Using reinforcement learning to create an agent that plays Pong and then evaluating the model

Ryan Puglia
University of Luxembourg
Email: ryan.puglia.001@student.uni.lu

This report has been produced under the supervision of:
Patrick Keller
University of Luxembourg
Email: patrick.keller@uni.lu

***Abstract*—This document is a template for the scientific and technical (S&T for short) report that is to be delivered by any BiCS student at the end of each Bachelor Semester Project (BSP). The Latex source files are available at: https://github.com/nicolasguelfi/lu.uni.course.bics.global**

This template is to be used using the Latex document preparation system or using any document preparation system. The whole document should be in 8000 words (± 20%) [1] for S2 to S6 students and 6000 (± 20%) for S1 students (excluding the annexes) and the proportions must be preserved. The other documents to be delivered (summaries, ...) should have their format adapted from this template.

A tutor (or any person having contributed to the BSP work) is not a co-author per se for a student's work. It is possible to exploit a BSP report to produce a scientific and technical publication. In this case, the authors list has to be discussed and agreed with the concerned parties.

1. Plagiarism statement

This 350 words section without this first paragraph must be included in the submitted report and placed after the conclusion. This section is not counting in the total words quantity.

I declare that I am aware of the following facts:

- As a student at the University of Luxembourg I must respect the rules of intellectual honesty, in particular not to resort to plagiarism, fraud or any other method that is illegal or contrary to scientific integrity.
- My report will be checked for plagiarism and if the plagiarism check is positive, an internal procedure will be started by my tutor. I am advised to request a pre-check by my tutor to avoid any issue.
- As declared in the assessment procedure of the University of Luxembourg, plagiarism is committed whenever the source of information used in an assignment, research report, paper or otherwise published/circulated piece of work is not properly acknowledged. In other words, plagiarism is the passing off as one's own the words, ideas or work of another person, without attribution to the author. The omission of such proper acknowledgement amounts to claiming authorship for the work of another person. Plagiarism is committed regardless of the language of the original work used. Plagiarism can be deliberate or accidental. Instances of plagiarism include, but are not limited to:
    1) Not putting quotation marks around a quote from another person's work
    2) Pretending to paraphrase while in fact quoting
    3) Citing incorrectly or incompletely
    4) Failing to cite the source of a quoted or paraphrased work
    5) Copying/reproducing sections of another person's work without acknowledging the source
    6) Paraphrasing another person's work without acknowledging the source
    7) Having another person write/author a work for oneself and submitting/publishing it (with permission, with or without compensation) in one's own name ('ghost-writing')
    8) Using another person's unpublished work without attribution and permission ('stealing')
    9) Presenting a piece of work as one's own that contains a high proportion of quoted/copied or paraphrased text (images, graphs, etc.), even if adequately referenced

Auto- or self-plagiarism, that is the reproduction of (portions of a) text previously written by the

1. i.e. approximately 16 pages double columns excluding the Plagiarism Statement

author without citing that text, i.e. passing previously authored text as new, may be regarded as fraud if deemed sufficiently severe.

2. Introduction (± 5% of total words)

In the dynamic landscape of artificial intelligence, reinforcement learning stands out as a powerful paradigm, offering a pathway for machines to learn and adapt through interaction with their environment. This paper embarks on an exploration of reinforcement learning, with a particular focus on the pivotal aspect of evaluating models within this framework. Reinforcement learning, often likened to a training process inspired by behavioral psychology, has showcased remarkable capabilities in diverse applications, ranging from robotics to gaming.

As one delves into the intricacies of reinforcement learning model evaluation, it is illuminating to consider its practical manifestation, such as in mastering complex games like Pong. The ability of reinforcement learning algorithms to autonomously learn optimal strategies and policies in game environments reflects a paradigm shift in AI capabilities. This paper aims to provide insights into the evaluative methodologies that underpin the success of reinforcement learning models, showcasing their adaptability and potential in navigating the challenges posed by dynamic and interactive scenarios, exemplified by games like Pong.

3. Project description (± 10% of total words)

3.1. Domains

3.1.1. Scientific .

- **Neural Networks and Deep Learning**
  In the field of Artificial Intelligence, the concepts of neural networks are of great use. Neural networks are computational models which are inspired by the structure and the way that a human brain operates, hence the name. These networks consist of neurons separated into various layers. The individual nodes usually have some numerical data assigned to them, and each neuron is connected with the neurons in the previous and following layers. Each of these connections have a weight assigned to them. Neural networks are trained on quantifiable input data to learn patterns and relationships in the input data. The weights are then adjusted to make accurate predictions from new input data. Deep Learning is a subfield of machine learning which focuses on these neural networks consisting of multiple layers. It is particularly useful of image and speech recognition, playing strategic games and natural language processing.
- **Reinforcement Learning**
  Reinforcement Learning is one of the main types of machine learning. It consists of an agent which learns to make rewarding decisions by interacting with an environment. The agent starts by making random decisions, analyzing what works through quantitative rewards or punishments, and adjusting its decisions to maximize the reward. To achieve this, the agent requires knowledge about the environment's possible actions, states and rewards. A training algorithm must also be chosen for the agent.
- **Evaluation Metrics**
  Evaluation metrics are a set of measures which are used to evaluate the performance of a machine learning model. There are many types of metrics such as accuracy, precision, mean squared error, etc... The type of metrics that are useful heavily depend on the type of problem that is trying to be solved using machine learning.

3.1.2. Technical.

- **The Python Programming Language**
  Python is a high level programming language known for its simplicity, readability and versatility. It is very useful when working in the fields of machine learning and data science due to the abundance of useful external libraries, such as TensorFlow and Keras.
- **OpenAI Gym** OpenAI Gym is a toolkit to develop reinforcement learning models. It provides many environments of popular games which allows users to test and evaluate their models. The environments usually have well-defined action spaces, observation spaces and rewards, which provide useful information for reinforcement learning models. OpenAI Gym also allows for the possibility of custom environments.
- **TensorFlow** Tensorflow is an open-source machine learning framework It provides a platform to build and deploy machine learning models for a wide range of tasks
- **Keras** Keras is an open-source neural network API. It is an integral part of TensorFlow. It provides an intuitive interface to build and train neural networks.

3.2. Targeted Deliverables

3.2.1. Scientific deliverables. The main goal of the scientific deliverable section of the paper is to answer the following scientific question: How can one reliably evaluate reinforcement learning models, and how can these evaluations be used to compare various models which aim to complete the same goal?

