NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

The Friv Reinforcement Learning Environment

George S. Stamatelis

Supervisor: **Panagiotis Stamatopoulos,** Assistant Professor

ATHENS

NOVEMBER 2021

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Περιβαλλον Ενισχυτικής Μηχανικής Μάθησης Βασιμένο στο Friv

Γεώργιος Σ. Σταματέλης

Επιβλέπων: **Παναγίωτης Σταματόπουλος,** Επίκουρος Καθηγητής

ΑΘΗΝΑ

ΝΟΕΜΒΡΙΟΣ 2021

BSc THESIS

The Friv Reinforcement Learning Environment

George S. Stamatelis
S.N.: 1115201800185

SUPERVISOR: **Panagiotis Stamatopoulos,** Assistant Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Περιβαλλον Ενισχυτικής Μηχανικής Μάθησης Βασιμένο στο Friv

Γεώργιος Σ. Σταματέλης
Α.Μ.: 1115201800185

ΕΠΙΒΛΕΠΩΝ: **Παναγίωτης Σταματόπουλος,** Επίκουρος Καθηγητής

ABSTRACT

The goal of the thesis is to provide a new environment based on the popular video game website FRIV to evaluate single player reinforcement learning agents. Specifically, it provides a good platform to examine an agent's capability to explore a large state space with sparse rewards. Moreover, it can also be used to experiment with transfer reinforcement learning.

SUBJECT AREA: Artificial Intelligence

KEYWORDS: Reinforcement Learning, Deep Learning, Convolutional Neural Networks, Computer Vision, Control, Video Games, Sequential Decision Making

ΠΕΡΙΛΗΨΗ

Ο σκοπός της πτυχιακής εργασίας είναι να εισάγει ένα νέο περιβάλλον ανάπτυξης και αξιολόγησης αλγορίθμων ενισχυτικής μηχανικής μάθησης βασισμένο στην ιστοσελίδα παιχνιδιών FRIV. Πιο συγκεκριμένα, είναι ενας καλός τρόπος αξιολόγησης της δυνατότητας ενός πράκτορα να εξερευνεί τον μεγαλο χωρο αναζήτησης εφόσον τα σήματα επιβράβευσης είναι αραιά. Επιπλέον, μπορει να χρησιμοποιηθεί για να γινουν πειραματα ενισχυτικής μηχανικής μάθησης μέσω μεταφοράς.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Τεχνιτή Νοημοσύνη

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Ενισχυτική Μηχανική Μάθηση, Βαθιά Μάθηση, Νευρωνικά Δίκτυα Συνελίξεων, Τεχνητή Οραση, Ελεγχος, Βιντεοπαιχνίδια, Ακολουθιακή Λήψη Αποφάσεων

Αφιερώνεται στην μητέρα μου

ACKNOWLEDGEMENTS

Για τη διεκπεραίωση της παρούσας Πτυχιακής Εργασίας, θα ήθελα να ευχαριστήσω τον επιβλέποντα, επ. καθ .Παναγιώτη Σταματόπουλο, για τη συνεργασία και την πολύτιμη συμβολή του στην ολοκλήρωση της.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

The following work was done as part of the BSc program of studies an the Department of Informatics and Telecommunications of the National and Kapodistrian University of Athens.

1. INTRODUCTION

In the past few years, a lot of research has been conducted in the field of deep reinforcement learning concerning games. On the one hand games are very easy to implement and experiment with, compared to other areas such as self driving vehicles or robotics. On the other hand they require a certain level of ”intelligence” to perform well. Moreover, most modern games are too large to be solved with traditional search strategies.
Deep Q learning [13] and its improvements have managed to produce very exiting and unexpected results compared to older strategies. That being said, as we will see, they still encounter a lot of problems and can perform really poorly even in simple video games.
Our goal in this thesis is to create and publicly distribute a collection of video games, based on the very popular website FRIV, in which modern reinforcement learning algorithms struggle. These games will be emulated on top of [6] , to make experimenting with new algorithms easy for researchers. Additionally, we will implement multiple levels of the same game, to make it easy to experiment with transfer learning from simpler to tougher tasks. We will also provide a performance baseline.
Chapter 2 contains the necessary background for this thesis. Chapter 3 provides an overview of other successful applications of deep reinforcement learning. In chapter 4 we discuss the games we implemented and in chapter 5 we present the performance of some popular reinforcement learning algorithms. Last but not least, in chapter 6 we propose some possible directions for future research.
The accompanying source code can be found **here**.

2. BACKGROUND

2.1 Introduction to reinforcement learning

We will start by formulating the reinforcement learning problem and providing some basic definitions and algorithms. We will also briefly discuss some of the most common artificial neural networks used in deep reinforcement learning (for more information [5] or [8]). Finally, we will present some of the most successful algorithms of deep reinforcement learning. The pseudo-code segments provided are from [22], the images(unless stated otherwise) are created by the author.

2.1.1 Characteristics of reinforcement learning

In the reinforcement learning setting, there is an agent that interacts with the environment. That agent is both the learner and the decision maker. Environment is everything outside the agent. The agent interacts with the environment and receives reward signals.
A reward R_t is a scalar number indicating how well an agent has done. Our goal is to find a sequence of actions that maximise that reward signal. The whole field of reinforcement learning is based on the reward hypothesis. All goals can be described by the maximisation of expected cumulative rewards. In practise this approach has proven very flexible and widely applicable.
At each time step t, the agent is at a state S_t. S_t is a Markov state iff

$$P[S_{t+1}|S_t] = P[S_{t+1}|S_1, ...S_t]$$

That means S_t contains all the useful information from the history. For that reason Markov states can also be called information states.

2.1.2 Markov Decision Processes(MDP)

A Markov process is any memoryless random process operating over Markov states. A Markov reward processes is a Markov process that observes rewards. A Markov decision process is a reward process that ”makes decisions”. Therefore, at each time step t it is at a state S_t. It chooses to take an action A_t and it transitions to a state S_{t+1}. It also observes a reward $R_{t+1} \in \mathbb{R}$.
In reinforcement learning literature, we usually denote the set of all possible states as S, the set of all possible rewards as R and the set of all possible actions at a given state as A(s).

For any Markov state s and a successor s' and an action a, we define the state transition probability as

$$p(s'|s, a) = Pr\{S_t = s'|S_{t-1} = s, A_{t-1} = a\}$$

Therefore we will denote a Markov decision process as a tupple <S,A,P,R>. S is the set of all possible states, R the set of all possible rewards, A(s) is the set of all possible actions from a state s and P is a matrix containing the state transition probability for every state

action pair.
We can also compute the expected rewards for all state action pairs as :

$$r(s,a) = E[R_t|S_{t-1} = S, A_{t-1} = a] = \sum_{r \in R} r \sum_{s' \in S} p(s', r|s, a)$$

The return is usually formulated as the total discounted reward from time step t.

$$G_t = R_{t+1} + \gamma * R_{t+1} + \gamma^2 R_{t+3} + ...$$

$\gamma \in [0, 1]$
Here we have considered that the episode termination is a self looping state (cycles back to itself) with zero reward forever. We use the parameter γ because having something ”good” now is preferable than having something ”good” 100 steps later. That being said, its use is optional and indeed some applications don’t require that. If that is the case you can formulate the return as

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ...$$

Its worth noting that there exist many different types of Markov decision processes.
Episodic Markov decision processes always terminate.
Infinite MDPs take place over countably infinite state spaces and/or action spaces. They might take place over continuous state/action spaces or over continious time steps.
Partially observed MDPs are MDPs with hidden states. The agent at state S_t doesn’t have access to the complete information regarding that state.
We will focus on discrete time /discrete action spaces episodic MDPs (since that’s what games are).

2.1.3 Policies and Value functions

The concept of expected return is used to formulate how good it is for the agent to be in a given state and/or perform an action a at that state.
A policy π is a mapping from states to probabilities of selecting each action. The value of states under policy π is the expected return starting from that state and taking actions according to π.

$$V_\pi(s) = E_\pi[G_t|S_t = s]$$

The value of taking action a in a state s under policy π is called action value function and it is given by the formula

$$q_\pi(s,a) = E_\pi[G_t|S_t = s, A_t = a] = E_\pi[\sum_{k=0}^{\infty} \gamma^k * R_{t+k+1}|S_t = s, A_t = a]$$

2.2 Bellman Equations

The value function can be decomposed in two parts, the immediate reward and the value function of the state the agent ends up in discounted by a factor of γ. Hence

$$V_\pi(s) = E_\pi[G_t|S_t = s] = E_\pi[R_{t+1} + \gamma * G_{t+1}|S_t = s]$$
$$= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma * E_\pi[G_{t+1}|S_{t+1} = s']]$$
$$= \sum_{a \in A(s)} \pi(a|s) * \sum_{s', r} p(s', r|s, a)[r + \gamma V_\pi(S')]$$

for every state s $\in$ S .
Theoretically we could use the bellman equations for each state and solve a linear system of equations to find the optimal state value functions or action value functions.

$$v_*(s) = \max_\pi v_\pi(s), \forall s \in S$$

$$q_*(s, a) = \max_\pi q_\pi(s, a), \forall s \in S$$

However, most interesting problems are of significant size. As a consequence, this approach is computationally impossible. We need to find a way to approximate v or q in order to solve large problems.
Notice that both v_π and q_π can be estimated from experience. For instance, we could follow π , maintain the average of actual returns for every time the agent has visited a state s and use that average to approximate the value for that state. If we let the agent run long enough that average will converge to $v_\pi(s)$. If we keep separate averages for each state action pairs then we can estimate q_π.
Both of the previous approaches are used in Monte Carlo methods which we will examine later.

2.2.1 Policy ordering and optimal policy

- A policy π is defined to be better than π' if its expected return is greater than, or equal to that of π' for each state.
- There is always at least one policy that is better than or equal to all the other policies.
- that is the optimal policy π_*.

2.3 Reinforcement learning Methods

Now we will discuss some of the most basic and widely used methods in reinforcement learning. As you will see, most of RL uses two ”functions”. One evaluates a policy and the other uses those evaluations to improve it. This idea is called Generalised Policy Iteration(GPI)

2.3.1 Dynamic Programming

In practise, classical DP methods are of limited utility. Nevertheless, they are very important for understanding the foundations of other algorithms.
We usually assume that the environment is a finite MDP and then use the bellman equations to evaluate a policy. After that we improve that policy and reevaluate it and so on. Since exact solutions are usually impossible for large problems, we will try to provide iterative solutions. At each iteration k, we use V_k to update V_{k+1} for each state s. The new value of s is given by the old values of it's successors plus the expected immediate one step reward. As k approaches infinity, the sequence of approximate value functions $v_0, v_1, ...$ converges to v_π. This kind of operation is called EXPECTED UPDATE.

Algorithm 1 Iterative Policy evaluation

```
Data: $\pi$ the policy to be evaluated
Result: V $\approx V_\pi$
V(s) $\leftarrow$ 0 for each state s in S
repeat
    $\Delta \leftarrow$ 0
    foreach s in S do
        V $\leftarrow$ V(s)
        V(s) $\leftarrow \sum_a \pi(a|s) * \sum_{s',r} p(s',r|s,a)[r + \gamma * V(s')]$
        $\Delta \leftarrow$ max( $\Delta$,|v-V(s)|)
    end
until $\Delta < \theta, \theta$ is a small positive number;
```

Now, using the previous algorithm we can find an optimal policy. We start off with a random policy and then use it to evaluate the current policy. For each state, we choose "Greedily" the best action and check if the action differs from the action of the policy. If that is the case, then we have improved over the original policy. When we go through all the states without changing a single action, we have discovered an optimal policy.

Algorithm 2 Policy Iteration

```
Result: V ≈ $V_*$ and $\pi \approx \pi*$
1)
initialise V(s) $\in \mathbb{R}$ and π(s) $\in$ A(s) arbitrarily for all states s
2)
Iterative Policy Evaluation for π
3)
policyStable $\leftarrow$ True
foreach s $\in$ S do
    oldAction $\leftarrow$ π(s)
    π(s) $\leftarrow$ argmax$_a \sum_{s',r} p(s',r|s,a)[r+\gamma * V(s)]$
    if oldAction != π(s) then
        policyStable $\leftarrow$ False
    end
end
4) if policyStable then
    return
end
Go back to step 2
```

While policy iteration converges, for each iteration we do a full sweep of policy evaluation. Therefore, the algorithm can be very slow. There is also an algorithm called value iteration. In that algorithm, policy evaluation is stopped exactly after one step .

