Εθνικο Μετσοβιο Πολυτεχνειο
Τμημα Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

Τομέας Τεχνολογίας Πληροφορικής & Υπολογιστών
Εργαστήριο Μικροϋπολογιστών & Ψηφιακών Συστημάτων

Training & Acceleration of Deep Reinforcement Learning Agents

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αναγνωστόπουλος Χ.
Κωνσταντίνος

Επιβλέπων: Δημήτριος Ι. Σούντρης
Καθηγητής

Αθήνα, Ιούλιος 2022

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Τομέας Τεχνολογίας Πληροφορικής & Υπολογιστών
Εργαστήριο Μικροϋπολογιστών & Ψηφιακών Συστημάτων

Training & Acceleration of Deep Reinforcement Learning Agents

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αναγνωστόπουλος Χ. Κωνσταντίνος

Επιβλέπων: Δημήτριος Ι. Σούντρης
Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 13η Ιουλίου 2022.

.....	.....	.....
Σούντρης Δημήτριος Καθηγητής	Θεοδωρίδης Γεώργιος Αναπληρωτής Καθηγητής	Ξύδης Σωτήριος Επίκουρος Καθηγητής

Αθήνα, Ιούλιος 2022.

.....

Αναγνωστόπουλος Χ. Κωνσταντίνος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Τα τελευταία χρόνια, οι εφαρμογές της μηχανικής μάθησης γίνονται ολοένα και πιο δημοφιλείς. Εκμεταλλευόμενοι την υπολογιστική ισχύ που τα σύγχρονα chip μας παρέχουν καθώς και τον μεγάλο όγκο δεδομένων που παράγεται καθημερινά από ανθρώπους και μηχανές, διάφορα μοντέλα μηχανικής μάθησης μπορούν να εκπαιδευτούν για να επιλύσουν ένα ευρύ φάσμα προβλημάτων, όπου κλασικές προγραμματιστικές προσεγγίσεις αδυνατούν. Η εκπαίδευση αυτών των μοντέλων βασίζεται σε τρία βασικά μοντέλα μάθησης: την επιβλεπόμενη μάθηση (supervised learning), την μη επιβλεπόμενη μάθηση (unsupervised learning) και την ενισχυτική μάθηση (reinforcement learning). Συνοπτικά, η επιβλεπόμενη μάθηση βασίζεται σε «χρυσά» ή επισημασμένα δεδομένα, για την επίβλεψη της διαδικασίας της εκπαίδευσης ενός μοντέλου. Η μη επιβλεπόμενη μάθηση βασίζεται σε τεχνικές όπου ανακαλύπτουν κρυμμένα μοτίβα σε ένα σύνολο μη επισημασμένων δεδομένων. Από την άλλη μεριά, η ενισχυτική μάθηση αποτελεί μία τελείως διαφορετική προσέγγιση και βασίζεται στη μεγιστοποίηση κάποιου είδους αμοιβής, όπου κάποιος πράκτορας λαμβάνει όταν αλληλεπιδρά με ένα συγκεκριμένο περιβάλλον. Ένα κλασσικό παράδειγμα ενισχυτικής μάθησης στον πραγματικό κόσμο αποτελεί αυτό του εκπαιδευτή σκύλων, ο οποίος προσπαθεί να εκπαιδεύσει ένα σκύλο να κάνει κάποιο συγκεκριμένο κόλπο. Όταν ο σκύλος συμπεριφέρεται προς τη σωστή κατεύθυνση, ο εκπαιδευτής τον επιβραβεύει με ένα μπισκότο. Ο σκύλος (πράκτορας), προσπαθεί να μεγιστοποιήσει τη συνολική του αμοιβή (δηλαδή τον αριθμό των μπισκότων που θα φάει), προσαρμόζοντας τη συμπεριφορά του κατάλληλα προς τη σωστή κατεύθυνση.
Επιπλέον, τα τελευταία χρόνια, τα νευρωνικά δίκτυα, ένα μοντέλο μηχανικής μάθησης που μιμείται τον τρόπο με τον οποίο λειτουργούν οι βιολογικοί νευρώνες, εμφανίζονται στο προσκήνιο της μηχανικής μάθησης εξαιτίας των υψηλών επιδόσεων τους σε διάφορα προβλήματα, όπως η επεξεργασία φυσικής γλώσσας, η αναγνώριση εικόνας, η δημιουργία τεχνιτών εικόνων και κειμένων κ.α. Η κλιμάκωση των νευρωνικών δικτύων (δηλαδή η προσθήκη περισσότερων νευρώνων και άρα περισσότερων παραμέτρων), συνήθως οδηγεί σε καλύτερα αποτελέσματα. Για παράδειγμα, σε εφαρμογές επεξεργασίας φυσικής γλώσσας, μοντέρνα νευρωνικά δίκτυα διαθέτουν δισεκατομμύρια παραμέτρους. Ξανά, η πρόοδος μας στον τομέα της μικροηλεκτρονικής μας επιτρέπει να χρησιμοποιούμε μοντέλα τέτοιας τάξης μεγέθους εξαιτίας της αυξημένης υπολογιστικής ισχύος που διαθέτουμε.
Στο πρώτο μέρος της παρούσας εργασίας, θα μελετήσουμε πώς η ενισχυτική μάθηση και τα νευρωνικά δίκτυα μπορούν να συνδυαστούν για την εκπαίδευση ευφυών πρακτόρων ικανών να αλληλεπιδρούν με πολύπλοκα περιβάλλοντα. Θα υλοποιήσουμε τέσσερις διαφορετικούς αλγορίθμους βαθιάς ενισχυτικής μάθησης, τους Deep Q Network, REINFORCE, Asyncrhonous Actor Critic & Proximal Policy Optimization, και θα χρησιμοποιήσουμε αυτές τις υλοποιήσεις για την εκπαίδευση ευφυών πρακτόρων στα περιβάλλοντα CartPole και DuckieTown. Ενώ το CartPole θεωρείται ως το εισαγωγικό περιβάλλον για την ενισχυτική μάθηση, το περιβάλλον DuckieTown είναι πιο πολύπλοκο και απαιτείται από τον πράκτορα να μάθει να πλοηγείται στους δρόμους μίας προσομοιωμένης πόλης.
Στο δεύτερο μέρος, θα επικεντρωθούμε στην ανάπτυξη του εκπαιδευμένου πράκτορα, στο περιβάλλον DuckieTown, στον πραγματικό κόσμο. Συγκεκριμένα, θα επικεντρωθούμε στα εξής δύο σενάρια: Το πρώτο σενάριο αφορά την επιτάχυνση της εμπρόσθιας διάδοσης (forward propagation) του νευρωνικού δικτύου για να επιτύχουμε μικρότερους χρόνους απόκρισης και ως εκ τούτου να κατασκευάσουμε έναν πιο «αποκριτικό» πράκτορα, χαρακτηριστικό που είναι επιθυμητό για όλα τα αυτόνομα οχήματα. Το δεύτερο σενάριο αφορά τον έλεγχο ενός "σμήνους' πρακτόρων από μία κεντρική συσκευή. Αυτή τη φορά, η κεντρική συσκευή πραγματοποιεί υπολογισμούς κατά παρτίδες (batch computations), με το μέγεθος της κάθε παρτίδας (batch size) να είναι ίσο με το πλήθος των πρακτόρων στο σμήνος. Γι΄ αυτό το σενάριο, θέτουμε ένα κατώτατο όριο των 100 FPS που πρέπει να επιτευχθεί για κάθε πράκτορα. Οι συσκευές που θα χρησιμοποιηθούν για την επιτάχυνση είναι η NVIDIA Xavier NX, όπου διαθέτει GPU ως επιταχυντή και η Xilinx Zynq UltraScale+ MPSoC ZCU10, που διαθέτει FPGA.
Όλη η σχετική δουλειά για την παρούσα εργασία μπορεί να βρεθεί στο Github repository εδώ.

Λέξεις κλειδιά : μηχανική μάθηση, νευρωνικά δίκτυα, βαθιά μάθηση, ενισχυτική μάθηση, βαθιά ενισχυτική μάθηση, επιτάχυνση υλικού

Abstract

In recent years, machine learning applications are becoming increasingly more popular. By leveraging the computing power that modern chips provide us with and large amounts of data that are produced daily by people or machines, machine learning models can be trained to solve a vast spectrum of problems that classical programming approaches cannot. Training machine learning models can be based on three different learning paradigms: supervised learning, unsupervised learning and reinforcement learning. In summary, supervised learning paradigm needs "golden" or "labeled" data in order to supervise the process of training a model and unsupervised learning lies on techniques that discover hidden patterns on a set of unlabeled data. Reinforcement learning on the other hand is a totally different approach and relies on the maximization of the reward an agent receives when he interacts with a specific environment. A common real-world reinforcement learning example is that of a dog trainer trying train a dog to perform certain tricks. When the dog acts towards the proper direction, the trainer rewards the dog with a biscuit. The dog (agent) tries to maximize its cumulative reward i.e. eat as many biscuits as possible, by adapting its behavior and act towards the correct direction.

Furthermore, in recent years, neural networks, a machine learning model that mimics the way biological neural networks work, have come to the fore due to state of the art results they achieve in various problems like natural language processing tasks, image classification, image or text generation etc. Scaling neural networks (adding more neurons and thus more parameters), usually leads to better results. For natural language processing tasks for example, current state of the art models contain billions of trainable parameters. Again, our advance in microelectronics enables us to use such models due to the increased computing power that we can utilize.

In the first part of the present thesis, we will study how reinforcement learning paradigm and neural networks can be combined in order to train intelligent agents, able to interact with complex environments. We will implement four different deep reinforcement learning algorithms, Deep Q Network, REINFORCE, Asynchronous Actor Critic & Proximal Policy Optimization and we will leverage those implementations to train intelligent agents able to interact with two environments, Cart Pole and DuckieTown. While CartPole environment is considered as the "hello world" environment for reinforcement learning, DuckieTown is a more complex one with an agent trying to learn how to properly drive through the roads of a simulated, animated city.

In the second part of the present thesis, we will focus one how we can deploy the trained DuckieTown agent in the real world. More specifically, we will focus on two scenarios. The first scenario regards the acceleration of the forward pass of the neural network model in order to achieve faster inference time and therefore create a more responsive agent, a trait that is desired for all autonomous vehicles. The second scenario regards the control of a swarm of agents by a central device. The central device this time performs batch computations, with the batch size being equal to the total number of agents in the swarm. For this scenario we set a minimum of 100 FPS that must be achieved for each agent. The devices that will be used for the acceleration are NVIDIA Xavier NX utilizing a GPU as a hardware accelerator and Xilinx Zynq UltraScale+ MPSoC ZCU104 utilizing an FPGA.

All related work regarding the entire thesis can be found in the Github repository here.

Keywords : machine learning, neural networks, deep learning, reinforcement learning, deep reinforcement learning, hardware acceleration

Ευχαριστίες

Ξεκινώντας, θα ήθελα να ευχαριστήσω θερμά τον διδακτορικό ερευνητή Δημήτριο Δανόπουλο για τη συνέχη του βοήθεια και υποστήριξη καθώς και για τον ενδιαφέρον που έδειξε καθ΄ όλη τη διαδικασία της υλοποίησης της παρούσας εργασίας. Επιπλέον, θα ήθελα να ευχαριστήσω τον καθηγητή μου Δημήτριο Σούντρη, για την αποδοχή της πρότασης μου για το παρόν θέμα της διπλωματικής εργασίας, επιτρέποντας μου με αυτό τον τρόπο να ερευνήσω πτυχές της μηχανικής μάθησης που κέντρισαν το ενδιαφέρον μου.
Τέλος, καθώς η παρούσα διπλωματική εργασία ολοκληρώθηκε παράλληλα με την εργασιακή μου απασχόληση, δεν θα μπορούσα να παραλείψω από τις ευχαριστίες τους συναδέλφους μου, Στέφανο και Σταύρο, για τις τεχνικές γνώσεις που μου μετέφεραν και τις γενικότερες συμβουλές τους, οι οποίες αναμφίβολα οδήγησαν σε μία πιο ολοκληρωμένη, τεχνικά, εργασία.

Contents

List of Figures

List of Algorithms

Chapter 1

Theoretical Background

1.1 Machine Leaning

1.1.1 Definition

Machine learning is the study of computer algorithms that appear to improve through experience by using data provided to them. As Arthur Samuel stated, an American pioneer in the field of artificial intelligence, "machine learning is the field of study that gives computers the ability to learn without being explicitly programmed".

A more formal definition regarding the learning ability of computers comes from computer scientist Tom Mitchel quoting, "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E."

Machine learning techniques and their ability to "learn from experience" shines in tasks where no programmer can write down an explicit solution to the problem due to their high complexity or their continuous change over time.

1.1.2 Machine Learning Paradigms

Learning paradigms state the pattern on which the learner manages to learn. There are three main paradigms on which different machine learning algorithms manage to learn. These paradigms are supervised, unsupervised and reinforcement learning and are further analyzed bellow:

- Supervised Learning : In supervised learning, the goal of the learner is to learn the function that maps input data to their corresponding labels. In order to do that, the learner needs a supervisor that will label training data with peoper labels. After the learning process, the learner can generalize its knowledge into new, unlabeled data and manage to classify them into their true labels.

- Unsupervised Learning : In unsupervised learning, there are no labeled data nor a supervisor. Instead, the learner manages to discover hidden structures and patterns in the data through observation. A common example is clustering algorithms that manage to group training examples into clusters that share similar features.

- Reinforcement Learning : Reinforcement learning, which is the main subject of the present thesis, involves machines and software agents that try to determine the best behavior within an environment in order to maximize their cumulative rewards. A real life example of reinforcement learning is that of a dog trainer and its dog. The dog (agent) tries to discover the optimal behavior so that the trainer will provide it with more biscuits (rewards). In the next chapters there will be a detailed explanation of the mathematical background and the algorithms involved with this learning paradigm.

1.1.3 Instance-Based vs Model-Based Machine Learning

A very helpful way to categorize machine learning algorithms is by the way the manage to generalize when supplied with new data. There are two main categories of algorithms that are described bellow:

- Instance-based machine learning : Algorithms that belong to this category manage to generalize when fed with new data by making comparisons between the new data and the training data that they were provided with during the training procedure. In order to achieve this, training data must not be discarded so that comparison can be performed. An example of such algorithms is kNN algorithm that decides about the label of an input by finding the label of the majority of the top k closest neighbors of the input data point.
- Model-based machine learning : Algorithms that belong to this category contain parameters whose values are learned during the training process using the training data that were provided. After the parameter's values have been determined, the training data can be discarded. An example of such algorithm is logistic regression where the parameters describing a hyperplane must be determined. After training, new data are labeled by observing their positions relative to the hyperplane. Data points belonging to one side are associated with one of the learned labels and data points belonging to the other side are associated with the other learned label.

The present thesis is heavily focused on neural networks which are models with, usually, enormous number of parameters that are determined in the training process. Hence, model-based machine learning techniques is what we will focus on.

1.2 Reinforcement Learning

Reinforcement Learning is an area of machine learning which studies how intelligent agents should take actions in an environment in order to maximize their a cumulative reward. As already mentioned, reinforcement learning is one of the three basic machine learning paradigms, with the other two being supervised learning and unsupervised learning. Its main characteristic is that it does need labeled input and output pairs and sub-optimal actions to be explicitly corrected. In this section, the necessary mathematical concepts behind reinforcement will be presented and the foundation for the better understanding deep reinforcement learning algorithms will be laid.

1.2.1 Learning Scenario

As mentioned, unlike supervised learning, the learner does not receive labeled data but instead, it receives information by interacting with an environment. After the learner or agent performs an action, it receives a real-valued reward and its current state in the environment. The reward it receives is related with its task and its corresponding goal. The described scenario is illustrated in figure 1.16.

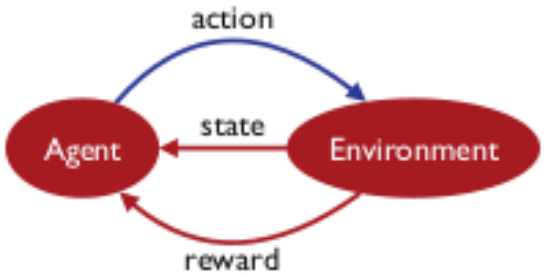

Figure 1.1: Reinforcement learning scenario [1].

The goal of the agent is to discover the best course of actions that maximize the rewards received from the environment. Of course, the agent receives only the immediate reward related to the action just taken and

there is no future reward feedback from the environment. Therefore, during the training, the agent faces the dilemma between exploring new states and actions that will provide new information about the environment and the rewards, and exploiting the already gathered knowledge to optimize the reward. This dilemma is known as the *exploration vs exploitation trade-off*.

Examples of such a learning scenario can be a computer bot trying to navigate inside a virtual world while aiming to achieve a certain goal or a self-driving car trying to avoid obstacles in a road while aiming to reach a certain destination. In both cases, the agent interacts with the environment via the decided actions and simultaneously it receives feedback through rewards. Actions that lead to unwanted states (e.g. crashing with an obstacle) yield less or negative rewards. The goal of the agent is to learn to make the correct decisions in order to maximize the cumulative reward and therefore reach its goal. In figure 1.2, an example of such a learning scenario is illustrated.

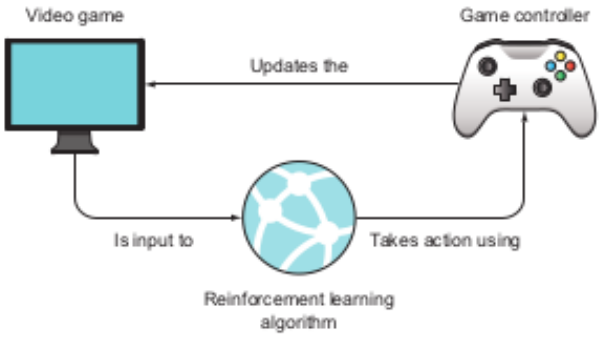

Figure 1.2: Reinforcement learning scenario example [2].

1.2.2 Markov Decision Process Model

In order to describe the learning problem, the environment and the interactions with it, the model of *Markov Decision Process* is adopted in reinforcement learning. A Markov decision problem is defined by: [1]

- A set of states S, possibly infinite.
- A start state $s_0 \in S$.
- A transition probability $\mathbb{P}[s'|s, a]$.
- A reward probability $\mathbb{P}[r'|s, a]$.

The model is called Markovian because it assumes that the Markovian property is valid. This Markovian assumption states that the probabilities of the actions and the rewards depend only on the current state s and not on the entire history of states and actions taken. Of course, there are many examples of real world problems that this hypothesis is clearly not valid. For example considering the frame of a camera input of a self-driving car as state, the action the car should make clearly does not depend from the frame itself as crucial information regarding the movement of other objects are lost. To avoid such problems, a common tactic is to turn the problem into an MDP. For the above-mentioned example, by considering two consecutive frames as the state, the lost information is finally included and the Markovian hypothesis seems more reasonable to be assumed [2].

Regarding the characteristics of the MDP, it can be discrete, meaning that the decisions are taken at a set of decision epochs $0, 1, .., T$ or continuous, meaning that the decisions are taken in arbitrary points in time. When T is finite, the MDP is said to have a *finite horizon*. Independently of T, the MDP is said to be *finite* when both S and A are finite sets. Generally, the reward $r(s, a)$ at state s when taking action a is a random variable, but in many cases, the reward is a deterministic function of the state-action pair.

Figure 1.3 illustrates the Markov Decision Process Model, where at time $t \in 0, 1, ..., T$ the agent performs an action $a_t \in A$ after observing the state s_t. The state reached is s_{t+1}, with a probability of $P[s_{t+1}|s_t, a_t]$ and the received reward is $r_{t+1} \in \mathbb{R}$, with a probability of $P[r_{t+1}|s_t, a_t]$.

Figure 1.3: MDP illustration [1].

1.2.3 Policy, State-Value & Action-Value Functions

The goal of the agent in an environment is to determine the appropriate action to take at each state. The strategy that the agent adopts is called *policy*. For example, in the game of hide and seek, the simplest strategy for the seeker is to move until a hider is spotted. A more formal definition of the policy function is the following:

Policy definition

A policy, is a mapping $\pi : S \to \Delta(A)$ where $\Delta(A)$ is the set of probability distributions over A. A policy π is deterministic if for any s there exists a unique $a \in A$ such that $\pi(s, a) = 1$.

This definition is that of a *stationary policy*, since the distribution of actions does not depend on time. A more general definition for the policy function is that of the *non-stationary policy* which is a sequence of mappings $\pi_t : S \to \Delta(A)$.

The agent's objective is to find a policy that maximizes the expected *return*. The return it receives following a deterministic policy is the following:

- For finite horizon (finite T): $\sum_{t=0}^{T} r(s_t, \pi(s_t))$.
- For infinite horizon: $\sum_{t=0}^{\infty} \gamma^t r(s_t, \pi(s_t))$, where $\gamma \in [0, 1)$ is the discount factor used to discount future rewards.

In other words, the return is a scalar used to summarize a possibly infinite sequence of immediate rewards. In the discounted case, early rewards are considered more valuable than later ones. This leads to the following definition:

Policy value definition

The value $V_\pi(s)$ of a policy π at a state $s \in S$ is defined as the expected reward returned when starting at s and following policy π:

- Finite horizon : $V_\pi(s) = \mathbb{E}_{a_t \sim \pi(s_t)} \left[\sum_{t=0}^{T} r(s_t, a_t) | s_0 = s \right]$.
- Infinite discounted horizon : $V_\pi(s) = \mathbb{E}_{a_t \sim \pi(s_t)} \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) | s_0 = s \right]$.

where the expectations are over random actions a_t according to the distribution $\pi(s_t)$ that provide reward values $r(s_t, a_t)$. The function V_π is called *state-value function for policy* π.

Action-value function

Another definition which will be proved very useful in the following chapters is that of the *action-value function*.

The action-value function Q associated to a policy π is defined for all $(s, a) \in S \times A$ as the expected return for taking action $a \in A$ at state $s \in S$ and then following policy π :

$$Q_\pi(s, a) = \mathbb{E}\left[r(s, a)\right] + \mathbb{E}_{a_t \sim \pi(s_t)} \left[\sum_{t=1}^{\infty} \gamma^t r(s_t, a_t) | s_0 = s, a_o = a \right] = \mathbb{E}\left[r(s, a) + \gamma V_\pi(s_1) | s_0 = s, a_0 = a\right]$$

It must be noted that $V_\pi(s) = \mathbb{E}_{a_t \sim \pi(s_t)} [Q_\pi(s, a)]$

All these definitions for the policy, state-value and action-value functions are critical for further understanding reinforcement learning and its algorithms.

1.2.4 Optimal Policies & Policy Evaluation

Optimal policy

As mentioned above, an agent seeks to maximize its reward while interacting in an environment. In other words, an agent starting from state $s \in S$ seeks a policy π with the largest value $V_\pi(s)$. It is proven that such a policy does exist for any starting state $s \in S$ [1]. That policy is called *optimal policy* and has the following definition:

A policy π^* is optimal if its value is maximal for every state $s \in S$, that is, for any policy π and any state $s \in S$, $V_{\pi^*}(s) \geq V_\pi(s)$.

Policy improvement theorem

The value function allows us to know how good a certain policy is, but it is essential to know whether we should change to a new policy or not. A way to answer this question is by considering selecting action a in state s and thereafter following the existing policy π. In case such choice provides greater rewards, one could argue that the new policy of selecting a every time s is encountered and then following policy π is a better policy overall. Indeed, this special case comes under a more general result called policy improvement theorem [8]. The theorem states that for any two policies π and π^*, the following holds :

$$(\forall s \in S, \mathbb{E}_{a \sim \pi^*} [Q_\pi(s, a)] \geq \mathbb{E}_{a \sim \pi} [Q_\pi(s, a)]) \Rightarrow (\forall s \in S, V_{\pi^*}(s) \geq V_\pi(s))$$

A strict inequality for at least one state s in the left-hand side implies a strict inequality for at least one s in the right-hand side.

Bellman's optimality condition

Finally, Richard Bellman, whose work has been a major contribution to Reinforcement Learning paradigm, proved that a policy π is optimal if and only if for any pair of $(s, a) \in S \times A$ with $\pi(s)(a) > 0$ the following holds :

$$a \in \underset{a' \in A}{argmax} Q_\pi(s, a')$$

This condition is known as Bellman's optimality condition.

Bellman's equations

Bellman moved even further and noticed that the value of a policy in a state s can be expressed in terms of its values at other states forming a system of linear equations. These equations are known as Bellman's equations and state the following:

The values $V_\pi(s)$ of a policy π at states $s \in S$ for an infinite horizon MDP obey the following system of linear equations:

$$\forall s \in S, V_\pi(s) = \mathbb{E}_{a_1 \sim \pi(s)}[r(s, a_1)] + \gamma \sum_{s'} \mathbb{P}[s'|s, \pi(s)] V_\pi(s')$$

With all these definitions & theorems, the presentation and explanation of the algorithms that are used in the next chapters and paragraphs of this thesis will be better understood. For now, the basic theory behind Reinforcement Learning has been covered.

1.2.5 Classification of Reinforcement Learning Algorithms

Intelligent agents can be created using different algorithms. These reinforcement learning algorithms can be separated into different categories based on different criteria. In this paragraph the taxonomy of reinforcement learning algorithms will be presented [2, 8].

Model-free vs model-based algorithms

A common distinction between reinforcement learning algorithms is made by whether the algorithm is given a model of the environment. With the term "model" we mean anything that the agent can use in order to predict the environment's response to its actions. Such information can be reward or state transition probability distributions. If such information is known and consequently a model for the environment does exist, the algorithm that is given those information is a *model-based algorithm*. On the other hand, if such information is unknown and a model for the environment does not exist, the algorithm that operates on the environment is called model-free. Model-based approaches rely on *planning* using the model information provided to them where as model-free approaches rely on *learning* the unknown information.

Examples of planning algorithms are the *Value Iteration* algorithm and the *Policy Iteration algorithm* [3]. On the other hand, examples of learning algorithms are the *Q-learning* algorithm and the *SARSA* algorithm which will be further presented later.

On-policy vs off-policy algorithms

Another way to classify reinforcement learning algorithms is by how they change their behavior during training. Generally, reinforcement learning algorithms include two components, a *learning policy*, which determines the action to take during training, and an *update rule* which defines a new estimate of the optimal value function. For an *off-policy* algorithm, the update rule does not depend on the learning policy. More generally, an off-policy algorithm evaluates or improves one policy by acting based on another policy. On the other hand, an *on-policy* algorithm evaluates and improves the current policy used for control [1].

An example of off-policy algorithm is *Q-learning* where as an example of on-policy algorithm is *SARSA* algorithm.

In the present thesis, model-free, both on-policy and off-policy algorithms will concern us. In the following two paragraphs, SARSA & Q-Learning algorithms will be presented us they constitute the basis for their deep reinforcement learning counterpart.

1.2.6 Q-Learning Algorithm

Q-Learning is a temporal-difference, off-policy algorithm introduced by Watkins in 1989 [8]. The algorithm aims at learning the Q function associated with the optimal policy π^*, independent of the learning policy which is followed during the algorithm. The algorithm's core is the Bellman's equation which ultimately translates into the following update rule :

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{a}_{\text{learning rate}} \cdot \overbrace{\Bigg(\underbrace{\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Bigg)}^{\text{temporal difference}}$$

As seen in the above equation, the algorithm tries to approximate the optimal action-value function by updating towards the target value. The complete algorithm is described in algorithm section 1.

Algorithm 1 Q-Learning algorithm

Algorithm parameters : learning rate $a \in [0, 1]$, $\epsilon \in [0, 1]$, discount factor $\gamma \in [0, 1)$.

- For every $(s, a) \in S \times A$ initialize $Q(s, a)$.
- Loop for each episode:
  - Initialize s.
  - Loop for each step of the episode until s is terminal:
    - Choose action a for state s using learning policy (e.g. $\epsilon - greedy$).
    - Take action a, observe reward r and new state s'.
    - $Q(s, a) \leftarrow Q(s, a) + a \left[r + \gamma \underset{a}{max} Q(s', a) - Q(s, a) \right]$.
    - $s \leftarrow s'$.

It is proven, that by following the algorithm above, Q converges to Q^* with a probability of 1 [8]. Furthermore, Q-Learning is classified as an off-policy algorithm because the policy of choosing the action s ($\epsilon - greedy$) is different than the one used to evaluate the Q value of the next action in state s' (which is selected as the one which yields the maximum Q value on state s', $\underset{a}{max} Q(s', a)$). The $\epsilon - greedy$ strategy is a common policy to bring a balance to the exploration-vs-exploitation trade-off. In order to explore the action space and not rely only one those actions currently known to maximize the rewards, with a probability of ϵ a random action is chosen while with a probability of $1 - \epsilon$, the best action learned so far is selected. Of course, this does not apply for the calculation of the target Q value where the action selected for state s' is the one the maximizes the Q function at this state.