3.2.2. Technical deliverables. The technical deliverable of this paper focuses on the implementation of an agent capable

of playing the game Pong in the environment provided by OpenAI Gym. The agent should learn by implementing reinforcement learning techniques and should be able to react to the inputs of the environment in such a way that it can consistently score points. The agent should also be able to be evaluated during and after the training process in order to see if there is any improvement.
OpenAI provides an environment to play Atari Pong with. Useful information can be extracted from this environment such as the game state, the rewards, the frames and possible actions. This information is in turn processed and given to the agent. The agent uses the processed information for every single decision it makes, and in turn learns from the outcomes of its decisions. This is done with the help of a training loop and appropriate training algorithms. This process is repeated until the agent can consistently make good decisions.
Some evaluation functions and metrics will also be implemented to evaluate and perhaps compare the model with another.

4. Pre-requisites ($[5\%..10\%]$ of total words)

Describe in these sections the main scientific and technical knowledge that is required to be known by you before starting the project. Do not describe in details this knowledge but only abstractly. All the content of this section shall not be used, even partially, in the deliverable sections. It is important not to include in this section all the knowledge you have been obliged to acquire in order to produce the deliverable. It should only state the knowledge the student possessed before starting the project and that was mandatory to possess to be capable to produce the deliverables. It explicitly defines the technical and scientific pre-condition for the project. It is also useful to avoid project failures due to over or under complex subjects.

4.1. Scientific pre-requisites

No scientific pre-requisites are required to understand the scientific deliverable section of this paper.

4.2. Technical pre-requisites

Knowledge of the Python programming language will be required to follow along and understand the technical deliverable section of this paper more clearly.

5. A Scientific Deliverable 1

For each scientific deliverable targeted in section 3.2 provide a full section with all the subsections described below.

5.1. Requirements ($\pm$ 15% of section's words)

How can one reliably evaluate reinforcement learning models, and how can these evaluations be used to compare various models which aim to complete the same goal?

The scientific deliverable aims to study the implementation and evaluation of reinforcement learning models. The functional requirements of the scientific deliverable section are as follows:

- **Getting an understanding of what reinforcement learning is and how it works:** Before delving into what goes into evaluating a reinforcement learning model, it is important to first understand how such a model functions and how it differs from other types of machine learning.
- **Understanding common evaluation metrics:** This requirement focuses on exploring pre-existing evaluation metrics, such as cumulative rewards, average rewards per episodes, etc...
- **Understanding reproducibility and statistical analysis:** To evaluate a model accurately, it is important to understand the reproducibility of the model to obtain consistent results, as well as how to analyze these results with statistical techniques.

Additionally, the scientific section must provide a logical progression from understanding the basic functionalities of reinforcement learning models to how such models are evaluated.

5.2. Design ($\pm$ 30% of section's words)

The first section of the production section provides an explanation of the core concepts of reinforcement learning. This section does not delve into too much detail, nor should it. It serves to provide the reader with a general understanding of how reinforcement learning works. After this section, the concepts of evaluation metrics are briefly explored. Finally, the production section goes over some important aspects of evaluating reinforcement learning models and how these evaluations can be used to compare multiple models against each other. The transition between these sections aims to be as logical and seamless as possible.

5.2.1. Understanding the core concepts of reinforcement learning. The purpose of this section is to provide the reader with an introduction to the topic of machine learning and reinforcement learning, as well as some comprehensive definitions of important subtopics.

It explains what separates reinforcement learning from other types of machine learning, as well the main functionalities of reinforcement learning. This includes agents, environments, states, policies, actions, rewards and exploration versus exploitation. Delving into these concepts is a key component of understanding how reinforcement learning works

5.2.2. Reviewing evaluation metrics for reinforcement learning. This section serves to introduce the concept of

evaluation metrics. The definition of evaluation metrics is given, along with some examples of the most popular evaluation metrics. This includes an explanation of cumulative rewards, average rewards and average episode lengths.
There are many more types of evaluation metrics, but going into detail for all of them would unnecessarily complicate this paper. This section serves more to understand why different evaluation metrics exist and what their purposes are.

5.2.3. Understanding reproducibility and the analysis of models using statistical methods. The last section focuses on the importance of the ability to replicate and verify experimental results and to employ statistical techniques for the analysis of reinforcement learning models. This section explains the importance of ensuring the reliability and credibility of a reinforcement learning model's results.

5.3. Production ($\pm$ 40% of section's words)

5.3.1. An understanding of reinforcement learning. Before delving into what reinforcement learning is and how it differs from other types of machine learning, we must first look at what machine learning is in general. Machine learning is a subset of artificial intelligence which focuses on developing algorithms which allow computers to learn patterns and make predictions without needing these predictions to be specifically programmed. These algorithms are able to perform tasks without specific instructions. Many fields have benefited from the development of machine learning techniques, such as image recognition, speech recognition, medical analysis and much more.

A large amount of data is usually required to train machine learning models. The model is usually exposed to a large dataset, often labeled, and attempts to adjust its parameters to make more accurate predictions. In more traditional types of machine learning, models receive labelled datasets and attempt to minimize between predicted results and actual results. Now that we have a general understanding of how machine learning works, what makes reinforcement learning stand out?

Reinforcement learning is one of the main three types of machine learning, with the other two types being supervised learning and unsupervised learning. As opposed to feeding a model with large datasets, in reinforcement learning, an agent learns by interacting with an environment. This agent receives feedback in terms of rewards or penalties and adjusts its behavior to achieve a predefined goal. In order to achieve this, there are many key concepts which need to be implemented in a reinforcement learning model:

- **Agent:** The agent is a fundamental component of a reinforcement learning model. The agent is an autonomous entity which is responsible for interacting with an environment with the goal of learning optimal actions to achieve a specific objective. The agent is able to perceive the state of the environment, and it is the primary decision maker.
- **Environment:** The environment is an external system or context which the agent is able to perceive and interact with. The environment provides feedback to the agent based on the selected actions of the agent.
- **State:** The state is the representation of the current situation or configuration of the environment. This can be provided in many different ways, such as images, scores and so on. Quite often, the state of the environment will change after a certain action has been taken by the agent.
- **Action:** The environment usually provides the agent or player with a set of actions. This set of actions can either be discrete or continuous. With discrete actions sets, the agent is able to choose from a predefined list of options. An example of this would be chess, where the agent only has a limited number of moves for each state. With continuous action sets, actions can take on a continuous range of values. An example of this would be moving or tilting a mechanical component.
- **Rewards:** Rewards are a numerical feedback provided by the environment to evaluate the efficacy of an action. The goal of the agent is to maximize this reward, most often cumulatively. Defining a reward function for the agent is a tricky endeavor, as it is quite easy to fall into the trap of local maximas. This refers to situations where the agent gets stuck in a suboptimal policy or strategy that appears optimal in its immediate vicinity, but is not the globally optimal solution for the entire problem.
- **Training policy:** A training policy refers to the way an agent employs a learning mechanism to adapt its strategy over time. There are many types of learning policies with their differing benefits, but the one thing that most have in common is a good balance between exploration and exploitation. This refers to the trade-off between exploring new strategies by selecting random action or exploiting known strategies by using high expected rewards. Finding a good balance between these two factors is crucial in ensuring the success of the reinforcement learning model.