Algorithm 3 Value Iteration

```
Result: $\pi \approx \pi*$
Initialize V(s) =0 for all s $\in$ S
repeat
    $\Delta \leftarrow$ 0
    foreach s in S do
        v $\leftarrow$ V(s)
        V(s) $\leftarrow$ max$_a \sum_{s',r} p(s',r|s,a)[r+\gamma * V(s')]$;
        $\Delta \leftarrow max(\Delta, |v - V(s)|)$
        end
    until until $\Delta < \theta$;
    π(s)=argmax$_a \sum_{r,s'} p(s',r|s,a)[r+\gamma * V(s')]$
```

Both DP methods however require operating over the entire state space and that can be very time consuming. In Asynchronous DP we update the estimates only for a subset of states rather than the entire state space at every policy evaluation iteration.
If the reader is interested in learning more about dynamic programming methods, they are advised to study [4].

2.3.2 Monte Carlo Methods

The most import assumption Monte Carlo(MC) methods make is that the task at hand is episodic. That means that every sequence of states and actions pairs terminates.

MC methods don't require complete knowledge of the environment, instead they use the average of samples of the return value G for each action pair. Moreover unlike DP and TD learning(which we will soon see) MC methods do not bootstrap. The estimates for every state are based on actual sampled experience. There are two different versions of Monte Carlo prediction.

- First visit MC. $V_\pi(s)$ is estimated as the average of returns following the first visits to s(s might be visited multiple times in the same episode).
- Every visit MC. $V_\pi(s)$ is the average of returns following all visits to s .

It is worth noting that first visit Monte Carlo methods have been studied since the 1940s. As the number of visits increases towards infinity both methods approach V_π.
Monte Carlo methods, can also face a serious problem as we shall see. Every state-action pair s,a is said to be visited if when the agent visits s it takes action a. Therefore, some pairs might never be visited if we act "greedily" during policy iteration. This problem/trade-off is known as exploration vs exploitation and it's extremely often encountered in reinforcement learning. One solution would be the exploring starts assumption. We could specify that the episodes start in s,a pair and that every pair has a nonzero probability of being chosen as start. While this could work, usually the ε-greedy approach is used. At each state, we pick an action greedily with probability 1-ε. Otherwise, we pick an action randomly.
All in all, starting with a random policy, we use either first visit MC or every visit MC for policy evaluation and (ε-)greedy or exploring starts to improve our policy.
Bellow I will provide two pseudocode segments. The first describes first visit MC prediction and the second on-policy control using first visit MC and ε-greedy policy.

Algorithm 4 First Visit MC prediction

```
Data: policy π to be evaluated
Result: $V \approx v_\pi$
Initialize returns$\leftarrow$an empty list for all s $\in$ S
repeat
    Generate an episode using $\pi$
    foreach state s in episode do
        G $\leftarrow$ the return that follows the occurence of S
        Append G to returns(s)
        V(s)=average(returns(s))
    end
until forever;
```

Algorithm 5 MC Control

```
Result: policy $\pi \approx \pi*$
Initialisation:
 foreach s,a $\in S, A(s)$ do
    Q(s,a) $\leftarrow$ arbitrary
     Returns(s,a) $\leftarrow$ empty list
     $\pi(a|s)$$\leftarrow$ a random policy
end
repeat
    Generate an episode using $\pi$
    foreach pair s,a in the episode  do
        G $\leftarrow$ the return that follows the first occurence of s,a in the episode
        append G to returns(s,a)
        Q(s,a) $\leftarrow$ average(returns(s,a))
    end
    foreach s in the episode do
        $A^* \leftarrow$ arg max$_a$ $Q(s,a)$
        foreach a $\in$ A(s)  do
            if a is Optimal then
                $\pi(a|s) = \epsilon/|A(s)|$
                else
                    $\pi(a|s) = 1 - \epsilon + \epsilon/|(s)|$
                    end
                end
            end
        until forever;
```

As you can notice, in order to perform control with MC methods, we have to run an entire episode before we can improve our policy. That is not the case with temporal difference learning as we will see.

2.3.3 Temporal Difference Learning

TD methods can be thought as a compromise between Monte Carlo methods and dynamic programming. In fact,when the state space is very large,variants of the TD method may be the only feasible option.
We first learn from some rewards and then we bootstrap(that is, guess the value of a state), without having to wait until the end of the episode like MC. Both TD and MC updates are sample updates, whereas DP updates are based on a complete distributions over all possible successors of each state. Therefore both TD and MC can be significantly faster than DP.
First we are going to examine the simplest case of the TD(0) algorithm, and then we are going to extend our discussion about n-step bootstrapping, TD(λ) and eligibility traces.
The quantity $R_{t+1} + \gamma * V(S_{t+1})$ is called the target for the TD(0) update , and the TD error is $R_{t+1} + \gamma * V(S_{t+1}) - V(S_t)$. The update rule for the TD(0) algorithm therefore is given by

$$V(S_t) \leftarrow V(S_t) + a * TDError(S_t)$$

It is worth noting that, a very important open question in reinforcement learning is the following.
" Since both TD and MC methods converge to V_* which is faster?"
Bellow is displayed the basic TD(0) algorithm for policy evaluation.

Algorithm 6 TD(0) policy evaluation

```
Data: policy π to be evaluated
Result: V $\approx v_*$
V(S)=0 for each s in S
foreach episode do
    Initialize S
    foreach step of the episode do
        A $\leftarrow$ action given by $\pi$ for S
        Take action A and observe S' and R
        V(s) $\leftarrow$ V(s) + α *[R+$\gamma V(S') - V(S)]$
        $S\leftarrow$ S'
        if S is Terminal then
            break
        end
    end
end
```

2.3.3.1 n step Bootstrapping

n step Bootstrapping TD is a compromise between MC and TD. The goal is to be able to easily transition from MC to TD(0) and vise versa, as smoothly as possible depending on the task at hand. The agent takes n steps forward, stores the rewards and makes a guess. For instance the 3 step update would be based on the first 3 rewards and the estimated value of the state 3 steps later.
As we already know, the n step return is given by

$$G_{t:t+n} = R_{t+1} + \gamma * R_{t+2} + \gamma^2 * R_{t+3} + ... + \gamma^{n-1} * R_{t+n} + \gamma^n * V(S_{t+n})$$

Therefore the n step TD update rule becomes

$$v(S_t) \leftarrow V(S_t) + \alpha * [G_{t:t+n} - V(S_t)]$$

2.3.3.2 Averaging over n step returns

This very popular method of temporal difference learning is often denoted as TD(λ) and it combines all of the n step returns .

$$G_{t:\lambda} = (1 - \lambda) * \sum_{n=1}^{\infty} \lambda^{n-1} * G_{t:n}$$

therefore, the update rule becomes

$$V(S_t) = V(S_t) + \alpha * [G_{t:\lambda} - V(S_t)]$$

This is called forward view TD(λ) and can be computationally expensive since we have to run entire trajectories. Luckily for us, there is an equivalent backward view TD(λ). But first,we have to discuss eligibility traces.

2.3.3.3 Eligibility traces

Imagine you are playing a board game or a card game against a player X. Player X smiles and does a certain move. Then they smile again and do the same move. After 2,3 moves you see them smile. What move do you think they are going to play? Now imagine,a lot of rounds go by and the player plays that move without smiling a few times. You see them smile again, is your assumption about their following move the same?
Eligibility traces assign credit to the most frequent states but if they have not occurred recently, that credit gradually decays. Mathematically:
$E_0(s) = 0, \forall s \in S$
$E_t(s) = \gamma * \lambda * E_{t-1}(S) + 1(S_t = S)$

2.3.3.4 Backward view TD (λ)

All in all, we keep an eligibility trace for each state s. The update rule now becomes

$$V(S_t) \leftarrow V(S_t) + a * TDError * E_t(s)$$

When λ=0, we are following the TD(0) method we discussed previously. When λ=1 TD(1) is the same as Monte Carlo methods.

2.3.3.5 Sarsa , Q learning and expected Sarsa

There are two different variations of control

- on policy control. That is, we follow a policy and evaluate it .
- off policy control. That is, we follow a policy while estimating the value of a target policy

First, we will examine the SARSA(State Action Reward State Action) algorithm, which performs on policy control. Unlike previous methods where we considered state to state transitions and learned the value of states, here we examine the values of state-action pairs. The update rule therefore is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha * [R_{t+1} + \gamma * Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

And if S_{t+1} is terminal we consider $Q(S_{t+1}, A_{t+1})$ to be zero.
The agent estimates q_π and then updates policy π using the ε-greedy method.

Algorithm 7 Sarsa

```
Initialise Q(s,a) for every action state pair
All terminal States are initialised to Zero
foreach episode do
    Initialise S
    Use the Q values to choose an action A(usually following an $\epsilon$- Greedy policy π )
    repeat
        Take action A and observe S’ ,R
        Choose action A’ from S’ using the Q values(ε-greedy)
        Q(S,A) $\leftarrow$ Q(S,A) + a*[R+ $\gamma$Q(S’,A’)-Q(S,A)]
        (S,A) $\leftarrow$ (S’,A’)
    until S is terminal;
end
```

The Q learning algorithm, on the other hand, performs off policy control. The agent runs a (sub-optimal) policy while ”greedily” improving another policy. All that is required for convergence, is that all state action pairs continue to be updated. If that is true then the algorithm converges to q_*.
The update rule is :

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha * [R_{t+1} + \gamma * \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

Algorithm 8 Q learning

```
Algorithm Parameters step size a $\in [0, 1]$ , small $\epsilon > 0$
Initialise Q(s,a) for each state s and it’s actions A(s)
Q(terminal,.) =0
foreach episode do
    Initialise S
    repeat
        Choose action A from S using policy derived from Q values
        Take action A observe reward R and state S’
        Q(S,A) $\leftarrow$ Q(S,A) + $\alpha$ [R + $\gamma$ max$_a Q(S', a) - Q(S, A)]$ $S \leftarrow$ S’
    until Until S is terminal;
end
```

There is also a variation of SARSA, that works just like Q learning, but instead of finding the maximum over the next state action pair, it uses the expected action value as a target. Therefore the update rule becomes

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha * [R_{t+1} + \gamma * E[Q(S_{t+1}, A_{t+1}|S_t)] - Q(S_t, A_t)]$$

That algorithm is called expected SARSA. While it is more computationally expensive, it eliminates the variance introduced by the selection of A_{t+1}. In practise, it usually performs slightly better than SARSA.

2.3.3.6 Double Q learning

If the action values contain random errors or if there is noise in the environment itself, traditional Q learning can overestimate it's targets leading to sub optimal policies.
Q learning uses the same parameters θ both for the selection and evaluation of an action. Double Q learning uses two (sets of) parameters, one is used for determining the greedy policy and the other for evaluating it. The target for the classic Q learning algorithm can be written as $y_t^Q = R_{t+1} + \gamma Q(S_{t+1}, arg\, \text{max}_a\, Q(S_{t+1}, a; \theta_t); \theta_t)$ The target for double Q learning is
$y_t^Q = R_{t+1} + \gamma Q(S_{t+1}, arg\, \text{max}_a\, Q(S_{t+1}, a; \theta_t); \theta_t')$
and only θ_t is updated at each time step.
θ_t' can be updated by symmetrically switching the roles of the parameters. In fact, in the original implementation of double Q learning, the algorithm randomly chooses at each time step whether to update θ_t or θ_t'

2.3.4 Model Based Reinforcement learning and planning

So far, we have examined model free methods in reinforcement learning. For the rest of section 2.3 we will examine model based reinforcement learning.
Model based RL combines planning and reinforcement learning.
As a model we define anything an agent can use to predict how the environment will react to an action. For instance, it might take a state s and an action a and output the new state s' (perhaps not the actual next state) and the predicted reward. As model based RL we define any approach to solve a mdp problem using a model and learning to approximate a global value or policy function. It is worth noting that in the planning and search community the theoretic framework behind model based reinforcement learning was described as learning real-time A * [11] .
There are three main categories of model based RL

- Model based RL with a learned model. We learn a model and a value or policy. A well known example is the dyna algorithm which we will discuss shortly.
- Using an already known model for planning to learn a global value or policy function. This approach was used in AlphaGO zero
- learning about a model and using it to plan LOCALLY without estimating global v or π .

2.3.4.1 Learning the dynamics of a model using observed data

This is the first step of performing model based RL. In control literature this is called system identification. There are different types of model an agent can learn.

- A forward model predicts a future state given current state and action. It is used very often.
$$(s_t, a_t) \rightarrow (s_{t+1})$$
- A backward/reverse model predicts possible precursors.
$$(S_{t+1}) \rightarrow (S_t, a_t)$$

- An inverse model predicts an action given a state and it's successor.

$$(S_t, S_{t+1}) \rightarrow a_t$$

A second distinction is about what supervised machine learning method we will use .

- Parametric approach (e.g linear regression or Neural Networks)
- Non parametric approach (e.g decision trees or Nearest neighbor)

Parametric approaches are by far the most popular.

2.3.4.2 Using the model to plan

The goal of this step is to use the model to recommend an action or improve a policy. There are quite a few considerations for planning.