A toy example that the Q-Learning algorithm is applicable is illustrated in figure 1.4. Gridworld is a classic environment where an agent is trying to reach its goal following the shortest path and avoiding the obstacles. In order to enforce the algorithm to find the shortest path, a reward of -1 is given to the agent for each action that does not lead him to an obstacle, a reward of -20 if an obstacle is hit and a reward of +30 if the goal is reached. By following Q-Learning algorithm, the optimal action-value function Q^* is learned and the agent is able to navigate through the gridworld by choosing $\underset{a}{max} Q(s, a)$. for whatever state he is in.

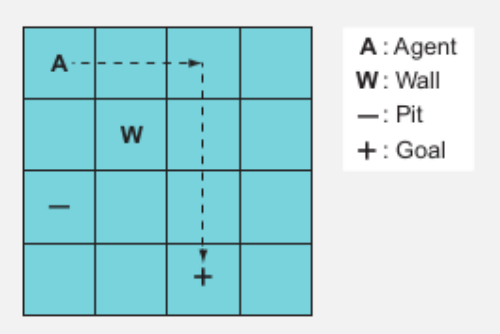

Figure 1.4: Gridworld game environment [2].

Q-Learning appears to be working fine for such an environment, but there are some disadvantages lying hidden. First, in case the initialization of the environment was random (i.e. the positions of the obstacles and the goal varied for different episodes), the algorithm would not be able to converge, as the action-values for different episodes would be different because a different optimal path would have to be followed. Another major disadvantage is the fact that for the algorithm to work, a discrete finite state space is assumed. A discrete finite state space allows us to initialize the Q-function in a table and then operate on it, trying to approximate $Q*$. In case of a continuous infinite state space, that would clearly be not possible. The "deep-learning counterpart" of Q-Learning comes to solve these problems, and this solution will be presented later [2].

1.2.7 SARSA Algorithm

SARSA algorithm is the on-policy equivalent of Q-Learning algorithm. Again, the algorithm is trying to learn the Q function associated with the optimal policy π^*. Its name reflects the way the algorithm operates and which follows the pattern $\boldsymbol{S}_1$ for the current state, $\boldsymbol{A}_1$ for the action taken, $\boldsymbol{R}$ for the reward the agent receives, $\boldsymbol{S}_2$ for the state the agents lands on and $\boldsymbol{A}_2$ for the next action it takes. There are minor differences in the update rule that are responsible for the on-policy nature of the algorithm. The update rule is presented in the following equation:

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{a}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{Q(s_{t+1}, a_{t+1})}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)}^{\text{temporal difference}}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{new value (temporal difference target)}}$$

The difference between SARSA and Q-Learning, lies in the update rule and the selection of the next action. When in state s, both algorithms select the action a based on $\epsilon - greedy$ strategy. Afterwards, Q-Learning updates by using the action which yields the greatest Q value when in next state s'. On the other hand, SARSA updates by selecting action a' in next state s' using $\epsilon - greedy$ strategy again. This action is then stored and used as the real action to take in the next state s'. In algorithm section 2, all the details of the SARSA algorithm are presented.

Algorithm 2 SARSA algorithm

Algorithm parameters : learning rate $a \in [0, 1]$, $\epsilon \in [0, 1]$, discount factor $\gamma \in [0, 1)$.

- For every $(s, a) \in S \times A$ initialize $Q(s, a)$.
- Loop for each episode:
  - Initialize s.
  - Choose action a for state s using learning policy (e.g. $\epsilon - greedy$).
  - Loop for each step of the episode until s is terminal:
    - Take action a, observe reward r and new state s'.
    - Take action a' from state s' using the same learning policy (e.g. $\epsilon - greedy$).
    - $Q(s, a) \leftarrow Q(s, a) + a\,[r + \gamma Q(s', a') - Q(s, a)]$.
    - $s \leftarrow s', a \leftarrow a'$.

Like Q-Learning, it is proven that SARSA converges to Q^* with a probability of 1 [8]. Unfortunately, SARSA suffers from the exact same disadvantages like Q-Learning presented in the previous paragraph.

1.3 Neural Networks

1.3.1 Inspiration

Artificial neural networks are computer systems inspired from biological neural networks which constitute the brains of animals. The realization that the brain performs computations in a totally different way compared to electronic computers pushed the study of artificial neural networks. With the main difference being the massive parallelism of information processing, brains have the capability of organizing their structure, meaning the neurons that they are composed of, in a way that it completes certain computations. These computations are related to processes like vision, hearing etc.

Like biological neural networks, artificial neural networks contain a number of interconnected neurons that are the building block of the network and the fundamental information processing unit. Furthermore, like biological neural networks have the ability to "alter" their structure, artificial neural networks can go through a learning process that carries out certain modifications on them with the ultimate goal of achieving a certain task.

Seeing neural networks as machines capable of adapting and altering their architecture, the following definition can be realized : "A neural network constitutes a large scale parallel and distributed processor that contains fundamental processing units called neurons capable of storing emperical knowledge render it available for future use. They resemble the functionality of the brain in the following two ways:

1. Knowledge is obtain from the environment through a learning process.

2. The strength of the connections between neurons, called synaptic weights, are used for storing the obtained knowledge [3]".

1.3.2 Structure & Functionality of a Neuron

As already mentioned above, the fundamental processing unit of a neural network is the neuron. The functionality of an artificial neuron can be compared to that of a biological neuron. In the biological model, each neuron receives input signals from dendrites and produces its output along its axon. That axon branches and is connected with other neurons. Regarding the artificial model, signals travel through axons of other neurons $(x_0, x_1, ...)$ and interact with the rest of the neurons $(w_0 \times x_0)$ through the synaptic weight (w_0) that characterizes each synapse. The values of the weights characterize the magnitude of the influence each input has to the neuron. After that, all the weighted inputs are summed and in the case the summation result is over a certain threshold the neuron is fired. This triggering is modeled through an activation function that determines the value of the output signal based on the inputs and the threshold. This analogy is illustrated in figure 1.5.

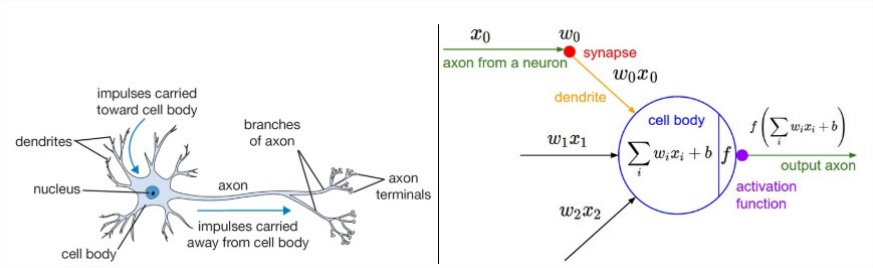

Figure 1.5: Comparison between biological and artificial neuron.

To recap, an artificial neuron consists of the three following elements:

1. A set of synaptic weights, one for each input of the neuron, that model the magnitude of influence its input has to the state of the neuron.

2. An adder, that sums all the weighted inputs.

3. An activation function that determines the value of the neuron's output by comparing the result of the adder to a certain threshold.

Based on the elements above, for a neuron k with m inputs, the output y_k is calculated with the following formula:

$$y_k = f\left(\sum_{i=1}^{m} w_{ki} x_i + b_k\right)$$

where x_i is each input of the neuron, w_{ki} the synaptic weight, b_k the threshold and f the activation function.

Some recently used activation functions are the following:

- Rectifier Linear Unit (ReLU) : This is a standard choice due to its simplicity and its non-linear nature which is highly needed in multi-layer neural networks.

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

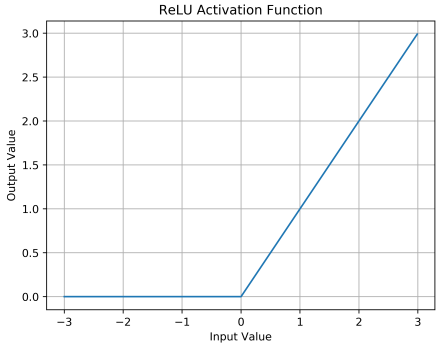

Figure 1.6: ReLU activation function.

- Sigmoid Function : This activation function is used when the output of the neuron must be interpreted as a probability as it squashes its input between 0 and 1.

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

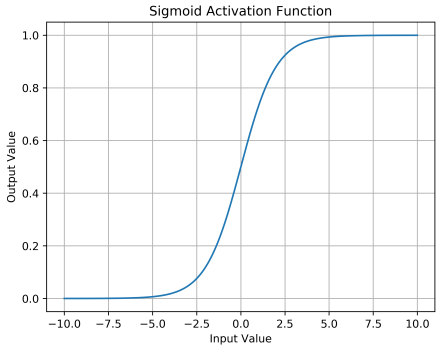

Figure 1.7: Sigmoid activation function.

- Tanh Function : This activation function has the ability of squashing its input between -1 and 1.

$$Tanh(x) = \frac{e^{-2x} - 1}{e^{-2x} + 1}$$

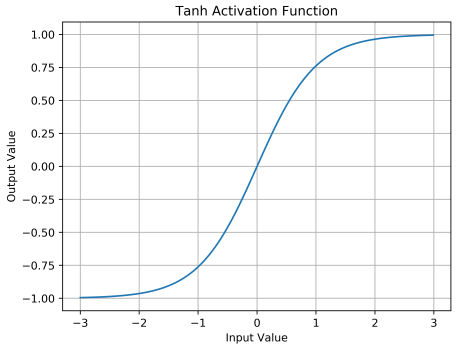

Figure 1.8: Tanh activation function.

- Softmax Function : This function is a generalization of the sigmoid function to multiple dimensions. It operates on an entire set of neuron outputs (belonging to the same neural layer) and ensures that the sum of the individual outputs will sum to one. It is widely used when the entire output of a neural layer must be interpreted as a probability distribution.

$$\sigma(\boldsymbol{z})_k = \frac{e^{z_k}}{\sum_{i=1}^{n} e^{z_i}}$$

where $\boldsymbol{z} = (z_1, z_2, ..., z_n)$ the plain output of the neurons adders in the layer.

The functionality of a single neuron can have a mathematical interpretation that it will be proven useful for understanding the need of multilayer networks containing more than one neurons.

In order to make things simple, let us assume an input vector $\boldsymbol{X} = [x_1, x_2]$ and a neuron with weights $\boldsymbol{W} = [w_{k1}, w_{k2}]$ and threshold b_k. The result of the neurons adder is :

$$v_k = x_1 * w_{k1} + x_2 * w_{k2} + b_k$$

It is clear, that this formula describes a line in the two dimensional space where the input data lives. Thus the neuron can be used as a classifier where data belonging to one side of the separation line are linked with one class and data belonging on the other side are inked to the other class. Of course, this example can be generalized in more than two dimension where the line would be a hyperplane.

This realization may give us an intuition of the way that a neuron works but signifies one great weakness. What if the data that have to be classified are not linearly separable? As a neuron is able to draw just hyperplanes, separating non-linear separable data is an impossible task. This realization is illustrated in figure 1.9.

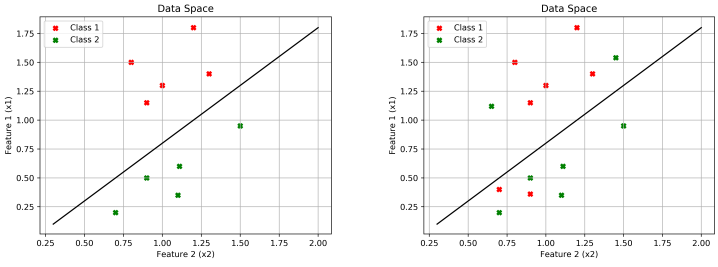

Figure 1.9: Linear & non-linear separable data.

This weakness of the neuron can be solved with the use of multiple neurons and it was the main motivation of developing multilayer neural networks that will be presented in later paragraphs.

1.3.3 Learning Procedure of a Neuron

As illustrated in figure 1.9, the operation of a neuron is equivalent to that of drawing a line in order to separate data belonging to two different classes. In order to draw the correct line, the values of weights w_{ki} and threshold b_k must be known. But how does a neuron know what weight values should be assigned to its synapses?

For the parameters of the neuron to have the correct values, a learning process must be designed. This learning process utilizes the gradient descent algorithm. For the explanation of the algorithm and how it is applied to the neuron, we will assume a supervised problem containing a set of labeled training data with each instance of the data being denoted as a vector $\boldsymbol{X}_i$. Gradient descend algorithm appears in three different flavors, stochastic gradient descend, mini-batch gradient descent and batch gradient descent. Each flavor of gradient descent, differs to the number of training instances it operates on. Stochastic gradient descent updates the neuron's weights after consuming exactly one training instance while batch gradient descent updates the weights after consuming all training data provided. Mini batch gradient descent is the intermediate case where batches of data with size grater than one but less than the entire training set are consumed. As stochastic and batch gradient descent can be considered as edge cases of mini-batch gradient descent, the learning process using the later will be described.

Along with the training data being denoted as a vector $\boldsymbol{X}_i$, let us assume that the output of the neuron described by a parameter vector $\boldsymbol{W}_n$ at the n^{th} step of the learning process, for a specific input is denoted as $y_k(\boldsymbol{X}_i|W_n)$. Furthermore, let $\hat{y_i}$ be the true label of the corresponding input vector. Then a loss function can be defined that quantifies the error between the output of the neuron and the true label of the corresponding training instance. Such loss functions can be the Mean Squared Error (MSE) loss function used for regression problems, that for a batch size of N instances is defined as:

$$L(\boldsymbol{W}_n) = \frac{\sum_{i=1}^{N} (y_k(\boldsymbol{X}_i|\boldsymbol{W}) - \hat{y_i})^2}{N}$$

or the Binary Cross-Entropy loss function used for pure classification tasks:

$$L(\boldsymbol{W}_n) = \sum_{i=1}^{N} y_k(\boldsymbol{X}_i|\boldsymbol{W}_n) log(\hat{y_i}) + (1 - y_k(\boldsymbol{X}_i|\boldsymbol{W}_n)) log(1 - \hat{y_i})$$

Minimization of the loss function is the goal of the learning process and thus, for each mini-batch of training instances, the following update is performed on neuron's parameters [9, 10]:

$$\boldsymbol{W}_{n+1} = \boldsymbol{W}_n - \gamma_n \boldsymbol{\nabla} L(\boldsymbol{W}_n)$$

where γ_n is a learning rate parameter possibly varying for different steps and $\boldsymbol{\nabla} L(\boldsymbol{W}_n) = [\frac{\partial L}{\partial b_k}, \frac{\partial L}{\partial w_1}, ...]$. Performing such updates, the parameters are changed so that the loss function converges to a minimum. The overall process is described in algorithm section 3.

Algorithm 3 Neuron mini-batch gradient descent

Algorithm parameters : number of epochs E, learning rate γ, batch size N.

- For each of the E epochs.
  - For each N sized batch of data in the training set containing data vectors $\boldsymbol{X}_i$.
    - Calculate neuron outputs $y_k(\boldsymbol{X}_i)$.
    - Calculate loss $L(\boldsymbol{W})$ for the whole batch.
    - Perform parameter update based on the following rule:

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \gamma \boldsymbol{\nabla} L(\boldsymbol{W})$$

1.3.4 Multilayer Neural Networks

As mentioned above, a single neuron in incapable of solving problems containing non-linearly separable data. To cope with this problem, networks of multiple neurons arranged in different layers are used. Such neural network is able to "draw" non-linear lines in problem such as figure's 1.9 data. In figure 1.10, an example of a neural network is illustrated.

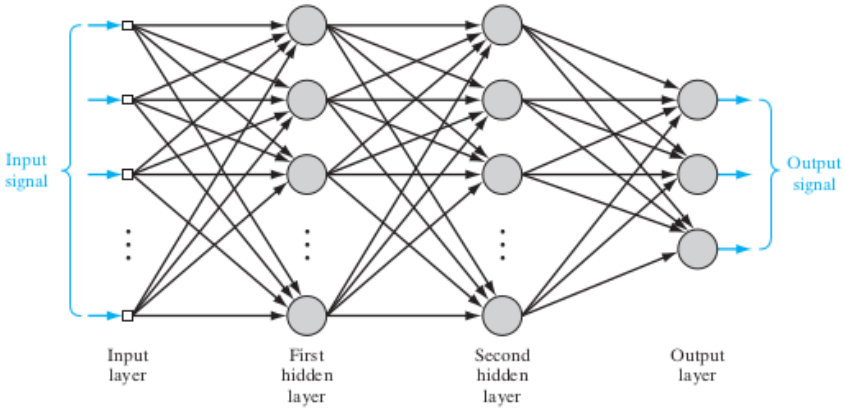

Figure 1.10: Multilayer neural network example [3].

Figure 1.10 illustrates the vanilla architecture of multilayer neural networks containing only simple layers of neurons. Other types of layer do exist and they will be presented in the next paragraph. The vanilla architecture possesses the three following characteristics [3]:

- Each neuron possesses a non-linear activation function.
- The network has one or more layer of neurons hidden after the input nodes and the output layer.
- The network exhibits a high degree of connectivity.

It is essential to note the critical role of the non-linear functions that neurons posses. In case each neuron did non have non-linear activation functions but output the raw result of their adders, multiple layers would have no reason of existence. As it will be pointed out in paragraph 1.3.6, each neural layer can be described through a matrix-vector multiplication and thus, multiple layers with no intermediate non-linear functions collapse into a two-layer input output model.

Regarding the training process of a neural network, this time the procedure is more complex compared to the learning procedure of a single neuron. Again, the gradient descent algorithm will be used, and the three different flavors, stochastic, mini-batch & batch are again possible. The problem lies in the fact that there is no obvious target value the hidden layer's neurons should have whereas the output layer's neurons do have a desired value, which is denoted as $\hat{y_k}$. To quantify the correction that must be done to hidden layer's neuron's parameters, the backpropagation algorithm is used.

Backpropagation algorithm utilizes the well known chain rule to calculate the gradient of the loss function for the different hidden layers. The calculation of the loss function back-propagates from the output layer towards the input and thus the name of the algorithm. After the gradients are computed, the parameters are changed using gradient descent.

For presenting the bacpropagation algorithm, let us assume again a supervised learning problem. We denote as y_j the output of the j^{th} neuron of the output layer and as d_j its desired output. The error e_j of the j^{th} neuron is given by the subtraction of the two terms $e_j = d_j - y_j$. Again, a loss function must defined, and for simplicity we use the MSE loss function and this, we get $L_j = \frac{1}{2}e_j^2$. For the entire output layer, we get:

$$L = \sum_{j \in C} L_j = \frac{1}{2} \sum_{j \in C} e_j^2$$

For training on a mini-batch, we average the loss function on the total number of instances in the batch. For simplicity, we will continue assuming training with exactly one instance.

As already stated, the the output of the adder of a neuron with m inputs is:

$$v_j = \sum_{i=1}^{m} w_{ji} x_i + b_j$$

and the final output is given after the application of the activation function :

$$y_j = \phi_j(v_j)$$

Of course, L is a function of the parameters of the output layer of the neural network, and thus, the correction $\Delta w_{i,j}$ that must me applied to them in order to minimize the loss function can be calculated and it is proportional to the partial derivative $\frac{\partial L}{\partial w_{ij}}$. Using the chain rule, we get the following formula:

$$\frac{\partial L}{\partial w_{ji}} = \frac{\partial L}{\partial e_j} \frac{\partial e_j}{\partial y_j} \frac{\partial y_j}{\partial v_j} \frac{\partial v_j}{\partial w_{ji}}$$

For the individual terms, after calculating the partial derivatives, we get:

$$\frac{\partial L}{\partial e_j} = e_j, \frac{\partial e_j}{\partial y_j} = -1, \frac{\partial y_j}{\partial v_j} = \phi_j'(v_j), \frac{\partial v_j}{\partial w_{ji}} = x_i$$

That said, the final correction that must be performed to the output layer's neurons' parameters is:

$$\Delta w_{ji} = -\gamma \cdot \frac{\partial L}{\partial w_{ji}} = \gamma \cdot \delta_j x_i$$

where δ_j is defined as the local gradient and it equals :

$$\delta_j = -\frac{\partial L}{\partial v_j} = -\frac{\partial L}{\partial e_j} \frac{\partial e_j}{\partial y_j} \frac{\partial y_j}{\partial v_j} = e_j \phi_j'(v_j)$$

Now, let us assume the j_{th} neuron of the first hidden layer exactly behind the output layer. Furthermore, we use index k for the neurons of the output layer. The local gradient δ_j, can be written as:

$$\delta_j = -\frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial v_j} = \ -\frac{\partial L}{\partial y_j} \varphi_j'(v_j)$$

For the term $\frac{\partial L}{\partial y_j}$ we have:

$$\frac{\partial L}{\partial y_j} = \sum_k e_k \frac{\partial e_k}{\partial y_j} = \sum_k e_k \frac{\partial e_k}{\partial v_k} \frac{\partial v_k}{\partial y_j}$$

, because $e_k = d_k - y_k = d_k - \varphi_k(v_k) \Rightarrow \frac{\partial e_k}{\partial v_k} = \varphi_k'(v_k)$

Furthermore, $v_k = \sum_{j=0}^{m} w_{kj} y_j(n) \Rightarrow \frac{\partial v_k}{\partial y_j} = w_{kj}$

Thus, the following equation is valid:

$$\frac{\partial L}{\partial y_j} = -\sum_k e_k \varphi_k'(v_k) w_{kj} = -\sum_k \delta_k w_{kj}$$

Therefore, the local gradient of a hidden neuron before the output layer is given by the following equation:

$$\delta_j = \varphi_j'(v_j) \sum_k \delta_k w_{kj}$$

What has been achieved is a relation between the local gradients of one neural layer with the exactly next one. In algorithm section 4, all steps of the back propagation are described:

Algorithm 4 Back propagation

- For each training instance (or mini-batch), perform a forward pass and calculate the loss function L.
  - For each layer in the neural network starting from the output layer and moving backwards:
    - If neuron is in output layer :
    $$\delta_j = e_j \phi'_j(v_j)$$
    - If neuron is in hidden layer :
    $$\delta_j = \phi'_j(v_j) \sum_k \delta_k w_{kj}$$
    ,where δ_k is the local gradients of the layer's neurons right in front.
    - Calculate parameter correction :
    $$\Delta w_{ji} = \gamma \cdot \delta_j \cdot x_i$$
    - Perform gradient descent update based on Δw_{ji}.

Having calculated the parameter corrections, gradient descent can now be applied to minimize the loss function.

1.3.5 Types of Layers

As already stated, the architecture of figure 1.10 is the vanilla architecture containing simple fully connected neural layers. More types of neural layers do exist, each one of them excelling in different kinds of tasks. In this paragraph, some of the mainly used neural layers will be presented.

Convolutional layers

Convolutional layers have been very successful in computer vision problems. They are capable of exploiting spatial relations between features of image data. As the same kernel is applied to the whole image, the number of parameters they contain is heavily reduced compared to an approach of using one neuron for each pixel of the input image. In other words, they systematize the idea of spatial invariance, exploiting it to learn useful representations with fewer parameters [4]. Their operation is illustrated in figure 1.11.

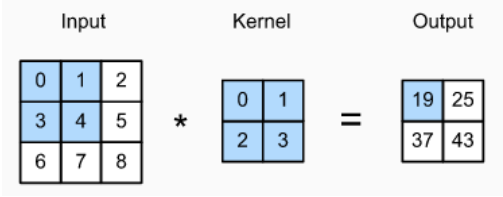

Figure 1.11: Operation of a convolutional layer [4].

As presented, the kernel containing trainable parameters is applied along the surface of the image with a selected step size and yields the results. It is essential to note that convolutional layers commonly contain more than one channels (more than one kernels), that simultaneously operate on the input data surface and learning different useful representations.

Recurrent layers

Recurrent layers are another type of neural layers which have been very successful in time series prediction and natural language processing tasks. Recurrent neurons' output is fed back into their input and thus, their calculations are performed on new data and their previous outputs as well forming a type of memory. In figure 1.12, the general concept of a recurrent layer is presented where we can clearly see that each hidden state $\boldsymbol{H}_t$ is a function of the input for the given time step $\boldsymbol{X}_i$ and the previous hidden state $\boldsymbol{H}_{t-1}$.

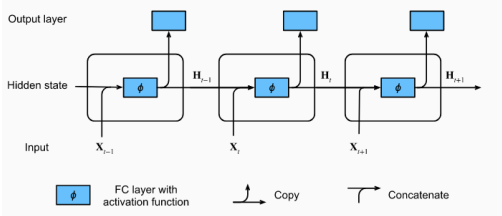

Figure 1.12: Operation of a recurrent neuron unfolded over time dimension [4].

As simple recurrent layers suffer from exploding or vanishing gradients during their training process after back-propagating through time, more advanced layers have been created like Long-Short-Term-Memory networks (LSTMs) and Gated Recurrent Units (GRUs) that perform some more complex calculations but the main idea of the output feedback into the input remains the same. An illustration that quickly presents the differences between simple RNNs and GRUs with LSTMs is presented in figure 1.13.

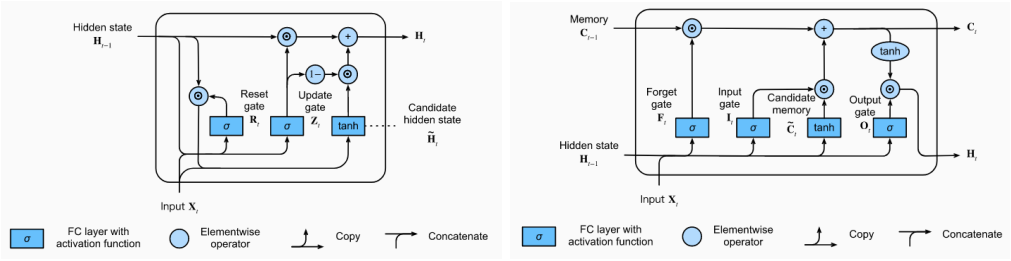

Figure 1.13: Operation of LSTMs (right) and GRUs (left)[4].

Attention Layers

Natural language processing was revolutionized in 2017 after "Attention is All you Need" was published [11]. This paper introduced a new type of layer, the attention layer, that implements the idea of using Queries and Keys to boost specific Values that contain more useful information to be passed in the next layer. Transformer models (like the well known BERT & GPT-3) make full use of those kind of layers to achieve state of the art results in various language tasks. In figure 1.14 a high level diagram of the attention mechanism is presented.

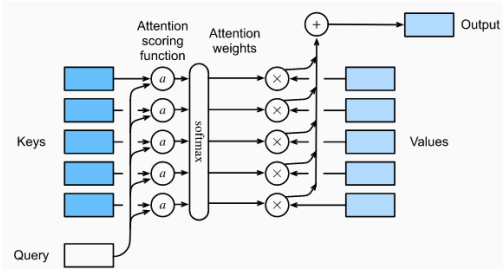

Figure 1.14: Attention mechanism[4].

Queries, keys and values are extracted after the input is processed by linear layers with parameters $\boldsymbol{W}_q, \boldsymbol{W}_k, \boldsymbol{W}_v$ respectively.

More layers that have impact on the operation of the neural network (pooling layers [12]), that make the training process more stable (batch-normalization layers [13]) or that help reduce over-fitting phenomena (dropout layers [14]) do exist but this paragraph cannot cover all of them.