Some of the main characteristics that differentiate reinforcement learning from other types of machine learning is not needing labelled input and output pairs to be presented. The reinforcement learning agent is usually thrown into the environment and learns optimal policies by itself.

5.3.2. Evaluation metrics. Evaluation metrics are quantitative measures used to assess the performance, accuracy, and effectiveness of a trained model or agent on a specific task or problem. With the use of these metrics, we can find a systematic way to quantify the performance of the agent in terms of achieving the desired results.

These metrics provide a way to compare the results of the agent to the results of other agents or models, especially benchmark metrics. Benchmark metrics serve as a standard for comparison.

By analyzing these metrics, programmers can more successfully identify issues with the structure of the current model and improve upon it. This process is of utmost importance when it comes to achieving desired results. It is very rarely the case that a model is optimally built for a purpose the first time. There are no perfect parameters for every type of model and task, so the parameters must be adjusted upon evaluating the model based on these metrics.

There are many different types of evaluation metrics which suit the needs of reinforcement learning. Some of the most commonly used evaluation metrics are the following:

- **Cumulative Rewards:** Cumulative rewards measure the sum of rewards obtained by the agent over a defined period. These types of rewards more accurately capture the impact of an agent's actions over an episode. Reinforcement learning often involves long-term decision-making, where the consequences of a single action might not be very apparent. These types of rewards help the agent consider the future and make decisions that lead to favorable outcomes over an extended period of time.

$$C_t = \sum_{k=0}^{T} r_{t+k+1}$$

- **Average Rewards:** The average reward is the cumulative reward divided by the number of steps taken in that time. This approach provides a more normalized measure of the agent's performance over an extended period of time.

$$A_t = \frac{1}{t} \sum_{k=0}^{t-1} r_{k+1}$$

- **Discounted Sum of Rewards:** Discounted sum of rewards considers the impact of future rewards by discounting them based on a factor, often labelled as gamma. These types of rewards encourage the agent to prioritize immediate rewards while still valuing future rewards. The function for this can be expressed as

$$R_t = \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k+1}$$

with R_t being the discounted sum of rewards at time t, γ being the discount factor, and r_{t+k+1} being the reward obtained at time $t + k + 1$
Discounted sum of rewards help account for the uncertainty and potential changes in the environment. The adjustment of the discount factor γ allow for a custom balance of the importance of short-term and long-term rewards.
- **Episodic Rewards:** As the name suggests, this type of evaluation metrics evaluate the agent's performance by analyzing the score or rewards at the end of an episode. Episodic rewards can often be useful when dealing with games with levels or episodes.

5.3.3. Understanding reproducibility and statistical method for the sake of comparisons. Understanding reproducibility and employing statistical methods are essential aspects of conducting rigorous and reliable research, especially in the context of comparing reinforcement learning models. Reproducibility refers to the ability to reproduce or replicate experimental results, ensuring that the findings are consistent and valid across different settings, datasets, or implementations.

Ensuring reproducibility in a model involves adopting good practices throughout the research and design process of the model. One of the most important practices to take is efficient and clear code documentation. This includes logging all the parameters which were used to create an agent, specifying which seeds were used in the environment, listing exact versions of external resource. Documenting these types of data is vital for ensuring reproducibility, as the steps can then be easily followed for other people to evaluate their models against yours for comparison.

Once the model and its expected outputs can be reproduced consistently, the task of statistical analysis of the model can begin. There is not much sense in overanalyzing a model if the results from it can not be reproduced. By adopting a comprehensive approach to statistical analysis, researchers can elucidate the significance of observed trends, assess model performance with confidence intervals, and employ hypothesis testing to validate key assumptions. The integration of statistics offers a probabilistic framework, enhancing the overall transparency of RL experiments.
The implementation of these statistical methods can greatly help to ensure the validity and robustness of models.

- **Hypothesis testing:** Hypothesis testing helps determine whether observed trends are statistically significant or could be attributed to random variation. The process involves formulating a hypothesis, collecting data, and performing statistical tests to draw conclusions.
- **Confidence intervals:** Confidence intervals provide a range of plausible values for performance metrics, quantifying uncertainty. In the context of reinforcement learning models, confidence intervals can be used to estimate the uncertainty associated with performance metrics, such as mean rewards or success rates. They are typically from performance metrics derived from experimental data.
- **Bayesian statistics:** Bayesian statistics is an approach to statistics which views probability as a

measure of belief or uncertainty. In the context of evaluating reinforcement learning models, Bayesian methods provide a flexible framework for updating and refining beliefs about model parameters based on both prior knowledge and observed data.

- **Cross validation:** This is a statistical technique commonly used to assess the performance of reinforcement learning models. The primary goal of cross-validation is to estimate how well a model will generalize to an independent dataset. It is a valuable technique which systematically assessing the performance of models on multiple subsets of the data.

A combination of statistical techniques such as these on accumulated data from the previously defined evaluation metrics provide a significant insight into the performance of a reinforcement learning model, particularly when comparing multiple models. By harnessing a comprehensive array of statistical methodologies, one can delve deeper into the nuanced aspects of model behavior, elucidating patterns, trends, and potential areas for improvement.

In the context of comparing multiple models, the application of statistical techniques takes on an added layer of significance. It becomes a discerning tool for model selection, helping to identify the nuances that set one model apart from another. The comparative statistical analysis not only quantifies the disparities in performance but also untangles the underlying factors contributing to these variations.

5.4. Assessment ($\pm$ 15% of section's words)

The research of the previous section aimed to answer the question "How can one reliably evaluate reinforcement learning models, and how can these evaluations be used to compare various models which aim to complete the same goal?". In order to answer this question, the research made sure to delve into various key aspects of the topic.

The initial segment provided a broad perspective on the mechanics of reinforcement learning. Although not delving into intricate details, the overview successfully accomplishes the objective of furnishing the reader with a foundational understanding of the operational principles underlying reinforcement learning models.
It serves to demystify the workings of these models, setting them apart from other categories within the realm of machine learning. This section succinctly elucidates key components of reinforcement learning, including agents, actions, rewards, and more, offering a brief yet insightful glimpse into the fundamental aspects that shape the landscape of this learning paradigm.

The subsequent segment centered on an exploration of diverse evaluation metrics applicable to the assessment of a reinforcement learning model. The primary objective was to elucidate the purposes served by these metrics and to underscore the rationale behind the existence of various metric types. Additionally, the section aimed to provide examples of some of the most commonly employed metrics without delving into exhaustive detail. Noteworthy metrics such as cumulative rewards, discounted sum of rewards, among others, were addressed, contributing to a more comprehensive overview of the evaluative landscape within the realm of reinforcement learning models.