- At which state do we start planning?
  We can choose the state randomly like in DP. Unfortunately, that approach does not scale well in high dimensions. We can also start only from already visited states. Therefore we only examine reachable states instead of the entire state space. We might also assign some states a priority. We will discuss about the prioritised sweeping algorithm in the next sections. Finally if our aim is to find more local information, we can only start from the current state.
- When do we start planing ? The number of real steps before a planing cycle .
- How much "Budget" is devoted on planning ?
- How to plan?
  In discrete planning which is the main approach to classic AI, discrete states are stored in a tree or an array and used to update V,Q or π. This is suitable for games. Some popular methods include the infamous minmax algorithm and Monte-Carlo tree search(MCTS).
  Differential planning (better known as value gradients in RL community) is a gradient based approach and requires a differential model of the environment. It is very popular in robotics but less applicable to discrete problems.

There are many more categories of planning methods. The interested reader can study [14].

2.3.5 Dyna

Dyna is a general framework for model based reinforcement learning. We will provide the basic idea behind it and then we will discuss a simple example of the Dyna Q algoritm.
Dyna combines planning,acting and learning in one algorithm. Real experience of the environment can be used for two things. It can be used to improve the value function or the policy using any reinforcement learning algorithm(direct RL). It can also be used to improve the model. More accurate models can also improve the value function or the

policy indirectly(indirect RL). Both of these approaches have some advantages and some disadvantages. Dyna combines them in a single algorithm.
In an ideal setting, as the agent interacts with the environment, the following happen simultaneously.

- The agent uses the experience gained and an RL algorithm to improve its value function or policy.
- The agent uses the same experience to improve the model.
- Simulated experience from the model is also used to improve the value function or policy through planning.

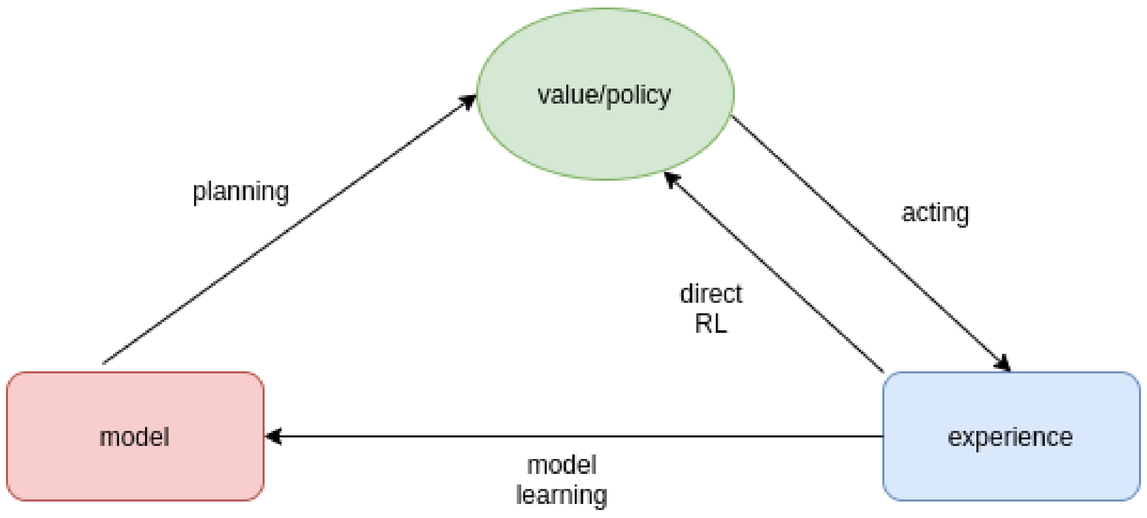


Figure 2.1: Integration of acting, learning and planning in Dyna

In a real implementation, we have to specify the order in which these happen. In the dyna-Q algorithm direct reinforcement learning using tabular Q learning is performed first. Model learning is then performed. The model is also table based and assumes the environment is deterministic. Planning takes place last.

Algorithm 9 Tabular dyna Q

```
Initialize Q(s,a) and Model(S,a) for all state action pairs
repeat
    S ← current state
    A ← ε-greedy(Q,S)
    Execute A , observe R and S'
    Q(S,A) ← Q(S,A) +α*[R+γ*max$_a Q(S', a) - Q(S, a)$]
    $Model(S, A)$← R,S'
    foreach i in range(0,n) do
        S ← random previously observed state
        A ← random action previously taken from S
        R,S' ← Model(S,A)
        Q(S,A) ← Q(S,A) +α*[R+γ*max$_a Q(S', a) - Q(S, a)$]
    end
until forever;
```

2.3.6 Prioritized sweeping

Uniform selection of state action pairs can be inefficient. For a lot of tasks, focusing on ”good” pairs can lead to much better performance. During planning, a lot of state action pairs examined yield very little rewards, hence evaluating them can be a waste of computational resources. We want to move backwards from states whose value has changed significantly. If a state's value remains the same over an extended period of time then we have probably picked a very good action. Also, if a state's value changes, chances are the values of it's predecessors will also change.
We can maintain a priority queue for every state action pair whose value changes significantly. At each step, we pop the top pair in the queue and calculate the effect updating it would have on it's predecessor pairs. For each of the later, if it is greater than a threshold we push that pair back in the queue with a new priority(if it is already in the queue).

2.3.7 Rollout algorithms

Rollout algorithms are decision time algorithms. They begin at the **current state**, stimulate many (Monte-Carlo) trajectories and average their returns. The average is used to update the value of the current state. Then, the agent picks the action that will move them to the state S' with the highest value. Notice that unlike traditional Monte-Carlo methods, we do not have to compute q* or q_π, we only produce estimates for the current state and a given policy called **rollout policy**. By picking the best action for the current state we perform **one step policy iteration**.
Unfortunately, simulating many trajectories can be slow. Luckily, we can sample many trajectories in parallel and/or perform pruning to speed up the process.
Rollout algorithms do not maintain long term memories of values or policies(we make immediate use of value calculations and then discard them) hence they are not considered learning algorithms.

For more information about rollout algorithms , the user can study [3].

2.3.8 Monte Carlo Tree search

Monte Carlo Tree search is a rollout algorithm with a twist. Instead of discarding value estimates from MC stimulation, we store them in order to direct simulations towards more highly rewarding trajectories.
MCTS builds and uses a tree in the following manner:

- **Selection step** Traverse the tree of already discovered states balancing exploration and exploitation(perhaps ε-greedy). The edges of the graph represent action values.
- **Expansion step** When the game reaches a state not on the tree, add that state to the tree as a new node.
- **Simulation** Simulate the rest of the episode randomly using the rollout policy (it is possible to use heuristic knowledge).
- **Backpropagation** Using the return of the simulated episode, update all the edges of the tree visited. No values are saved for the states and actions visited by the rollout policy.

MCTS has been applied to board games with tremendous success outperforming alpha beta pruning.

2.3.9 Function Approximation

So far, we have used tabular methods (where a value function or a policy is stored in a tabular form). This approach has to very serious problems. First of all, in most interesting problems there can be too many states or state-action pairs to store. In addition, even if we store them, learning can be to slow. To deal with this problem we will try and ”learn” a function $\widehat{V}$ that approximates V_π or a function $\widehat{Q}$ that approximates Q_π. The three most common types of value function approximation are :

- from state s to $\widehat{V}(s,w)$
- from state-action pair (s,a) to $\widehat{q}(s,a,w)$
- from state s to $\widehat{q}(s,a_1,w)...\widehat{q}(s,a_m,w)$ where $\{a_1,a_2,...,a_m\} = A(s)$

We also use a feature vector for each state$X(s) = \begin{bmatrix} X_1(S) \\ X_2(S) \\ ... \\ X_n(S) \end{bmatrix}$ where the features might be the positions of all the pieces in chess or the distance from 4 walls in robotics. Then using that feature vector and the targets of each RL algorithm , we solve a regular regression problem. It is also worth noting, that unlike traditional supervised learning , our target values might change as our policy improves. In the next three subsections we are going to examine some examples of function approximation on traditional algorithms.

2.3.9.1 Monte Carlo Methods

As we already know, MC target is the total return G_t. We will use the sequence of states and their returns as training data and we will try minimise the empirical error:

$$J(w) = \sum_{i=1}^{T} (G_i - \widehat{V}(S_i))^2$$

The update rule is

$$w \leftarrow w + \alpha * [G_t - \widehat{V}(S, w)] * \frac{d\widehat{V}(S, w)}{dw}$$

For the simplest model of linear regression, the value function approximation becomes

$$V(S, w) = w^T * X(S)$$

and the derivative with respect to w

$$\frac{d\widehat{V}(S, w)}{dw} = X(s)$$

Algorithm 10 MC Evaluation with Gradient Descent

```
Data: A policy $\pi$ to be evaluated and a differentiable function $\widehat{v}$
Result: $\widehat{v} \approx v_\pi$
Initialise weights w =0
repeat
    Generate an episode $S_0, A_0, R_1, S_1, A_1, ..., R_T, A_T$ using $\pi$
    foreach t=0,1,...,T-1 do
        w $\leftarrow w + \alpha * [G_t - \widehat{V}(S, w)] * \frac{d\widehat{V}(S,w)}{dw}$
    end
until forever;
```

2.3.9.2 TD(0)

The target now is $R_{t+1} + \gamma * \widehat{V}(S_{t+1}, w)$ therefore the error becomes

$$J(w) = \sum_{t=1}^{T} (R_{t+1} + \gamma * \widehat{V}(S_{t+1}, w) - V(S_t))^2$$

If we are at a state S, take action A, observe reward R and end up at state S', the gradient descent update rule becomes:

$$w \leftarrow w + a * [R + \gamma * \widehat{V}(S', w) - \widehat{V}(S, W)] \frac{d\widehat{V}(S, W)}{dW}$$

2.3.9.3 Backward view TD(λ)

For each state S_t The TD error is :

$$\delta_t = R_{t+1} + \gamma * \widehat{V}(S_{t+1}, w) - \widehat{V}(S_t, w)$$

and the eligibility trace is

$$E_t = \gamma * E_{t-1} + x(S_t)$$

Therefore the update rule at that state is

$$w \leftarrow w + a * \delta_t * E_t$$

2.3.10 Policy Gradient Methods

So far, we have discussed algorithms trying to approximate the value, or action-value function. PGM try to approximate the optimal policy using a parametric function $\pi(a|s,\theta)$ and performing gradient ascent. At each time step t

$$\theta_{t+1} \leftarrow \theta_t + a * \frac{d\widehat{J}(\theta_t)}{d\theta}$$

where $\frac{d\widehat{J}(\theta_t)}{d\theta}$ is a stochastic estimate of the performance measure's gradient.
We want actions with the highest performance to be assigned the highest probability, while still maintaining some exploration. We can use the softmax policy

$$\pi(\alpha|s,\theta) = \frac{exp(h(s,a,\theta))}{\sum_b exp(h(s,b,\theta))}$$

Where h can be a neural network or a linear function approximator, or any other machine learning model.
Unlike action-value methods, policy gradient methods can actually find stochastic policies. They also have stronger convergence guarantees.

2.3.10.1 Policy gradient theorem

We assume that we are in an episodic task and that every episode starts at state s_0. We define the performance as

$$J(\theta) = V_{\pi_\theta}(s_0)$$

The policy gradient theorem establishes that

$$\nabla J(\theta) \propto \sum_s \mu(s) * \sum_a q_\pi(s,a) * \nabla_\theta \pi(a|s,\theta)$$

where $\mu(s)$ is called on-policy distribution under π and is given by

$$\mu(s) = \frac{\text{number of time steps spent on s on average}}{\sum_{s'} \text{number of time steps spent on s' on average}}$$

2.3.10.2 REINFORCE ALGORITHM(Monte-Carlo policy gradient)

Since μ is a distribution, we can rewrite the policy gradient theorem as

$$\nabla_\theta J(\theta) \propto E_\pi[\sum_a q_\pi(S_t,a) \nabla_\theta \pi(a|S_t,\theta)]$$

If we now replace a with sample A_t we get

$$\nabla_\theta J(\theta) = E_\pi[G_t * \frac{\nabla_\theta \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}]$$

Therefore, the stochastic gradient ascent rule is:

$$\theta_{t+1} \leftarrow \theta_t + a * G_t \frac{\nabla_\theta \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)}$$

the fraction can be also written as $\nabla_\theta ln\pi(A_t|S_t,\theta)$ and is usually referred to in literature as **eligibility vector**.
The algorithm using this update rule is called REINFORCE. Since it uses complete returns from time t it is a Monte-Carlo algorithm and it is well defined only for episodic tasks.
If we are using linear action preferences , while following a softmax policy the eligibility vector can be written as

$$\nabla_\theta ln\pi(a|s,\theta) = X(s,a) - \sum_b (\pi(b|s,\theta) * X(s,b)$$

It is important to note that although REINFORCE has good theoretical properties it can produce slow learning.