1.3.6 Neural Networks as Matrix Operations

In paragraph 1.3.2, it is stated that the output of the j^{th} neuron of a layer given its parameters $w_{j1}, ..., w_{jn}, b_j$ and its inputs $x_1, ..., x_n$ is given by the following formula:

$$y_j = f\left(\sum_{i=1}^{n} w_{ji} \cdot x_i + b_j\right)$$

It is clearly seen that if we assume the weight vector of the neuron $W_j = [w_{j1}, ..., w_{jn}]$ and the input vector $X = [x_1, ..., x_n]$, the output can be obtained as the dot product of the weight vector and the input vector like:

$$y_j = f\left(XW_j^T + b_j\right)$$

To make things even simpler, one can assume the extended parameter vector of the neuron as $W_j = [w_{j1}, ..., w_{jn}, b_j]$ and the extended input vector as $X = [x_1, ..., x_n, 1]$, then the output is obtained only by the dot product of the vectors:

$$y_j = f\left(XW_j^T\right)$$

As long as the output of one neuron can be modeled as a matrix multiplication operation, then the output of an entire neural layer can be modeled the same way as well. Let us assume the weight matrix of the neural layer :

$$\boldsymbol{W} = \begin{bmatrix} w_{11} & ... & w_{1n} \\ ... & w_{ji} & ... \\ w_{m1} & ... & w_{mn} \end{bmatrix}$$

Each row of this matrix contains the weight parameters of each n-input neuron in the layer. Furthermore, neurons' thresholds are contained in a vector $B = [b_1, ..., b_m]$. The output of the layer is an m-element vector $Y = [y_1, ..., y_m]$ which is derived but the formula:

$$Y = f\left(X\boldsymbol{W}^T + B\right)$$

Of course, function f is applied to the resulted vector in an element-wise way. Furthermore, as stated in paragraph 1.3.4, in case of multilayer neural networks, it is necessary for f to be non-linear, otherwise the multilayer neural network would be equivalent to a collapsed one-layer version of it. To prove this statement, let us assume a two layer neural network, with $\boldsymbol{W}_1, \boldsymbol{W}_2$ and B_1, B_2 the parameters of the respective layers. If H is the output of the hidden layer and O the output of the output layer, then if f is linear:

$$H = X\boldsymbol{W}_1^T + B_1$$

$$O = H\boldsymbol{W}_2^T + B_2 = (X\boldsymbol{W}_1^T + B_1)\boldsymbol{W}_2^T + B_2 = X\boldsymbol{W}_1^T\boldsymbol{W}_2^T + B_1\boldsymbol{W}_2^T + B_2 = X\boldsymbol{W}^T + B$$

It is clear that without a non-linear activation function, the neural network collapses into a single layer with parameters $\boldsymbol{W}, B$. Of course the same logic applies for an arbitrary number of layers and not just two-layer neural network.

1.4 Deep Reinforcement Learning Algorithms

In section 1.2, the mathematical background of Reinforcement Learning paradigm was presented, along with two basic algorithms that aim to learn the optimal Q-function, Q^*. As stated in paragraphs 1.2.6 & 1.2.7, Q-Learning & SARSA algorithms have some serious disadvantages and limitations. Both algorithms cannot be applied when the state space is infinite (as they rely on initializing action-values for every state on a table) and in the example environment of paragraph 1.2.6 they cannot cope with a random initialization of the environment. Basically, tabular approaches do not ultimately learn how to properly behave but they memorize the best course of actions the agent should make for a given environment through trial and error. Thus, even minor changes in the environment can totally change the ability of the agent to navigate properly.

In section 1.3, artificial neural networks where presented. Neural networks have revolutionized the field of machine learning achieving state of the art results in numerous tasks. Neural networks can be combined with reinforcement learning paradigm and create the sub-field of *Deep Reinforcement Learning*. As neural network can process unstructured data and create meaningful representations out of them, they can deal with the disadvantages of Q-Learning and SARSA algorithms and achieve state of the art results.

In this section, the deep reinforcement learning counterpart of Q-Learning algorithm will be presented and more deep reinforcement learning algorithms utilizing neural networks. Specific applications along with the code implementation of each algorithm will be presented in the next chapter.

1.4.1 Deep Q Learning

The fundamental idea behind Deep Q-learning algorithm is the following : instead of making a tabular representation of the optimal Q function, an approach which is impossible when the state space is infinite, let's approximate the optimal Q-function with a neural network. It is well known that neural networks can act as universal function approximators and thus, Q^* can be approximated with a neural network [15, 4].

The original Q function, accepts to its input the state and the action to be performed and returns the value of that specific action. The action that should be performed is the one that yields the greatest value $\underset{a}{max} Q(s, a)$. Deep Q-Learning algorithm, approximates the Q function using a neural network, but this time, the neural network does net accept the state and the action to its input in order to output the value of the corresponding action, but accepts just the state. The output of the neural network has a dimension equal to the number of possible actions and the selected action is again the one with the greatest value. This the the "modified" Q-function Deepmind used to play Atari games and it is illustrated along with the the original Q-function in figure 1.15 [16].

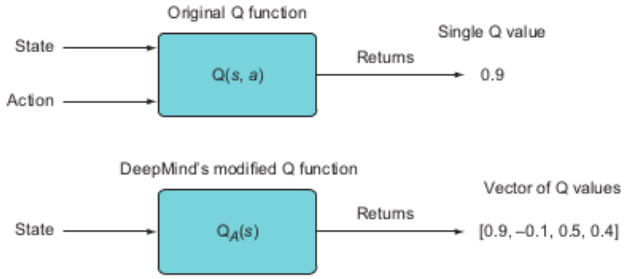

Figure 1.15: Deepminds "modified" Q-function [2].

The neural network used is trained in a supervised way aiming to minimize the loss between its current best action prediction value (that led the agent to state s_2) and the target value $r + \gamma \cdot max\left(Q_A\left(s_2\right)\right)$. But a simple substitution of the Q-table used in Q-Learning with a neural network introduces training instabilities that undermine the performance of the algorithm and thus the algorithm must be further modified. The modifications that Deepmind proposed to increase the stability of the learning procedure are the addition of an experience replay buffer that stores past experiences and the use of a separate network to produce the target values for the Q-Learning update [16, 17].

The experience replay buffer

An empty buffer of finite size is initiated. After each time step, the agent's experience $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$ is stored in the buffer. Of course, the buffer will ultimately contain experiences from various episodes. When updating the network's parameters using Q-Learning update, the loss is computed over a mini-batch of experiences randomly sampled from the experience replay buffer.

This mini-batch approach has two main advantages compared to the on-line version of updating for every new experience. First, learning from consecutive samples is inefficient due to strong correlations between the samples. By using random samples from the experience replay buffer correlations are broken and thus the variance of the updates is reduced. Second, by using mini-batches the behavior distribution is averaged over many previous states smoothing out learning and avoiding oscillations (catastrophic forgetting) [17].

The target network

As already mentioned, the training of the network is performed so that the loss between the predicted action's value and the target value $r + \gamma \cdot max\left(Q_A(s_2)\right)$ is minimized. The stability of learning is increased if those target values are extracted by a copy of the network that is using an older set of parameters. This way, a delay is added between the time an update is performed on the main network and the time this update affects the network producing the target values. Of course, the target network is updated after a given number of updates is performed on the main network [17, 18].

Those additions improve the stability of the training process but Deep Q-Learning algorithm still suffers from catastrophic forgetting [18]. To sum up, the algorithm is presented in algorithm section 5

Algorithm 5 Deep Q-Learning algorithm

Algorithm parameters : experience replay buffer size N, mini-batch size B, learning rate a, $\epsilon \in [0, 1]$, discount factor $\gamma \in [0, 1)$.

- Initialize Q-Network with parameters θ.
- Initialize target $\hat{Q}$-Network with parameters $\hat{\theta}$.
- Loop for each episode:
  - Initialize s.
  - Loop for each step of the episode until s is terminal:
    - With probability ϵ select random action a_t else select $a_t = argmax_a Q(s, a)$.
    - Take action a_t, observe reward r_{t+1} and new state s_{t+1}.
    - Store experience $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$ in experience replay buffer (replace existing experiences if buffer is full).
    - Sample mini-batch of transitions $e_j = (s_j, a_j, r_{j+1}, s_{j+1})$ of size B.
    - Set $y_j = \begin{cases} r_{j+1} & \text{if } s_{j+1} \text{ is terminal state} \\ r_{j+1} + \gamma \cdot argmax_a \hat{Q}(s_{j+1}, a) & \text{otherwise} \end{cases}$
    - Perform gradient descent on $L\left(y_j, Q(s_j, a_j)\right)$ with respect to network parameters θ.
    - Every C steps update $\hat{Q}$-Network parameters $\hat{\theta} = \theta$.

1.4.2 Policy Gradient Methods & REINFORCE

Deep Q-Learning algorithm presented in the previous paragraph is an algorithm which trains a neural network to approximate the optimal action-value function Q^*. Afterwards, the agent performs the action that maximizes the action-value function on the corresponding state. In this paragraph, the algorithm to be presented will not learn the action-value function but a parameterized policy function. Again, as neural

networks can act as universal function approximators, a neural network will be used to approximate the policy function.

To begin with, let us assume a parameter vector $\boldsymbol{\theta}$ to represent the parameterization of the policy function. Then, we write $\pi(a|s,\theta) = Pr[a|S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}]$ for the probability of action a taken at time t. By measuring the performance of the current policy described by parameter vector $\boldsymbol{\theta}$, with a performance measure $J(\boldsymbol{\theta})$, we can update the parameter vector in order to maximize the performance measure using gradient ascent algorithm:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + a\nabla J(\theta_t)$$

Methods following this schema are called *policy gradient methods* and REINFORCE algorithm belongs to this method category [8]. Due to the update rule, it is necessary for the parameterization to be differentiable with respect to the parameters, a property neural networks do posses.

Since the policy function maps actions to probabilities for each different state, a neural network that tries to approximate a policy function should have outputs that are interpreted as probabilities. Using a softmax activation function to the output layer guarantees that the sum of the outputs of last layer's neurons (one neuron for each possible action) is 1 and thus, they can be interpreted as probabilities. The final action that the agent performs is sampled from the action space using the probabilities provided from the output of the policy network.

An advantage of this process is that it inherently deals with the exploration vs exploitation dilemma. In DQN algorithm, it was necessary to introduce some randomness in the process by selecting a random action with a probability of ϵ in order to make exploration of the state-action space possible. On the other hand, by sampling an action using the probabilities provided by the policy network introduces the desired randomness in order to make exploration possible [2].

Another advantage of policy gradient methods is their ability to approximate stochastic policies. If the environment is deterministic, the final probabilities that the network will output for each state will be deterministic as well, with a probability mass of 1 to the proper action. On the other hand, if the environment is stochastic, the network will distribute the probability mass properly to different actions. Action-value methods like DQN have no way of finding stochastic optimal policies [8].

Lastly, another theoretical advantage is that by using a policy parameterization, the action probabilities change smoothly as a function of the learned parameters. Instead, when using $\epsilon-$greedy selection, the action probabilities may vary dramatically even for a small change in the estimated action values, if that change results in an different action having the maximal value. Mainly for this reason, policy-gradient methods posses stronger guarantees of convergence compared to action-value methods.

At this point, it is essential to define the performance measure $J(\boldsymbol{\theta})$ in order to be able to describe the complete algorithm. For the shake of simplicity, we will assume that an episode starts from a random state s_0 and the no discounting case ($\gamma = 1$) will be considered. We denote as $\pi_{\boldsymbol{\theta}}$ the policy determined by parameters $\boldsymbol{\theta}$. Then, for this episode, we define the performance as :

$$J(\boldsymbol{\theta}) = v_{\pi_{\boldsymbol{\theta}}}(s_0)$$

which is the value for policy $\pi_{\boldsymbol{\theta}}$ in state s_0. According to Policy Gradient Theorem, the gradient of the performance measure defined in the above equation is [8]:

$$\nabla J(\boldsymbol{\theta}) \propto \mathbb{E}_\pi \left[\sum_a q_\pi(S_t, a) \nabla \pi(a|S_t, \boldsymbol{\theta}) \right]$$

Now, by multiplying and dividing with $\pi(a, S_t, \boldsymbol{\theta})$ and then replacing a by the sample $A_t \sim \pi$, we get:

$$\nabla J(\boldsymbol{\theta}) \propto \mathbb{E}_\pi \left[q_\pi(S_t, A_t) \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})} \right] = \mathbb{E}_\pi \left[G_t \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})} \right] = \mathbb{E}_\pi \left[G_t \nabla ln\left(\pi(A_t|S_t, \boldsymbol{\theta})\right) \right]$$

Of course, as already mentioned, for the shake of simplicity, the analysis above assumes the no discounting case where γ equals 1. In algorithm section 6, the more general, discounting case scenario is presented.

Algorithm 6 REINFORCE : Monte-Carlo Policy Gradient Control

Algorithm parameters : learning rate a, $\epsilon \in [0, 1]$, discount factor $\gamma \in [0, 1)$.

- Initialize policy network with parameters $\boldsymbol{\theta}$.
- Loop for each episode:
  - Generate an episode $s_0, a_0, r_1, ..., s_{T-1}, a_{T-1}, r_T$, following policy $\pi(\cdot|\cdot, \boldsymbol{\theta})$.
  - For each step of the generated episode $t = 0, 1, ..., T-1$:
    - $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$
    - $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + a\gamma^t G \nabla ln\left(\pi(a_t|s_t, \boldsymbol{\theta})\right)$

Note that algorithm 6 is labeled as "Monte-Carlo", as the parameter update is performed after the completion of an entire episode.

1.4.3 Actor-Critic Methods & Asynchronous Actor-Critic

As presented in the previous paragraphs, Deep Q-Learning algorithm attempts to approximate the optimal action value function Q^* and REINFORCE algorithm attempts to approximate the optimal policy function π^*. Actor-critic methods attempt to combine these two different approaches by learning both a policy and a value function. The reason such combination is developed, is because the critic, which tries to approximate the value function, performs boostrapping (updates the value estimate for a state from the estimated values of subsequent states). The bias introduced through bootstrapping is often beneficial as it reduces variance and it accelerates learning [8]. On the other hand, the actor tries to approximate the optimal policy function as described in the previous paragraph.

The key idea behind the actor-critic approach is the calculation of the *advantage*. Advantage is an attempt to quantify how much better or worse a state we transitioned to is, compared to what we expected. By definition, what we expect about a state s_t is given by the value function $V(s_t)$. The actual value of that state after performing an action a_t and receiving a reward r_{t+1} to move to the next state s_{t+1} is $r_{t+1} + \gamma \cdot V(s_{t+1})$. That said, the advantage is simply the difference:

$$A_t = r_{t+1} + \gamma \cdot V(s_{t+1}) - V(s_t)$$

By using the advantage, the update rule for the actor approximating the optimal policy function with a parameters $\boldsymbol{\theta}$ is similar to that of REINFORCE algorithm with a slight modification:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + a\gamma^t \cdot A_t \cdot \nabla ln\left(\pi(a_t|s_t, \boldsymbol{\theta})\right)$$

Regarding the critic, its parameters $\boldsymbol{w}$ are updated the same way they did in Deep Q Learning algorithm trying to match the target value of $r_{t+1} + \gamma \cdot V(s_{t+1})$ by utilizing a loss function L (like mean square error).

Using these ideas, the one-step version (online) of the Advantage Actor-Critic is presented in algorithm section 7. The algorithm is characterized as one-step because the update to the parameters is performed after each step of each episode.

The problem with the one-step approach is the same with the one faced in Deep Q Learning. When updating the value function, the target value is not accurate and this leads to training instabilities. The problem can be solved by using more accurate target values by calculating the total return that follows after the transition to one state. This approach is the episodic (Monte Carlo) Advantage Actor-Critic. It is characterized as episodic because the update to the parameters is performed after the completion of an entire episode, when the rewards of each action are known and the return after each transition can be calculated. That said, the formula for the return at time step t is given by the following formula:

$$A_t = G_t - V(s_t) = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k - V(s_t) = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... + \gamma^{T-1-t} r_T - V(s_t)$$

Algorithm 7 One-step Advantage Actor-Critic

Algorithm parameters : learning rates $a_{\boldsymbol{w}}, a_{\boldsymbol{\theta}}$, $\epsilon \in [0, 1]$, discount factor $\gamma \in [0, 1)$.

- Initialize policy network with parameters $\boldsymbol{\theta}$.
- Initialize value network with parameters $\boldsymbol{w}$.
- Loop for each episode:
  - For each step $t = 0, 1, ..., T-1$:
    - $A_t \leftarrow r_{t+1} + \gamma \cdot V(s_{t+1}) - V(s_t)$, if s_{t+1} is terminal, $V(s_{t+1}) = 0$
    - $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + a_{\boldsymbol{\theta}} \gamma^t A_t \nabla ln\left(\pi(a_t|s_t, \boldsymbol{\theta})\right)$
    - $target \leftarrow r_{t+1} + \gamma \cdot V(s_{t+1})$, if s_{t+1} is terminal, $V(s_{t+1}) = 0$
    - $\boldsymbol{w} \leftarrow \boldsymbol{w} + a_{\boldsymbol{w}} \nabla L(target, V(s_t))$

Although episodic Advantage Actor-Critic solves the accuracy problem of the target values for the value function, it is not considered a true Actor-Critic method as it does not perform bootstrapping. As already told, bootstrapping is when we are making a prediction out of a prediction (calculating value the value of the next state $V(s_{t+1})$) like we did in the one-step version of the algorithm. As the episodic version does not bootstrap, it does not gain the advantages of increased bias that Actor-Critic methods do provide. Episodic Advantage Actor-Critic is often called as REINFORCE with baseline [8]. The complete algorithm description can be found in algorithm section 8.

Algorithm 8 Episodic Advantage Actor-Critic (REINFORCE with baseline)

Algorithm parameters : learning rates $a_{\boldsymbol{w}}, a_{\boldsymbol{\theta}}$, $\epsilon \in [0, 1]$, discount factor $\gamma \in [0, 1)$.

- Initialize policy network with parameters $\boldsymbol{\theta}$.
- Initialize value network with parameters $\boldsymbol{w}$.
- Loop for each episode:
  - Generate an episode $s_0, a_0, r_1, ..., s_{T-1}, a_{T-1}, r_T$, following policy $\pi(\cdot|\cdot, \boldsymbol{\theta})$.
  - For each step of the generated episode $t = 0, 1, ..., T-1$:
    - $A_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k - V(s_t)$
    - $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + a_{\boldsymbol{\theta}} \gamma^t A_t \nabla ln\left(\pi(a_t|s_t, \boldsymbol{\theta})\right)$
    - $target \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$
    - $\boldsymbol{w} \leftarrow \boldsymbol{w} + a_{\boldsymbol{w}} \nabla L(target, V(s_t))$

An intermediate solution that offers both the bias Actor-Critic methods offer and increased stability compared to 1-step Advantage Actor-Critic is the N-step Advantage Actor-Critic. In this modified version of the algorithm, the update to the parameters is performed every N steps or by the time the episode finished when this happens before N steps. This way we can have a more accurate target value for the value function while still being able to perform bootstrapping as we do not have the rewards for the action after the N^{th} step. The advantage is now calculated with the following formula:

$$A_t = r_{t+1} + \gamma r_{t+2} + ... + \gamma^{N-t-2} r_N + \gamma^{N-t-1} V(s_N) - V(s_t)$$

It can be clearly seen why the N-step approach performs bootstrapping as $V(s_N)$ appears as a term in the advantage, meaning that the algorithm is making a prediction out of a prediction. Of course, if the episode ends before the N^{th} step, only the observed rewards are discounted and summed and the value of the terminal state is considered to be zero. The complete algorithm is now presented in algorithm section 9.

Algorithm 9 N-step Advantage Actor-Critic

Algorithm parameters : learning rates $a_{\boldsymbol{w}}, a_{\boldsymbol{\theta}}$, $\epsilon \in [0,1]$, discount factor $\gamma \in [0,1)$, number of steps N.

- Initialize policy network with parameters $\boldsymbol{\theta}$.
- Initialize value network with parameters $\boldsymbol{w}$.
- Loop for each episode:
  - Generate an episode $s_0, a_0, r_1, ...,$ following policy $\pi(\cdot|\cdot, \boldsymbol{\theta})$ until the N^{th} state or the end of the episode (if the end comes before N^{th} state).
  - For each step of the generated episode $t = 0, 1, ...$:
    - $A_t \leftarrow r_{t+1} + \gamma r_{t+2} + ...\gamma^{N-t-2} r_N + \gamma^{N-t-1} V(s_N) - V(s_t)$ (with terms referring to steps after the episode end being zero)
    - $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + a_{\boldsymbol{\theta}} \gamma^t A_t \nabla ln\left(\pi(a_t|s_t, \boldsymbol{\theta})\right)$
    - $target \leftarrow r_{t+1} + \gamma r_{t+2} + ... + \gamma^{N-t-2} r_N + \gamma^{N-t-1} V(s_N)$ (with terms referring to steps after the episode end being zero)
    - $\boldsymbol{w} \leftarrow \boldsymbol{w} + a_{\boldsymbol{w}} \nabla L(target, V(s_t))$

Having presented the idea of the Advantage Actor-Critic algorithm, we will proceed to the Asynchronous Advantage Actor-Critic, shortly referred to as A3C. This variation of the original Actor-Critic method exploits parallel actor learners that are trained on their own replica of the environment while updating the initial model parameters that they share. In more detail, every agent shares the same parameters $\boldsymbol{\theta}, \boldsymbol{w}$ for their policy and value networks respectively and asynchronously, perform updates to them, that are visible to everyone, the same way it was presented in the previous algorithm sections. [2, 19]

A3C algorithm appears to increase the stability of learning and decrease the training time needed for convergence. The core of the Asynchronous Advantage Actor-Critic algorithm can be any version of the actor-critic, either on-line, Monte-Carlo or N-step. In any case, the main difference between the asynchronous approach with the original methods is the parallel nature that comes by training multiple agents simultaneously [19].

It must be noted, that the asynchronous approach can be used along with the rest of the algorithms like REINFORCE and Deep Q Learning, offering the same advantages. Furthermore, for the case of Deep Q Learning, asynchronous approaches enables us to not rely on experience replay buffer to stabilize the training. This is a major advantage because if, for any case, a type of recurrent neural network should be used, it could not rely on an experience replay buffer due to its need to operate on sequential experiences.

1.4.4 Proximal Policy Optimization

Proximal Policy Optimization, shortly referred to as PPO, is an Actor-Critic method, developed by OpenAI [5]. It aims to overcome the problems that common Actor-Critic approaches face while keeping the implementation complexity at a minimum, unlike Actor-Critic with Experience Replay (ACER) & Trust Region Policy Optimization (TRPO) which are other attempts to overcome those problems [20, 21]. While Actor-Critic methods do provide significant improvements compared to vanilla policy gradient methods, they are still sensitive to the choice of step size. Small step sizes can lead to significantly slow learning progress where as large step sizes lead to noise and catastrophic drops in performance.

In order to deal with these problems, the general idea behind PPO is to try to make the updates of the policy function delicate enough so that catastrophic drops in performance are not observed. Like TRPO, PPO uses the probability ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ in the objective under optimization. Trying to maximize the objective $L(\theta) = \hat{\mathbb{E}}_t\left[r_t(\theta) A_t\right]$ would lead to excessively large policy updates resulting in the problems mentioned above. That said, it is essential that changes of the policy that move $r_t(\theta)$ away from 1 are penalized. The objective that the OpenAI team proposes is the following:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[min(r_t(\theta) A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon) A_t)\right]$$

where ϵ is a hyperparameter. The second term inside the min is responsible for clipping the probability ratio when its values are outside of the interval $[1-\epsilon, 1+\epsilon]$. The minimum of the the clipped and unclipped objective is finally taken so that the final objective is a lower bound of the unclipped objective. This way, the change in probability ratio is ignored when it would make the objective improve and on the contrary, it is included when it makes the objective worse [5].

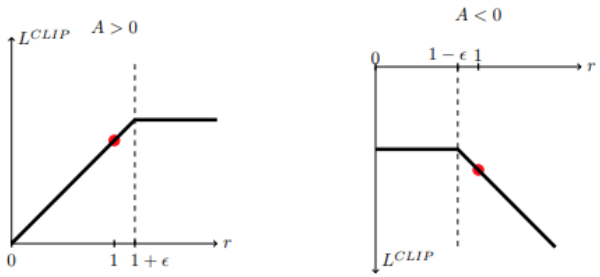

Figure 1.16: Main objective's behavior for positive and negative advantages [5].

While the objective L^{CLIP} is used by the policy network so that the optimal policy π^* is discovered, for the value network, a standard mean square error loss $L^{VF}(\theta)$ can be used like in the previous algorithms. Furthermore, in order to ensure sufficient exploration, the objective can further be augmented by adding an entropy bonus. If policy and value networks share a number of parameters, the overall objective must combine both the policy objective and the value function error term. The complete formula is presented in the equation above:

$$L^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}\left[L^{CLIP}(\theta) - c_1 L^{VF}(\theta) + c_2 S[\pi_\theta](s_t)\right]$$

where c_1, c_2 are coefficients and S denotes the entropy bonus.

Lastly, PPO algorithm can take advantage of multiple agents acting in the same environment like the A3C algorithm. Massively parallel approaches can yield state of the art results while increasing the stability and the convergence speed of the learning process [22]. The complete algorithm, as presented in the original paper utilizing multiple agents, is illustrated in algorithm section 10.

Algorithm 10 Proximal Policy Optimization

- For episode=1,2,... :
  - For actor=1,2,...,N :
    - Run policy $\pi_{\theta_{old}}$ in environment for T timesteps
    - Compute advantage estimates $A_1, ..., A_T$
  - Optimize L with respect to θ, with K epochs and minibatch size $M \leq NT$
  - $\theta_{old} \leftarrow \theta$

Chapter 2

Training Deep Reinforcement Learning Agents

2.1 Main Development Tools

This section will cover the two main tools used for training Deep Reinforcement Learning agents, OpenAI Gym and Pytorch. In short, OpenAI Gym is a library developed by OpenAI team for providing easy access to a collection of environments for the agents to interact. Their work enables engineers and researchers to focus their attention on the training algorithms than designing the environments. Pytorch on the other hand is a machine learning library developed by Facebook for creating and training neural network architectures.

2.1.1 OpenAI Gym

In this previous chapter, there has been a thorough presentation of the theoretical background regarding reinforcement learning, neural networks and deep reinforcement learning algorithms. In every algorithm presented, the environment was always taken for granted. In reality, when trying to train an agent to interact properly with an environment, a detailed simulation of this environment must be created because having a robot agent to learn through trial and error in the real world is obviously not an efficient idea (regarding both time and money).

A researcher, when trying to develop a new algorithm to achieve state of the art results or a student, when trying to introduce himself in the world of reinforcement learning, should have to deal with the learning paradigm and the algorithms rather than the design of a simulation of an environment. For this reason, OpenAI has created a toolkit that enables an individual to deal with reinforcement learning exclusively. This toolkit is OpenAI Gym and it enables the development and comparison of reinforcement learning algorithms. Gym provides a growing collection of benchmark environments that expose a common and easy to use interface for the interaction with them [23]. Indicatively, three environments from the entire collection are illustrated in figure 2.1. On the left, the Cart Pole problem is illustrated where the agent's goal is to try to balance the pole by moving either left or right. In the center, the well known Atari 2600 Assault game is presented where the agent's goal is to play the game successfully. Finally, on the right, the agent's goal is to learn control the ant by making it walk forward as fast as possible.

As already stated, all the environments expose a common interface for the agent to interact with them. The API is simple and its fundamental component is the *environment* object returned by *make()* function which expects as input the name of the environment to be created. Afterwards, the following methods of the *environment* object are used for the interaction and the development [24]:

- *reset()* : resets the environment by returning an initial state
- *render()* : responsible for rendering the next frame of the simulation
- *step()* : takes an action as an input and returns the next state, the received reward, a flag that notifies whether a terminal state has been reached and info useful for debugging

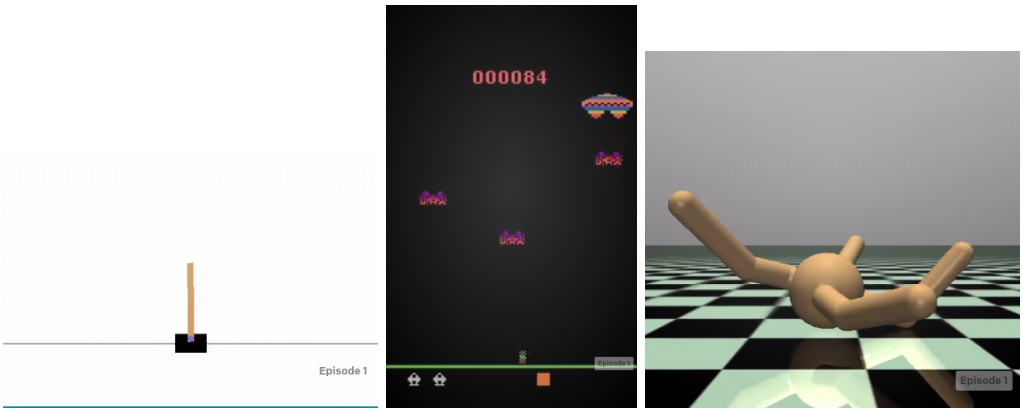

Figure 2.1: Cart pole problem (left), Atari 2600 Assault game (center), Ant (right).