Conclusively, the paper delved into the significance of reproducibility and the application of statistical methods within the framework of evaluating and comparing reinforcement learning models. While the intricacies of these topics could be explored at a microscopic level, the primary aim of this paper diverges from such exhaustive depth. Rather, its overarching purpose lies in offering a broad perspective on commendable practices in the evaluation and comparison of models within the context of reinforcement learning.

The production section, in essence, furnishes a robust portrayal of essential practices, striking a balance between comprehensiveness and specificity, thereby providing a foundational understanding of model assessment and comparison best practices.

6. A Technical Deliverable 1

For each technical deliverable targeted in section 3.2 provide a full section with all the subsections described below. The cumulative volume of all deliverable sections represents 75% of the paper's volume in words. Volumes below are indicated relative to the section.

6.1. Requirements ($\pm$ 15% of section's words)

For the technical deliverable section, there are several requirements to be fulfilled. The program created for this program should be able to fulfill the following requirements:

6.1.1. Gather information from the OpenAI Gym environment and preprocess this information. The program should be able to interact and use the OpenAI Gym environment. This information will be needed to train agents later on. In order to make this process smoother, the information gathered from the environment should be selectively preprocessed, so that only necessary information from the environment is provided to any agents.

6.1.2. Create an agent which is capable of interacting with the preprocessed data. As one of the main goals of this section is to train and evaluate a custom agent's capabilities of playing Pong, it is naturally important that the agent is able to interact with the provided preprocessed data about the environment. The agent should be able to take in this information and provide an output which corresponds to one of the possible actions which the environment provides.

6.1.3. Train the agent and save the model. The program should provide a way for reinforcement learning agent's to train on pre-processed data to further its ability to make better decisions. Once an agent has been trained for a desired number of steps, the program should also be save the agent's training process, so that it can be trained or tested later.

6.1.4. Evaluate the agent's capabilities of playing Pong. The program should also provide a way to test the agent and evaluate the agent's capabilities based on its performance. This function is very useful when trying to evaluate if the training process has had any impact on the agent's average performance.

6.2. Design (± 30% of section's words)

6.2.1. The preprocess_frame class. The design of the preprocess_frame class is composed of multiple functions which have the goal of extracting information from the OpenAI Gym Pong environment and preprocessing it for later use.
The class uses various methods from the rl.core.Processor, cv2 and PIL.Image libraries. There are three functions in the preprocess_frame class with different functionalities:

- **process_observation** The purpose of this function is to extract the most important information about the current state of the game environment, namely the positions of the paddles and ball, for future use. The function takes a single frame from the game environment as an input. This frame is first cropped, so that the score is no longer visible. The next step is to turn the image into a grayscale image. The image is turned into a 84x84 pixel image and each pixel is given an RGB value of (0,0,0) (black) or (255,255,255) (white) depending on if the current RGB value is smaller than (144,144,144). The function finally returns a NumPy array of 0s and 255s of size 84x84, with each entry corresponding to a black or white pixel in the cropped image.

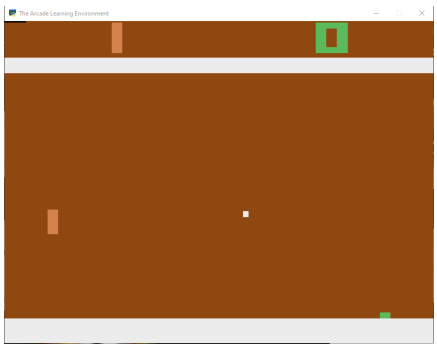

Figure 1: An example frame from the OpenAI Gym Pong environment in "human" render mode

- **process_state_batch** This function processes an entire batch of states from the environment and returns. Each element in the arrays with the value 255 also are turned into 1 instead. The reason this is not done in process_observation earlier is for visualization purposes.
- **process_reward** This function processes the reward from the game environment for later use in an agent and returns it. The NumPy clip method is also used to assure that no rewards are returned outside the desired boundary.

6.2.2. The CustomModel class. This class serves as a blueprint for creating a neural network for the reinforcement learning agent. The input shape of the model and the number of actions in the action space of the environment are passed to the model as the variables input_shape and actions.
The model is initialized in this class and various layers are added to it. The input layer of the model receives the given input shape and the output shape is determined to be the size of the given action space.
Several so called 'hidden layers' are also created, which better help agents train. The following types of layers are present in the model:

- **Permute layer** This type of layers permutes the dimensions of the input tensor
- **Conv2D layer** This convolutional layer apply a desired number of filters of a given input size. Various features, such as blurring, sharpening, edge detection and more are extracted in this layer using the filters. The input "stride" is used to determine how many pixels the filter moves at a time.
- **Activation "relu"** This layer applies a rectified linear unit activation function to the previous layer. This introduces non-linearity to the neural network. The purpose of this is to allow the model to learn more complex patterns in the given data.
- **Dense** Dense layers are fully connected layers. An input size "n" is given to this type of layer to determine the number of neurons. Every single neuron in the Dense layer is connected to every neuron in a previous layer.
- **Flatten** The purpose of Flatten layers is to transform multidimensional inputs into a one-dimensional array.

6.2.3. The Memory class. The Memory class provides a blueprint to create of the SequentialMemory library from rl.memory. The size of the memory and the window length are given as an input for this class. Window length determines how many frames are given as an input to the agent for a single decision.

6.2.4. The Agent class. This class provides a blueprint for creating and managing an agent which interacts with the OpenAI Gym Pong environment. The following parameters are passed when creating an instance of the Agent class: input_shape, actions, mem_len_lim, window_len, checkpoing_filename. The main design functionalities are as follows:

- **Memory Initialization** The agent class uses the custom Memory class to manage the replay memory for experience while training. While training, the agent analyzes the entries in memory to improve.
- **Neural Network initialization** The agent creates an instance of the CustomModel class. This network is responsible for processing input frames from the OpenAI Gym Pong environment and making predictions for the best action to take.
- **Load saved weights** When creating an instance of the Agent class, the class checks whether a file exists with the same name as the "checkpoint_filename" variable. If this is the case, then the class loads in the saved weights from that file so that the agent can continue the training from a previously saved state.
- **Create the Deep Q learning agent** The create_dqn_agent method creates a Deep Q-Network agent using the Keras-rl library. The agent is configured with various parameters such as the neural network model, the memory, the agent's exploration policy, the learning rate and other training-related settings. Once all the necessary parameters have been defined, the agent is then compiled and is ready for training.

6.3. Production (± 40% of section's words)

In this section, the process of creating and implementing the program and agent, including considerations and the general thought process.

6.3.1. Selecting the language, key libraries and environment. The first step of the process was deciding which programming language to use for the project, as well as how the environment for the pong game. In terms of programming languages, Python naturally seemed like the natural choice, due to the availability of many open-source machine learning libraries such as TensorFlow, Keras, PyTorch and so on.