2.3.10.3 REINFORCE with baseline

The policy gradient theorem can be modified to compare the action value with an arbitrary baseline $b(s)$

$$\nabla J(\theta) \propto \sum_s \mu(s) * \sum_a (q_\pi(s,a) - b(s)) * \nabla_\theta \pi(a|s,\theta)$$

Using similar steps as the previous section, the update rule becomes

$$\theta_{t+1} \leftarrow \theta_t + a * (G_t - b(S_t)) \frac{\nabla_\theta \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)}$$

The baseline leaves the expected value of the update unaffected, but it can significantly reduce the variance. As a result , it can significantly speed up the learning process. Usually, we choose a parametric estimate of v(s), $\widehat{v}(s,w)$ as the baseline function. The weight vector w can be learned using one of the function approximation methods discussed previously.

2.3.10.4 ACTOR CRITIC METHODS

The goal of actor critic methods is to improve the REINFORCE algorithm by introducing bias through bootstrapping. One step actor critic methods replace the MC full return of REINFORCE with one step immediate return $+$ prediction. Just like TD methods, we can choose the degree of bootstrapping and include eligibility traces. The update rule for one step bootstrapping is :

$$\theta_{t+1} \leftarrow \theta_t + \alpha * (R_{t+1} + \gamma \widehat{v}(S_{t+1},w)) - \widehat{v}(S_t,w)) * \frac{\nabla_\theta \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)})$$

The actor critic algorithm maintains two sets of parameters

- A critic that maintains action value parameters w as we have discussed in the previous subsections.
- An actor that updates θ as suggested by the critic(Since the critic modifies the estimate of v, they direct the updates of θ).

The pseudocode for Actor Critic with one step bootstrapping can be found bellow.

Algorithm 11 One step Actor Critic(Episodic)

```
Data: $\pi$(a|s,$\theta$) , $\widehat{v}(s, w)$
Two different stepsizes $a_\theta > 0$ and $a_w > 0$
Initialise policy parameter $\theta$ and state value weights $w$
while 1 do
    Initialise S
    I $\leftarrow$ 1
    while S not terminal do
        Take action a according to parametric policy $\pi$
        Observe S',R
            /* if S' is terminal then $\widehat{v}(S', w) = 0$                    */
        $\delta \leftarrow$ R $+\gamma * \widehat{v}(S', W) - \widehat{v}(S, w)$
        w $\leftarrow$ w + $a_w * I * \delta * \nabla_w \widehat{v}(S, w)$
        $\theta \leftarrow \theta + a_\theta * I * \delta * \nabla_\theta ln\pi(A|S, \theta)$
        I $\leftarrow \gamma * I$
        S $\leftarrow$ S'
    end
end
```

There is also a variation of the actor critic algorithm in which the actor calculates an advantage function A along with the value function.

2.3.11 Trust Region Policy Optimisation

When using policy gradient methods, large updates of the policy parameters can guide the agent to follow poor policies. If this happens a few times, the model can be trapped to following poor policies for ever. Even if the model is initially following a good policy, large updates of the policy parameters can force it to ”forget”.
The goal of TRPO [17](and PPO [18]) is to force the model to make ”small” updates of the policy parameters. There is a extensive mathematical background behind these algorithms, which is beyond the scope of this paper.
TRPO maximises an objective function called surrogate objective subject to the size of the policy update.

$$\max_\theta E[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)} A_t - \beta KL[\pi_{\theta old}(.|s_t), \pi_\theta(.|s_t)]$$

where θ_{old} is the set of parameters before the update , KL is the Kullback–Leibler divergence and β is a hyper-parameter. The problem with this maximisation is that it is hard to choose a value for β that performs well in different tasks(e.g atari games and robotics). That's why TRPO solves the following problem instead :

$$\max_{\theta} E[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)} * A_t]$$

subject to

$$E[KL[\pi_{\theta old}(.|s_t), \pi_\theta(.|s_t)] \leq \delta$$

where δ is a positive number

2.3.12 Proximal Policy Optimisation

2.3.12.1 Clipped Surrogate Objective

The main objective is

$$L^{CLIP}(\theta) = E_t[min(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)} A_t, clip(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}, 1-\epsilon, 1+\epsilon) * A_t)]$$

The first term is the same as TRPO. ϵ is a hyperparameter used to keep $\frac{\pi_\theta}{\pi_{\theta old}}$ in a certain range. Because of the min operator the final objective is a pessimistic(lower) bound on the unclipped objective.
All in all , L^{CLIP} is a lower bound on the TRPO objective, with a penalty for having too large policy update

2.3.12.2 Adaptive KL Penalty Coefficient

A penalty can be used to keep KL divergence close to a target value d_{targ}. In each policy update, the algorithm performs the following two steps :

- Optimize
  $$L^{KPEN} = E_t[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)} A_t - \beta * KL(\pi_{\theta old}(.|s_t), \pi_\theta(.|s_t)]$$
  with several SGD epochs
- Compute $d = E_t[KL(\pi_{\theta old}(.|s_t), \pi_\theta(.|s_t))]$
  if d is smaller than $d_{targ}/1.5$ then $\beta \leftarrow \beta/2$.
  if d is larger than $d_{targ} * 1.5$ then $\beta \leftarrow \beta * 2$

In practise the algorithm converges regardless of the initial value of β

2.3.12.3 Algorithm

The surrogate loss from the previous sections can be computed from traditional policy gradient methods. If using automatic differentiation, one simply replaces traditional policy gradient objective with CLIP or KLPEN objective and performs gradient ascent. A combination of both is also possible. Many techniques also combined a learned value function V(s).

Both TRPO and PPO can be used for discrete and continuous action spaces, and they can be improved by simulating multiple environments in parallel. In our experiments on the Friv Learning Environment (chapter 4) we notice that Proximal Policy Optimisation uses up far less memory and CPU resources than Deep Q learning(2.5.0.2).

2.4 Neural Networks

2.4.1 Perceptron

The perceptron algorithm was developed by Rossenblat in 1962 and it has a very important historical value to the field of machine learning. It is a binary linear classifier that aims to learn a threshold function separating the data vectors in 2 classes.
The perceptron function is given by the following formula:

$$f(x_i) = \begin{cases} 1 & \text{if } w^T * x_i + b \geq 0 \\ \text{-1,} & \text{otherwise.} \end{cases}$$

x_i is a training example and b is called bias. Usually the target label is denoted as t_i.
The algorithm aims to discover a vector w such that for the data points x_i in the first class $f(x_i) \geq 0$ and for the rest $f(x_i) < 0$. Consequently, if a data point is correctly classified the following will be true

$$(w^T x_i + b) * t_i \geq 0$$

Therefore, the algorithm aims to minimise the following quantity called perceptron criterion

$$E_P(x) = -\sum_{x_i \in D} w^T * (w^T * x_i + b) * t_i$$

Where D is the set of all the incorrectly classified elements. The stochastic gradient descent update rule on the parameter w is

$$w^{(\tau+1)} = w^{(\tau)} + a\nabla_w E_P(w) = w^{(\tau)} + (w^{(\tau)T} x + b) * t \tag{2.1}$$

The perceptron training algorithm can be naturally interpreted in the following manner. Looping through each element x_i calculate the perceptron function. If it is correctly classified then the weight vector remains unchanged. If it is incorrectly classified then remove $w^T * x_i + b$ from the weight vector.
If an exact linearly separable solution exists, the algorithm is guaranteed to converge in a finite number of steps.

2.4.2 Feed forward neural networks

Deep feed forward neural networks are often called multi layer perceptrons (MLP) because the are a network of perceptron classifiers. They are the most simple yet most important neural network architecture and they consist of multiple layers from 2, to several thousands. Other more complex architectures are built on top of them.
The first layer is called input layer and the last layer is called output layer. Every layer in between is called hidden layer. Each layer consists of several nodes. Each node of a

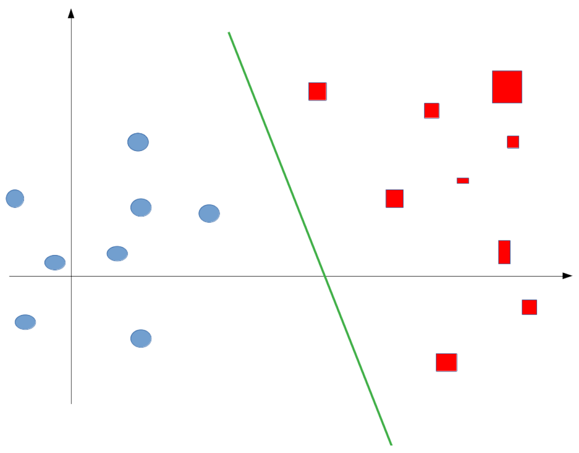


Figure 2.2: perceptron algorithm applied on linearly seperable data

layer is usually connected with all the nodes of the previous layer. The values of those connections are called weights. If a node n is connected with nodes 1,2,...m trough weights $w_1, w_2, ..., w_m$ then the output of the node is

$$output(n) = \sum_{i=1}^{m} output(i) * w_i$$

The output of each node usually passes through an activation function. In most modern architectures that is the reLU function it can however be another function such as the sigmoid or the tanh function.
They are called feed forward because computation only moves forward there are no cycles or loops. They can be used for classification and regression (supervised learning) ,as feature extractors for other classifiers ,and they can perform unsupervised learning .

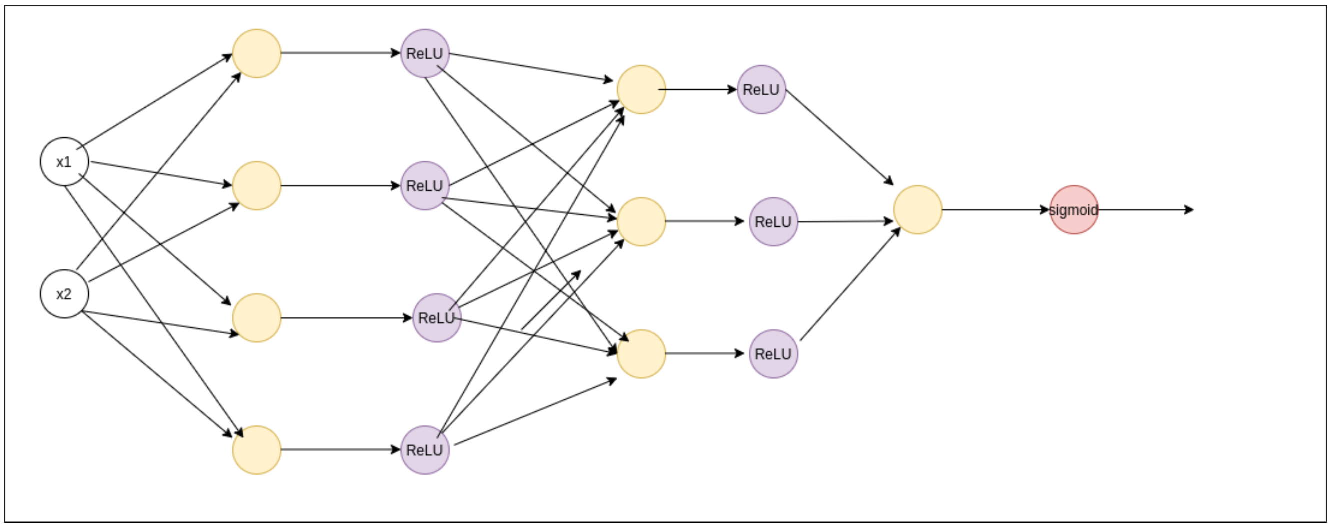


Figure 2.3: A simple Feed Forward Neural Network performing binary classification of 2 dimensional data. It has 2 hidden fully connected layers . The biases at each layer have been emitted from the diagram for simplicity.

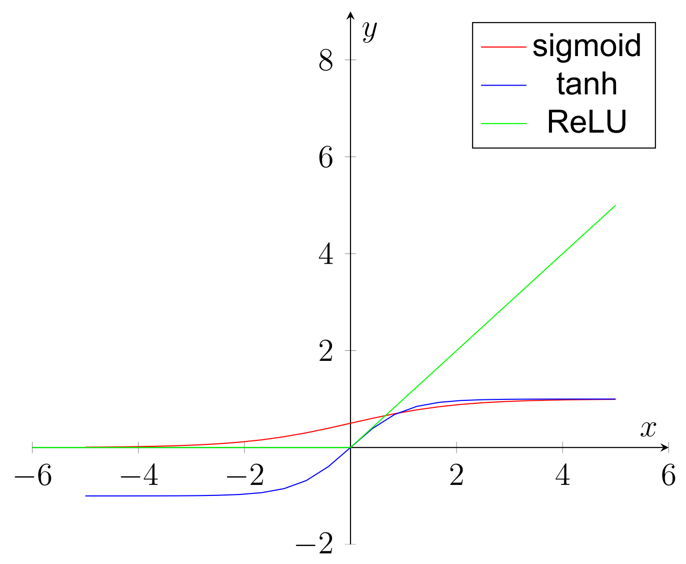


Figure 2.4: The 3 most common activation functions in Neural Networks

2.4.3 Convolutional Neural Networks

Convolutional Neural Networks(often denoted as CNNs or ConvNets) are a class of artificial Neural Networks most often used for visual data (or data with certain properties). They take advantage of hierarchical patterns in data to improve the performance of feed forward neural networks. The input is a tensor with shape (number of inputs x height x width x input channels).
Some of the hidden layers (as well as the input layer) perform convolution operation. That is a dot product of the input matrix with a matrix called convolution kernel. The kernel slides along the input matrix generating another matrix called feature map. The feature map goes through an activation function before being provided to the next layer.
Other layers can be pooling layers. Pooling layers decrease the dimension of the data by combining outputs of multiple nearby neurons into a single value. Today, most pooling layers use the maximum value of each cluster as output (max pooling). Other forms of pooling such as average polling can be used with good results.
Usually a sequence of convolution layers and pooling layers are followed by some feed forward fully connected layers. Other classifiers such as support vector machines can be used instead of the feed forward layers.