The action space of the environment can been obtained via the *action_space* attribute that also supports random sampling an action from it via the *sample()* method. Furthermore, it is possible to obtain the state space via the *observation_space* attribute.

To summarize, in the code section above, an indicative example of the API's usage is presented for performing random actions in the Cart-Pole environment until a terminal state is reached.

```
import gym

# Create CartPole environment
env = gym.make('CartPole-v0')
env.reset()
done = False

while not done:
    env.render()
    action = env.action_space.sample()
    state, reward, done, info = env.step(action)
```

2.1.2 Pytorch

Another important aspect of the present thesis is neural networks. Theoretical background regarding neural networks was presented in the previous chapter and it has been shown the a lot of matrix math is involved in their operation. Furthermore, as seen in paragraph 1.3.4, training of multilayer neural networks (or any other type of neural layers they posses), involves the computation of derivatives from the output towards the input of the network for making proper correction to each neuron's parameters. This process is known as the back propagation algorithm.

For this purpose, automatic differentiation libraries have been developed that are responsible for tracking the performed computations and later replays them backward to compute the gradients. This way, any kind of neural network architecture can be created and the library is responsible for calculating the gradients for the training process.

Pytorch is one of these libraries and in the last years it has been the most popular choice by engineers in either industry and academy [25]. The fundamental block of the library's operation is the *Tensor*. A tensor is basically a multidimensional rectangular array of numbers similar to the well known Numpy arrays. A tensor can hold information regarding the input to a neural network, the output of a neural layer or the parameters of a neural layer. Operations between Pytorch tensors can be computed on classic CPUs but also on hardware accelerators like GPUs. Overall, three main modules constitute the whole operation of the framework [26]:

- Autograd module : as already stated, this module is responsible for tracking the performed operations between tensors and later replays them backward in order to compute the gradients [27].
- Optim module : this module implements the optimization algorithms (Gradient Descent, Adam e.t.c.) that leverage the computed gradients in order to update the parameters of a neural network.
- nn module : this module provides a user friendly way of defining network architectures (computation graphs) that later autograd can perform operation tracking.

For now, no example code will be given for training a neural network in Pytorch as it is not as compact as using the OpenAI Gym API in the previous paragraph but numerous code sections leveraging Pytorch to train Deep Reinforcement Learning agents will be later presented.

2.2 Deep Reinforcement Learning Framework

As part of the present thesis, there has been an attempt to create a framework implementing four different deep reinforcement learning algorithms the theory of which has been presented in the first chapter. The goal of the whole implementation is to allow the training of different agents for various environments (having either discrete or continuous action spaces) without the need of re-implementing the whole or part of an algorithm and by just plugging in the desired neural network model and the environment object that exposes the same API that openAI gym environments expose.

In this section, a general description of the frameworks operation and usage will be presented showing how to create a neural network model and train an agent using the selected algorithm. The complete code regarding the implementation of each algorithm will be presented in the appendix while the complete repository for the framework can be found here.

2.2.1 Neural Network Definition

For each implemented algorithm to work, two arguments must be passed to its initialization function, the neural network model and the environment that the agent will operate in. In order for a neural network object to be valid, it must implement three essential methods: *infer_step(), infer_batch() & infer_action()*. Each method's functionality is described bellow:

- *infer_step()* : This method expects as input a single state that the agent lies in. After performing the calculations, it outputs : the Q-value of the state, and the selected action in case of a deepQ-network, the action distribution (policy), the selected action and the value of the state in case of an actor critic model or the action distribution (policy) and the selected action in case of a policy network utilized for REINFORCE algorithm.
- *infer_batch()* : Like *infer_step()* but it does not return a selected action and it operates in a batch of states, therefore returning a batch of distributions or values depending on the model.
- *infer_action()* : This method operates on a single state and returns just the selected action of the model. It is used for evaluation purposes.

That said, the creation of a neural network that will be used in the REINFORCE algorithm for a discrete action space environment having a continuous valued four-dimensional observation space is illustrated in the following code section. The network's architecture does not concern us for the time as the goal is the understanding of the implementation of the methods mentioned above.

```
1 import torch
2 import torch.nn as nn
3
4 class SimpleReinforce(nn.Module):
5
```

```
    def __init__(self):
        super(SimpleReinforce, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(4, 32),
            nn.ReLU(),
            nn.Linear(32, 16),
            nn.ReLU(),
            nn.Linear(16,2),
            nn.Softmax(dim=1)
        )

    def infer_step(self, x):
        act_prob = self.model.forward(x)
        dist = torch.distributions.Categorical(probs=act_prob)
        action = dist.sample().item()
        return dist, action

    def infer_batch(self, x):
        act_prob = self.model.forward(x)
        dist = torch.distributions.Categorical(probs=act_prob)
        return dist

    def infer_action(self, x):
        act_prob = self.model.forward(x)
        dist = torch.distributions.Categorical(probs=act_prob)
        action = dist.sample().item()
        return action
```

As it can be seen, Pytorch's distributions are heavily used. Given a selected action, the probability for this specific function can be obtained via the distribution with using: *dist.log_prob(action)*.

2.2.2 Training the Agent

Having defined the model that implements the three methods that are required from the framework, one can simply train the network using one of the implemented by the framework algorithms. The following code section illustrates how the neural network defined above can be trained using REINFORCE algorithm:

```
import gym
import matplotlib.pyplot as plt
from models import SimpleReinforce
from diploma_framework.algorithms import Reinforce

env = gym.make('CartPole-v0')
model = SimpleReinforce()

alg = Reinforce(environment=env,
          model=model,
          lr=1e-03,
          max_frames=150_000,
          num_steps=200,
          gamma=0.99)

rewards_rein = alg.run(eval_window=1000,
                  n_evaluations=10,
                  early_stopping=False,
```

```
                    reward_threshold=197.5)

alg.save_model('models/trained_reinforce.joblib')
```

2.3 Cart Pole Problem

This section constitutes a first attempt to benchmark the algorithms presented in section 1.4 on Cart-Pole environment. In the following paragraphs there will be a brief description of the specific environment and then the performance of the different algorithms will be discussed.

2.3.1 The CartPole Environment

Cart pole environment can be considered the "Hello World" problem of reinforcement learning. As described in documentation, the environment simulates a pole which is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum starts upright, and the goal is to prevent it from falling over. An illustration of the environment can be seen in figure 2.2.

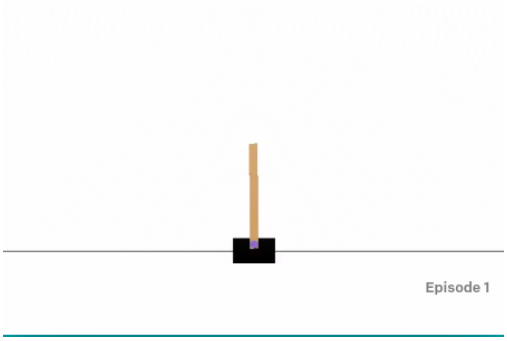

Figure 2.2: Cart pole environment.

After obtaining one environment instance using *env = gym.make('CartPole-v0')*, one can check the environment's observation space and action space using *env.observation_space, env.action_space* respectively. For the observation space, each state consists of four components that are described in the table 2.3.1.

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	∼-0.418 rad	∼0.418 rad
3	Pole Tip Velocity	-Inf	Inf

Table 2.1: Observation space of Cart Pole environment.

It should be noted that one can use the raw pixels from the rendered environment as the observations for training an agent instead of the provided 4-dimensional observation vector. For the solution presented in the current section, the 4-dimensional observation vector will be used.

Regarding the action space for the specific environment, there are two discrete actions encoded with 0 and 1. Action 0 is translated into moving the cart to the left and action 1 is translated into moving the cart to the right.

Termination of the episode comes with the following conditions :

- Pole angle is more than $12°$ or less than $-12°$.

- Center of the cart reaches the limits of display (more than 4.8 units away from the center).
- Episode length is greater than 200 frames.

For every step that is taken that leads the pole to an angle within the above limits and the cart again within the above limits, a reward of +1 is received. In case a step leads the cart or the pole to an unwanted state then a reward of 0 is received. Of course, at the 200^{th} frame the game is won.

2.3.2 DQN Agent

The first algorithm to be tested on Cart Pole environment is DQN. Theoretical background about DQN was described in section 1.4 and the detailed implementation in Python can be found in appendix and also in the provided Github repository. In order to evaluate the algorithm, during the training process the average cumulative reward is calculated periodically after a certain number of frames has been seen by the agent. For every experiment from now on, the average cumulative reward of 10 episodes will be calculated every 1000 frames. It is necessary to average out the cumulative rewards on a number of episodes in order for the related curves to be smooth and provide more information.

In case we were not interested about comparing the performance of different deep reinforcement learning algorithms, we would initiate any attempt to train an agent with *early_stopping* parameter set to *True*. This way, when the stopping criteria are met (in case of the Cart Pole environment, having an agent that returns an average cumulative reward of 200 after the evaluation), the training process would terminate. Although, because we wish to compare the performance and the stability of different algorithms, the training process will not be terminated upon certain stop criteria are met, but instead the algorithm will be left to reach a maximum number of frames seen by the agent. For DQN and every algorithm after that, the maximum number of frames seen by an agent is set to 150,000.

As Cart Pole environment can be considered as the "hello world" problem of reinforcement learning, the network that will be used for DQN and the rest of the algorithms is a simple feed-forward neural network without great complexity. In more detail, the network used in the present task consists of 2 hidden layers of neurons and the output layer. Each neuron in the hidden layers uses a ReLU activation function. Definition of the model is presented in the following code section.

```
import torch
import torch.nn as nn

class SimpleDQN(nn.Module):

    def __init__(self, n_states, n_actions):
        super(SimpleDQN, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(n_states, 32),
            nn.ReLU(),
            nn.Linear(32,16),
            nn.ReLU(),
            nn.Linear(16,n_actions)
        )

    def infer_step(self, x):
        qval = self.model(x)
        action = qval.detach().argmax().item()
        return qval, action

    def infer_batch(self, x):
        return self.model(x)

    def infer_action(self, x):
```

```
25          return self.model(x).detach().argmax().item()
26
27  model = SimpleDQN(4, 2)
```

It is true that deep reinforcement learning algorithms are really sensitive to the selection of their hyperparameter's values. In case of the current environment, a really quick trial and error search lead to selection of the hyperparameters presented in table 2.2.

Parameter	Value	Parameter	Value
sync_freq	1000	epsilon_start	1
lr	10^{-3}	epsilon_end	0
batch_size	128	epsilon_decay	250
max_frames	150000	gamma	0.9

Table 2.2: Selected hyperparameters used for DQN algorithm on Cart Pole environment.

Running the algorithm and logging the cumulative rewards after the selected number of frames as described above leads to figure 2.3.

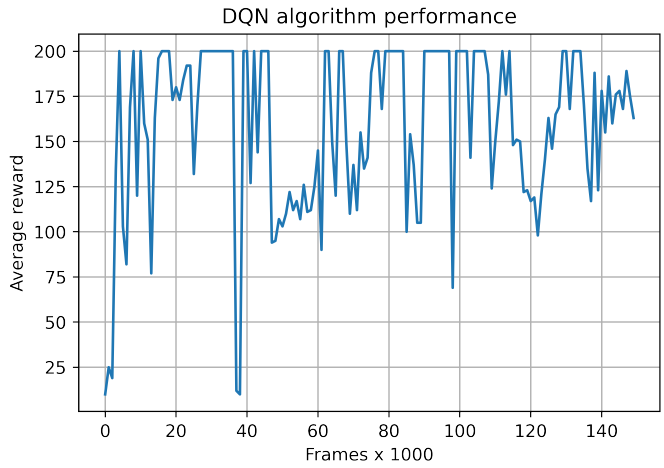

Figure 2.3: Average cumulative rewards during DQN agent training.

It is clear that the agent immediately starts learning. The first evaluation happens at the 1000^{th} frame with the cumulative reward being fairly low signifying that he agent still makes random moves leading the pole to fall quickly since a reward of about 25 is gained. The situation changes after the 3000^{th} frame where the agent seems to already master the game with the evaluation returning and average cumulative reward of 200 after averaging out the rewards of 10 different episodes. As the CartPole-v0 environment considers the problem solved after 200 frames of not losing (i.e. a cumulative reward of 200), the agents seems to already have learned how to win the game. The overall training process that terminates after 150,000 frames lasts 3 minutes and 30 seconds.

As already stated at DQN algorithm's theoretical background, the training process seems to lack stability. Having the early stopping feature of the process disabled, the process carries on even after the agent seems to be capable of winning the game. As the process carries on, it appears that the agent has severe sudden losses of performance. For example, around the 40000^{th} frame, the agent seems to have completely forgotten how to "play the game". The same behavior appears along the entire training process with sudden and severe performance drops. This is the main problem that DQN algorithm faces, named "Catastrophic Forgetting".

On the other hand, although catastrophic is a serious problem, it can easily be avoided by enabling the early stopping feature and terminating the process after a certain average cumulative reward is achieved. This

can certainly be done as we have apriori knowledge of the cumulative reward of a won episode. However, there are environments that such information is not known or cannot be known. By enabling the early stopping feature, DQN algorithm seems to have a fast convergence.

2.3.3 REINFORCE Agent

Moving on, we evaluate REINFORCE algorithm for Cart Pole problem. Once again, the neural network that is used is a simple feed forward network without great complexity. Again, it consists of 2 hidden layers and one output layer consisting of 32, 16 and 2 neurons respectively. Because the network will act as a parameterization of the policy function, which outputs the probabilities of each action, a softmax activation function is applied at the output layer while the rest of the layers have a common ReLU activation function. The definition of the model is presented in the following code section:

```
import torch
import torch.nn as nn

class SimpleReinforce(nn.Module):

    def __init__(self):
        super(SimpleReinforce, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(4, 32),
            nn.ReLU(),
            nn.Linear(32, 16),
            nn.ReLU(),
            nn.Linear(16,2),
            nn.Softmax(dim=1)
        )

    def infer_step(self, x):
        act_prob = self.model.forward(x)
        dist = torch.distributions.Categorical(probs=act_prob)
        action = dist.sample().item()
        return dist, action

    def infer_batch(self, x):
        act_prob = self.model.forward(x)
        dist = torch.distributions.Categorical(probs=act_prob)
        return dist

    def infer_action(self, x):
        act_prob = self.model.forward(x)
        dist = torch.distributions.Categorical(probs=act_prob)
        action = dist.sample().item()
        return action
```

REINFORCE is executed using the set of hyperparameter values presented in table 2.3.

Parameter	Value	Parameter	Value
lr	10^{-3}	num_steps	200
max_frames	150000	gamma	0.99

Table 2.3: Selected hyperparameters used for REINFORCE algorithm on Cart Pole environment.

The plot of the logged cumulative rewards plotted after the training process is completed is illustrated in figure 2.4.

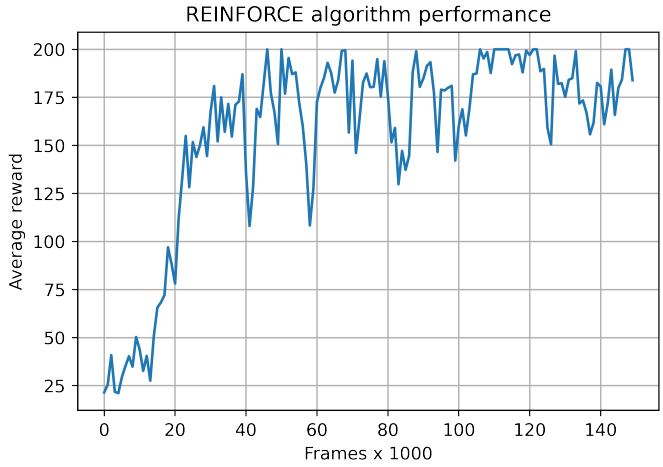

Figure 2.4: Average cumulative rewards during REINFORCE agent training.

Once again, it is clear that the agent goes through a successful learning process as the average cumulative reward increases and finally reaches the ultimate goal of 200 therefore winning the game. Convergence speed is clearly slower compared to that of DQN but the overall training process does not suffer from severe performance drops although it still is not stable. Completion of the training process after 150,000 episodes lasts 1 minute and 33 seconds, which is significantly faster compared to DQN.

2.3.4 A3C Agent

A3C algorithm belongs in the Actor Critic class of methods utilizing two neural networks to parameterize both the policy function with the actor and the value function with the critic. For that reason it is necessary to implement two neural networks, one to be the actor and another one to be the critic. For Cart Pole problem, as before, we define two minimalistic neural networks without great complexity. The actor network, consists of 2 hidden layers and the output layer.

As before, a ReLU activation function is applied to the output of each hidden layer and a softmax function at the output of the output layer. The layers consist of 25, 50 and 2 neurons respectively. Regarding the critic network, since the actor's first layers learn a representation of the state space, we use this representation for the critic network as well. This means the both the actor and the critic share the first two layers. However, using Pytorch's *detach()* method, we detach from the computation graph the output of the shared layers when feeding it to the rest of the critic network in order to prevent the critic from getting involved with the parameter updates of the shared layers. That said, critic's unique layers are two, a hidden layer composed by 25 neurons and the output layer containing simple one neuron. The definition of the model can be found in the following code section:

```
import torch
import torch.nn as nn

class ActorCritic(nn.Module):

    # Define double-headed model, one
    # head for actor, another one for critic

    def __init__(self):
        super(ActorCritic, self).__init__()
        self.l1 = nn.Linear(4,25)
        self.l2 = nn.Linear(25, 50)
```

```
        self.actor_lin1 = nn.Linear(50,2)
        self.l3 = nn.Linear(50,25)
        self.critic_lin1 = nn.Linear(25,1)

    def forward(self, x):
        y = F.relu(self.l1(x))
        y = F.relu(self.l2(y))
        actor =  F.log_softmax(self.actor_lin1(y), dim=1)
        c = F.relu(self.l3(y.detach()))
        critic = self.critic_lin1(c)
        return actor, critic

    def infer_step(self, x):
        action_probs, value = self.forward(x)
        dist = torch.distributions.Categorical(logits=action_probs)
        action = dist.sample().item()
        return dist, action, value

    def infer_batch(self, x):
        action_probs, value = self.forward(x)
        dist = torch.distributions.Categorical(logits=action_probs)
        return dist, value

    def infer_action(self, x):
        dist, _ = self.forward(x)
        dist = torch.distributions.Categorical(logits=dist)
        return dist.sample().cpu().numpy()[0]
```

A3C is executed using the set of hyperparameter values presented in table 2.4. It must be noted the *actor_weight* and *critic_weight* parameters refer to the weight multiplied with the losses of two networks which are later summed for the calculation of the total loss to be minimized.

Parameter	**Value**	**Perparameter**	**Value**
n_workers	32	actor_weight	1
lr	10^{-3}	critic_weight	0.1
max_frames	150000	gamma	0.99
num_steps	200		

Table 2.4: Selected hyperparameters used for A3C algorithm on Cart Pole environment.

The plot of the logged cumulative rewards plotted after the training process is completed is illustrated in figure 2.5. Once again, it is clear that the agent progressively learns to play the game and finally masters is achieving an average cumulative reward of 200.Compared to the two previous methods, A3C seems to provide really stable training as there are no performance drops after a specific point. Furthermore, due to its parallel nature, utilizing 32 different workers to explore the environment in simultaneously, the training process achieves the illustrated results in just 1 minute and 20 seconds. However in case early stopping feature was enabled, the converge of the algorithm would not be that fast as the network managed to master the game after seeing approximately 80,000 frames.

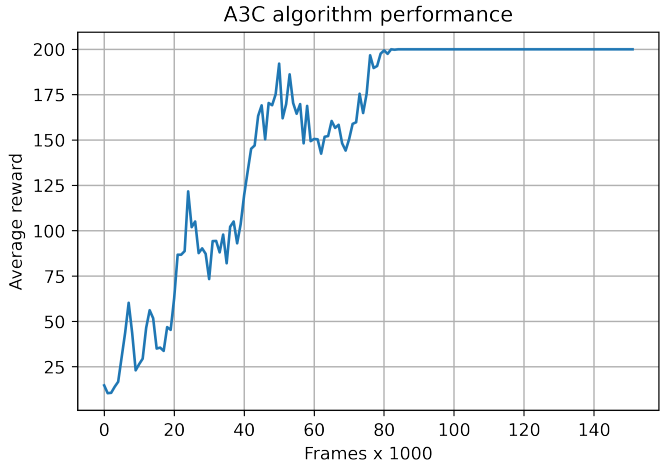

Figure 2.5: Average cumulative rewards during A3C agent training.

2.3.5 PPO Agent

The final experiment that is conducted, is training a PPO agent. PPO is considered an Actor Critic method and it supports the use of multiple parallel workers. However, the current implementation of PPO involves exactly one worker. For a more fair comparison with A3C, the same actor and critic neural networks are used that are presented in the last code section. The algorithm is executed using the set of hyperparameter values presented in table 2.5.

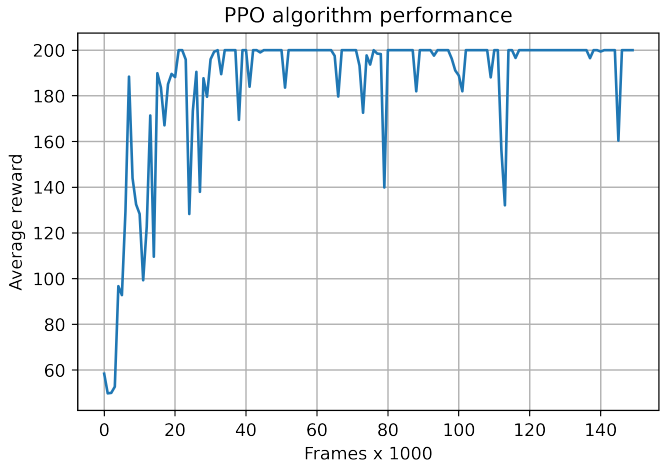

Figure 2.6: Average cumulative rewards during PPO agent training.

In figure 2.6, the plot of the logged cumulative rewards is illustrated. Compared to A3C's evaluation figure, PPO seems to have a very fast convergence mastering the game at approximately 20,000 frames. Although performance drops do exist during the rest of the training process, they are immediately corrected with the agent returning to "master" performance collecting a cumulative reward of 200. However, since the implementation of PPO does not involve multiple parallel workers, the training time is significantly greater than that of A3C taking 6 minutes and 32 seconds to complete the overall process.

Parameter	Value	Perparameter	Value
lr	10^{-3}	gamma	0.99
batch_size	4	lamb	1
epochs	4	actor_weight	1
max_frames	150000	critic_weight	0.1
num_steps	150	entropy_weight	0.001
clip_param	0.2		

Table 2.5: Selected hyperparameters used for PPO algorithm on Cart Pole environment.

2.3.6 Overall Comparison

The comparison of the four different algorithms that are used to train the agent is really interesting. In figure 2.7, the evaluated average cumulative rewards for different algorithms are illustrated. The plot on the right illustrates the same results with DQN performance excluded to make it more easily readable.

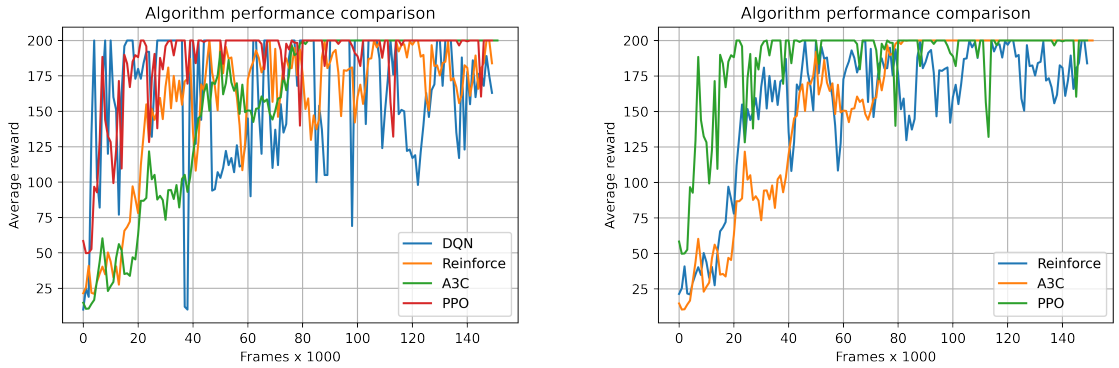

Figure 2.7: Performance comparison of different algorithms.

Having put the four different performance curves on the same plot, the superiority of A3C and PPO in the Cart Pole environment is clear. A3C produces a really stable training while PPO although being a little bit more unstable has the fastest convergence. Another useful plot is that of training duration for each algorithm which is illustrated in figure 2.8.

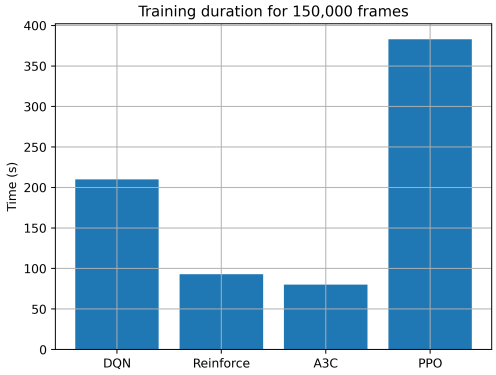

Figure 2.8: Training duration for 150,000 frames.

Training duration results are expected. A3C is the fastest approach at it utilizes parallel workers that simultaneously explore the environment and update the network's parameters. On the other hand, PPO is

the slowest as the implementation that is used does not utilize parallel workers like A3C and also, for each time gradient descent is applied to the networks' parameters, it is applied for multiple epochs (4 for this specific example) and thus, it increases the training time. Furthermore, the fact that PPO has to optimize two networks, the actor and the critic, instead of one like DQN and REINFORCE, is another reason that leads to increased training duration.

2.4 Exploring More Complex Environments - DuckieTown

In section 2.3, a brief introduction to OpenAI's Cart Pole environment was made and then four different deep reinforcement learning algorithms were evaluated. It is true that Cart Pole environment is a fairly simple case. Although its observation space is continuous, it has low dimensionality and a discrete action space and thus, no complex neural network must be utilized. In the present section, we will deal with a more complex problem using a more difficult environment called DuckieTown and we will utilize a more complex neural network. In the following paragraphs, there will be a description of the environment and the performance of PPO algorithm used for training the agent will be presented.

2.4.1 The DuckieTown Environment

DuckieTown environment [28] is a self-driving car simulator exposing an API which is the same as OpenAI's Gym. DuckieTown offers a wide range of tasks, from simple lane-following to full city navigation with dynamic obstacles enabling the training and experimentation with reinforcement learning agents. An illustration of the environment with an agent siting in a random position can be seen in figure 2.9.

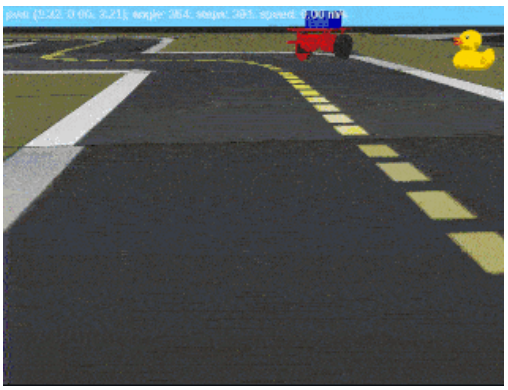

Figure 2.9: DuckieTown environment.

While the previously studied CartPole environment's state consisted of vectors containing four elements describing cart position, cart velocity, pole angle and pole tip velocity, DuckieTown's states are the actual RGB images that are rendered and are visible to human eyes.

As far as the action space is concerned, the original environment has a continuous action space. This means that instead of giving discrete values to *step()* method that encode a specific action, continuous values are given that represent an actual quantity. In case of DuckieTown, an action is a two dimension vector $\mathbf{a} = [a_1, a_2]$ where a_i is the angular velocity of each wheel of a robot moving in the environment and $a_i \in [-1, 1]$.

Regarding the rewards, DuckieTown implements a complex reward function that returns higher values when an agent has the proper position in the right lane and moves with the highest possible speed simultaneously. Reward decreases as the agent does not have the proper position in lane (e.g. it is not at the center of it) and it becomes negative when the agent collides with an obstacle or leaves the road.

2.4.2 PPO Agent Training

In order to create an intelligent agent able to drive in DuckieTown, Proximal Policy Optimization algorithm is used. This algorithm was chosen due to its fast convergence and its stable nature, characteristics that were demonstrated in CartPole environment. Although PPO's behaviour was really desirable in CartPole problem, this does not necessarily mean that it will be the optimal training algorithm for DuckieTown as well. However, we proceed with PPO primary for research reasons since approaches utilizing off-policy Q-Learning already exist [29].

