Scenario: Basic – Agent: Deep Q-Learning Network*

The next step was setting up a Deep Q-Learning Network (DQN) agent. The agent was coded from scratch. It was trained using a CPU and it had the following parameters:



The neural network consisted of two convolutional layers and two dense layers. The first of the dense layers had a relu activation, and the second had a linear activation. The input was a stack of grey-scale frames taken directly from the game screen. The following block of code was used to create the DQN.



The trained agent played a series of 20 episodes and it obtained the following results:

Total Score = [94.0, 35.0, 65.0, -362.0, 94.0, 58.0, 70.0, 82.0, 82.0, 82.0, 65.0, -58.0, 44.0, 94.0, -101.0, 65.0, 94.0, -387.0, 94.0, 32.0]

It was definitely an improvement over the baseline but not as good as it could get. In particular, in this first experiment it was noticed that the agent only moved left. If the target was located to the right of the agent, it would not hit it (this happened in the case of the -362 score shown above). The first item inspected was the image. Figure 4 showcases a colored version of the frame stack used as input for the DQN.



***Figure 4.*** Basic scenario with 30x45 resolution (colored frame)

It is evident that the enemy in the scenario is not noticeable. For this reason, another experiment was run but passing frames with higher resolution to the DQN. Figure 5 and Figure 6 display the full resolution (480x640) as well as the augmented resolution image passed to the DQN in the second experiment (120x156). Notice that the frame in Figure 5 is colored whereas the image passed to the DQN was in greyscale.



***Figure 5.*** Basic scenario with 480x640) resolution (colored frame)



***Figure 6.*** Basic scenario with 120x156 resolution (colored frame)

In this second experiment the agent was able to move left and right to find the enemy. Scores attained by the agent were also higher:

Total Score = [94.0, 58.0, 82.0, 58.0, 5.0, 12.0, 58.0, 94.0, 70.0, 70.0, 53.0, 82.0, 58.0, 94.0, 24.0, 94.0, 70.0, 82.0, 70.0, 70.0]

In this case the agent was able to shoot the enemy in every episode.

For the basic scenario it was attempted to train agents using the Reinforce Policy Gradient algorithm as well as using the Actor-Critic algorithm. In both of these cases the algorithms did not converge.

*4.3. Scenario: Basic – Agent: Proximal Policy Optimization*

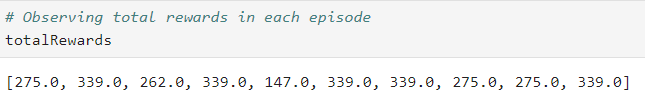
A final experiment was run using the sample factory library. In this case a GPU was required, thereby the agent was trained on Google Colab, which it is not the ideal environment to use the library according to the library’s documentation. In spite of this limitation the agent trained using Proximal Policy Optimization (PPO) was able to shoot the enemy within two shots on every episode. A record of the total score is not available (but there is a video available, which is attached to the assignment).

The PPO agent was trained using the following settings: number of workers = 8, number of environments per worker = 4, training steps = 4,000,000.

*4.4. Scenario: Health Gathering Supreme – Agent: Baseline*

The setup of the random agent was the same as the one used for the basic scenario, with one difference. The actions in this case were: turn left, turn right, move forward.

The rewards obtained by the random agent were the following:



From section 2.1 we know that in this scenario each tick that the agent is alive returns a reward. Thereby, these rewards are not so impressive. It became evident while watching the agent play that it was stuck in one place for the duration of each episode.

*4.5. Scenario: Health Gathering Supreme – Agent: Deep Q-Learning Network*

The neural network architecture as well as the DQN algorithm configuration was the same as the configuration that attained the highest returns in the Basic scenario. The rewards obtained were better than the baseline:

Total Score = [ 343.0, 313.0, 344.0, 278.0, 305.0, 281.0, 281.0, 281.0, 281.0, 343.0, 343.0, 249.0, 280.0, 270.0, 280.0, 334.0, 441.0, 439.0, 601.0, 440.0]

The agent was not stuck anymore when using the DQN; however, it was noticed that once the agent bumped into a wall, the agent would keep moving towards the wall. It was hypothesized that given that the stacked frames were in greyscale the agent could not tell the walls from the life packs apart, since both are grey.

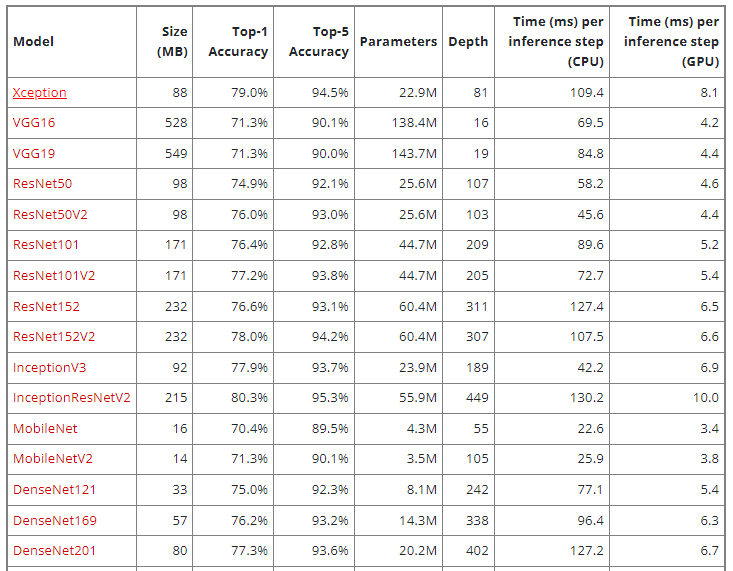
Because of this, the neural network was modified to take three channels as an input. On a first run of this second experiment the agent did not improve much. The second experiment was run a second time, but for 25 epochs with 5,000 training steps per epoch. With this setup the agent got the following rewards during testing:

Total Score = [312.0, 279.0, 278.0, 280.0, 280.0, 280.0, 279.0, 440.0, 280.0, 280.0, 280.0, 243.0, 599.0, 344.0, 440.0, 344.0, 151.0, 440.0, 311.0, 280.0]

This version of the agent did not perform better than the original DQN agent.