The process of implementing key features from these libraries for personal projects is made simple due to the abundance of guides and their documentations. In terms of choosing a library for the machine learning related features, the choice for this particular project was TensorFlow.
TensorFlow is a versatile framework which supports a wide range of machine learning related tasks, such as deep learning, reinforcement learning and more. As this project focuses on reinforcement learning, TensorFlow was a natural fit.

The next step was to choose the environment for Pong. Instead of opting for a custom environment setup, I chose to leverage a pre-existing one. This decision aligns with the project's primary focus on developing and evaluating the reinforcement learning agent, making the use of a readily available Pong environment more pragmatic.
I chose to use the OpenAI Gym Pong environment. This environment's accessibility, well-documented interface, and community support simplify implementation of reinforcement learning models.

6.3.2. Setting up the environment and understanding the parameters. OpenAI Gym makes it very simple to set up any desired environments. The command "env = gym.make("ALE/Pong-v5") sets up the environment for Pong. Additionally, the argument "render_mode" can be passed in to visualize the game environment. When setting up a reinforcement learning model, it is of utmost importance that the game parameters, actions, states and rewards are clearly defined, so that this information can be given to a potential agent. Upon reading the documentation, I learned that the OpenAI Gym environment provides six possible actions. These are as follows:

0	1	2	3	4	5
NOOP	FIRE	RIGHT	LEFT	RIGHTFIRE	LEFTFIRE

TABLE 1: Table of actions in OpenAI Gym

In order to use one of these actions manually in the environment, the built-in step(action) function can be used. This function also returns four values:

- **observation:** The observation is a representation of the current state of the environment, typically as an array or tensor.
- **reward:** The reward is a numerical value which represents how well the agent or player is performing at the current time step. Positive rewards indicate desirable actions, such as scoring a point.
  In the case of the Pong environment, the following rewards can be obtained at any given time step: -1 if the opponent scores a point; 0 if the ball is still in play; 1 if the agent or player scores a point.
- **done:** This flag is a boolean value which indicates whether the game episode has terminated or not. An episode refers to a single run or instance of an agent interacting with the environment from an initial state to a terminal state.
- **info:** This is a dictionary containing some additional information about the game, but this information will not be used when training the agent.

The first three of the values would play an important part in training a potential agent, since they will provide feedback about the agent's selected actions during training. Once I had understood the most important parameters of the game, the next step was to choose how to process this information to provide it to a potential agent.

The environment also provides an opponent to play against. This opponent is a hard-coded bot which simply follows the position of the ball with its paddle. The behavior of the bot is consistent, which makes sure that our agent can train properly.

6.3.3. Preprocessing. Preprocessing involves transforming and manipulating raw observations from the environment before feeding them to the agent's training algorithm. For this particular project, I decided the image processing would be used. The OpenAI Gym provides the observation in the forms of images, which are conveyed through arrays and tensors. While it is possible to simply feed these tensors to the agent, they do contain quite a bit of unnecessary information, which would complicate the training process of the agent. Below is an example of a frame which is provided by the environment.

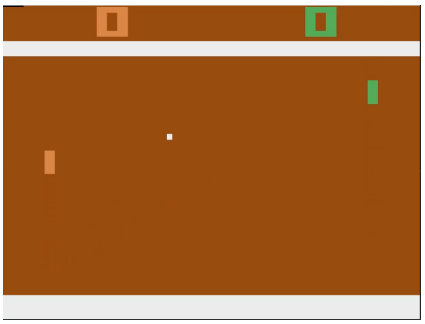

Figure 2: 2nd example frame from the OpenAI Gym Pong environment in "human" render mode

There are several issues with the provided frame for training purposes. Firstly, the frame displays the current score of the game as well as some unnecessary white bars. Any potential agent will not need to see the score, as they will receive the rewards for each time step anyway.
Secondly, the frame is too colorful. While the vibrant representation might be more aesthetically pleasing to a human observer, there is not really any need for more than two colors to differentiate the ball and paddles from the background. If the number of possible inputs is limited, it can make the training process a lot simpler.

The preprocessing of the observation will be done in a separate class called preprocess_frame. The first step of the preprocessing will be cropping the frames which the environment provides. I decided to do this with so that only the game is visible using a NumPy array of size (84, 84).
Once this is done, the image is converted to grayscale using the cv2 library's built-in grayscale function. While grayscale reduces the number of colors, the number can still be further reduced to two colors. I decided to do this by using the RGB values of the remaining pixels' colors. An RGB value of 144 is used as a threshold, so any values below that become black and values above it become white. The result is the following:

6.3.4. Deciding the reinforcement learning method and the layout of the neural network. Once the preprocessing was completed, the next step was to decide how a potential neural network would process this information. For this project, I decided to go with a deep learning neural network, as well as a Q learning approach.

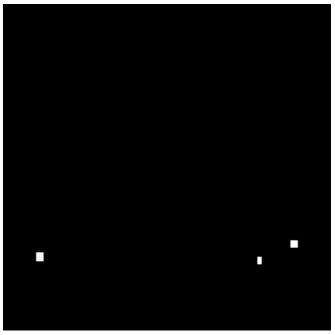

Figure 3: Example of a processed frame displayed with cv2

There are several approaches to implement reinforcement learning for such problems such as policy-gradient methods, Monte-Carlo methods, and more, but for this project, Deep Q learning was the logical choice. Deep Q learning is effective in handling high-dimensional input spaces, like images. This enables the model to learn relevant features from the pixel data.

Defining the structure of the neural network is not something that can simply be estimated for any given project, and consists of tweaking and restructuring to see desired results. A Deep Q learning neural network is composed of various interconnected layers. The layers can usually be categorized into various types. The simplest to define layers are usually the input and out layers.
As the action space of the environment consists of six distinct actions which are referenced in table 1, the output layer of the model should logically consist of six neurons, each corresponding to one of these actions. This output layer is responsible for producing the network's predictions based on input data.

The selection of the input layer is a critical decision in designing the neural network architecture for Pong reinforcement learning. Initially, it may seem intuitive to choose an input size of (84, 84), assigning one neuron for each pixel in the processed image.
However, relying solely on a single frame for predictions lacks the temporal information necessary to accurately anticipate the ball's movements.

To address this limitation and provide the model with a history of the ball's positions, multiple frames are concatenated to form a single input. In this case, I opted for a sequence of 12 frames for each action. This would give us the following number of neurons:

$$12 * (84 * 84) = 84672$$

This choice allows the model to capture the temporal dynamics of the game, enhancing its ability to make more informed and accurate predictions. It's worth noting

that the specific number of frames, in this case, can be adjusted based on individual preferences and the observed performance of the model.