In order to perform proper training on DuckieTown environment, there is one major change that needs to be implemented to basic PPO algorithm that was used on CartPole environment. During training on CartPole, we consider as a state an one dimensional vector that contains information about the pole's and the cart's velocity and position. Since the state contains information about velocity and not just positions, previous states can be considered independent of the current state. We should recall what was presented in paragraph 1.2.2, that the learning problem that Reinforcement Learning deals with is described using the Markov Decision Process model. MDP model makes an important assumption, that at each step, actions and rewards depend only on the current state and not on the entire history of the states and actions taken. If we move to DuckieTown, states are just RGB frames that literally contain what the agent sees thought its camera sensor. At this moment we should pause and think about the following scenario. Could a human decide whether he should turn a car left or right, accelerate or break if just a single frame of a camera installed on a car was presented to him? Clearly, a single frame does not hold any information about car's velocity or acceleration, which is useful information in order to properly drive the car. That said, a state, at the form that DuckieTown outputs it is not independent of the previous states and does not fulfill the Markovian assumption that needs to be valid in order for the agent to be trained properly. For this reason, PPO is redesigned and instead of using frames as the states, a state is considered to be multiple frames stacked together forming a tensor of dimension $(1 \times \#\text{ stacked frames} * 3 \times height \times width)$, where 1 is the size of the batch containing exactly one state and 3 is the number of channels in each frame (RGB). Stacking multiple frames together is a standard way to make the Markovian assumption valid [29] since the dependence between frames that have a time difference greater than the size of the stack is much weaker. We will refer to the new implementation of PPO algorithm that used stacked frames as a state as "Stacked Frame PPO". The implementation can be found with the rest of the algorithms in the appendix.

Setting up the environment

As described in the previous paragraph, DuckieTown original environment has a continuous action space. However, the deep reinforcement algorithms that are already implemented in the framework operate on environments with discrete actions spaces. For this reason, it is necessary to convert the environment's continuous action space to discrete if we wish to use the same implementations of deep reinforcement learning algorithms.

Hopefully, doing such a conversion is fairly easy by using OpenAI's gym wrappers. Wrappers allow us to perform transformations of action spaces and rewards. As in Péter Almási et al approach with Deep Q-Learning, the action space is turned to discrete with each action mapped to specific values of angular velocities for each wheel. The same mappings are used as those presented in [29]. In table 2.6 details regarding those mappings are presented.

Discrete action	Left wheel angular velocity	Right wheel angular velocity	Decription
0	0.04	0.4	Turn left
1	0.4	0.04	Turn right
2	0.3	0.3	Move forward

Table 2.6: Mappings of discrete actions to continuous values of wheel angular velocities.

In the following code section, the definition of the wrapper used to implement these mappings in the actual environment is presented.

```python
import gym
from gym import spaces
import numpy as np

class DiscreteWrapper(gym.ActionWrapper):
    """
    Duckietown environment with discrete actions (left, right, forward)
    instead of continuous control
    """

    def __init__(self, env):
        gym.ActionWrapper.__init__(self, env)
        self.action_space = spaces.Discrete(3)

    def action(self, action):
        # Turn left
        if action == 0:
            vels = [0.04, 0.4]
        # Turn right
        elif action == 1:
            vels = [0.4, 0.04]
        # Go forward
        elif action == 2:
            vels = [0.3, 0.3]
        else:
            assert False, "unknown action"
        return np.array(vels)

    def reverse_action(self, action):
        raise NotImplementedError()
```

Apart from actions, in order to follow the same reward distribution that is used in Péter Almási's et al approach, a wrapper for the rewards also needs to be created. The defined wrapper is presented in the following code section. The final reward function is basically the following:

$$reward = \begin{cases} 10 \cdot speed \cdot dot\_dir - 100 \cdot dist + 400 \cdot col\_pen & \text{,when robot in correct lane} \\ 400 \cdot col\_pen & \text{,when robot in wrong lane} \\ -40 & \text{,when robot leave the track} \end{cases}$$

where *speed* is the speed of the robot in the simulator, *dot_dir* is calculated as the dot product of the vectors pointing towards the heading of the robot and the tangent of the curve, *dist* is the distance from the center of the right lane, and *col_pen* is a penalty for collisions.

```python
class DtRewardWrapper(gym.RewardWrapper):
    """
    Wrapper for modifying rewards to follow what is implemented in
    https://arxiv.org/abs/2009.11212
    """
    def __init__(self, env):
        super(DtRewardWrapper, self).__init__(env)

    def reward(self, reward):
        if reward == -1000:
            reward = -40
        elif reward < 0:
```

```
            reward = reward * 10
        elif reward >=0:
            reward = reward * 10
        return reward
```

Finally, in order to apply these transformations defined by the wrappers to a DuckieTown environment, one has simply to pass an environment instance throught the wrapper classes like presented bellow:

```
env = DtRewardWrapper(env)
env = DiscreteWrapper(env)
```

Creating the model

Having set up the environment's important details, it is time to define the models the will be used, the actor network and the critic network. Since the state that will be the input to the models will be stacked images, a simple feed forward network consisting of plane dense layers cannot be used. Instead, our networks will be convolutional neural networks able to process images and discover useful features in them.

As presented in Péter Almási's et al approach, the frames coming from the environment pass through an image preprocessing process. Preprocessing involves the following steps:

1. Resizing : While the original DuckieTown environment's states are frames with size 480×640 (height $\times$ width), in order to reduce inference time of the models the environment is tweaked to return frames with size 60×80.
2. Cropping : The part of the image that does not contain useful information is cropped. The remaining image has a size of 40×80 and contains everything bellow the horizon.
3. Normalization : Pixel RGB values are scaled in the range of [0,1] for better training results.

Contrary to Péter Almási's et al approach, no color segmentation is performed.

Given that the model will operate in stacked frames as described above, the final input to the model has a shape of $15 \times 40 \times 80$. In order to process the stacked frames, a convolutional core is used that it is shared between the actor and the critic heads. The convolutional core's output is also detached when passed to the critic's head input so that its parameters are tuned based only on actor's output. Convolutional core's architecture is the same as Péter Almási's et al approach [29]. Complete architecture of both actor and critic are illustrated in figure 2.10. In total, actor and critic together contain 4,558,436 trainable parameters. The model is defined in the following code section, in a class that implements the necessary methods for the model object to be plugged in the deep reinforcement learning framework's Stacked Frame PPO algorithm.

```
import torch
import torch.nn as nn
from torchvision import transforms as T
from torch.nn import functional as F

class CNNActorCritic(nn.Module):

    def __init__(self, n_output, device):

        super(CNNActorCritic, self).__init__()

        self.device = device
        self.transform = T.Compose([
                    T.Normalize(mean=[0]*15, std=[255]*15),   # Turn input to 0-1 range from 0-255
                    ])
```

```
        self.conv_core = nn.Sequential(
            nn.Conv2d(15, 32, 3),
            nn.LeakyReLU(),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(32, 32, 3),
            nn.LeakyReLU(),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(32, 64, 3),
            nn.LeakyReLU(),
            nn.MaxPool2d(2, 2),
            nn.Flatten(),
        ).to(self.device)

        self.actor_head = nn.Sequential(
            nn.Linear(1536,1024),
            nn.LeakyReLU(),
            nn.Linear(1024,512),
            nn.LeakyReLU(),
            nn.Linear(512,256),
            nn.LeakyReLU(),
            nn.Linear(256,128),
            nn.LeakyReLU(),
            nn.Linear(128, n_output)
        ).to(device=self.device)

        self.critic_head = nn.Sequential(
            nn.Linear(1536,1024),
            nn.LeakyReLU(),
            nn.Linear(1024,512),
            nn.LeakyReLU(),
            nn.Linear(512,256),
            nn.LeakyReLU(),
            nn.Linear(256,128),
            nn.LeakyReLU(),
            nn.Linear(128, 1)
        ).to(device=self.device)

    def forward(self, x):
        x = torch.permute(x, (0, 3, 1, 2))  # Place channel axis in correct position
        x = self.transform(x)               # Apply transform
        x = T.functional.crop(x, top=20, left=0, height=40, width=80)
        x = x.to(device=self.device)
        visual_repr = self.conv_core(x).squeeze(-1).squeeze(-1)   # Calculate ResNet output
        return F.log_softmax(self.actor_head(visual_repr), dim=1).to('cpu'),
        ↪  self.critic_head(visual_repr.detach()).to('cpu') # Calculate policy probs and value

    def infer_step(self, x):
        action_probs, value = self.forward(x)
        dist = torch.distributions.Categorical(logits=action_probs)
        action = dist.sample().item()
        return dist, action, value

    def infer_batch(self, x):
        action_probs, value = self.forward(x)
        dist = torch.distributions.Categorical(logits=action_probs)
```

```
        return dist, value

    def infer_action(self, x):
        dist, _ = self.forward(x)
        dist = torch.distributions.Categorical(logits=dist)
        return dist.sample().cpu().numpy()[0]
```

As it can be seen, scaling and cropping of the original input frames are performed during the forward pass of the model.

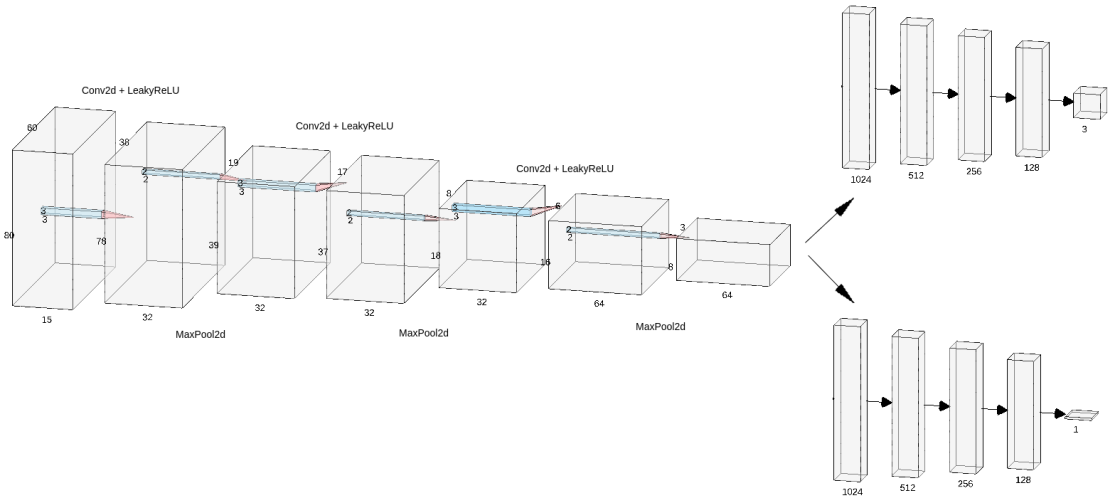

Figure 2.10: Model architecture.

Training precedure

Having Stacked Frame PPO implemented, preprocessing steps included in model's forward pass and the model architecture defined, we initiate the training of an agent on DuckieTown. The map is the empty loop and the goal of the agent is simply to learn to drive the vehicle properly by following the proper lane. In empty loop there are no pedestrians or other vehicles that the agent can collide with. Hyperparameters chosen for the Stacked Frame PPO algorithm for the training procedure are presented in table 2.7. Every training loop was run on a single AMD Ryzen 7 3700X 8-Core CPU and no hardware accelerator was used. On the contrary, Péter Almási's et al approach claims to have used an NVIDIA DGX Workstation, which contains 4 pieces of V100 GPUs.

Parameter	Value	Parameter	Value
lr	2×10^{-5}	gamma	0.99
batch_size	8	lamb	1
epochs	3	actor_weight	1
max_frames	800,000	critic_weight	0.5
num_steps	500	entropy_weight	0.01
clip_param	0.2	stacked_frames	5

Table 2.7: Selected hyperparameters used for PPO algorithm on DuckieTown environment.

While training, evaluation is performed every 5,000 frames by calculating both the average played frames and the average cumulative reward of 5 different episodes. Compared to approaches used for CartPole environment, the number of seen frames after which an evaluation if performed is increased from 1,000 to

5,000 and the number of episodes to average is reduced from 10 to 5 in order to decrease the training time. Furthermore, for the training procedure, early stopping is enabled with a reward threshold of 30,000 and a frame threshold of 350,000. Given that the environment itself is set up with a maximum number of frames of 3,500, what will kick in first and terminate the training procedure is the reward threshold. In figure 2.11, average played frames and average cumulative rewards during the training procedure are illustrated in the corresponding plots.

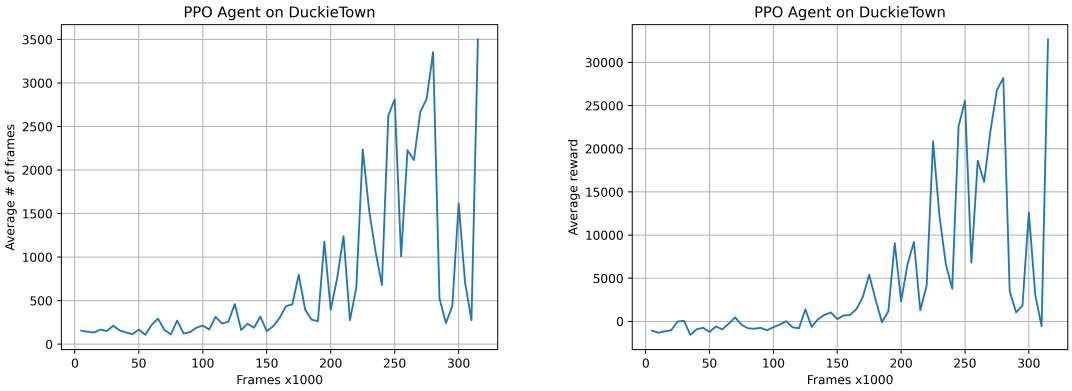

Figure 2.11: Average played frames (left) and average cumulative rewards (right) during PPO agent training.

As it can be seen, the agent hits the threshold of an average cumulative reward of 30,000 and the training process is terminated. The trained model is saved for later use and fine tuning. It is clear that the agent manages to learn proper behavior since the cumulative reward has increased considerable and the average played frames have reached 3,500, a number that an agent performing random actions would never manage to achieve. The achieved average cumulative reward of approximately 32,000 that is finally reached, is managed after the agent sees 315,000 frames and the whole process takes approximately 3 hours and 15 minutes. The final model is tested visually in order to confirm that it is capable of properly driving the vehicle. For obvious reasons, a video of the agent actually driving the vehicle cannot be presented but instead, the model itself is stored in the present thesis' repository along with the code for recreating the results.

Since the reward threshold that was used was selected arbitrarily and based on what Péter Almási's et al paper has described, training was performed again, this time with the reward threshold set to 50,000. This was done in order to check whether the model could perform even better with further training or it reached the cumulative reward limit for the environment and for the given number of max episode length of 3,500.

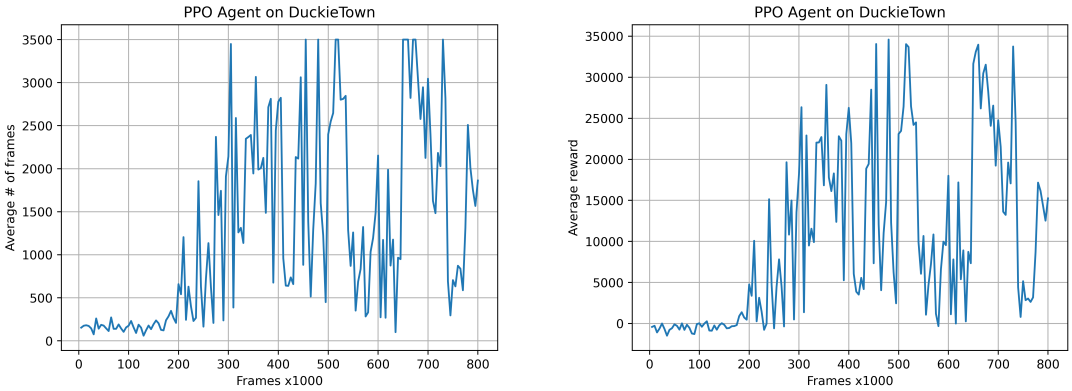

Figure 2.12: Average played frames (left) and average cumulative rewards (right) during PPO agent training (increased reward threshold).

In figure 2.12, the progress of the training procedure is illustrated again. The maximum cumulative reward was observed at frame 480,000 reaching approximately 35,000 which is a little more than what was achieved previously. The overall training procedure presents instabilities with various catastrophic forgetting events dropping the cumulative reward which is then increased again. The training procedure terminates after a preset number of frames (800,000) and it lasts approximately 11 hours. As before, the best model was saved and evaluated visually, exhibiting similar behavior as the previous model.

Apart from training results, it is really worthy to compare the observed training duration with Péter Almási's et al approach. As already stated, training on 315,000 frames lasts approximately 3 hours and 15 minutes without the need of any hardware acceleration. In [29], it is stated that training on 500,000 frames using 4 V100 GPUs lasts approximately 40 hours. Assuming that the implementation of the DQN algorithm that has been used is proper, this duration difference even when using dedicated hardware accelerators signifies the superiority of Proximal Policy Optimization algorithm compared to DQN.

Chapter 3

Accelerating Deep Learning Models

3.1 Presentation of the Problem

As we have already mentioned in chapter 1, neural networks can act as universal function approximators [4, 15]. For a neural network to be able to approximate a latent function, it has to contain enough trainable parameters. As technology progresses and computation power increases, researchers are able to tackle harder problems using larger neural networks. From trying to identify handwritten digits using simple feed-forward networks to natural language processing tasks like explaining jokes using 540-billion parameter transformer models [30], neural networks have proven their capability to excel at various tasks.

While larger architectures are able to solve increasingly more difficult problems, the same increase of trainable parameters creates new problems. As previously described, each neural network layer can be described via a matrix multiplication. Therefore, larger models lead to more matrix multiplications with bigger matrices. This increase in complexity can lead to inference and training that are both power-hungry and slow.

In chapter 2, we successfully trained a neural network to be able to drive a vehicle within an environment that provided a simple simulation of city roads. Our model was nowhere near state of the art models used in NLP tasks containing billion of parameters. It contains approximately 2.3 million trainable parameters (just the actor) and yet the training process is certainly not an easy task for a basic computer requiring almost 3 hours. Having the final network and its parameters we can use them in an actual robot that can actually move autonomously in a real-world recreation of the simulated environment. It is common sense, that the task of driving is a latency-critical problem. Decisions must be executed instantly since the vehicle is moving. For example, lets suppose that we have a moving car traveling with a speed of 100 km/hour. If this car is an autonomous car and a neural network decides about each action the car performs, a 0.2s latency of the system to perform the calculations that are done by the neural network and decide for the next action means that the car has moved approximately 5.5 meters without supervision. This distance is important and can be proven lethal in various scenarios.

Although in our case the neural network is not that complex and the inference duration is less than 0.2s and the moving robot certainly moves with a speed of less than 100 km/hour, it is an nice test case for trying to reduce the inference time of the neural network in order to minimize the latency. In the presented chapter, we well describe every technology used, both software and hardware along with the various steps that are made in order accelerate the inference of the already trained neural network.

3.2 The ONNX Format

ONNX format stands for Open Neural Network Exchange format and it is an open format built to represent machine learning models. ONNX defines a common set of operators - the building blocks of machine learning and deep learning models - and a common file format to enable AI developers to use models with a variety of frameworks, tools, runtimes, and compilers [31]. ONNX was originally named Toffee and was developed by the PyTorch team at Facebook. In September 2017 it was renamed to ONNX and announced by Facebook

and Microsoft. Later, IBM, Huawei, Intel, AMD, Arm and Qualcomm announced support for the initiative.

The goal of creating a global format for neural network models after defining a common set of operators is to enable developers to easily move between frameworks at different stages of the development process. Furthermore, it allows hardware vendors and others to improve the performance of artificial neural networks of multiple frameworks at once by targeting the ONNX representation.

3.2.1 Converting models to ONNX format

In order to leverage ONNX format's flexibility to work easily on different hardware and calculate the speedup each device offers we will start by converting our currently trained model to ONNX format. Since during inference time every decision is taken from the actor head of the actor-critic neural network, we begin by isolating the actor in order to perform only the useful computations. The actor model consists of the convolutional core of the initial actor-critic model which is responsible for extracting a useful representation for the given stacked frames, and the actor head that contains four different feed-forward layers.

After isolating the actor, *torch.onnx* module is used in order to convert the Pytorch model into ONNX format. This module is provided by Pytorch library and is eases the conversion of any Pytorch model. In the following code section, the conversion step is presented:

```
torch.onnx.export(model,                    # model being run
                  dummy_input,              # model input (or a tuple for multiple inputs)
                  "../models/actor.onnx",   # where to save the model (can be a file or file-like object)
                  export_params=True,       # store the trained parameter weights inside the model file
                  opset_version=11,         # the ONNX version to export the model to
                  do_constant_folding=True, # whether to execute constant folding for optimization
                  input_names = ['input'],  # the model's input names
                  output_names = ['output'], # the model's output names
                  dynamic_axes={'input' : {0 : 'batch_size'},    # variable length axes
                                'output' : {0 : 'batch_size'}})
```

The original Pytorch model is stored in the *model* variable and *dummy_input* contains a dummy input so that *torch.onnx* module can trace the graph that represents the neural network. Dynamic axes are the dimension of the input and the output that can vary, in our case it is the dimension of the batch size which, in general, can have an arbitrary value. After running the piece of code presented above, the initial Pytorch model is stored in the actor.onnx file and can be later loaded into any ONNX inference session.

An ONNX inference session is responsible for loading the appropriate model and then perform inference given numpy arrays as inputs. The following code section contains the definition of the ONNXActor model that leverages an ONNX inference session in order to perform the forward propagation of the neural network.

```
import torch, onnxruntime, numpy as np

class ONNXActor():
    # Implements actor using ONNX runtime

    def __init__(self, onnx_path, providers):
        # Initiliaze model
        self.ort_session = onnxruntime.InferenceSession(onnx_path, providers=providers)

    def forward(self, x):
        # Implements forward pass of model
        output = self.ort_session.run(None, {'input' : x.numpy().astype(np.float32)})[0]
        return torch.Tensor(output)

    def infer_action(self, x):
        # Utilizes torch distributions to return an action
```

```
        dist_probs = self.forward(x)
        dist = torch.distributions.Categorical(logits=dist_probs)
        return dist.sample().numpy()[0]
```

As it can be seen, all the computations performed in the forward pass are now handled by the ONNX runtime session which is initiated when an ONNXActor object is initialized. The *providers* variable that is passed to the inference session specifies the hardware that the computations will be performed on. ONNX supports various platforms, like CUDA and TensorRT for accelerating the inference. We will further analyze the performance achieved by different providers and the coming paragraphs.

3.3 The Embedded Devices

An embedded system is a computer system that has a dedicated function within a larger mechanical or electronic system. It is *embedded* as part of a complete device often including electrical or electronic hardware and mechanical parts. Because an embedded system typically controls physical operations of the machine that it is embedded within, it often has real-time computing constraints.

In our case, since the scenario of our problem is to deploy the trained neural network on an actual robot which will try to navigate through a real-world DuckieTown, we need such a computer system to perform the necessary computations efficiently and finally move the robot properly through the environment.

In the following paragraphs, the process of deploying the neural network on two different embedded devices will be described along with the performance boost the is achieved via each devices hardware accelerator. The first device that will be examined is NVIDIA's Jetsion Xavier NX and the second device will be Xilinx' Zynq UltraScale+ MPSoC ZCU104. Each device's hardware accelerators will be leveraged in order to decrease the inference time of the deployed neural network and achieve low-latency autonomous driving.

3.3.1 Jetson Xavier NX

The NVIDIA Jetson Xavier NX Developer Kit constitutes a development board that includes a Jetson Xavier NX module at its heart. The Jetson Xavier NX module is a small system on module with dimensions of 70x45mm that manages to deliver 21 TOPS (at 15W or 20W) or up to 14 TOPS (at 10W). It can run multiple modern neural networks in parallel and process data from multiple high-resolution sensors—a requirement for full AI systems [6].

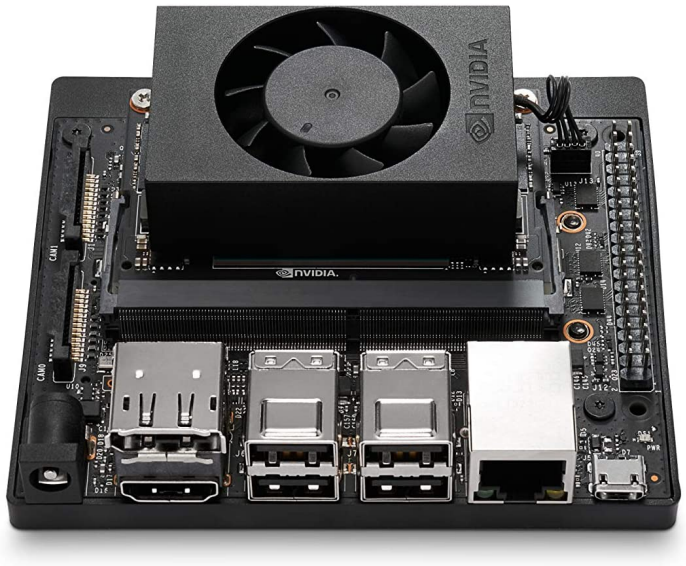

Figure 3.1: The Jetson Xavier NX developer kit [6].

The Jetson Xavier NX module contains a 6-core NVIDIA Carmel ARM v8.2 64-bit CPU with 6MB L2 and 4MB L3 cache. Its accelerator is a 384-core NVIDIA Volta GPU with 48 Tensor Cores. Finally it contains 8 GB of 128-bit LPDDR4x memory. The developer kit contains USB ports, ethernet port, HDMI and other modules enabling the communication with peripherals. In figure 3.1 the Jetson Xavier NX developer kit is presented and in figure 3.2 we can see the Jetson Xavier NX module that lies in the heart of the developer kit [6].

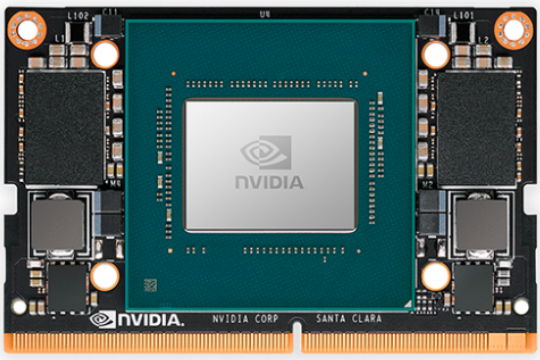

Figure 3.2: The Jetson Xavier NX module [6].

3.3.2 Xilinx Zynq UltraScale+ MPSoC ZCU104

The Xilinx Zynq UltraScale+ MPSoC ZCU104 evaluation kit enables designers to jumpstart designs for embedded vision applications such as surveillance, Advanced Driver Assisted Systems (ADAS), machine vision, Augmented Reality (AR), drones and medical imaging. This kit features a Zynq UltraScale+ MPSoC EV device with video codec and supports many common peripherals and interfaces for embedded vision use case. The included ZU7EV device is equipped with a quad-core ARM Cortex-A53 applications processor, dual-core Cortex-R5 real-time processor, Mali-400 MP2 graphics processing unit, 4KP60 capable H.264/H.265 video codec, and 16nm FinFET+ programmable logic [32].

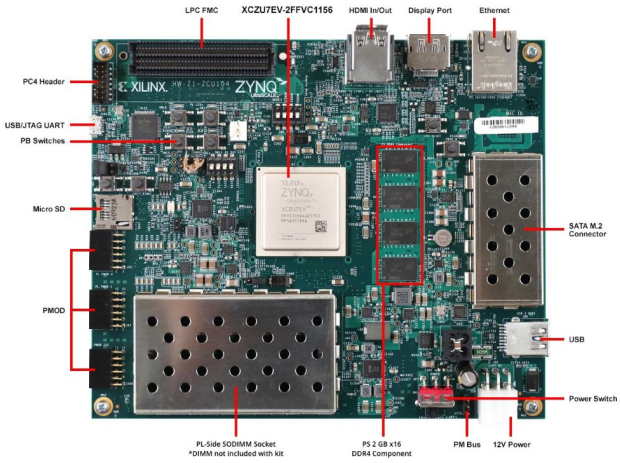

Figure 3.3: The ZCU104 evaluation kit.