2.4.4 Long Short Term Memory Neural Networks

Long Short Term Memory Neural Networks(LSTMs) were initially proposed by Sepp Hochreiter and Juergen Schmidhuber in 1997 to deal with the vanishing gradient problem in regular recurrent neural networks. Vanilla recurrent neural networks can remember very long dependencies in the input sequence causing a lot of practical(computational) problems. While training RNNs long term gradients may become very small or extremely large causing severe numerical instability. LSTMs regulate that problem but they can still suffer from it. Since their initial introduction there have been some improvements such as Peephole LSTMs and Convolutional LSTMs. Since we are not going to use LSTMs in the Friv Learning Environment we are going to discuss about the simple LSTMs with forget gates. The interested reader can view the original paper [10] or their wikipedia page for a briefer

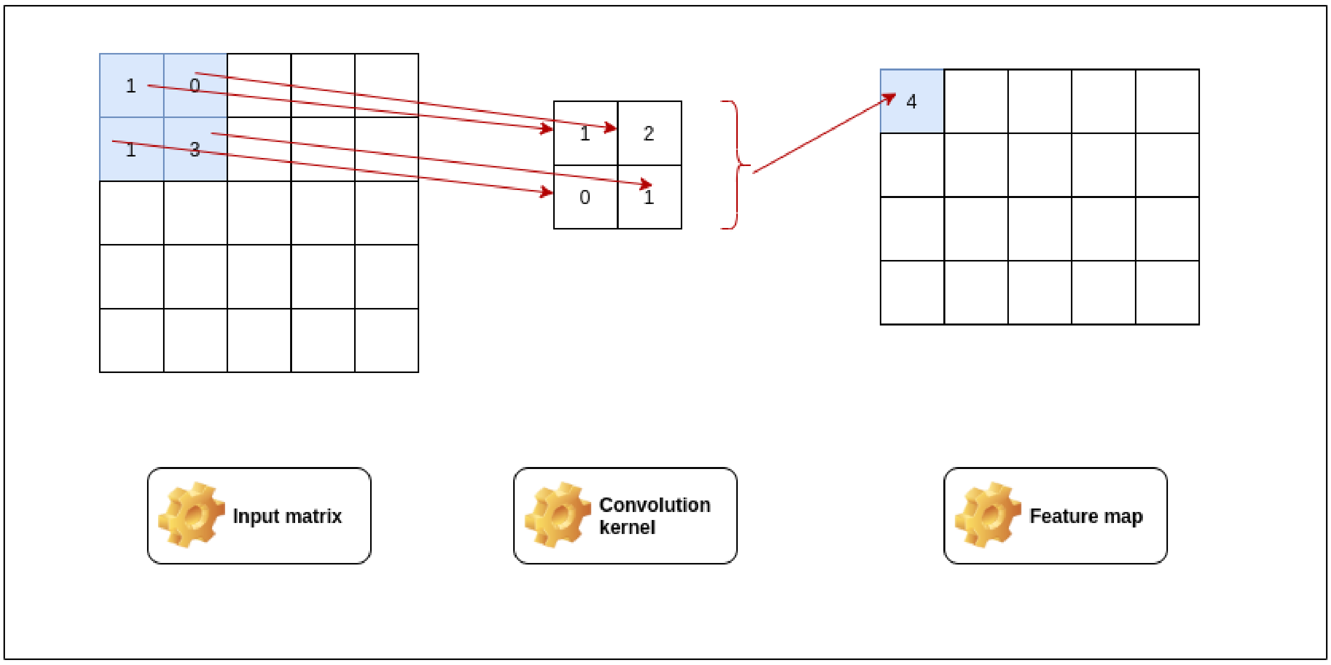


Figure 2.5: Convolution layer example

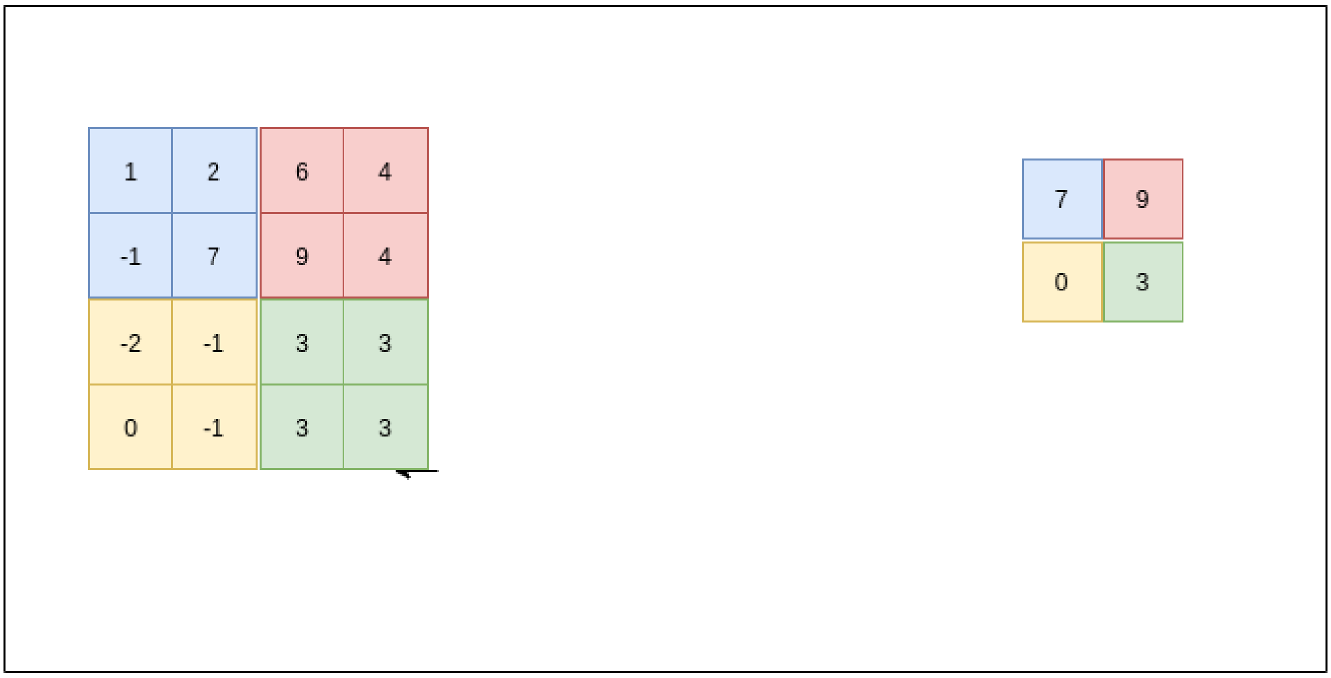


Figure 2.6: pooling layer example. 2x2 kernel , stride=1

introduction.
LSTMs are one of the most important achievements of deep learning and they have successfully been used in various sequence to sequence mapping problems such as machine translation.In the field of Reinforcement learning they are used in partially observed environments.
LSTMs consist of a sequence of units. Unlike feed forward neural networks and CNNs, LSTMs have both feedforward and feedback connections between units. A common LSTM unit consists of a cell an input gate an output gate and a forget gate. The cell remembers values through time and gates deal with the flow of information in and out of the cell.
We will now examine the variables assosiated with a single LSTM cell. A unit can contain

many cells.

- h denotes the number of hidden units, d denotes the dimension of the input
- $x_t \in \mathbb{R}^d$ is the input vector
- f_t denotes the output of the forget gate , i_t denotes the input/update gate's output and o_t denotes the output gate's output.
- h_t denotes the output vector of the LSTM unit
- $\hat{c}_t$ denotes the cell input activation vector
- c_t represents the cell state vector
- $W \in \mathbb{R}^{hxd}$, $U \in \mathbb{R}^{hxh}$ are weight matrices and $b \in \mathbb{R}^h$ is the bias
- s is the sigmoid function and tanh is the hyperbolic tangent function

The equations determining the values of the previous variables during a single forward pass are the following.

$$\begin{gathered}
f_t = s(W_f * x_t + U_f * h_{t-1} + b_f) \\
i_t = s(W_i * x_t + U_i * h_{t-1} + b_i) \\
o_t = s(W_o * x_t + U_o * h_{t-1} + b_o) \\
\widehat{c_t} = tanh(W_c * x_t + U_c * h_{t-1} + b_c) \\
c_t = f_t \circ c_{t-1} + i_t \circ \widehat{c_t} \\
h_t = o_t \circ tahn(c_t)
\end{gathered}$$

As you can see, their values depend on both the current input and the previous output. The initial values c and h are both 0. $\circ$ represents the hadammard product.

2.5 Deep Reinforcement learning

Deep Reinforcement learning(DRL) is used to refer to reinforcement learning with value function approximation, or with policy approximation, where the parametric function approximating the value or policy is a Neural Network.

2.5.1 Deadly Triad

It has been shown from Tsitsiklis and Van Roy in [25] that when the following 3 conditions are combined there is a high probability that instability and divergence of the RL algorithm will occur.

- off policy learning
- function approximation (Especially non linear)

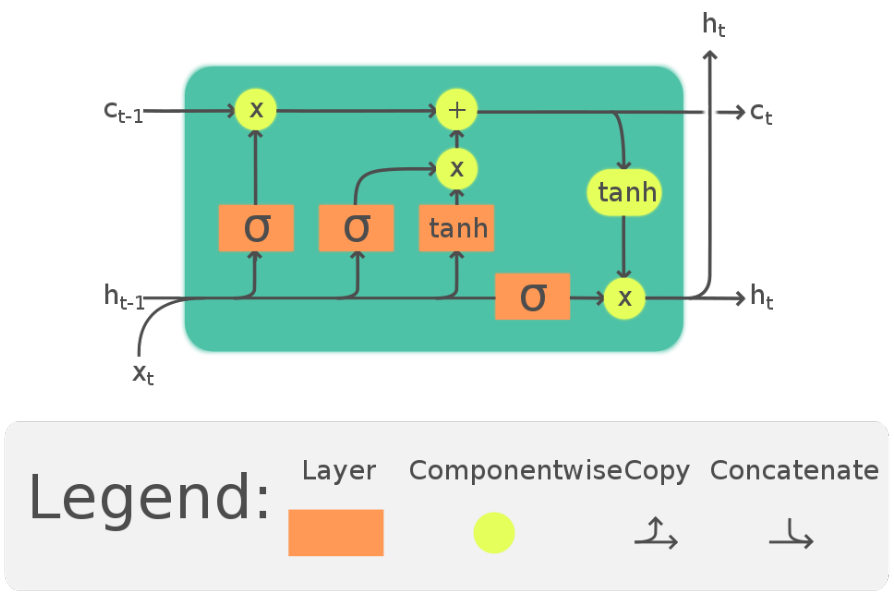


Figure 2.7:
THE LSTM Cell
By Guillaume Chevalier - File:The_LSTM_Cell.svg, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=109362147

- Bootstrapping

These conditions are usually referred to as the deadly triad in RL literature and for years they discouraged RL researchers from using neural networks.