The choice of the intermediate layers is a much more complex and tedious process. Initially, I started out with one conventional intermediate Dense layer consisting of 200 neurons. The exact number of neurons was a fairly arbitrary choice. Each of these neurons was connected to the input and output layer and have weights assigned to them which are tweaked. During training, I was not seeing the desired results, so I would end up tweaking this layout a number of times. The most important of these changes was introducing multiple Keras Conv2D layers alongside the standard Dense and Permute Layers.
Conv2D, or 2D convolutional layers, play a big part in processing visual inputs. They apply filters to capture spatial features of in the input frames. I decided to have multiple of these layers with a decreasing stride length. This allows for hierarchical feature extractions. Layers with higher stride lengths capture high-level and more abstract features, whereas layers with lower stride lengths are able to capture more fine-grained details.
For each of these Conv2D layers, a "relu", or rectified linear activation function, is used. The function has the following properties:

$$relu(x) = max(x, 0)$$

$$\text{relu}(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{if } x \leq 0 \end{cases}$$

If the input value is below 0, then 0 is returned. Otherwise, the input value is returned. "relu" introduces non-linearity to the model, which allows for more complex patterns and relationships.
Once all the most important characteristics of the model are defined, it can then be compiled. This entire process is done inside a 'model' class.

6.3.5. Creating the reinforcement learning agent. When creating a reinforcement learning agent, there are many aspects to consider when creating it, such as memory management, saving the agent, implementing the previous neural network architecture and most importantly, the policies. The DQNAgent library from rl.agents provides a useful template for creating reinforcement learning agents. This agent is capable of implementing Deep Q learning algorithms. DQNAgent was created in the context of playing Atari games, and it also has great compatibility with OpenAI Gym.

I initially implemented a memory management system for the agent using the straightforward SequentialMemory library. This library allowed easy selection of the memory limit and window length, with the window length set to 12 frames to align with the decision to provide the neural network with this number of frames. The chosen memory limit, set at 600,000, ensures sufficient data storage for effective training.

Subsequently, attention was directed towards establishing a mechanism for saving the model's weights. The ModelIntervalCheckpoint library simplifies this process. When a new DQNAgent model is compiled, it checks for the existence of a specified filename, which is provided during the instantiation of the Agent class. If the filename exists, the model attempts to load previously saved weights from that file; otherwise, a new file with the given filename is created. The models are stored in the "h5f" file type.

The selection of a training policy is a crucial step in building a reinforcement learning agent. In this project, I opted for the Epsilon-Greedy policy as the agent's guiding strategy. Epsilon-Greedy provides a straightforward mechanism for the agent to balance between exploration and exploitation, optimizing its actions to maximize cumulative rewards.
The Epsilon-Greedy policy provides a simple way for the agent to balance between exploring new actions and exploiting known actions to maximize rewards. This policy works by letting the agent choose the action with the highest estimated value with a probability of $1 - \epsilon$. This is considered the "greedy" action. The ϵ value will always be in the range $[0, 1]$, so that means that the agent also has a probability of ϵ of choosing a random action.

$$\text{current choice}(x) = \begin{cases} max(Q), & \text{with } probability 1 - \epsilon \\ rand(Q), & \text{with } probability \epsilon \end{cases}$$

with Q being the set of calculated Q values for each action

When the agent first begins training, the value of ϵ is set to 1. This ensures that the agent tries completely new actions all the time, since it does not have much data to learn from. Over time, the ϵ value is slowly lowered as the agent's memory grows. This ensures that the agent still explores new actions while not completely disregarding any previous strategies. It allows the agent to improve on what it has learned by trying new things.

With these important parameters defined, including the previously defined preprocessing class and neural network architecture, the DQNAgent can be compiled using these features. The agent requires some additional parameters to be set before compilation. The most notable ones include:

- **nb_steps_warmup:** This determines the number of random steps the agent takes before applying starting to use its learned policy. When training the model for the first time, this was set to 50000, but once the model had been saved, the value was set to 0. This is so that the model instantly picks up from where it stopped during retraining.

- **target_model_update:** This determines the frequency at which the Deep Q network model is updated. Updating the model for every step would be very impractical, so this target is used. The frequency in our model is every 1000 steps.
- **learning rate "lr":** This refers to the step size during the update of the Deep Q network's weights. It determines how much the model's weights are adjusted by. For our model, a learning rate of 0.00025 was chosen.

6.3.6. Creating the training and testing environment. In the main function, users have the flexibility to choose between training, evaluating, or visually displaying the agent's performance upon program startup.

The training function allows the agent to learn over a specified number of steps. The agent trains using the DQNAgent's built in fit() function.

The evaluation function assesses the agent's performance across 200 episodes and compares it to a random action bot, with results visualized on a graph. The agent plays using the DQNAgent's built in test() function. To enhance efficiency, render mode is disabled during training and evaluation.

However, for the display function, render mode is set to "human," providing a visual representation of the agent's gameplay over a few episodes. It's important to note that this function is designed solely for showcasing the agent's abilities, without collecting data for evaluations.

6.4. Assessment (± 15% of section's words)

The core objective of this project was to develop a proficient reinforcement learning agent capable of excelling in OpenAI Gym's Pong environment. The journey involved comprehending fundamental concepts in machine learning and reinforcement learning and seamlessly integrating them into the program. Although the initial challenge was creating an agent adept at interacting with the environment and generating predictions, the primary focus shifted to meticulously fine-tuning specific aspects. The greater challenge lay in optimizing the agent's performance and ensuring its proficiency in playing Pong.

Key insights and results obtained through various evaluation methods simplified this process by instilling a sense of confidence in the agent's learning capabilities. To assess the agent's performance, I consistently compared its results with those of a bot playing randomly. At the outset of the agent's training, both the agent and the random bot exhibited nearly identical results. Despite the agent utilizing continuous rewards for determining its outcomes, I opted to use Pong game scores for evaluations. An episode of the Pong environment ends when one of the players reaches a score of 21. The average score for the bot playing randomly over the course of 200 episodes was 0.735 points. The agent naturally had a similar average at the beginning of training. During the agent's training process, which would span many hours, intermediate evaluations were made to check up on the progress of the agent. After the agent's parameters and structure had been modified and tweaked enough times, the average score of the agent started going up. The average score was always evaluated over 200 test episodes.

After close to 2 million training steps, the agent now averages 13.83 points per game, with the agent also being capable of winning games against the hard-coded bot on some occasions. Below is a figure displaying the results of the agent at 2 million steps compared to the bot, which makes random decisions.