3.4 Accelerating Inference: The Case of a Single Agent

3.4.1 Jetson Xavier NX

For the acceleration of the forward pass of the actor model using Jetson Xavier NX, different configurations will be tested. As a baseline, we will use the average inference time of the actor model using plain Pytorch code run on top of Xavier's CPU. The rest of the configurations that will be tested are presented in table 3.1.

Framework	Hardware	Data type
Pytorch	CPU	fp32
Pytorch	GPU	fp32
Pytorch	GPU	fp16
ONNX	CPU	fp32
ONNX	GPU	fp32
ONNX	Tensrorrt	fp32
ONNX	CPU	int8
ONNX	GPU	int8
ONNX	Tensorrt	int8

Table 3.1: Inference configurations for NVIDIA Xavier NX.

As far as different data types, used for representing the model's parameters, are concerned, both Pytorch and ONNX provide tools for easily casting the initial *fp32* model to *fp16* and *int8*. For example, using Pytorch one can easily convert a model to *fp16* with just one line of code by using the *half()* method which returns the initial model converted to *fp16*. On the other hand, ONNX provides a quantization function that given an initial *fp32* ONNX file describing the model, it saves a different file describing the same model cast to *int8*. These tools make it easier for machine learning engineers to discover the best configuration without having to dive deeper to more low level hardware optimization every time a model needs to be placed into production. In the following code section, an overview of how to cast a model to different datatype for each framework is presented:

```
import torch
from onnxruntime.quantization import quantize_dynamic
from model import Actor

if __name__ == "__main__":

    # Cast Pytorch model to fp16
    model = Actor('cuda')
    model.load_state_dict(state_dict=torch.load('models/actor_state_dict.pt'))
    model = model.half()

    # Cast ONNX model to int8
    quantize_dynamic('models/actor.onnx', 'models/actor_8bit_dynamic.onnx')
```

Of course, while *fp16* and *int8* formats can decrease the inference time, they introduce errors to the final output of the model for obvious reasons. This makes it necessary for us to extract some statistics for these errors and find out if they lie withing a tolerable range. Overall, in table 3.2 the error introuced at the output of the model is presented. Since the output of the actor is a three element vector which contains the probabilities of each of the possible actions with a logarithmic function having being applied on them, we apply a exponential function to the output of the model to negate the logarithm and make the error calculation more interpretable. This way, we are sure the the output of the model is the actual probability and not the log-ed probability.

Framework	Hardware	Data type	Mean absolute error	Max absolute error
Pytorch	CPU	fp32	-	-
Pytorch	GPU	fp32	-	-
Pytorch	GPU	fp16	0.00010325558	0.00045317410
ONNX	CPU	fp32	2.400577e-08	1.7881393e-07
ONNX	GPU	fp32	2.400577e-08	1.7881393e-07
ONNX	CPU	int8	0.00051964243	0.00328561660
ONNX	GPU	int8	0.00051964243	0.00328561660

Table 3.2: Mean and max absolute error of every configuration for NVIDIA Xavier NX.

The results presented in the table clearly indicate the lowering the precision increases the error and this is something that was expected. Furthermore, the conversion to ONNX format introduces some error even when the *fp32* data type is used but is clearly neglectable. Overall the maximum absolute error is observed when *int8* data type is used with the ONNX format and it is approximately a 0.32% difference of the initial probability that the model should output. This amount of error can still be considered neglectable.

As far as the inference time is concerned, we will use the Pytorch *fp32* model run on Xavier's CPU as the baseline for the speedup that each other configuration provides. By default, Pytorch utilizes every core of Xavier's CPU meaning that our baseline already utilizes some kind of acceleration. The average inference time achieved for a single input is 10.509635ms. We have to underline again that we do not utilize batch sizes greater that one due to the nature of the agent that has to infer the previous action in order to receive the next state that will be fed into its input. In table 3.3 we present the average inference time achieved for each configuration. The results are graphically presented in the bar plot of figure 3.5. Furthermore, in figure 3.4, a box plot for all the configuration inferences is presented to graphically depict the variability of the inference duration for specific configurations.

As it can be seen, the best average inference time is achieved when using ONNX framework and the *fp32* data type run on top of Xavier's GPU. This configuration manages to reduce the average inference time to 2.369108ms. It is essential to note that ONNX framework provides acceleration even when Xavier's CPU is utilized. This is due to graph optimizations that the ONNX runtime provides [33].

An interesting observation regards the *int8* data type. One would expect that lowering the precision would lead to more efficient computations but the results suggest that this is not the case. Every configuration that utilizes *int8* seems to perform worse than the same configuration utilizing the *fp32* datatype. Furthermore, GPU seems to really struggle when performing the computations using *int8*. A possible explanation for the first obeservation is the following: The ONNX conversion to *int8* is dynamic. This mean that at inference time depending on the input and the output of each layer a scaling parameter is calculated. The scaling parameter Q is the difference between two consecutive values a matrix element can have. After performing the computations using *int8* datatype, each layer has to fall back to *fp32* format for the dynamic quantization process to repeat. All this process can add an overhead that is not worth it due to the limited size of our model. In case a larger model was used, the overhead would be miniscule compared to the time that would be required to calculate the outputs of larger and more complex layers. Regarding the GPU's inability to perform better than CPU when *int8* is used, this is due to missing kernels that are required for integer calculations to be performed.

Framework	Hardware	Data type	Average inference time (ms)	Frames per second (FPS)
Pytorch	CPU	fp32	10.509635	95.150787
Pytorch	GPU	fp32	4.199729	238.110619
Pytorch	GPU	fp16	3.112479	321.287308
ONNX	CPU	fp32	3.026185	330.449033
ONNX	GPU	fp32	2.369108	422.099741
ONNX	CPU	int8	3.905829	256.027594
ONNX	GPU	int8	6.276672	159.320089

Table 3.3: Average inference time & FPS achieved with NVIDIA Xavier NX.

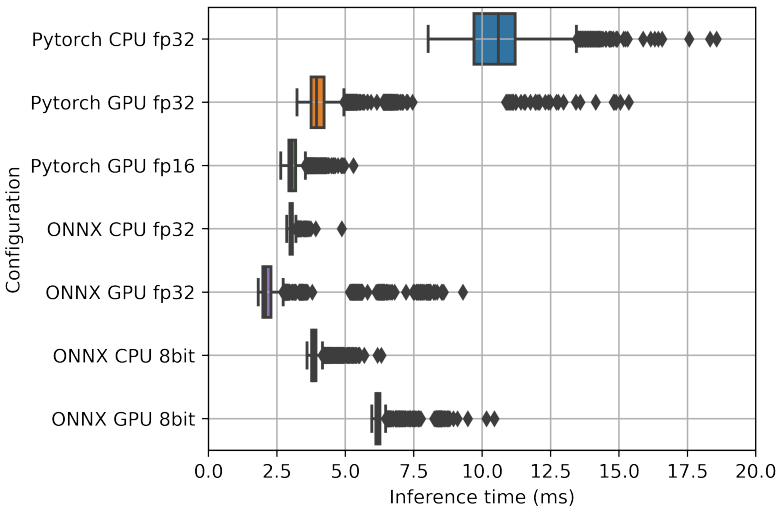

Figure 3.4: Boxplot for NVIDIA Xavier NX inference.

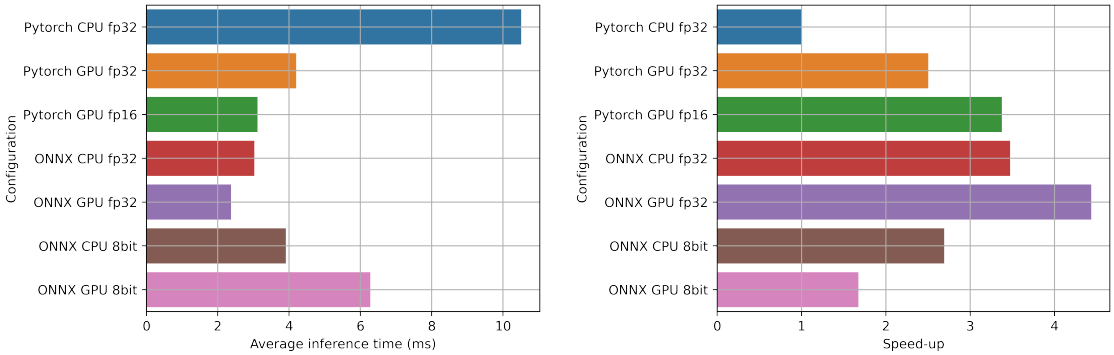

Figure 3.5: Average inference time and speedup for NVIDIA Xavier NX.

3.4.2 Xilinx Zynq UltraScale+ MPSoC ZCU104

For the acceleration of the forward pass of the actor using Xilinx Zynq UltraScale+ MPSoC ZCU104, four configurations will be evaluated. The first and the simplest one is the classic Pytorch code run on top of the CPU that the board provides. This configuration will serve us a baseline to compare with the rest of the configurations. The second and third configurations are again code run on top of the CPU but this time ONNX is leveraged for *fp32* and *int8* data types. Finally, the fourth configuration utilizes the board's programmable logic. Using the FPGA of the board, a DPU (Deep Learning Processor Unit) will be leveraged to accelerate the model's inference. The Xilinx Deep Learning Processor Unit is a programmable engine dedicated for convolutional neural network. The unit contains register configure module, data controller module, and convolution computing module.

For proper configuration of the programmable logic, Xilinx provides Vitis AI. Vitis AI is a comprehensive AI inference development platform for Xilinx devices, boards, and Alveo data center acceleration cards. It consists of a rich set of AI models, optimized deep-learning processor unit (DPU) cores, tools, libraries, and example designs for AI on edge and data center ends. It is designed with high efficiency and ease of use in mind, unleashing the full potential of AI acceleration on Xilinx FPGAs and adaptive SoCs [7].

Vitis AI offers tools for optimizing, quantizing and compiling neural network models into DPU instructions. Vitis AI optimizer (figure 3.6) is responsible for pruning a neural network resulting in models having less

Framework	Hardware	Data type
Pytorch	CPU	fp32
ONNX	CPU	fp32
ONNX	CPU	int
Vitis AI	DPU+CPU	int8+fp32

Table 3.4: Inference configurations for ZCU104.

parameters which are therefore able to achieve faster inference. The quantizer (figure 3.7) is then responsible to perform quantization of a model's parameters. Since the initial parameters are stored in *fp32*, the optimizer converts them to fixed point datatype like *int8*. Fixed point representation consumes less memory, achieves faster inference and more energy efficient computations. The Vitis AI compiler (figure 3.8) maps the AI model to a high-efficient instruction set and data flow. It also performs sophisticated optimizations such as layer fusion, instruction scheduling, and reuses on-chip memory as much as possible. Finally, Vitis AI also provides high level libraries and APIs for efficient inference using DPU cores. The complete schematic illustrating each layer of abstraction is presented in figure 3.9.

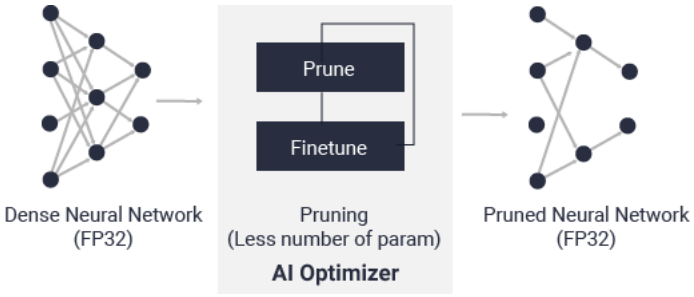

Figure 3.6: Vitis AI optimizer [7].

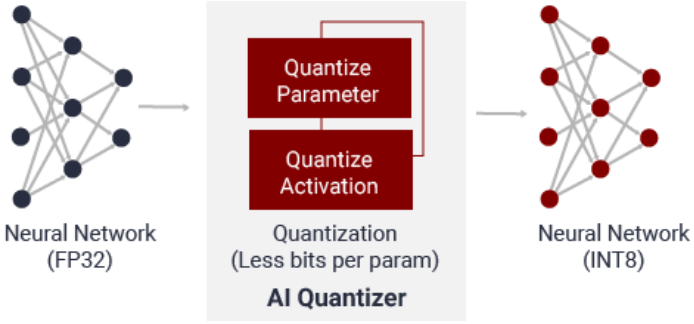

Figure 3.7: Vitis AI quantizer [7].

For our case, we will make use of each tools in order to produce the desired *.xmodel* file which contains the set of the instructions for the DPUs of our device. The process of converting the initial trained Pytorch model to the *.xmodel* is presented in the appendix and in the Github repository of the present thesis.

Having obtained the *.xmodel* file, we are able to print the graph that describes the operations that take place inside the model. In figure 3.12 the corresponding graph is illustrated. We can see that the graph is split into four main sections. Sections marked with blue color are intended to be run on DPU. Sections marked with red color are intended to be run on CPU. Each section corresponds to the following operations:

- The first section describes the pre-processing that the model performs. This includes scaling and slicing the initial input frame. This subgraph is intended to be run on CPU.

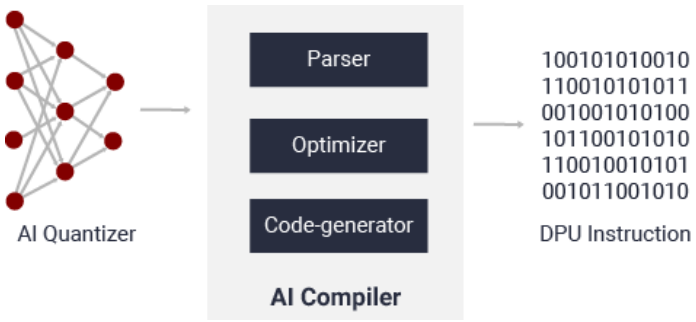

Figure 3.8: Vitis AI compiler [7].

- The second section describes the convolutional part of the model. It includes all the convolutional and max pooling layers. This subgraph is intended to be run on DPU.
- The third part of the model describes the fully connected part of the model. As we can see, fully connected layers are intended to be run on CPU.
- The fourth part of the model describes the model's last fully connected layer. Vitis AI compiler automatically transforms fully connected layers into convolutional layers whenever this is possible.

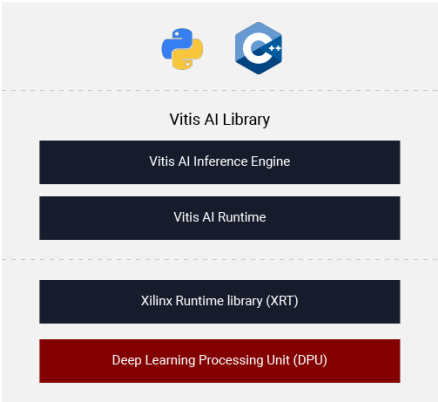

Figure 3.9: Vitis AI library [7].

Further investigating the documentation of Vitis AI, it is stated the fully connected layers are not supported in the framework. The compiler tries to convert them to equivalent convolutional layers whenever this is possible and the layers that are not converted are executed in CPU. For this reason, Vitis AI compiler is unable to create a unified graph for the whole model to be run on DPU. In order to proceed with using the DPUs, a hybrid model will be created that will consist of two parts. The first part is the convolutional part and it will run on the programmable logic's DPUs, the second part is the fully connected part and it will run on the CPU. In the code section of appendix B.5, we present the implementation of the hybrid model that inputs the data into the DPU accelerated convolutional layers and the DPU outputs are later fed into the fully connected layers. For the fully connected part, Numpy library is used and the layer's weights are properly loaded from *.npy* files. We also have to note that the preprocessing step that regards scaling and slicing, as well as conversion of the float input to *int8* input that is handled by the GPUs is implemented in the *preprocess_fn()* method. The full application code that is run on ZC104 device is presented in the appendix and in the thesis' Github repository.

Having implemented our hybrid model, we are ready to calculate quantization errors and inference speed for the four configurations mentioned above. In table 3.5 the introduced errors are calculated for the output of the models for the various configurations. At a first glance we see that the DPU configuration seems to

introduce really serious inaccuracies due to the static *int8* quantization taking place. On the other hand, dynamic quantization used by ONNX framework does not have the same problem.

Framework	Hardware	Data type	Mean absolute error	Max absolute error
Pytorch	CPU	fp32	-	-
ONNX	CPU	fp32	1.323844e-08	1.1920929e-07
ONNX	CPU	int8	0.00050873094	0.002666235
Vitis AI	DPU+CPU	int8+fp32	0.11339324799	0.517653040

Table 3.5: Mean and max absolute error of every configuration for ZCU104.

Regarding the inference duration, in table 3.6 the average inference duration and the FPS achieved by each configuration on ZCU104 is presented. Figure 3.10 illustrates a boxplot for the inference duration where we can see the deviation in inference time that some configurations introduce. Figure 3.11 illustrates again the average inference time as well as the speedup that each configuration achieves.

Framework	Hardware	Data type	Average inference time (ms)	Frames per second (FPS)
Pytorch	CPU	fp32	31.096884	32.157563
ONNX	CPU	fp32	11.864230	84.286971
ONNX	CPU	int8	10.761447	92.924304
Vitis AI	DPU+CPU	int8+fp32	32.210083	31.046179

Table 3.6: Average inference time & FPS achieved with ZCU104.

As it can be seen, the most promising configuration is the ONNX *int8*. The Vitis AI approach seems to really underperform as its average inference time is greater than the baseline's. The reason this is happening is straightforward. As already mentioned above, DPUs handle only the convolutional section of the neural network and fully connected layers are handled by the CPU. If we try to count the number of parameters appearing in each part of the network we end up with 32096 parameters for the convolutional part and with 2263299 parameters in the fully connect part, the majority of which is in the first fully connected layer ($1536 \times 1024 + 1024$ parameters). That said, we can see that the DPU only accelerates a small fraction of the total computations without seriously affecting the overall inference time. The small increase in the inference time can be explained due to the overhead for transferring data to the DPU and from the DPU.

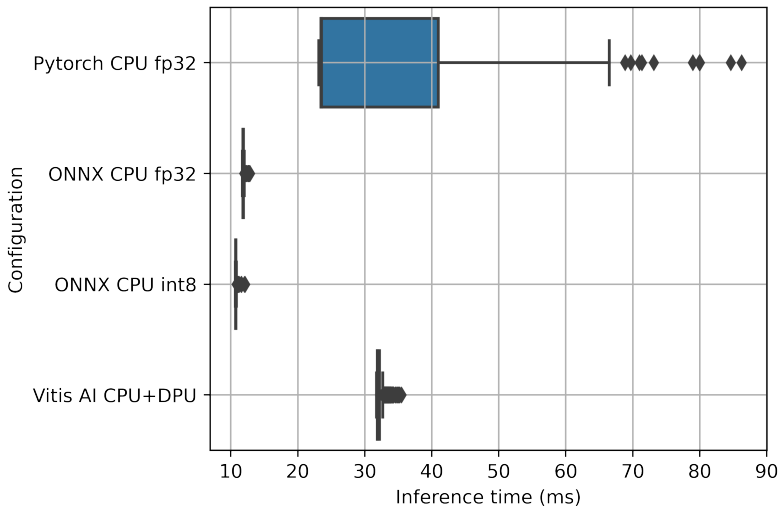

Figure 3.10: Boxplot for ZCU104 inference.

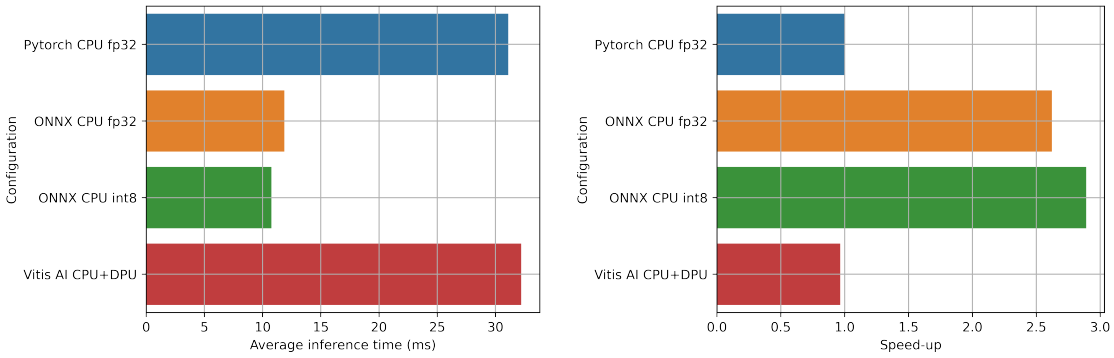

Figure 3.11: Average inference time and speedup for ZCU104.

Regarding the ONNX *int8* configuration, this time it manages to perform better than its *fp32* counterpart, something that did not happen when we performed the computations on NVIDIA Jetson NX. As we already stated, the dynamic quantization that is performed can add an unnecessary overhead that can lead to slower inference time. This time, the cortex A53 CPU that the ZCU104 uses is less efficient than the NVIDIA Carmel ARMv8.2 CPU of the Xavier NX and matrix multiplications are complex enough for the overhead the dynamic quantization introduces to be worth it. Similar behavior would be expected if we tried to benchmark a more complex model with greater number of parameters.

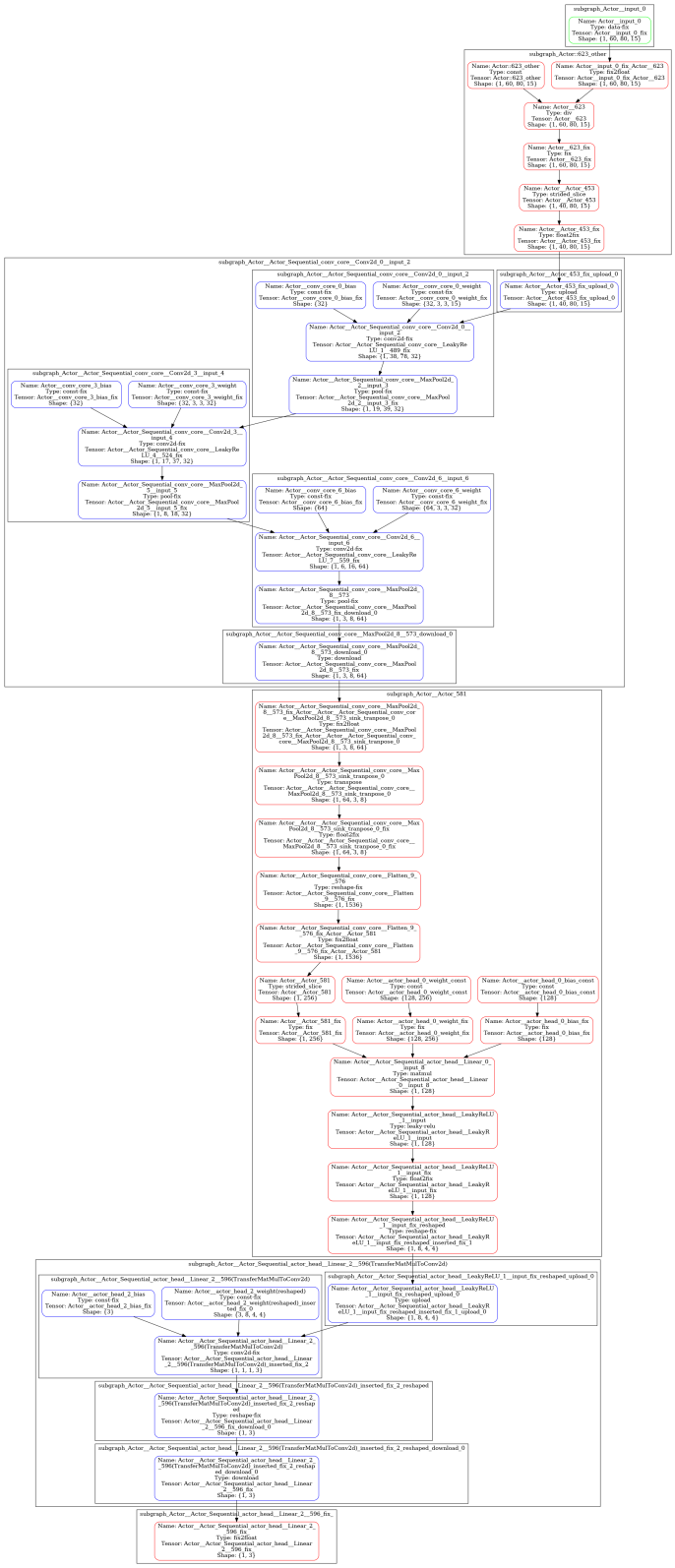

Figure 3.12: Graph representing the produced *.xmodel* file.

3.5 Accelerating Inference: The Case of Multiple Agents

3.5.1 New Scenario

In the previous section we investigated how to speed up the inference of a single agent which is deployed in an environment. As we stated, in inference time the agent takes in a single state produced by the environment and outputs a single action which is fed back to environment in order to produce the next state. This means that the batch size that it is used in inference time is strictly fixed to one as there is only one state under process from the agent.

This limitation results in low utilization of hardware accelerators, especially GPUs which can take advantage of batch sizes greater than one and perform efficient, parallel computations across the batch dimension. For this reason, we can leverage hardware accelerators to perform inference for more than one robots which can be guided from the same neural network. This idea leads to the following scenario:

Let's suppose that we need to deploy multiple self guided vehicles using a trained neural network like the one that we currently used in the previous experiments. Those guided vehicles do not have the computational power to perform inference on their own, so the will transmit their state, as their sensors receive it, to a central device that will perform the inference for them. The central device, as it manipulates multiple vehicles, it gathers different states and performs inference using a batch size greater than one and equal to the number of vehicles under its responsibility. For the shake of the scenario, we will assume that each vehicle needs to perform and action for more than 100 states per second, so the FPS that each vehicle must operate is greater than 100. Certainly, inferring a batch of size greater than one is more time consuming although the average inference time of each state in the batch might be lower. This means that if more vehicles are under the central device's supervision, the FPS for each vehicle will be lower. That said, will try to discover which is the maximum number of vehicles that each hardware accelerator and each configuration can handle for the vehicles under central device's supervision to experience minimum FPS of 100. In figure 3.13, the scenario described above is illustrated. As it can be seen, the central device collects states from each vehicle, performs batch computation, and transmits back the proper actions each vehicle should perform.

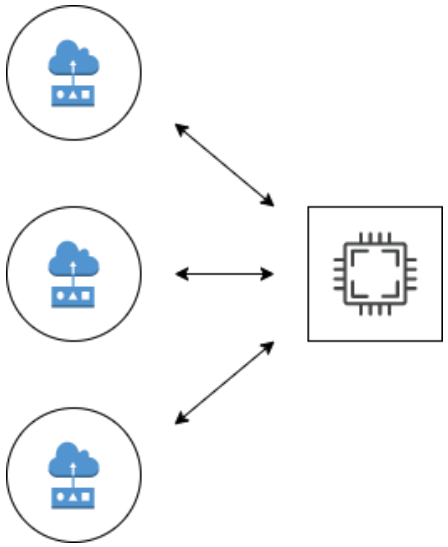

Figure 3.13: Multiple agents scenario, controlled by one neural network in the central device.

3.5.2 Jetson Xavier NX

As stated in the previous paragraph, increasing the batch size utilizes a hardware accelerator more efficiently. Specifically for Jetson Xavier NX, in figure 3.14 we can how the inference time per batch is increasing while we increase the batch size. As we can see, doubling the batch size does not result in double inference time per batch. This means that on average, the computations for each state inside a batch are performed faster. This conclusion can be validated by figure 3.16, where we can see clearly that while the batch size increases, the average inference time per state is lower.

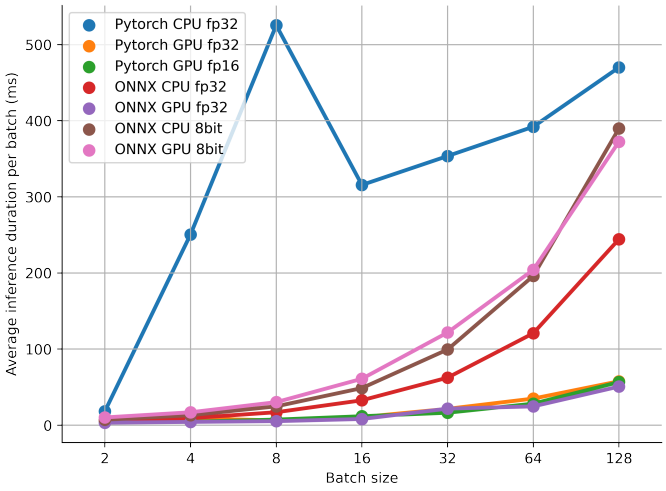

Figure 3.14: Average inference time per batch for batch computations.