2.5.2 Deep Q learning

Mnih at all introduced DQN in 2015 [13]. DQN uses a convolutional neural network to approximate the optimal action value function

$$Q^*(S_t, a) = \max_{\pi}[R_{t+1} + \gamma * \max_{a} Q(S_{t+1}, a)]$$

DQN uses experience replay and target networks to address instability issues.
First, we will discuss experience replay. Replay buffer(sometimes referred in literature as replay memory, or simply experience replay) is a fixed size buffer usually implemented as a circular buffer. The oldest transition is removed from the buffer in order to make space for a new one. It holds the most recent tuples (S_t, a_t, r_t, S_{t+1}) These tuples are sampled randomly at fixed intervals and used for training. As far as the sampling is concerned, usually all tuples are assigned the same probability, but there are other sampling strategies such as prioritised experience replay.
A Target network is a neural network that keeps its parameters separate from the main NN and only updates them periodically. It therefore manages to reduce correlation between Q(s,a) and target $r + \gamma * \max_a Q(s', a)$. The loss function for state t using a target network becomes

$$(R_{t+1} + \gamma * \max_{a} Q_{target}(S', a, w^*) - Q(S, a, w))^2$$

The gradient descent update rule(for w not w^*) is:

$$w_{t+1} \leftarrow w_t + \alpha * [R_{t+1} + \gamma * \max_{a} \widehat{Q}_{target}(S_{t+1}, a, w_t^*) - \widehat{Q}(S_t, A_t, w_t)] * \nabla_w \widehat{Q}(S_t, A_t, w_t)$$

Mnih et. all also regularised the update(value inside the brackets) in the range $[-1, 1]$ to improve stability.
All in all, the agent starts playing the game randomly. At each time step it stores the $<$state,action,reward,successor $>$ tuples in the experience replay. The state is a "screen-shot" of the game console. It samples some of the tuples to train the neural network, and uses it to pick the optimal action. The newly explored tuples are again stored in the buffer for further training. An $\epsilon-$greedy strategy is used to increase exploration. The main network's parameters are periodically copied to the target network.

2.5.3 Dueling Architecture

Dueling architecture [27] estimates both value function V(s) and an advantage function A(s,a). In traditional CNNS there are a number of convolution layers(and polling, RELU layers), followed by a feed forward NN. Now the convolution layers will be followed by two separate branches of feed forward neural networks F_1 and F_2. One estimates V the other estimates A. We can obtain Q with the following equation.

$$Q(s, a; w, F_1, F_2) \leftarrow V(s; w, F_1) + ((A(s, a; w, F_2) - \max_{a'} A(s, a'; w, F_2))$$

where F_1 denotes the parameters of the feed forward NN responsible for estimating V and F_2 the parameters of the other one.
Wang et al. propose to replace max operator with average for better stability.Hence Q is recovered by the following equation

$$Q(s, a; w, F_1, F_2) \leftarrow V(s; w, F_1) + ((A(s, a; w, F_2) - \frac{1}{A(s)} \sum_{a'} A(s, a'; w, F_2))$$

Using the dwelling architecture(Along with some algorithmic improvements) the authors managed to improve over previous state of the art performance in the Atari Domain(section 3.2).

2.5.4 Double Deep Q learning

There is no need for additional neural networks as the target network in traditional deep Q learning is used as the second value function. The greedy policy is evaluated according to the online network and it's value is estimated using the target network. The target of the double Q learning algorithm is

$$y_t^{DoubleDQN} = R_{t+1} + \gamma * Q_{target}(S_{t+1}, arg \max_a Q_{online}(S_{t+1}, a))$$

The rest of the components of the original DQN algorithm remain the same. The weights of the online network are periodically copied to the target network. Double DQN [26] finds better policies than classic DQN on the Atari 2600 Domain.

2.5.5 Noisy Nets for exploration

Noisy nets [7] are artificial neural networks whose weights and biases are perturbed by a parametric function of noise. The output of the ANNs therefore is

$$y = f_\theta(x)$$

where

$$\theta := \mu + \Sigma * \epsilon$$

With $*$ denoting the hadamard product.
μ and Σ are learnable parameters and ϵ is noise with zero mean. The loss of the ANN is an expectation over the noise

$$\widehat{L}((\mu, \Sigma)) = E[L(\theta)]$$

optimisation happens only with respect to μ and Σ Consider a linear layer of an traditional ANN with p inputs and q outputs $y = wx + b$ the corresponding noise linear layer is defined as

$$y = (\mu^W + \sigma^W * \epsilon^W)x + \mu^b + \sigma^b * \epsilon^b$$

e^b and e^w are not learnable parameters, they are noise.
The researchers in deep mind that came up with the idea of Noisy Nets experimented on two different choices for the noise distribution.

- Independent Gaussian noise
  The noise applied to each weight and bias is independently drawn from a unit Gaussian distribution.
- Factorised Gaussian noise
  p unit noise Gaussian variables are used for noise of the inputs and q for noise of the outputs.

The noisy layers have been applied successfully to both DQN and dueling architecture DQN. Because of the introduced noise, $\epsilon-$ greedy policy iteration is no longer necessary. The policy greedily optimizes the randomised action value Q.
Noisy Net agents outperformed traditional DQN and Dueling DQN and A3C agents for the majority of Atari games.

2.5.6 Distributional Reinforcement Learning

So far, Q learning (and its variants) try to maximise the expected return. In the distributional RL [1] setting the goal is to approximate the distribution of the return. In order to achieve that, a variant of Bellman's equation has been introduced. The distributional Q learning algorithm aims to minimise the Kullbeck-Leibler divergence between the current distribution and the target distribution.
Explaining this approach in depth is beyond the scope of this thesis. The reader can take a look at [1]

2.5.7 Rainbow (combination of improvements)

The Rainbow [9] agent integrates all the improvements of section 2.5, as well as multi step learning into a single agent.
Rainbow agent was tested in the Atari domain and it outperformed all previously published baselines both in data efficiency and in final performance.

2.6 Asynchronous Reinforcement Learning

So far the methods of deep RL we have examined store the data from each timestep on a replay buffer and randomly sample some to train the ANN. This approach, while very powerful, uses a lot of memory per iteration(as we will see in the experiments).
Asynchronous methods [12] execute multiple agents in parallel on multiple instances of the environment instead of using experience replay. This approach has both theoretical and practical benefits.
It has been applied successfully on one step Q learning, n step Q learning, one step SARSA and actor critic. In fact, the Asynchronous Advantage Actor Critic(A3C) is one of the most successful agents to date.

3. RELATED WORK

3.1 TD Gammon

TD Gammon [24] is a program developed by Gerald Tesauro in 1992, designed to play backgammon. It has tremendous historical value to the field of artificial intelligence, and remains one of the most impressive application of reinforcement learning to date. In that time, there was already another program called neurogammon that could play backgammon really well using a large corpus of states evaluated by experts. TD Gamon managed to outperform it using zero human knowledge.
It uses temporal difference learning $TD(\lambda)$ with non linear function approximation through feed forward neural networks (which were considerably smaller than today's standards).
The game of backgammon consists of 32 pieces, each of them can be in one of 24 possible locations. The game tree has an effective branching factor of about 400 making it impossible to use traditional heuristic methods and tree search.
The neural network takes as input 198 units. For each point in the board 4 units represent the number of white pieces and 4 units the number of black pieces. There is also a unit for white pieces on the bar and one for black pieces on the bar. Moreover, one unit represents the white pieces taken from the table and one unit represents the black pieces taken from table. Finally, two more units determine whether it is the black or the white player's turn.
The outputs of the hidden units were passing through sigmoid activation function. Most of today's architectures use relu instead. The neural network outputs the probability of the current player winning.
It was trained using self play. The reward is +1 if the current player wins, -1 if the current player looses and 0 in any other case. Since the parameters where initialised randomly early games lasted from hundreds to thousands of moves. After a few dozen episodes the performance increased significantly. After about 300.000 episodes it performed as well as neurogammon. A few years latter Tessauro came up TD Gammon 1.0 which combined human knowledge(labeled examples) and self play and performed even better. Apart from the impact TD Gammon had on the field of AI, it changed the way top human players play the game.

3.2 Arcade earning environment and Atari

The Arcade Learning environment [2] was developed by Bellemare et all in 2012 and helped achieve significant breakthroughs in Reinforcement Learning. It provides an interface to hundreds of Atari 2600 game environments. It is a benchmark for evaluating and comparing RL algorithms. The authors applied standard RL algorithms with linear function approximation without achieving impressive results. That being said, ALE has been one the most important papers behind almost all the achievements in deep reinforcement learning of the past decade.
Mnih et all used DQN with experience replay as previously discussed on ALE and managed to improve previous work. Raw pixel images (With some cropping and filtering) were given as input on the networks. It outperformed all previous approaches on 6 of the games and surpassed human experts in 3. Unlike TD-gammon, the neural networks did not "need" carefully constructed feature representations to perform well.
Since then, deep Q learning variations, proximal policy optimisation algorithms, massively parallel algorithms, exploration strategies, and other ideas have been used to improve on

the ALE domain, and their performance keeps getting better and better.

3.3 Pygame learning environment (PLE)

PLE [23] is a collection of 9 single player video games, emulated in python using pygame. It has been designed to allow easy evaluation of reinforcement learning algorithm on those games, relieving the practitioners of having to deal with graphics and implementation. Researchers can also add more games by implementing a few methods (They do not have to start from scratch). Only some very basic python libraries are required to use it. Most machine learning practitioners have already installed them for other software packages.

3.4 Dota 2

Dota 2 is a real time strategy multiplayer game. It presents a very difficult challenge for the following reasons

- long time horizons
- partial observability
- high dimensionality of both action space and observation space
- complex rules
- The game is being actively developed and constantly changing. Therefore the dynamics of the environments change over time.

Researchers in OpenAI developed a program called OpenAI Five [15] that managed to beat two world champions in dota 2. They actually emulated a simple version of dota 2. Each player can choose one of 17 users (instead of 117). Also, there is no support for items which allow a player to temporarily control multiple units at the same time. The exact implementation of the model is very complex and we will not describe it in full depth. We will provide a brief overview of the methods they used,and the problems they faced.
The observation space is very complex, the model takes as input approximately 16 thousand values. Those values are a flattened set of data arrays.
The action space consists of a single primary action and a set of parameter actions. For instance the action could be to attack or to cast a spell and the parameters could be the targets. The available number of action varies from time to time and averages around 8.1. In the emulated game, the agent can take one of 6 main action types and the appropriate secondary actions.
Designing reward signals is also pretty challenging compared to previous applications such as PLE or ALE. The goal is obviously to beat the other team(zero sum game). That being said, some rewards are given to the whole team and others to the hero who took the action. As the game progresses, each player's "power" increases significantly. As a result, the learning procedure might end up focusing completely on the later stages of the game. To combat this all rewards other than win/loss rewards decay over time.
Additionally, team based rewards can increase variance when a different agent takes a good action. To combat that each hero earns a linear combination of a raw reward and

the average team reward.
Moving forward, the neural network architecture, consists of a 4096-unit LSTM that takes the flattened observation and the hero embedding as input and outputs two predictions. One for the value function and one for the action. Each of the five heroes in a team is controlled by one neural network of this architecture. The Adam optimiser is used to train the network with back propagation through time.
Agents learn to maximise cumulative reward with Proximal Policy Optimisation(PPO). A central pool of optimisers each running on its own GPU stores game data from self play asynchronously in local experience replay buffers. Each optimiser samples a mini-batch of data randomly from it's local buffer and uses it to compute the gradient. The gradients are then averaged over the entire pool and used to update the parameters of the model. The entire system was trained on large distributed platform over google cloud for months.

3.5 Alpha Go (with human knowledge)

The game of Go is considered the most challenging classic game for AI. It has an enormous search tree making it impossible to learn efficiently. In order to "solve" it researchers at deep mind combined some classic algorithms as well some new ideas[20]. They also combined a dataset of board positions labeled by human experts and self play.
The main idea is to train two different neural networks and use them to perform MCTS. A value network used to evaluate board position and a policy network used to sample actions. Value networks can help decrease the depth of the search tree. As they both become more accurate the rollout policy becomes more accurate forcing Monte Carlo Tree Search to search more relevant states.
The board position is converted to a 19x19 image and provided to a sequence of convolution and pooling layers for feature extraction. After that, the learned representation is provided to a value network and a policy network. The training pipeline consists of the following three steps.

- First, a policy network ρ_σ is trained using the labeled dataset. Moreover, a fast rollout policy ρ_π with linear function approximation is trained as well.
- A policy network ρ_ρ is trained. ρ_ρ is initialised with the values of ρ_σ. The rewards are +1 if the current player wins, -1 if the current player looses and 0 otherwise.
- Finally, a value network v_θ is trained using self play. It has a similar architecture to the policy networks but outputs a single prediction instead of a probability distribution.

When training is done both networks are combined in a Monte Carlo Tree Search algorithm that selects actions with lookahead. The fast rollout policy is also used for evaluating states. The predicted value of a state is a linear combination of the value networks output and the complete episodic reward of a game played by the rollout policy.
Alpha Go is the first computer program that beat a professional Go player. It also outperformed all other commercial and public Go programs at the time.