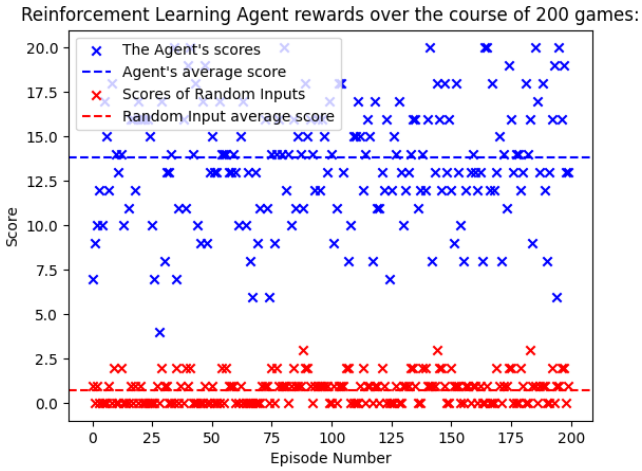

Figure 4: Scores of our RL agent compared to scores of random bot

Using the data from these evaluations, it is fair to say the main requirements of the technical deliverable section were fulfilled, namely creating a reinforcement learning agent capable of playing Pong.

Acknowledgment

The authors would like to thank the BiCS management and education team for the amazing work done.

7. Conclusion

This Bachelor Semester Project was successfully able to fulfill the scientific and technical requirement. In the scientific section, the paper gave an explanation on how to evaluate and compare reinforcement learning models without going into too much detail. Key aspects of reinforcement learning, its respective evaluation metrics and some useful statistical method were presented.

For the technical section, this paper successfully provides a detailed explanation of the implementation and evaluation of a reinforcement learning model designed to play the game Pong. With the help of the technical section, readers should be able to reproduce the steps and create a similar model. The main requirements of the technical section were fulfilled, and the model was capable of playing the game.

References

[BiCS(2021)] BiCS Bachelor Semester Project Report Template. https://github.com/nicolasguelfi/lu.uni.course.bics.global University of Luxembourg, BiCS - Bachelor in Computer Science (2021).

[BiCS(2021)] Bachelor in Computer Science: BiCS Semester Projects Reference Document. Technical report, University of Luxembourg (2021)

[Armstrong and Green(2017)] J Scott Armstrong and Kesten C Green. Guidelines for science: Evidence and checklists. *Scholarly Commons*, pages 1–24, 2017. https://repository.upenn.edu/marketing_papers/181/

[1] Fenjiro, Youssef, and Houda Benbrahim. "Deep reinforcement learning overview of the state of the art." *Journal of Automation Mobile Robotics and Intelligent Systems* 12.3 (2018): 20-39. file:///C:/Users/ryanu/Downloads/Deep_reinforcement_learning_overvie.pdf

[2] Ayodele, Taiwo Oladipupo. "Machine learning overview." *New Advances in Machine Learning* 2 (2010): 9-18. https://books.google.lu/books?hl=en&lr=&id=XAqhDwAAQBAJ&oi=fnd&pg=PA9&dq=machine+learning+overview&ots=r3Fn9WBdJs&sig=vn6dlkHPoGBrTaqAx7s5sGLb32o&redir_esc=y#v=onepage&q=machine%20learning%20overview&f=false

[3] Gottesman, Omer, et al. "Evaluating reinforcement learning algorithms in observational health settings." *arXiv preprint* arXiv:1805.12298 (2018). https://openreview.net/pdf?id=HJgAmITcgm

8. Appendix

All images and additional material go there.

8.1. Source Code

Below is the source code used to create the reinforcement learning agent.

8.1.1. The main file.

Listing 1: main.py

```
import gym
import numpy as np
import matplotlib.pyplot as plt
import preprocess_frame
from rl.callbacks import Callback
import random
import visualkeras

import agent

processor = preprocess_frame.Frame_Processor()
total_steps = 600000
target_update_frequency = 10000
step = 0


def train_model():
    """
    The purpose of this function is to let an agent train in an
        environment and readjust its weights
    """
    env = gym.make("ALE/Pong-v5")
    env.reset()
    valid_actions = env.action_space.n
    img_shape = (84, 84)
    window_length = 12
    input_shape = (window_length, img_shape[0],
        img_shape[1])
    game_agent = agent.Agent(input_shape, valid_actions,
        total_steps, window_length, "checkpoint_file.h5f")
    env = gym.make("ALE/Pong-v5")
    env.reset()

    # Training function: This function trains the agent for a
        given number of steps and saves the weights of the
        model every given interval
    training_func = game_agent.dqAgent.fit(env,
        nb_steps=total_steps,
        callbacks=[game_agent.checkpoint_callback],
                                               log_interval=target_update_fre
                                                   visualize=False)
    env.close()


def test_model():
    """
    This function is here to collect data on the performance
        of the agent. The performance of the agent is then
        compared
    to either a different agent or random actions
    """

```

```
env = gym.make("ALE/Pong-v5")
env.reset()
ep_num = 200

class TestCallback(Callback):
    def __init__(self):
        self.episode_rewards = []

    def on_episode_end(self, episode, logs):
        # Collect the total episode reward
        self.episode_rewards.append(logs['episode_reward'])

test_callback = TestCallback()
valid_actions = env.action_space.n
img_shape = (84, 84)
window_length = 12
input_shape = (window_length, img_shape[0],
    img_shape[1])
game_agent = agent.Agent(input_shape, valid_actions,
    total_steps, window_length, "checkpoint_file.h5f")
game_agent.dqAgent.test(env, nb_episodes=ep_num,
    visualize=False, callbacks=[test_callback])
adjusted_rewards = [element + 21 for element in
    test_callback.episode_rewards]
env.reset()

games_played = 0
total_reward = 0
rand_total_reward = []
while True:
    action = random.randint(0, 5)
    observation, reward, done, info = env.step(action)
    total_reward += reward
    if done:
        rand_total_reward.append(total_reward)
        if games_played == ep_num - 1 :
            break
        else:
            games_played += 1
            total_reward = 0
            env.reset()

env.close()

rand_total_reward = [element + 21 for element in
    rand_total_reward]
plt.scatter(np.arange(len(adjusted_rewards)),
    adjusted_rewards, label="The Agent's scores",
    color="blue", marker="x")
plt.axhline(y=np.mean(adjusted_rewards), color="blue",
    linestyle="--", label="Agent's average score")
plt.scatter(np.arange(len(rand_total_reward)),
    rand_total_reward, label="Scores of Random
    Inputs", color="red", marker="x")
plt.axhline(y=np.mean(rand_total_reward), color="red",
    linestyle="--", label="Random Input average
    score")
plt.ylim(-1, 22)
plt.xlabel('Episode Number')
plt.ylabel('Score')
plt.title(f"Reinforcement Learning Agent rewards over
    the course of {(len(adjusted_rewards))} games:")
plt.legend()
print(f"Agent Episode Rewards:
    {adjusted_rewards}\nAverage Agent Reward:
    {np.mean(adjusted_rewards)}")
print(f"Random Action Rewards:
    {rand_total_reward}\nRandom Action Average:
    {np.mean(rand_total_reward)}")
plt.show()

def show_agent_playing():
    """
    This function is simply to display the agent interacting
        with the environment and playing to the best of its
        ability
    """
    env = gym.make("ALE/Pong-v5",
        render_mode="human")
    env.reset()
    valid_actions = env.action_space.n
    img_shape = (84, 84)
    window_length = 12
    input_shape = (window_length, img_shape[0],
        img_shape[1])
    game_agent = agent.Agent(input_shape, valid_actions,
        total_steps, window_length, "checkpoint_file.h5f")
    game_agent.dqAgent.test(env, nb_episodes=5,
        visualize=False)


def custom_test():
    # env = gym.make("ALE/Pong-v5",
        render_mode="human")
    observation = env.reset()
    np.set_printoptions(threshold=np.inf)
    processor = preprocess_frame.Frame_Processor()

    for i in range(10000):
        observation =
            processor.process_observation(observation)
        # observation =
            processor.process_state_batch(observation)

        action = game_agent.dqAgent.forward(observation)

        observation, reward, done, info = env.step(action)

        observation, reward, done, info =
            processor.process_step(observation, reward,
            done, info)

        game_agent.dqAgent.backward(reward,
            terminal=done)

        print(f"Action: {action}", f"Reward: {reward}")


while True:
    user_input = input("Type 1 to train the model\n"
                       "Type 2 to test the model\n"
                       "Type 3 to display the agent
                           playing\n"
                       "Type 4 to end the program")
    if user_input == "1":
        train_model()
        break
    elif user_input == "2":
```

```
        test_model()
        break
    elif user_input == "3":
        show_agent_playing()
    elif user_input == "4":
        print("Goodbye")
        break
    else:
        print("Not a valid input")
```