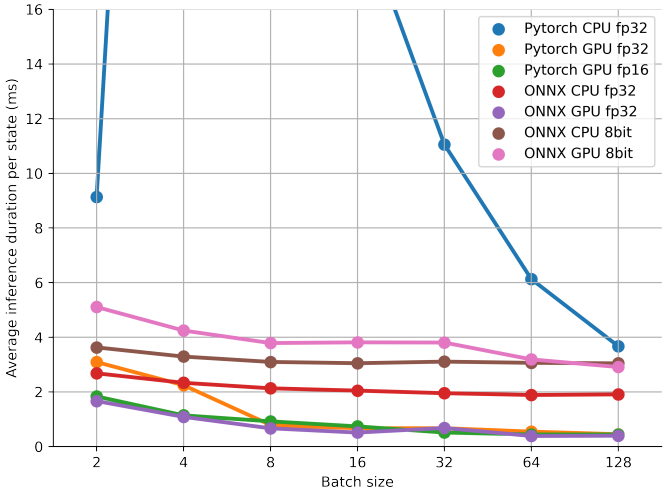

Figure 3.15: Average inference time per state for batch computations.

For testing how many devices each configuration used on Jetson Xavier NX can support, we need to plot the FPS each configuration achieves for different batch sizes. It is important to note that to calculate the FPS we use the following formula:

$$FPS = \frac{1000}{\text{Batch Inference Duration (ms)}}$$

The reason that we use the batch inference duration and not the average inference duration per state is that each agent will have to wait for the output for the entire batch to be computed in order to get its reply from the central device. The results for this investigation are illustrated in figure 3.16. As we can see, the configuration that supports the most is ONNX with *fp32* data type run ton top of GPU. The configuration with the worst performance is Pytorch with *fp32* data type run on top of NX's CPU. As expected, every configuration that utilizes the CPU for the computation to take place is able to support way less agents in order to achieve a minimum FPS of 100.

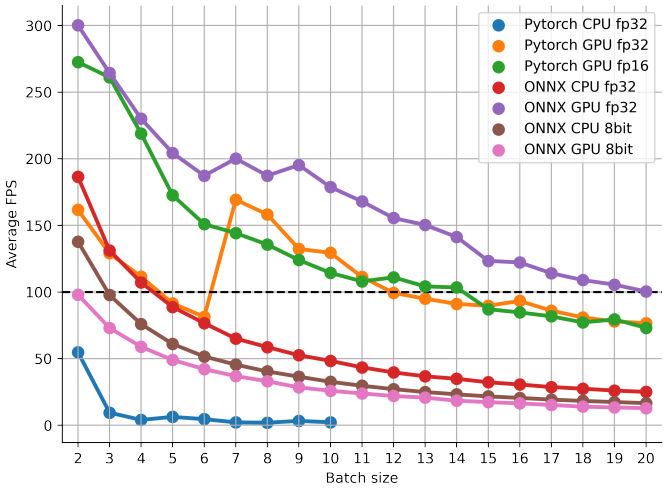

Figure 3.16: Average FPS for batch computations.

Epilogue

Ah the first chapter of the current thesis, we presented the theoretical background regarding reinforcement learning, neural networks and how they can be blended together. As next step, we presented four different deep reinforcement learning algorithms, Deep Q Learning, REINFORCE, Asynchronous Actor Critic and Proximal Policy Optimization.

Using these deep reinforcement learning algorithms, we moved on to the second chapter were we implemented all of them in Python. Specifically, a deep reinforcement learning framework was developed that enabled us to reuse all of these algorithms for different neural network architectures and for different environments. These made the experimentation really easy and enabled us to train deep reinforcement learning agents initially for the simple Cart Pole environment and then for the more complex DuckieTown environment. Regarding DuckieTown environment, we compared with the work of Peter Almasi et al. [29]. Using a similar neural network architecture, but a different algorithm to train the deep reinforcement learning agent, we managed to achieve similar results training only on a CPU for a approximately 3 hours instead of using multiple V100 GPUs for approximately 40 hours as it is stated in Peter Almasi's et al. approach.

On the third and last chapter of the present thesis, we focused on the deployment of the trained deep reinforcement learning agent in the real world. Specifically, our goal was to speed up the computations that take place during the neural network forward pass in order to minimize the inference duration. The motivation behind this attempt is that real world autonomous agents should respond quickly in order to minimize the time interval between two consecutive actions, where the agent is basically "blind". For this purpose the devices that we used where NVIDIA Xavier NX and Xilinx Zynq UltraScale+ MPSoC ZCU104.

Our work focused on two different scenarios, the scenario of the single agent and the scenario of multiple agents. In the single agent scenario, an autonomous agent uses one of the two devices mentioned above in order to perform the computations of the neural network. The configuration that produced the best results was ONNX code run on top of NVIDIA Xaviers NX's GPU which yielded a speed up of approximately 4.4 with the average inference time for a single state equal to 2.369108ms, a duration that translates into approximately 422 FPS. In the multiple agents scenario, we have a swarm of agents that are controlled from a central device, in our case NVIDIA Xavier NX. The central device performs batch inference, with the batch size being the number of agents in the swarm. After our evaluation, by setting an minimum FPS of 100 for each agent, we found out again the the best configuration was ONNX code run on top of NVIDIA Xavier NX's GPU which was able to support 20 different agents in the swarm with an average FPS of 100 for each one of them.

As future work, regarding the learning part, more techniques can be utilized in order to increase the efficiency of the learning process. Currently, the neural network has to see hundreds of thousands of frames in order to be aple to drive the vehicle inside DuckieTown environment. The reason for this is that it lacks prior knowledge. Human beings do have this prior knowledge, they understand the mechanics at the core of the environment they operate in and they are able to learn more efficiently. For this reason the following two approaches can be used : First, pretraining of the neural network through imitation learning can be performed using data coming from a real human driving in the simulator of the DuckieTown environment. This approach is a supervised learning task that requires gathered target data in order to work. To avoid data gathering, one can use the second approach which is a bit bolder. Instead of pretraining the neural network with a supervised learning task, the neural network can be pretrained with a self supervised task. More specifically, a world model can be developed which will produce a meaningful embedding for each state of the DuckieTown. The self supervision can be applied by forcing the model to predict the next state of the environment given the current state and a randomly selected action.

Regarding, the acceleration part, more work can be put into accelerating the inference using FPGAs. Currently, Xilinx compiler limitations made it impossible for us to run inference of the entire network on the DPUs provided by ZCU104's programmable logic.

Appendix A

Implementations of Deep Reinforcement Learning Algorithms

A.1 Basic Deep Reinforcement Learning Algorithm Class

```python
from typing import Callable, Tuple, Union
from diploma_framework.evaluation import test_env
import joblib, torch, numpy as np

class DeepRLAlgorithm():
    """
    Implements basic methods that every algorithm in the framework will inherit
    """

    def get_model(self) -> torch.nn.Module :
        """
        Returns th algorithms neural network model
        """
        return self.model

    def save_model(self, path : str) -> None :
        """
        Saves the algorithm's model to the specified path
        """
        try:
            joblib.dump(self.model, path)
        except:
            raise Exception(f'Poblem storing model to {path}')

    def evaluate(self,
                 n_evaluations : int,
                 custom_test : Union[Callable,None] = None) -> Tuple[float, float] :
        """
        Evaluate model on environment n_evaluations times and
        extract average metrics. Evaluation function can be the predifined one
        or a custom user defined.
        """
        if custom_test is None:
            evaluations = [test_env(self.env, self.model, vis=False) for _ in range(n_evaluations)]
        else:
```

```
            evaluations = [custom_test(self.env, self.model, vis=False) for _ in range(n_evaluations)]

        average_reward = np.mean([metric[0] for metric in evaluations])
        average_nframes = np.mean([metric[1] for metric in evaluations])

        return average_reward, average_nframes
```

A.2 Deep Q-Learning Algorithm

```
import torch, gym, copy, math, random, logging
import numpy as np

from tqdm import tqdm
from collections import deque
from typing import Union
import torch.nn as nn
import torch.optim as optim

from diploma_framework.algorithms._generic import DeepRLAlgorithm

logger = logging.getLogger('deepRL')

class DQN(DeepRLAlgorithm):
    """
    Implements Deep Q-Learning algorithm
    """

    def __init__(self,
                 environment: Union[object, str],
                 model: nn.Module,
                 sync_freq: int = 1000,
                 lr: float = 1e-03,
                 memory_size: int = 2000,
                 batch_size: int = 128,
                 max_frames: int = 150_000,
                 epsilon_start: float = 1.0,
                 epsilon_end: float = 0.0,
                 epsilon_decay: int = 250,
                 gamma: float = 0.9) -> None :

        if isinstance(environment, str):
            self.env = gym.make(environment)
        else:
            self.env = environment

        self.model = model
        self.target_model = copy.deepcopy(model)
        self.sync_freq = sync_freq
        self.batch_size = batch_size
        self.max_frames = max_frames
        self.epsilon_start = epsilon_start
```

```
        self.epsilon_end = epsilon_end
        self.epsilon_decay = epsilon_decay
        self.gamma = gamma

        self.optimizer = optim.Adam(self.model.parameters(), lr=lr)
        self.criterion = nn.MSELoss()
        self.replay_buffer = deque(maxlen=memory_size)

    def run(self,
            eval_window: int = 1000,
            n_evaluations: int = 10,
            early_stopping: bool = True,
            reward_threshold: float = 197.5) -> list:
        """
        Run the DQN algorithm with hyperparameters specified in arguments.
        Returns list of test rewards throughout the agent's training loop.
        """
        logger.info('Initializing training')
        test_rewards = []
        test_frames = []
        frame_idx = 0
        early_stop = False

        with tqdm(total = self.max_frames) as pbar:
            while frame_idx < self.max_frames and not early_stop:

                state0_ = self.env.reset()
                state0 = torch.from_numpy(state0_).float().unsqueeze(dim=0)
                done =  False

                # In episode :
                while not done and not early_stop:

                    epsilon = self.epsilon_end + (self.epsilon_start-self.epsilon_end)*math.exp(-1 *frame_idx
                    ↪  / self.epsilon_decay)
                    frame_idx += 1
                    qval, action = self.model.infer_step(state0)

                    # Explore or exploit
                    if (random.random() < epsilon):
                        action = np.random.randint(0,self.env.action_space.n)
                    else:
                        pass

                    # Make the action
                    state1_, reward, done, _ = self.env.step(action)
                    state1 = torch.from_numpy(state1_).float().unsqueeze(dim=0)

                    # Store experience
                    exp = (state0, action, reward, state1, done)
                    self.replay_buffer.append(exp)
                    state0 = state1

                    # Train network after new experience is added
                    if len(self.replay_buffer) > self.batch_size:

```

```
                        batch = random.sample(self.replay_buffer, self.batch_size)
                        state0_batch, action_batch, reward_batch, state1_batch, done_batch =
                        ↪  self._get_batch_data(batch)

                        Q1 = self.model.infer_batch(state0_batch)
                        with torch.inference_mode():
                            Q2 = self.target_model.infer_batch(state1_batch)

                        # Bellman criterion
                        Y = reward_batch + self.gamma * ((1-done_batch) * torch.max(Q2, dim=1)[0])
                        X = Q1.gather(dim=1, index=action_batch.long().unsqueeze(dim=1)).squeeze()
                        loss = self.criterion(X, Y.detach())

                        # Perform update
                        self.optimizer.zero_grad()
                        loss.backward()
                        self.optimizer.step()

                        # Update target network every sync_freq steps
                        if (frame_idx % self.sync_freq == 0):
                            self.target_model.load_state_dict(self.model.state_dict())

                    if frame_idx % eval_window == 0:
                        reward_metric, frame_metric = self.evaluate(n_evaluations)
                        test_rewards.append(reward_metric)
                        test_frames.append(frame_metric)
                        pbar.update(eval_window)
                        pbar.set_description(f'Reward {reward_metric} - Frames {frame_metric}')
                        if reward_metric > reward_threshold and early_stopping:
                            early_stop = True
                            logger.info('Early stopping criteria met')

        return test_rewards

    def _get_batch_data(self, batch: tuple) -> tuple:
        """
        Given a batch of tuples of the form :
        (s0, a1, r1, s1, done), returns batches of the five didfferet components
        of the initial batch
        """
        state0_batch = []
        action_batch = []
        reward_batch = []
        state1_batch = []
        done_batch = []

        for experience in batch:
            state0, action, reward, state1, done = experience
            state0_batch.append(state0)
            action_batch.append(action)
            reward_batch.append(reward)
            state1_batch.append(state1)
            done_batch.append(done)

        state0_batch = torch.cat(state0_batch)
        action_batch = torch.Tensor(action_batch)
```

```
        reward_batch = torch.Tensor(reward_batch)
        state1_batch = torch.cat(state1_batch)
        done_batch = torch.Tensor(done_batch)

        return state0_batch, action_batch, reward_batch, state1_batch, done_batch
```

A.3 REINFORCE Algorithm

```
import torch, gym, logging
import numpy as np

from tqdm import tqdm
from typing import Union
import torch.nn as nn
import torch.optim as optim

from diploma_framework.algorithms._generic import DeepRLAlgorithm

logger = logging.getLogger('deepRL')

class Reinforce(DeepRLAlgorithm):
    """
    Class that implements REINFORCE algorithm
    """

    def __init__(self,
                 environment: Union[object, str],
                 model: nn.Module,
                 lr: float = 1e-03,
                 max_frames: int = 150_000,
                 num_steps: int = 150,
                 gamma: float = 0.99) -> None:


        if isinstance(environment, str):
            self.env = gym.make(environment)
        else:
            self.env = environment

        self.model = model
        self.max_frames = max_frames
        self.num_steps = num_steps
        self.gamma = gamma

        self.optimizer = optim.Adam(self.model.parameters(), lr=lr)

    def run(self,
            eval_window: int = 1000,
            n_evaluations: int = 10,
            early_stopping: bool = True,
            reward_threshold: float = 197.5) -> list:
```

```
        """
        Run REINFORCE algorithm with hyperparameters specified in arguments.
        Returns list of test rewards throughout the agent's training loop.
        """
        logger.info('Initializing training')
        test_rewards = []
        test_frames = []
        frame_idx = 0
        early_stop = False

        with tqdm(total = self.max_frames) as pbar:
            while frame_idx < self.max_frames and not early_stop:

                curr_state = self.env.reset()
                done = False
                transitions = []
                cumulative_reward = 0

                for _ in range(self.num_steps):

                    act_dist, action =
                    ↪  self.model.infer_step(torch.from_numpy(curr_state).float().unsqueeze(dim=0))
                    prev_state = curr_state
                    curr_state, reward, done, _ = self.env.step(action)
                    frame_idx += 1
                    cumulative_reward = reward + self.gamma * cumulative_reward
                    transitions.append((prev_state, action, cumulative_reward))

                    if frame_idx % eval_window == 0:
                        reward_metric, frame_metric = self.evaluate(n_evaluations)
                        test_rewards.append(reward_metric)
                        test_frames.append(frame_metric)
                        pbar.update(eval_window)
                        pbar.set_description(f'Reward {reward_metric} - Frames {frame_metric}')
                        if reward_metric > reward_threshold and early_stopping:
                            early_stop = True
                            logger.info('Early stopping criteria met')

                    if done:
                        break

                # Do not update is criterions for early stop are met
                if not (early_stopping and early_stop):

                    # Get batches
                    returns_batch = torch.Tensor([r for (s,a,r) in transitions]).flip(dims=(0,))
                    returns_batch /= returns_batch.max()

                    # List of numpy arrays to numpy and hen to Tensor for performance boost
                    state_batch = torch.Tensor(np.asarray([s for (s,a,r) in transitions]))
                    action_batch = torch.Tensor([a for (s,a,r) in transitions])
                    pred_dist_batch = self.model.infer_batch(state_batch)
                    prob_batch = pred_dist_batch.log_prob(action_batch)

                    # Calculate loss based on log prob and discounted rewards
                    loss = self.criterion(prob_batch, returns_batch)
```

```
                    self.optimizer.zero_grad()
                    loss.backward()
                    self.optimizer.step()

        return test_rewards

    def criterion(self,
                  predicted_probs_batch: torch.Tensor,
                  returns_batch: torch.Tensor) -> float:
        """
        Reinforce algorithm loss function
        """
        # log is already applied to predicted_probs_batch
        return -1 * torch.sum(returns_batch*predicted_probs_batch)
```

A.4 Asynchronous Actor-Critic Algorithm

```
import torch, gym, copy, logging
import multiprocessing as mp
import torch.nn as nn
import torch.optim as optim

from tqdm import tqdm
from typing import Union
from diploma_framework.algorithms._generic import DeepRLAlgorithm

test_lock = mp.Lock()

logger = logging.getLogger('deepRL')

class A3C(DeepRLAlgorithm):
    """
    Implements A3C algorithm
    """

    def __init__(self,
                 environment: Union[object, str],
                 model: nn.Module,
                 n_workers: int = 8,
                 lr: float = 1e-03,
                 max_frames: int = 150000,
                 num_steps: int = 200,
                 actor_weight: float = 1,
                 critic_weight: float = 0.1,
                 gamma: float = 0.99) -> None:

        if isinstance(environment, str):
            self.env = gym.make(environment)
        else:
            self.env = environment

```

```
        self.model = model
        # Move model parameters to shared memory
        self.model.share_memory()

        self.n_workers = n_workers
        self.lr = lr
        self.max_frames = max_frames
        self.num_steps = num_steps
        self.actor_weight = actor_weight
        self.critic_weight = critic_weight
        self.gamma = gamma

        # Shared variable to indicate early stopping
        self.early_stop = mp.Value('B', False)

    def run(self,
            eval_window: int = 1000,
            n_evaluations: int = 10,
            early_stopping: bool = True,
            reward_threshold: float = 197.5) -> list:
        """
        Run A3C algorithm with hyperparameters specified in arguments.
        Returns list of test rewards throughout the agent's training loop.
        """
        logger.info('Initializing training')
        manager = mp.Manager()
        test_rewards = manager.list()
        test_frames = manager.list()
        actor_loss = manager.list()
        critic_loss = manager.list()

        processes = []
        frame_counter = mp.Value('Q', 0)

        with tqdm(total = self.max_frames) as pbar:
            for i in range(self.n_workers):
                p = mp.Process(target=self._worker, args=(i, test_rewards, test_frames, actor_loss,
                ↪  critic_loss,
                                                          frame_counter, eval_window, n_evaluations,
                                                          ↪  early_stopping,
                                                          reward_threshold, pbar))
                p.start()
                processes.append(p)

            for p in processes:
                p.join()
            for p in processes:
                p.terminate()

        return test_rewards, test_frames, actor_loss, critic_loss

    def _worker(self,
                worker_id: int,
                test_rewards: list,
                test_frames: list,
                actor_losses: list,
```

```
            critic_losses: list,
            frame_counter: mp.Value,
            eval_window: int,
            n_evaluations: int,
            early_stopping: bool,
            reward_threshold: float,
            pbar: tqdm) -> None:
        """
        Process performed per different worker
        """
        env = copy.deepcopy(self.env)
        env.reset()
        optimizer = optim.Adam(self.model.parameters(), lr=self.lr)
        self.early_stop.value = False

        # Loop for epochs
        while frame_counter.value < self.max_frames and not self.early_stop.value:

            optimizer.zero_grad()
            values, logprobs, rewards = [], [], []
            done = False

            state_ = env.reset()
            state = torch.from_numpy(state_).float().unsqueeze(0)
            step_counter = 0

            bootstraping_value = torch.Tensor([0])
            # Loop for steps in episode
            while step_counter < self.num_steps and not done:

                dist, action, value = self.model.infer_step(state)
                values.append(value)
                logprob_ = dist.log_prob(torch.Tensor([action]))
                logprobs.append(logprob_)
                state_, reward, done, _ = env.step(action)
                state = torch.from_numpy(state_).float().unsqueeze(0)
                if done:
                    env.reset()
                if step_counter == self.num_steps and not done:
                    # Get value of next state
                    _, _, value = self.model.infer_step(state)
                    bootstraping_value = value.detach()
                rewards.append(reward)

                test_lock.acquire()
                step_counter +=1
                frame_counter.value += 1
                if frame_counter.value % eval_window == 0:
                    counter = frame_counter.value
                    reward_metric, frame_metric = self.evaluate(n_evaluations)
                    test_rewards.append(reward_metric)
                    test_frames.append(frame_metric)
                    pbar.n = int(counter / eval_window * 1000)
                    pbar.refresh()
                    pbar.set_description(f'Reward {reward_metric} - Frames {frame_metric}')
                    if reward_metric > reward_threshold and early_stopping:
```

```
                        logger.info('Early stopping criteria met')
                        self.early_stop.value = True
                test_lock.release()

            # Do not update is criterions for early stop are met
            if not (early_stopping and self.early_stop.value):
                actor_loss, critic_loss = self._update_params(optimizer, values, rewards, logprobs,
                ↪  bootstraping_value)
                # Only worker 0 logs losses
                if worker_id == 0:
                    actor_losses.append(actor_loss.item())
                    critic_losses.append(critic_loss.item())

    def _update_params(self,
                       optimizer: torch.optim,
                       values: list,
                       rewards: list,
                       logprobs: list,
                       boostraping_value: torch.Tensor
                       ) -> tuple:
        """
        Implemenetaion of model's parameter update step
        """
        rewards = torch.Tensor(rewards).flip(dims=(0,)).view(-1)
        logprobs = torch.stack(logprobs).flip(dims=(0,)).view(-1)
        values = torch.stack(values).flip(dims=(0,)).view(-1)

        returns = []

        ret_ = boostraping_value
        for i in range(rewards.shape[0]):
            ret_ = rewards[i] + self.gamma * ret_
            returns.append(ret_)
        returns = torch.stack(returns).view(-1).detach()
        loss, actor_loss, critic_loss = self._criterion(logprobs, values, returns)
        loss.backward()
        optimizer.step()
        return actor_loss, critic_loss

    def _criterion(self,
                   logprobs: torch.Tensor,
                   values: torch.Tensor,
                   returns: torch.Tensor) -> tuple:
        """
        Loss function for the A3C algorithm
        """
        actor_loss = -1*logprobs*((returns-values).detach())
        critic_loss = torch.pow(values-returns, 2)
        loss =  self.actor_weight*actor_loss.mean() + self.critic_weight*critic_loss.mean()

        return loss, actor_loss.mean(), critic_loss.mean()
```

A.5 Proximal Policy Optimization Algorithm

```python
import gym, torch, logging
import torch.optim as optim
import torch.nn as nn
import numpy as np

from tqdm import tqdm
from typing import Union
from diploma_framework.algorithms._generic import DeepRLAlgorithm

logger = logging.getLogger('deepRL')

class PPO(DeepRLAlgorithm):

    """
    Implements PPO algorithm

    """

    def __init__(self,
                 environment: Union[str, object],
                 model: nn.Module,
                 lr: float = 1e-03,
                 batch_size: int = 32,
                 epochs: int = 4,
                 max_frames: int = 150_000,
                 num_steps: int = 100,
                 clip_param: float = 0.2,
                 gamma: float = 0.99,
                 lamb: float = 1.0,
                 actor_weight: float = 1.0,
                 critic_weight: float = 0.5,
                 entropy_weight: float = 0.001
                ) -> None:

        if isinstance(environment, str):
            self.env = gym.make(environment)
        else:
            self.env = environment

        self.model = model
        self.optimizer = optim.Adam(self.model.parameters(), lr=lr)
        self.batch_size = batch_size
        self.epochs = epochs
        self.max_frames = max_frames
        self.num_steps = num_steps
        self.clip_param = clip_param
        self.gamma = gamma
        self.lamb = lamb
        self.actor_weight = actor_weight
        self.critic_weight = critic_weight
        self.entropy_weight = entropy_weight

    def run(self,
```

```
            eval_window: int = 1000,
            n_evaluations: int = 10,
            early_stopping: bool = True,
            reward_threshold: float = 197.5) -> list:
        """
        Run the PPO algorithm with hyperparameters specified in arguments.
        Returns list of test rewards throughout the agent's training loop.

        eval_window : number of frames between each evaluation
        """
        logger.info('Initializing training')

        test_rewards = []
        test_frames = []
        frame_idx = 0
        early_stop = False

        with tqdm(total = self.max_frames) as pbar:
            while frame_idx < self.max_frames and not early_stop:

                log_probs = []
                values = []
                states = []
                actions = []
                rewards = []
                masks = []
                entropy = 0

                state = self.env.reset()
                for _ in range(self.num_steps):

                    state = torch.FloatTensor(state).unsqueeze(0)
                    dist, action, value = self.model.infer_step(state)

                    next_state, reward, done, _ = self.env.step(action)
                    entropy += dist.entropy().mean()

                    action_log_probs = dist.log_prob(torch.Tensor([action]))
                    log_probs.append(action_log_probs)
                    values.append(value)
                    rewards.append(reward)
                    masks.append(1-done)

                    states.append(state)
                    actions.append(action)

                    state = next_state
                    frame_idx += 1
                    if frame_idx % eval_window == 0:
                        reward_metric, frame_metric = self.evaluate(n_evaluations)
                        test_rewards.append(reward_metric)
                        test_frames.append(frame_metric)
                        pbar.update(eval_window)
                        pbar.set_description(f'Reward {reward_metric} - Frames {frame_metric}')
                        if reward_metric > reward_threshold and early_stopping:
                            early_stop = True
```

```
                            logger.info('Early stopping criteria met')


                    if done: break

                next_state = torch.FloatTensor(next_state).unsqueeze(0)
                _, _, next_value = self.model.infer_step(next_state)

                returns = self._compute_returns(next_value, rewards, masks, values)

                returns   = torch.cat(returns).detach()
                log_probs = torch.cat(log_probs).detach()
                values    = torch.cat(values).detach()
                states    = torch.cat(states, dim=0)
                actions   = torch.LongTensor(actions)
                advantage = returns -  values

                self._update_params(states, actions, log_probs, returns, advantage)

        return test_rewards, test_frames

    def _compute_returns(self,
                         next_value: float,
                         rewards: list,
                         masks: list,
                         values: list) -> list:
        """
        Calculates return at each time step. Uses delta presented in PPO paper.
        """
        values = values + [next_value]
        gae = 0
        returns = []

        for i in reversed(range(len(rewards))):
            delta = rewards[i] + self.gamma * values[i + 1] * masks[i] - values[i]
            gae = delta + self.gamma * self.lamb * masks[i] * gae
            returns.insert(0, gae + values[i])

        return returns

    def _get_batch(self,
                   states: np.ndarray,
                   actions: np.ndarray,
                   log_probs: np.ndarray,
                   returns: np.ndarray,
                   advantage: np.ndarray) -> tuple:
        """
        Responsible for sampling a random batch out of the total saved data.
        Returns sampled states, actions, log_probs, returns and advantages
        """
        total_experiences = states.size(0)
        for _ in range(total_experiences // self.batch_size):
            selections = np.random.randint(0, total_experiences, self.batch_size)
            yield states[selections,:], actions[selections], log_probs[selections], returns[selections,:],
            ↪  advantage[selections, :]
```

```
    def _update_params(self,
                   states: np.ndarray,
                   actions: np.ndarray,
                   log_probs: np.ndarray,
                   returns: np.ndarray,
                   advantages: np.ndarray) -> None:
        """
        Performs the basic parameter update of PPO algorithm
        """
        for _ in range(self.epochs):
            for state_batch, action_batch, old_log_probs_batch, return_batch, advantage_batch in
            ↪  self._get_batch(states, actions, log_probs, returns, advantages):

                dist_batch, value_batch = self.model.infer_batch(state_batch)
                entropy = dist_batch.entropy().mean()

                new_log_probs_batch = dist_batch.log_prob(action_batch)

                ratio = (new_log_probs_batch - old_log_probs_batch).exp()
                surr1 = ratio*advantage_batch
                surr2 = torch.clamp(ratio, 1-self.clip_param, 1+self.clip_param) * advantage_batch

                actor_loss = -torch.min(surr1, surr2).mean()
                critic_loss = (return_batch - value_batch).pow(2).mean()

                loss = self.critic_weight*critic_loss + self.actor_weight*actor_loss - self.entropy_weight *
                ↪  entropy

                self.optimizer.zero_grad()
                loss.backward()
                self.optimizer.step()
```