3.6 Alpha Go without human knowledge

After the success of Alpha Go with human labeled examples and reinforcement learning, researchers in Deepmind developed another program that performs even better without

previous human knowledge [21], while also being significantly simpler. It managed to beat the previously published Alpha Go program by 100-0.
There is only a single CNN f_θ that outputs both a probability vector and a scalar value. The input of the network is the raw representation of black and white stones and its history. The probability vector for state s , provides an estimate for the optimal policy starting from state s. Some noise is also added during training, to increase exploration. The scalar value provides an estimate of the state's value.
During self play, in every time step a Monte Carlo Tree Search is executed guided by f_θ. The output of the search is probability vector π and a reward $z \in \{-1, 1\}$ depending on the winner. The goal is to train the Neural Network such that

$$f_\theta(s) = (\pi, z)$$

Each edge (s,a) of the tree stores three values

- a prior probability P(s,a)
- a visitation count N(s,a)
- an action value Q(s,a)

Each simulation starts from the root and selects moves that maximize $Q(s,a) + U(s,a)$ where $U(s,a) \propto \frac{P(s,a)}{1+N(s,a)}$ until it reaches a leaf node s'. s' is then expanded for one step using the neural network.

$$(P(s', *), V(s')) = f_\theta(s)$$

For each of the edges (s,a) traversed by the simulation the following two operations are performed

- $N(s,a)$ is increased by 1
- $Q(s,,a) = \frac{1}{N(s,a) * \sum_{s'|s,a->s'} V(s')}$

As we previously mentioned MCTS also outputs an action probability vector $\pi = (a_1, a_2, ...a_n)$. Each action a_i is assigned a probability

$$\pi_{a_i} = N(s,a)^{\frac{1}{\tau}}$$

with τ being a temperature parameter.
The Neural Network loss function sums over the mean squared error of the value estimate and the cross entropy loss of the policy plus a Tikhonov regulariser. Hence it is given by the following equation.

$$l = (z - v)^2 - \pi^T logp + c||\theta||^2$$

Using this pipeline, a "small" neural network was trained for 3 days and it managed to outperform some of the previous state of the art programs that took months to train. Training progressed smoothly without running into the problem of catastrophic inference. A larger network was trained using the same pipeline for approximately 40 days beating all state of the art programs by a very large margin.

3.7 Other Games

In the past few years, a lot of researchers have managed to successfully apply the previous algorithms to a variety of different games, both single agent and multi agent. A survey by Shao et all [19] provides an overview.
It also worth noting that RL has been successfully used in a variety of fields such as robotics and combinatorial optimisation.

4. THE FRIV LEARNING ENVIRONMENT(FLE)

Friv is a website containing a lot of games. Some are directly from Friv, some are adapted from other platforms. It provides a very rich pool of completely different games. From platformer and shooting games, to car racing and maze games. Some are endless whereas others are pretty quick. Some have very realistic physics (e.g hill climb racing) others not. Some have a narrow action space, others a very large. The last, is the key difference between the atari games, which are all played by the same console.
We emulated some of the games in python on top of Openai's gym [6]. We created multiple levels for 9 games. The total number of environments is 23.
Obviously the graphics are going to be somewhat simpler. That being said, the collision detection mechanics, the physics simulation(when required) and the game difficulty are pretty close to the actual games. For spinSoccer and carParking we used the box2d physics engine to make sure the physics and collisions are realistic. For the rest we use very accurate Axis Aligned Bounding Boxes and pygame's colliderect() function.

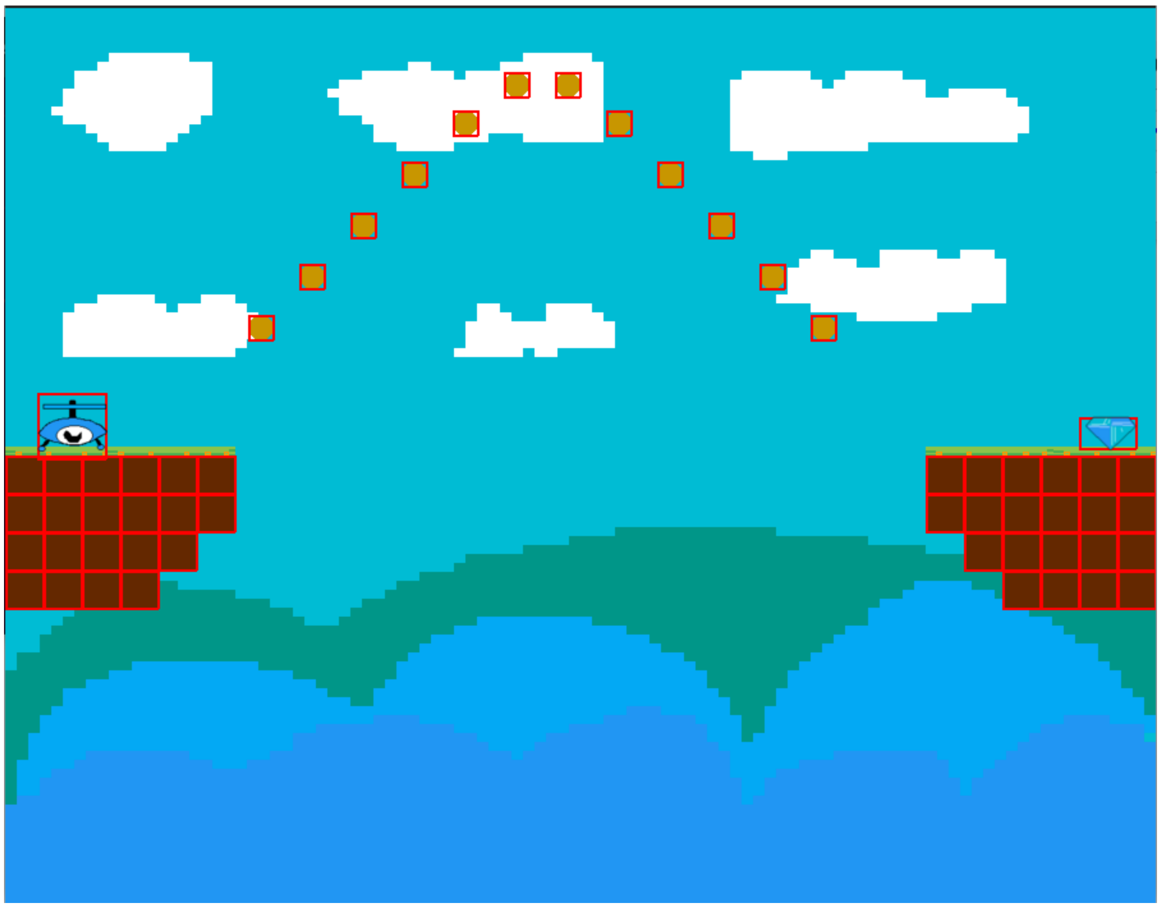


Figure 4.1: Collision detection in Eyecopter Gemland

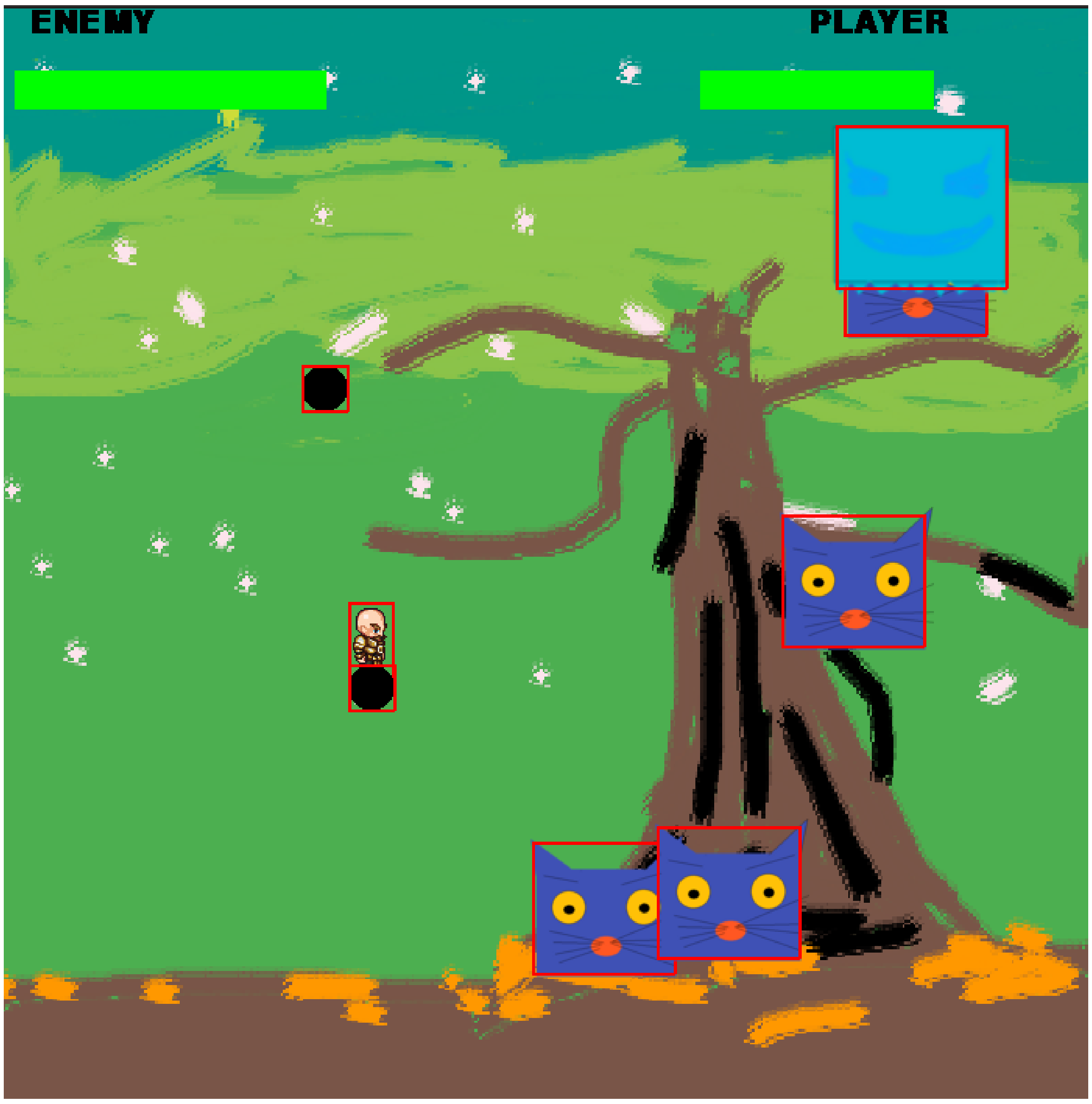


Figure 4.2: Collision detection in boss level pumpkin

As you can see the AABBs are the smallest possible, while containing the entire objects. If you play the games yourself you will notice the collision mechanics are just as accurate as the original games on FRIV.
First we will provide a brief overview of each game, and then we are going to provide a table evaluating some of the most common reinforcement learning (DQN,PPO,A2C) algorithms on each game. We will use the Stable Baselines 3 library [16]. We will compare the performance of the agents, with the following

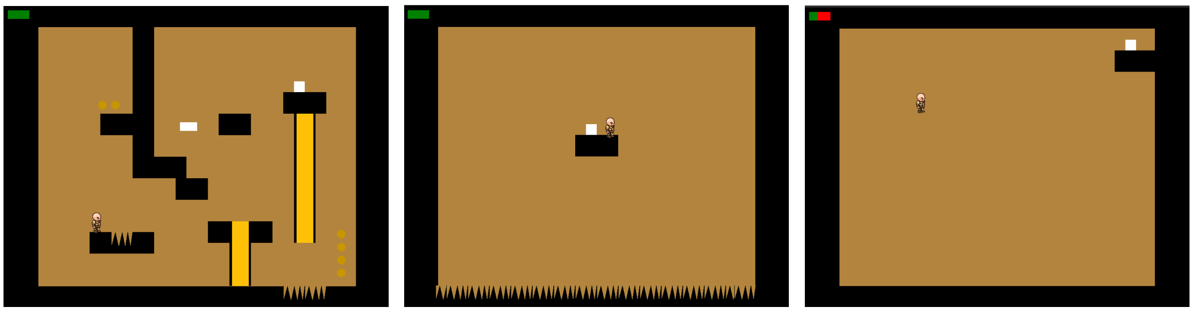
- A human agent. A human will play 5 games to get used to the rules and mechanics of the games. Then we will evaluate their performance on an average of 5 games. The human agent is also the developer/tester of the games, so they will be very competent.
- A random agent. Again the performance will be an average of 5 games
- A constant agent playing the same action and a perturb agent. The perturb agent will play the same constant action with probability p and with probability (1-p) it will pick an action randomly.

As we will see those games provide an interesting challenge due to the sparsity of rewards. Additionally, multiple levels of the same games can be used for experimenting with methods for transferring knowledge from simpler to harder tasks.

4.1 Agent Platformer

This is a very interesting experiment because it is a very easy game for human players. Nevertheless, the rewards are sparse and sometimes misleading. The agent can move left right fly or jump and must get to the white box. They collect bonus points from coins. In the first level coins are slightly far away from the white box, which can mislead the agent.The reward signals are given in the following manner.