8.1.2. The agent class.

Listing 2: agent.py

```
#This file is responsible for creating the agent which
    interracts with the game environment
import keras.models
import numpy as np
from rl.agents import DQNAgent
from rl.policy import EpsGreedyQPolicy,
    LinearAnnealedPolicy
from rl.callbacks import ModelIntervalCheckpoint
from keras.optimizers import Adam
from memory import Memory
from model import CustomModel
from keras.models import load_model
from preprocess_frame import Frame_Processor


class Agent:
    def __init__(self, input_shape, actions, mem_lim_len,
        window_len, checkpoint_filename):
        self.checkpoint_filename = checkpoint_filename
        self.checkpoint_callback =
            ModelIntervalCheckpoint(self.checkpoint_filename,
            interval=1000)
        self.actions = actions
        self.mem_lim_len = mem_lim_len
        self.winow_len = window_len
        self.memory = Memory(self.mem_lim_len,
            self.winow_len)
        self.input_shape = input_shape
        self.neural_model =
            CustomModel((self.input_shape),
            self.actions).model
        self.try_load_weights()
        self.processor = Frame_Processor()
        self.dqAgent = self.create_dqn_agent()
        self.total_memory = []
        self.try_load_weights()

    def create_dqn_agent(self):
        epsilon_policy =
            LinearAnnealedPolicy(EpsGreedyQPolicy(),
            attr='eps', value_max=0.6, value_min=0.1,
            value_test=0.05, nb_steps=5000000)
        agent = DQNAgent(
            model=self.neural_model,
            memory=self.memory.mem,
            policy=epsilon_policy,
            processor=self.processor,
            nb_actions=self.actions,
            nb_steps_warmup=0,
            gamma=.99,
            target_model_update=1000,
            train_interval=12,
            delta_clip=1
        )
        agent.compile(Adam(lr=0.00025), metrics=["mae"])
        return agent


    def model_summary(self):
        print(self.neural_model.summary())

    #This function attempts to load a checkpoint if there is
        one available
    def try_load_weights(self):
        try:
            print("Checkpoint file found")
            self.neural_model.load_weights(self.checkpoint_filename)
        except:
            print("No checkpoint file found")
```

8.1.3. The model class.

Listing 3: model.py

```
#This file is the blueprint for the neural network which the
    agent will use
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense, Activation, Flatten,
    Permute, Conv2D
#from keras.initializers import he_normal


class CustomModel:
    def __init__(self, input_shape, actions):
        self.input_shape = input_shape
        self.actions = actions
        self.model = self.build_model()

    def build_model(self):

        model = Sequential()
        #These layers will help the model recognise what is
            going on on the screen.
        #We lower the stride length for each layer
        model.add(Permute((2, 3, 1),
            input_shape=self.input_shape))
        model.add(Conv2D(32, (8, 8), strides=(4, 4),
            kernel_initializer="he_normal"))
        model.add(Activation('relu'))
        model.add(Conv2D(64, (4, 4), strides=(2, 2),
            kernel_initializer="he_normal"))
        model.add(Activation('relu'))
        model.add(Conv2D(64, (2, 2), strides=(1, 1),
            kernel_initializer="he_normal"))
        model.add(Activation('relu'))

        #The layers above give us a multidimentional array,
            so we need to flatten these for our intended
            usage
        model.add(Flatten())

        model.add(Dense(500))
        model.add(Dense(1000))
        model.add(Activation('relu'))
        model.add(Dense(self.actions))
```

```
        model.add(Activation('linear'))

        return model

    def load_model_func(self):
        self.model =
            keras.models.load_model("DQN_CHECKPOINT.hf5")
```

8.1.4. The memory class.

Listing 4: memory.py

```
#This file is responsible for creating the memory for the
    agent
from rl.memory import SequentialMemory

class Memory():
    def __init__(self, limit_len, win_len):
        #win_len determines how many of the previous
            frames are fed to the agent for every
            step/decision
        self.win_len = win_len
        self.mem = SequentialMemory(limit=limit_len,
            window_length=self.win_len)
```

8.1.5. The preprocssing class.

Listing 5: preprocess_frame.py

```
import cv2
import numpy as np
from rl.core import Processor
from PIL import Image

class Frame_Processor(Processor):
    def process_observation(self, observation):
        shape = (84, 84)

        observation = observation.astype(np.uint8)
        observation = observation[34:34 + 160, :160]

        # If the frame has multiple color channels (e.g.,
            RGB), convert to grayscales
        if len(observation.shape) == 3:
            observation = cv2.cvtColor(observation,
                cv2.COLOR_RGB2GRAY)
        # We then use a threshhold (144) to convert the
            image to an array of 0s and 255s
        observation[observation < 144] = 0
        observation[observation >= 144] = 255
        observation = cv2.resize(observation, shape,
            interpolation=cv2.INTER_NEAREST)

        return observation

    def process_state_batch(self, batch):
        return batch.astype('float32') / 255

    def process_reward(self, reward):
        return np.clip(reward, -1.0, 1.0)
```