A.6 Stacked Frame Proximal Policy Optimization Algorithm

```
import gym, torch, logging, copy
import torch.optim as optim
import torch.nn as nn
import numpy as np

from tqdm import tqdm
from typing import Callable, List, Tuple, Union
from collections import deque
from diploma_framework.algorithms._generic import DeepRLAlgorithm

logger = logging.getLogger('deepRL')

class StackedFramePPO(DeepRLAlgorithm):
    """
    Implements PPO algorithm but considers as one state the selected
    number of the last frames. This is used for converting an environment
    that cannot be modeled as a Markov Decision Process into a Markov
```

```
    Decision Process. Normally used when the environment provides as state
    the whole displayef frame.
    """

    def __init__(self,
                 environment: Union[str, object],
                 model: nn.Module,
                 lr: float = 1e-03,
                 batch_size: int = 32,
                 epochs: int = 4,
                 max_frames : int = 150_000,
                 num_steps: int = 100,
                 clip_param: float = 0.2,
                 gamma: float = 0.99,
                 lamb: float = 1.0,
                 actor_weight: float = 1.0,
                 critic_weight: float = 0.5,
                 entropy_weight: float = 0.001,
                 stacked_frames: int = 5
                ) -> None:

        if isinstance(environment, str):
            self.env = gym.make(environment)
        else:
            self.env = environment

        self.model = model
        self.optimizer = optim.Adam(self.model.parameters(), lr=lr)
        self.batch_size = batch_size
        self.epochs = epochs
        self.max_frames = max_frames
        self.num_steps = num_steps
        self.clip_param = clip_param
        self.gamma = gamma
        self.lamb = lamb
        self.actor_weight = actor_weight
        self.critic_weight = critic_weight
        self.entropy_weight = entropy_weight
        self.stacked_frames = stacked_frames

    def run(self,
            eval_window: int,
            n_evaluations: int,
            early_stopping: bool,
            reward_threshold: float,
            frames_threshold: float,
            return_best: bool = True,
            test_function: Union[Callable, None] = None) -> Tuple[List[float]] :
        """
        Run the PPO algorithm with hyperparameters specified in arguments.
        Returns list of test rewards throughout the agent's training loop.

        eval_window : number of frames between each evaluation
        """
        logger.info('Initializing training')

```

```
        test_rewards = []
        test_frames = []
        frame_idx = 0
        early_stop = False
        best_reward = float('-inf')
        best_model = None

        with tqdm(total = self.max_frames) as pbar:
            while frame_idx < self.max_frames and not early_stop:

                log_probs = []
                values = []
                states = []
                actions = []
                rewards = []
                masks = []
                entropy = 0

                frame = self.env.reset()
                stacked_frames = deque([torch.zeros(size=frame.shape).unsqueeze(0)]*self.stacked_frames,
                                       maxlen=self.stacked_frames)

                for _ in range(self.num_steps):

                    frame = torch.FloatTensor(frame).unsqueeze(0)
                    stacked_frames.append(frame)
                    state = torch.cat(tuple(stacked_frames), dim=-1)
                    dist, action, value = self.model.infer_step(state)

                    next_frame, reward, done, _ = self.env.step(action)
                    entropy += dist.entropy().mean()

                    action_log_probs = dist.log_prob(torch.Tensor([action]))
                    log_probs.append(action_log_probs)
                    values.append(value)
                    rewards.append(reward)
                    masks.append(1-done)

                    states.append(state)
                    actions.append(action)

                    frame = next_frame
                    frame_idx += 1
                    if frame_idx % eval_window == 0:
                        reward_metric, frame_metric = self.evaluate(n_evaluations, test_function)
                        test_rewards.append(reward_metric)
                        test_frames.append(frame_metric)
                        pbar.update(eval_window)
                        pbar.set_description(f'Reward {reward_metric} - Frames {frame_metric}')
                        if return_best and reward_metric > best_reward:
                            best_model = copy.deepcopy(self.model)
                            best_reward = reward_metric
                            logger.info(f'Current best at frame {int(frame_idx)} with reward
                            ↪  {best_reward:.2f}')
                        if (reward_metric > reward_threshold or frame_metric > frames_threshold) and
                        ↪  early_stopping:
```

```
                            early_stop = True
                            logger.info('Early stopping criteria met')

                    if done: break

                next_frame = torch.FloatTensor(next_frame).unsqueeze(0)
                stacked_frames.append(next_frame)
                next_state = torch.cat(tuple(stacked_frames), dim=-1)
                _, _, next_value = self.model.infer_step(next_state)

                returns = self._compute_returns(next_value, rewards, masks, values)

                returns   = torch.cat(returns).detach()
                # Normalize returns by subtracting mean and deviding with std
                returns  = (returns - torch.mean(returns)) / (torch.std(returns) + 1e-10)
                log_probs = torch.cat(log_probs).detach()
                values    = torch.cat(values).detach()
                # Normalize returns by subtracting mean and deviding with std
                values = (values - torch.mean(values)) / (torch.std(values) + 1e-10)
                states    = torch.cat(states, dim=0)
                actions   = torch.LongTensor(actions)
                advantage = returns -  values
                # Normalize advantages by subtracting mean and deviding with std
                advantage  =(advantage - torch.mean(advantage)) / (torch.std(advantage) + 1e-10)

                self._update_params(states, actions, log_probs, returns, advantage)

        if return_best:
            self.model = best_model

        return test_rewards, test_frames

    def _compute_returns(self,
                    next_value: float,
                    rewards: list,
                    masks: list,
                    values: list) -> list :
        """
        Calculates return at each time step. Uses delta presented in PPO paper.
        """
        values = values + [next_value]
        gae = 0
        returns = []

        for i in reversed(range(len(rewards))):
            delta = rewards[i] + self.gamma * values[i + 1] * masks[i] - values[i]
            gae = delta + self.gamma * self.lamb * masks[i] * gae
            returns.insert(0, gae + values[i])

        return returns

    def _get_batch(self,
               states: np.ndarray,
               actions: np.ndarray,
               log_probs: np.ndarray,
               returns: np.ndarray,
```

```
              advantage: np.ndarray) -> tuple:
        """
        Responsible for sampling a random batch out of the total saved data.
        Returns sampled states, actions, log_probs, returns and advantages
        """
        total_experiences = states.size(0)
        for _ in range(total_experiences // self.batch_size):
            selections = np.random.randint(0, total_experiences, self.batch_size)
            yield states[selections,:], actions[selections], log_probs[selections], returns[selections,:],
            ↪  advantage[selections, :]

    def _update_params(self,
                       states: np.ndarray,
                       actions: np.ndarray,
                       log_probs: np.ndarray,
                       returns: np.ndarray,
                       advantages: np.ndarray) -> None:
        """
        Performs the basic parameter update of PPO algorithm
        """
        for _ in range(self.epochs):
            for state_batch, action_batch, old_log_probs_batch, return_batch, advantage_batch in
            ↪  self._get_batch(states, actions, log_probs, returns, advantages):

                dist_batch, value_batch = self.model.infer_batch(state_batch)
                value_batch = (value_batch - torch.mean(value_batch)) / (torch.std(value_batch) + 1e-10)
                entropy = dist_batch.entropy().mean()

                new_log_probs_batch = dist_batch.log_prob(action_batch)

                ratio = (new_log_probs_batch - old_log_probs_batch).exp()
                surr1 = ratio*advantage_batch
                surr2 = torch.clamp(ratio, 1-self.clip_param, 1+self.clip_param) * advantage_batch

                actor_loss = -torch.min(surr1, surr2).mean()
                critic_loss = (return_batch - value_batch).pow(2).mean()

                loss = (self.critic_weight*critic_loss) + (self.actor_weight*actor_loss) -
                ↪  (self.entropy_weight * entropy)

                self.optimizer.zero_grad()
                loss.backward()
                self.optimizer.step()
```

Appendix B

Neural Network Acceleration

B.1 Conversion to ONNX

```
import torch, onnxruntime, numpy as np
from torchvision import transforms as T
from torch.nn import functional as F

class Actor(torch.nn.Module):
    #Implements the inference of only the actor model coming from the acotr critic object

    def __init__(self,
                 actor_critic_model,
                 device = 'cpu'):
        #Initilizes actor by copying necesseary layers
        super(Actor, self).__init__()
        self.device = device
        self.conv_core = actor_critic_model.conv_core.to(device)
        self.actor_head = actor_critic_model.actor_head.to(device)
        self.transform = actor_critic_model.transform

    def forward(self, x):
        #Implements forward pass of model
        x = torch.permute(x, (0, 3, 1, 2))   # Place channel axis in correct position
        #x = self.transform(x)               # Apply transform
        x = x / 255
        #x = T.functional.crop(x, top=20, left=0, height=40, width=80)
        x = x[:,:,20:,:]
        x = x.to(device=self.device)
        visual_repr = self.conv_core(x).squeeze(-1).squeeze(-1)
        dist = F.log_softmax(self.actor_head(visual_repr), dim=1)
        return dist

    def infer_action(self, x):
        # Utilizes torch distributions to return an action
        dist_probs = self.forward(x)
        dist = torch.distributions.Categorical(logits=dist_probs)
        return dist.sample().cpu().numpy()[0]

model = Actor(model)

```

```
dummy_input = saved_states[10] # Randomly selected input state

torch.onnx.export(model,                                # model being run
                  dummy_input,     # model input (or a tuple for multiple inputs)
                  "../models/actor.onnx",    # where to save the model (can be a file or file-like object)
                  export_params=True,        # store the trained parameter weights inside the model file
                  opset_version=11,          # the ONNX version to export the model to
                  do_constant_folding=True,  # whether to execute constant folding for optimization
                  input_names = ['input'],   # the model's input names
                  output_names = ['output'], # the model's output names
                  dynamic_axes={'input' : {0 : 'batch_size'},    # variable length axes
                                'output' : {0 : 'batch_size'}})
```

B.2 ONNX Actor Model

```
import torch, onnxruntime, numpy as np

class ONNXActor():
    # Implements actor using ONNX runtime

    def __init__(self, onnx_path, providers):
        # Initiliaze model
        self.ort_session = onnxruntime.InferenceSession(onnx_path, providers=providers)

    def forward(self, x):
        # Implements forward pass of model
        output = self.ort_session.run(None, {'input' : x.numpy().astype(np.float32)})[0]
        return torch.Tensor(output)

    def infer_action(self, x):
        # Utilizes torch distributions to return an action
        dist_probs = self.forward(x)
        dist = torch.distributions.Categorical(logits=dist_probs)
        return dist.sample().numpy()[0]


model_onnx = ONNXActor(onnx_path='../models/actor.onnx', providers=['CPUExecutionProvider'])
```

B.3 Vitis AI Compilation Process

B.3.1 Quantization utilities

```
import torch, joblib
import torchvision
import torch.nn as nn
import torch.nn.functional as F

```

```
from tqdm import tqdm
from torch.utils.data import Dataset
from utilities.models import Actor

class StatesDataset(Dataset):

    def __init__(self, path, golden_model_path='models/actor_state_dict.pt'):

        states = joblib.load(path)
        model = Actor()
        model.load_state_dict(state_dict=torch.load(golden_model_path))
        self.processed_states = []
        for state in states:
            #state = state.permute((0, 3, 1, 2))
            #state = state / 255
            #state = state[:,:,20:,:]
            self.processed_states.append(state)
        self.target = [model.forward(state) for state in self.processed_states]

    def __len__(self):
        return len(self.processed_states)

    def __getitem__(self, index):
        return self.processed_states[index].squeeze(0), self.target[index].squeeze(0)


def test(model, device, test_loader):
    '''
    test the model
    '''
    avg_error = 0
    total_samples = 0
    model.eval()
    with torch.no_grad():
        for data, target in tqdm(test_loader):
            data, target = data.to(device), target.to(device)
            output = model(data)

            batch_avg_error = torch.abs(output - target).flatten().mean().item()
            avg_error += (batch_avg_error * output.shape[0])
            total_samples += output.shape[0]
    avg_error /= total_samples
    print(f'Average absolute error {avg_error}')
    return avg_error
```

B.3.2 Quantization script

```
import os
import sys
import argparse
import random
import torch
import torchvision
```

```
import torch.nn as nn
import torch.nn.functional as F
from pytorch_nndct.apis import torch_quantizer, dump_xmodel

from common import *
from utilities.models import Actor

DIVIDER = '-----------------------------------------'


def quantize(build_dir,quant_mode,batchsize):

  dset_dir = build_dir + '/dataset'
  float_model = build_dir + '/float_model'
  quant_model = build_dir + '/quant_model'


  # use GPU if available
  if (torch.cuda.device_count() > 0):
    print('You have',torch.cuda.device_count(),'CUDA devices available')
    for i in range(torch.cuda.device_count()):
      print(' Device',str(i),': ',torch.cuda.get_device_name(i))
    print('Selecting device 0..')
    device = torch.device('cuda:0')
  else:
    print('No CUDA devices available..selecting CPU')
    device = torch.device('cpu')

  # load trained model
  model = Actor().to(device)
  model.load_state_dict(torch.load('models/actor_state_dict.pt'))

  # force to merge BN with CONV for better quantization accuracy
  optimize = 1

  # override batchsize if in test mode
  if (quant_mode=='test'):
    batchsize = 1

  rand_in = torch.randint(low=0, high=256, size=[batchsize, 60, 80, 15], dtype=torch.float32)
  #rand_in = torch.randint(low=0, high=256, size=[batchsize, 15, 40, 80], dtype=torch.float32)

  quantizer = torch_quantizer(quant_mode=quant_mode, bitwidth=8, module=model, input_args=(rand_in),
                              output_dir=quant_model, device=torch.device('cpu'))
  quantized_model = quantizer.quant_model


  # data loader
  test_dataset = StatesDataset(path='data/test_set_1.joblib')

  test_loader = torch.utils.data.DataLoader(test_dataset,
                                            batch_size=batchsize,
                                            shuffle=False)

  # evaluate
  test(quantized_model, device, test_loader)
```

```
63
64
65    # export config
66    if quant_mode == 'calib':
67      quantizer.export_quant_config()
68    if quant_mode == 'test':
69      quantizer.export_xmodel(deploy_check=False, output_dir=quant_model)
70
71    return
72
73
74
75  def run_main():
76
77    # construct the argument parser and parse the arguments
78    ap = argparse.ArgumentParser()
79    ap.add_argument('-d',  '--build_dir',  type=str, default='build',    help='Path to build folder. Default is
      ↪  build')
80    ap.add_argument('-q',  '--quant_mode', type=str, default='calib',    choices=['calib','test'],
      ↪  help='Quantization mode (calib or test). Default is calib')
81    ap.add_argument('-b',  '--batchsize',  type=int, default=100,        help='Testing batchsize - must be an
      ↪  integer. Default is 100')
82    args = ap.parse_args()
83
84    print('\n'+DIVIDER)
85    print('PyTorch version : ',torch.__version__)
86    print(sys.version)
87    print(DIVIDER)
88    print(' Command line options:')
89    print ('--build_dir    : ',args.build_dir)
90    print ('--quant_mode   : ',args.quant_mode)
91    print ('--batchsize    : ',args.batchsize)
92    print(DIVIDER)
93
94    quantize(args.build_dir,args.quant_mode,args.batchsize)
95
96    return
97
98
99
100 if __name__ == '__main__':
101     run_main()
```

B.3.3 Compilation script

```
1  if [ $1 = zcu102 ]; then
2        ARCH=/opt/vitis_ai/compiler/arch/DPUCZDX8G/ZCU102/arch.json
3        TARGET=zcu102
4        echo "-----------------------------------------"
5        echo "COMPILING MODEL FOR ZCU102.."
6        echo "-----------------------------------------"
7  elif [ $1 = zcu104 ]; then
8        ARCH=/opt/vitis_ai/compiler/arch/DPUCZDX8G/ZCU104/arch.json
```

```
      TARGET=zcu104
      echo "-----------------------------------------"
      echo "COMPILING MODEL FOR ZCU104.."
      echo "-----------------------------------------"
elif [ $1 = vck190 ]; then
      ARCH=/opt/vitis_ai/compiler/arch/DPUCVDX8G/VCK190/arch.json
      TARGET=vck190
      echo "-----------------------------------------"
      echo "COMPILING MODEL FOR VCK190.."
      echo "-----------------------------------------"
elif [ $1 = u50 ]; then
      ARCH=/opt/vitis_ai/compiler/arch/DPUCAHX8H/U50/arch.json
      TARGET=u50
      echo "-----------------------------------------"
      echo "COMPILING MODEL FOR ALVEO U50.."
      echo "-----------------------------------------"
else
      echo  "Target not found. Valid choices are: zcu102, zcu104, vck190, u50 ..exiting"
      exit 1
fi

BUILD=$2
LOG=$3

compile() {
  vai_c_xir \
  --xmodel      ${BUILD}/quant_model/Actor_int.xmodel \
  --arch        $ARCH \
  --net_name    Actor_${TARGET} \
  --output_dir  ${BUILD}/compiled_model
}

compile 2>&1 | tee ${LOG}/compile_$TARGET.log


echo "-----------------------------------------"
echo "MODEL COMPILED"
echo "-----------------------------------------"
```

B.3.4 Complete pipeline script

```
rm -rf build
export BUILD=./build
export LOG=${BUILD}/logs
mkdir -p ${LOG}

python -u quantize.py -d ${BUILD} --quant_mode calib 2>&1 | tee ${LOG}/quant_calib.log

python -u quantize.py -d ${BUILD} --quant_mode test  2>&1 | tee ${LOG}/quant_test.log

source compile.sh zcu104 ${BUILD} ${LOG}

xir png build/compiled_model/Actor_zcu104.xmodel model.png
```

B.4 DPU Actor Model

```python
class FPGAActor():

    def __init__(self, dpu_runner):

        self.dpu_runner = dpu_runner
        self.numpy_actor_head = []

        input_fixpos = self.dpu_runner.get_input_tensors()[0].get_attr("fix_point")
        print(input_fixpos)
        self.input_scale = 2**input_fixpos

        output_fixpos = self.dpu_runner.get_output_tensors()[0].get_attr("fix_point")
        self.output_scale = 1 / (2**output_fixpos)

        input_tensors = self.dpu_runner.get_input_tensors()
        output_tensors = self.dpu_runner.get_output_tensors()
        self.input_ndim = tuple(input_tensors[0].dims)
        self.output_ndim = tuple(output_tensors[0].dims)
        self.load_numpy_head()

        print('Input tensor name: ', input_tensors[0].name)
        print('Input tensor dim: ', input_tensors[0].dims)
        print('Input tensor dtype: ', input_tensors[0].dtype)
        print('Output tensor name: ', output_tensors[0].name)
        print('Output tensor dim: ', output_tensors[0].dims)
        print('Output tensor dtype: ', output_tensors[0].dtype)

    def preprocess_fn(self, state):
        """Preprocessing of state"""
        state = state[:,20:,:,:] * (1/255) * (self.input_scale)
        return state.astype(np.int8)

    def load_numpy_head(self):
        # load linear numpy head
        for layer_idx in range(0,10,2):
            weights = np.load(f'model_linear_layers/weights_{layer_idx}.npy')
            bias = np.expand_dims(np.load(f'model_linear_layers/bias_{layer_idx}.npy'),1)
            self.numpy_actor_head.append((weights, bias))

    def forward(self, x):
        # Implements forward pass using numpy linear layer

        x = self.preprocess_fn(x)
        visual_repr = np.empty(self.output_ndim, dtype=np.int8, order="C")
        job_id = self.dpu_runner.execute_async(x, visual_repr)
        self.dpu_runner.wait(job_id)
        visual_repr = visual_repr * self.output_scale

        # layer 1
```

```
        weights, bias = self.numpy_actor_head[0]
        out = weights @ visual_repr.T + bias
        out = np.where(out > 0, out, out * 0.01)
        # layer 2
        weights, bias = self.numpy_actor_head[1]
        out = weights @ out + bias
        out = np.where(out > 0, out, out * 0.01)
        # layer 3
        weights, bias = self.numpy_actor_head[2]
        out = weights @ out + bias
        out = np.where(out > 0, out, out * 0.01)
        # layer 4
        weights, bias = self.numpy_actor_head[3]
        out = weights @ out + bias
        out = np.where(out > 0, out, out * 0.01)
        # layer 5
        weights, bias = self.numpy_actor_head[4]
        out = (weights @ out + bias).T
        out = np.exp(out) / np.sum(np.exp(out), axis=1)
        #out = np.log(out)
        return out
```

B.5 ZCU104 Application Code

```
from ctypes import *
from typing import List
import numpy as np
import vart
import os
import pathlib
import xir
import time
import sys
import json
import argparse
import vitis_ai_library

_divider = '------------------------------'


class Timer():
    """
    Class that implements a timer
    """

    def __init__(self) :
        """Initilize times"""
        self.t0 = None
        self.total_time = 0
        self.n_laps = 0

```

```
    def start(self):
        """Start timer"""
        self.t0 = time.perf_counter()

    def stop(self):
        """Stops timer"""
        dt = (time.perf_counter() - self.t0)
        self.total_time += dt
        self.n_laps += 1
        return dt

    def reset(self):
        """Resets timer"""
        self.t0 = None
        self.total_time = 0
        self.n_laps = 0

    def get_average_time(self):
        """Returns average lap time"""
        return self.total_time / self.n_laps

    def get_laps(self):
        """Returns number of laps"""
        return self.n_laps

def log_results(durations_list, dest_file):
    """Log durations in json"""
    with open(dest_file, 'w') as f:
        json.dump(durations_list, f)

class FPGAActor():

    def __init__(self, dpu_runner):

        self.dpu_runner = dpu_runner
        self.numpy_actor_head = []

        input_fixpos = self.dpu_runner.get_input_tensors()[0].get_attr("fix_point")
        print(input_fixpos)
        self.input_scale = 2**input_fixpos

        output_fixpos = self.dpu_runner.get_output_tensors()[0].get_attr("fix_point")
        self.output_scale = 1 / (2**output_fixpos)

        input_tensors = self.dpu_runner.get_input_tensors()
        output_tensors = self.dpu_runner.get_output_tensors()
        self.input_ndim = tuple(input_tensors[0].dims)
        self.output_ndim = tuple(output_tensors[0].dims)
        self.load_numpy_head()

        print('Input tensor name: ', input_tensors[0].name)
        print('Input tensor dim: ', input_tensors[0].dims)
        print('Input tensor dtype: ', input_tensors[0].dtype)
        print('Output tensor name: ', output_tensors[0].name)
        print('Output tensor dim: ', output_tensors[0].dims)
        print('Output tensor dtype: ', output_tensors[0].dtype)
```

```
84
85      def preprocess_fn(self, state):
86          """Preprocessing of state"""
87          state = state[:,20:,:,:] * (1/255) * (self.input_scale)
88          return state.astype(np.int8)
89
90      def load_numpy_head(self):
91          # load linear numpy head
92          for layer_idx in range(0,10,2):
93              weights = np.load(f'model_linear_layers/weights_{layer_idx}.npy')
94              bias = np.expand_dims(np.load(f'model_linear_layers/bias_{layer_idx}.npy'),1)
95              self.numpy_actor_head.append((weights, bias))
96
97      def forward(self, x):
98          # Implements forward pass using numpy linear layer
99
100         x = self.preprocess_fn(x)
101         visual_repr = np.empty(self.output_ndim, dtype=np.int8, order="C")
102         job_id = self.dpu_runner.execute_async(x, visual_repr)
103         self.dpu_runner.wait(job_id)
104         visual_repr = visual_repr * self.output_scale
105
106         # layer 1
107         weights, bias = self.numpy_actor_head[0]
108         out = weights @ visual_repr.T + bias
109         out = np.where(out > 0, out, out * 0.01)
110         # layer 2
111         weights, bias = self.numpy_actor_head[1]
112         out = weights @ out + bias
113         out = np.where(out > 0, out, out * 0.01)
114         # layer 3
115         weights, bias = self.numpy_actor_head[2]
116         out = weights @ out + bias
117         out = np.where(out > 0, out, out * 0.01)
118         # layer 4
119         weights, bias = self.numpy_actor_head[3]
120         out = weights @ out + bias
121         out = np.where(out > 0, out, out * 0.01)
122         # layer 5
123         weights, bias = self.numpy_actor_head[4]
124         out = (weights @ out + bias).T
125         out = np.exp(out) / np.sum(np.exp(out), axis=1)
126         #out = np.log(out)
127         return out
128
129 def get_child_subgraph_dpu(graph: "Graph") -> List["Subgraph"]:
130     assert graph is not None, "'graph' should not be None."
131     root_subgraph = graph.get_root_subgraph()
132     assert (root_subgraph is not None), "Failed to get root subgraph of input Graph object."
133     if root_subgraph.is_leaf:
134         return []
135     child_subgraphs = root_subgraph.toposort_child_subgraph()
136     assert child_subgraphs is not None and len(child_subgraphs) > 0
137     return [
138         cs
139         for cs in child_subgraphs
```

```
        if cs.has_attr("device") and cs.get_attr("device").upper() == "DPU"
    ]

def app(states_path, model):

    states = np.load(states_path)
    graph = xir.Graph.deserialize(model)
    subgraphs = get_child_subgraph_dpu(graph)
    dpu_runner = vart.Runner.create_runner(subgraphs[0], "run")
    model = FPGAActor(dpu_runner)
    timer = Timer()
    outputs = np.empty(shape=(states.shape[0], 3))
    durations = []
    for i, state in enumerate(states):
        timer.start()
        out = model.forward(state)
        outputs[i] = out
        dt = timer.stop()
        durations.append(dt)
        if i == 0:
            print(f'Model output shape : {out.shape}')

    print(f'Averege inference time {timer.get_average_time() * 1000}ms on {timer.get_laps()} states.')
    golden_outputs = np.load('golden_out.npy')
    error = np.abs(outputs - golden_outputs).flatten()
    max_error = np.max(error)
    mean_error = np.mean(error)
    print(f'Max absolute error: {max_error} Average absolute error: {mean_error}')
    log_results(durations_list=durations, dest_file='zcu_fpga.json')

# only used if script is run as 'main' from command line
def main():

  # construct the argument parser and parse the arguments
  ap = argparse.ArgumentParser()
  ap.add_argument('-d', '--states_path', type=str, default='test_set_small.npy', help='Path to joblibfile of
  ↪  states')
  ap.add_argument('-m', '--model',     type=str, default='Actor_zcu104.xmodel', help='Path of xmodel. Default
  ↪  is Actor_zcu104.xmodel')
  args = ap.parse_args()

  print ('Command line options:')
  print (' -states_path : ', args.states_path)
  print (' -model   : ', args.model)

  app(args.states_path, args.model)

if __name__ == '__main__':
  main()
```

Bibliography

[1] Mahryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2nd edition, 2018.

[2] Alexander Zai and Brandon Brown. *Deep Reinforcement Learning in Action*. Alexander Zai, 1st edition, 2020.

[3] Simon Haykin. *Neural Networks and Learning Machines*. Pearson, 3rd edition, 2009.

[4] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021.

[5] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[6] Nvidia jetson xavier nx [online]. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/.

[7] Vitis ai [online]. https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html.

[8] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning*. The MIT Press, 2nd edition, 2018.

[9] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2nd edition, 2006.

[10] Sergios Theodoridis and Konstantinos Koutroumbas. *Pattern Recogintion*. Elsevier, 4th edition, 2009.

[11] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[12] Hossein Gholamalinezhad and Hossein Khosravi. Pooling methods in deep neural networks, a review, 2020.

[13] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[15] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[16] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015.

[18] Melrose Roderick, James MacGlashan, and Stefanie Tellex. Implementing the deep q-network, 2017.

[19] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.

[20] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic with experience replay, 2017.

[21] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2017.

[22] Nikita Rudin, David Hoeller, Philipp Reist, and Marco Hutter. Learning to walk in minutes using massively parallel deep reinforcement learning, 2021.

[23] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[24] Openai gym documentation [online]. https://gym.openai.com/docs/.

[25] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[26] Pytorch documentation [online]. https://pytorch.org/docs/stable/index.html.

[27] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[28] Maxime Chevalier-Boisvert, Florian Golemo, Yanjun Cao, Bhairav Mehta, and Liam Paull. Duckietown environments for openai gym. https://github.com/duckietown/gym-duckietown, 2018.

[29] Peter Almasi, Robert Moni, and Balint Gyires-Toth. Robust reinforcement learning-based autonomous driving agent for simulation and real world. *2020 International Joint Conference on Neural Networks (IJCNN)*, Jul 2020.

[30] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways. 4 2022.

[31] Onnx [online]. https://onnx.ai/.

[32] Xilinx zynq ultrascale+ mpsoc zcu104 evaluation kit [online]. https://www.xilinx.com/products/boards-and-kits/zcu104.html.

[33] Onnx graph optimizations [online]. https://onnxruntime.ai/docs/performance/graph-optimizations.html.