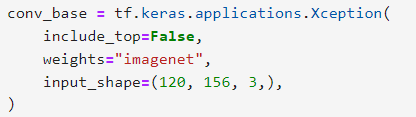
Given that at least part of this problem can be framed as an object detection experiment it was assumed that by improving the convolutional layers, using feature extraction networks, the agent’s performance could be improved. Thereby, a third experiment was run, this time using the convolutional base of the Xception network.

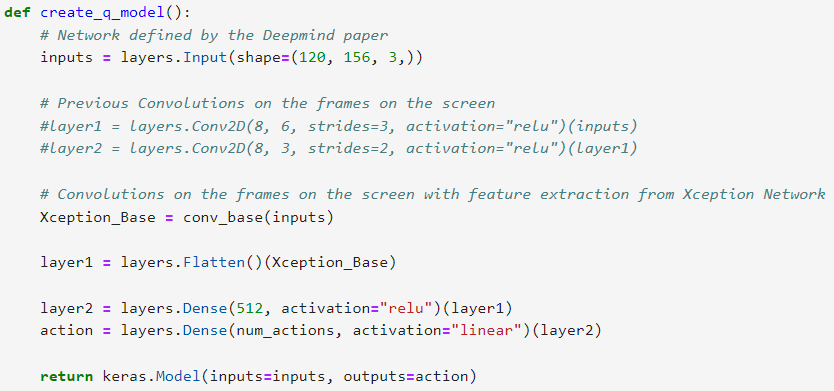
The Xception network is available with Keras and it is ranked among the top performant CNNs (see Figure 7).



***Figure 7.*** ranking of CNNs available in Keras

Besides being in the top of the ranking, the Xception network is lightweight, it has far fewer parameters than VGG. This was another reason to select this convolutional base. The following blocks of code contain the lines to download the pre-trained weights for the Xception network (trained with ImageNet dataset), as well as the modified DQN.



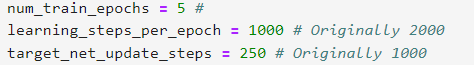


The training of the DQN was slowed dramatically by this addition, even when epochs were reduced back to 5 and training steps per epoch were reduced back to 2,000.

Surprisingly, this did not improve the agent’s performance.

Two more experiments were run using the convolutional base of the ResNet50v2 network. This network was selected because it is compact, it is faster on the CPU than the Xception network, and still performs well (see chart in Figure 7).

The following parameters were modified for these two experiments:



One experiment was run in Google Colab (to have access to a GPU) and the other run was in the local computer with CPU. Not surprisingly, the run in Colab using the GPU was twice as fast. For the agent trained with the GPU the following rewards were obtained:

Total Score = [21.0, 331.0, 270.0, 419.0, 141.0, 480.0, 263.0, 266.0, 325.0, 420.0, 190.0, 142.0, 328.0, 264.0, 321.0, 319.0, 328.0, 141.0, 421.0, 267.0]

The results obtained by the agent trained used the CPU were very disappointing:

Total Score = [197.0, 204.0, 198.0, 208.0, 213.0, 264.0, 201.0, 204.0, 211.0, 201.0, 200.0, 200.0, 201.0, 201.0, 210.0, 201.0, 209.0, 88.0, 214.0, 208.0]

It seems to be the case that the agent trained in the GPU was more eager to explore the map and, consequently, it performed better. However, this better performance was only possible because health packs are distributed randomly in the map. The agent was not aiming for the health packs, it would take steps and stand in front of a health pack only to turn and walk away from it.

The agent trained with the CPU only turned right.

*4.6. Scenario: Health Gathering Supreme – Agent: Proximal Policy Optimization*

Once again, a final experiment was run using the sample factory library. In this case a GPU was required as well, thereby the agent was trained on Google Colab. Just as in the case of the Basic scenario, the agent trained using Proximal Policy Optimization (PPO) performed really well. The agent was clearly seeking the health packs during each episode. A record of the total rewards is not available (but there is a video available, which is attached to the assignment).

The PPO agent was trained using the following settings: number of workers = 8, number of environments per worker = 4, training steps = 4,000,000.

*4.7. Scenario: My Way Home – Agent: DQN and PPO*

This scenario proved to be the hardest. Not even the training using the PPO algorithm provided by the sample factory library was able to solve the game. A record of the total rewards is not available (but there is a video available, which is attached to the assignment).

**V. Conclusion**

The goal of training an agent to play a game was completed for some of the VizDoom scenarios. The scenarios that were not solved might require more training and/or an exhaustive fine-tunning of hyper parameters.

Both Deep Q-Learning Network and Proximal Policy Optimization proved to be powerful algorithms to train reinforcement learning agents. The remaining algorithms, Policy Gradient and Actor-Critic might not converge because of the problems listed above: the first may be converging to a local minimum and the second might be suffering as a result of policy function updates that are too large.

The next step extending this work would be training agents for the Star Craft environment (<http://starcraftgym.com/>), a real-time strategy game. This game gets one step closer to the goal of this author, which is training RL agents to play in a free-market environment. It is worthwhile mentioning that Deep Mind already solved this environment with the AlphaStar program (<https://www.deepmind.com/blog/alphastar-mastering-the-real-time-strategy-game-starcraft-ii>).