- -1 if the agent falls on the spikes and loses .
- + $\frac{0.25}{6}$ if the agent collects a coin.
- + 0.75 if the Agent wins. In the last two environments there are no coins so the agent gets +1 for winning.
- We also added a large negative reward of -1 in case the game has not finished in 4000 time steps. That reward is not enough to lead the agent to victory but by observing the training processes we noticed it encouraged the agent to explore a lot more states.



(a) level 1 (b) level 2 (c) level 3

Figure 4.3: The three levels of Agent Platformer

4.2 Super Onion Boy

The agent controls an onion-looking ”boy” that must navigate to the end of the maze and collect the blue dot avoiding enemies. It can move left or right and/or jump, it can also jump on top of boxes. There is also a green lever that can throw it high in the air to collect more coins. There are bonus reward for jumping on top of different enemies and killing them and for collecting coins. There are three different types of enemies. Regular pumpkins, flying vegetables and a large ball. To kill the ball you must jump on it twice. After the first jump it moves a lot faster and it can kill other enemies as well.

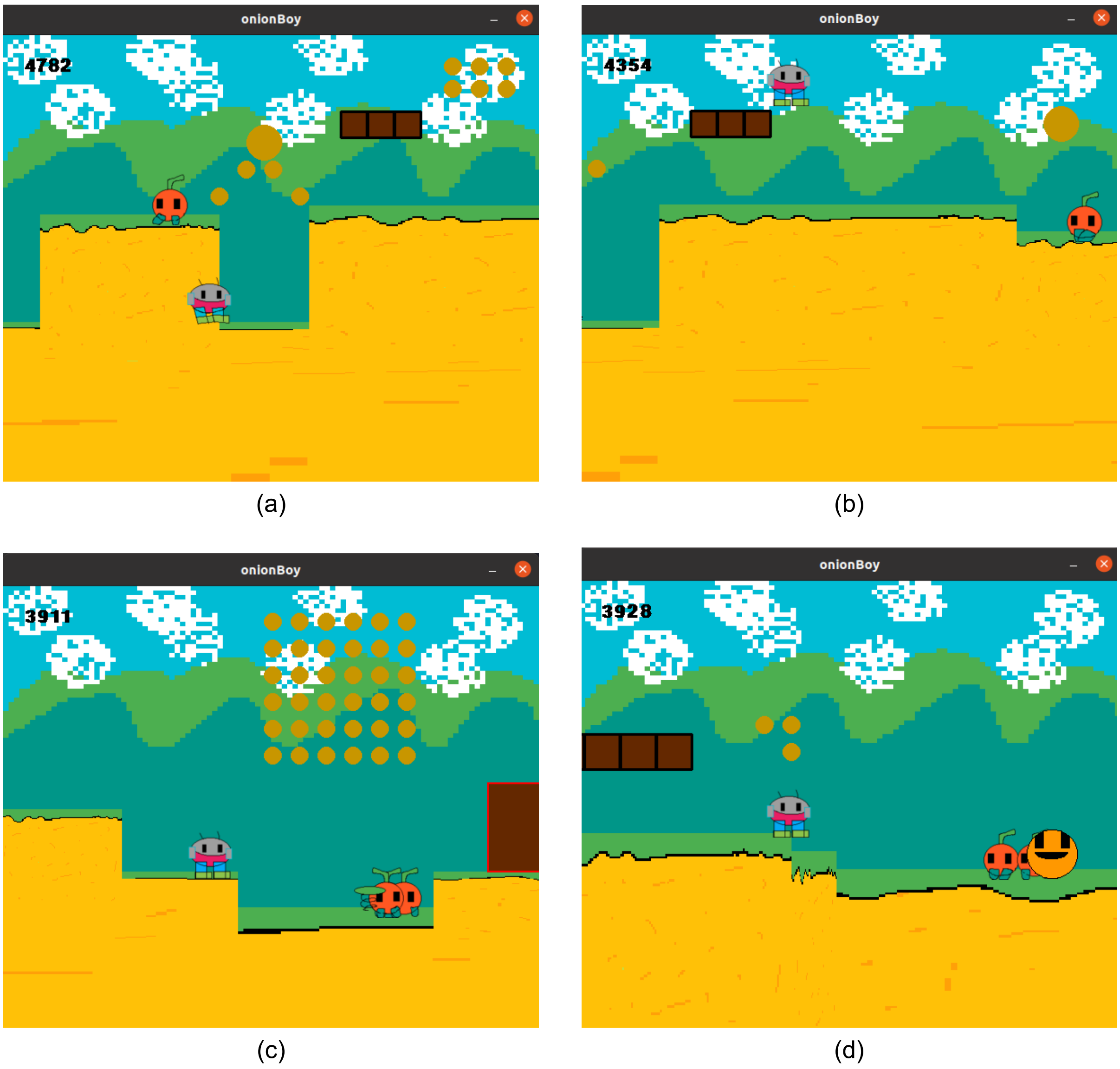


Figure 4.4: Super onion boy environment

4.3 Car Parking

In this game, the agent controls a car and must park it on the yellow parking spot without hitting the walls or other cars. The reward is +1 if parking is successful and -1 if the car collides with an obstacle or if time runs out. There are no immediate rewards because we wanted to see the agents' ability to explore a large state space. We used box2d to make sure the car's wheel and tire dynamics are realistic. The agent can pedal forward, reverse and/or move the wheel slightly to the left or the right.

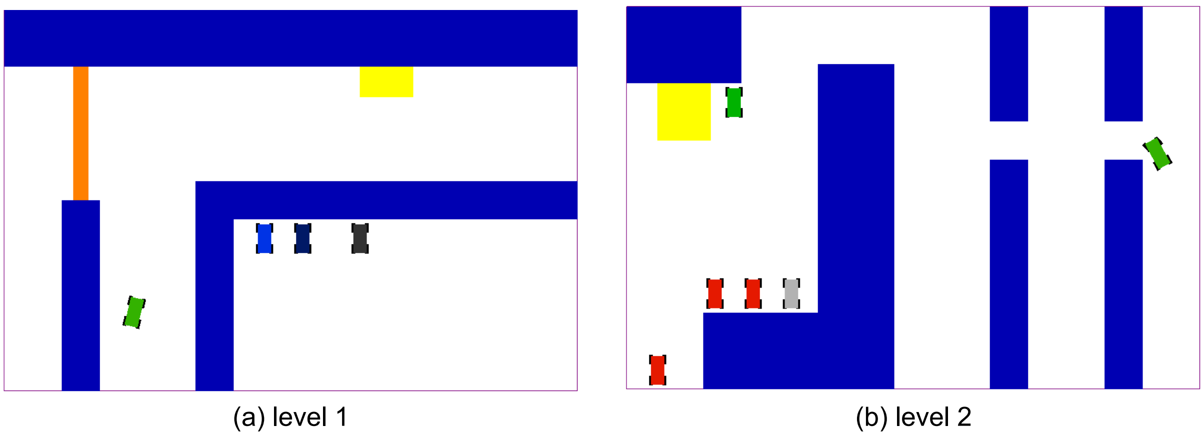
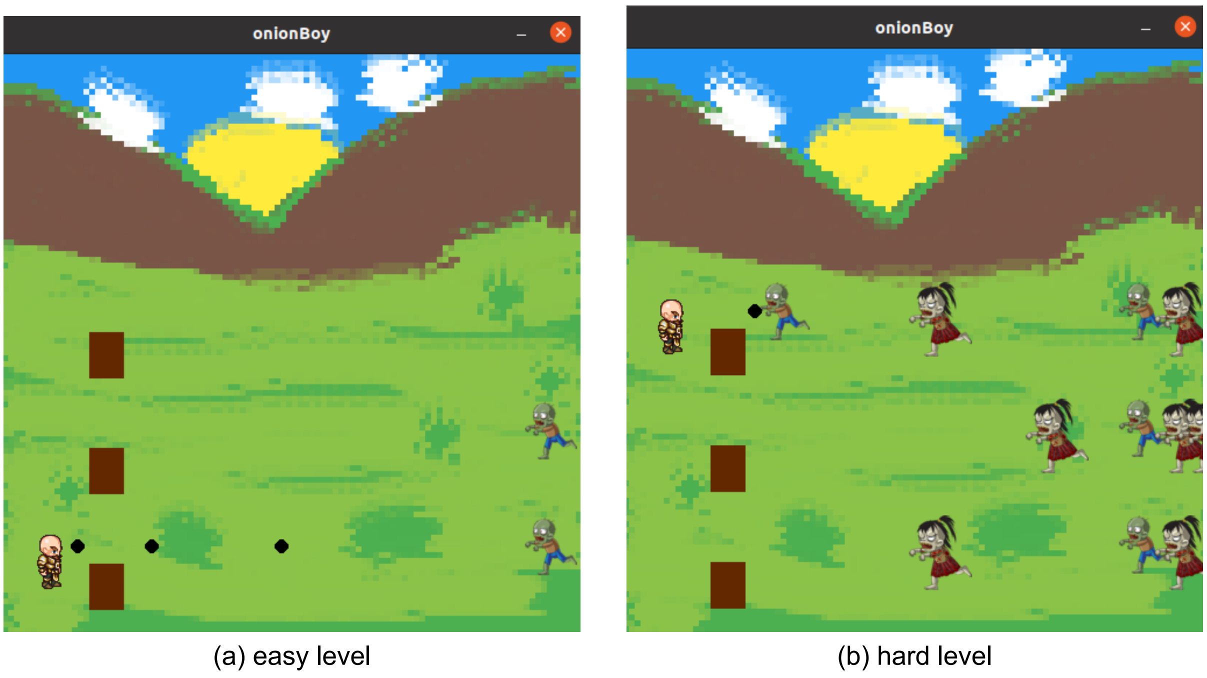

(a) level 1 (b) level 2

Figure 4.5: Car parking environment

4.4 Zombie Onslaught

In these game the agent must defend the military base from the incoming zombies. The zombies try to break the crates and get past the soldier. Some zombies are weaker and they die after being shot twice some are stronger and die after being shot 4 times. The reward for each zombie killed is $\frac{1}{\text{number of total zombies to kill}}$.
We provide two levels of zombie onslaught, one is relatively easy and consists only of weak zombies, the other is very hard and the human testers did not manage to complete it. The zombies are spawned in fixed intervals in order to make the experiments reproducible. We also added a ”loading” time interval to prevent the program from firing bullets faster than the human tester and therefore having an unfair advantage.

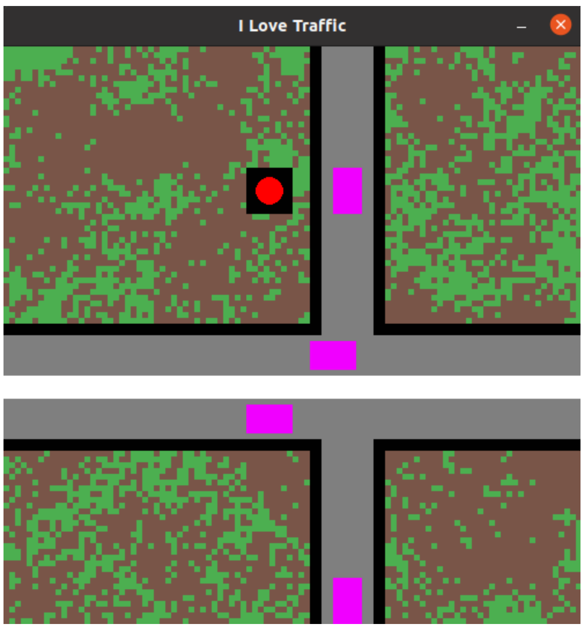


(a) easy level (b) hard level

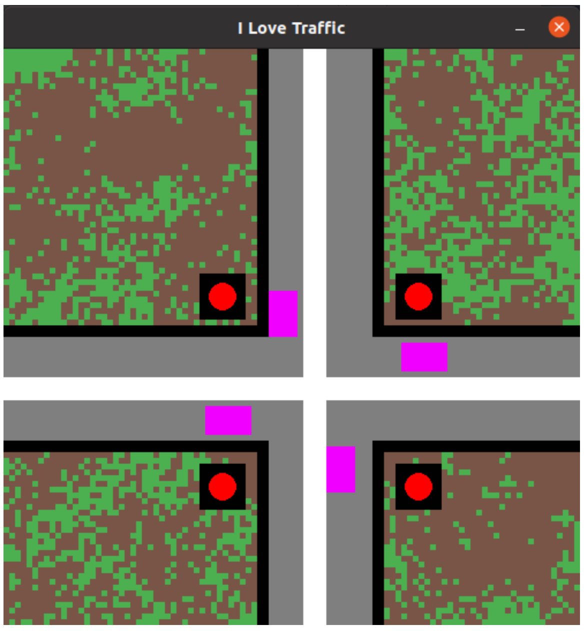
Figure 4.6: Zombie onslaught environment

4.5 I love traffic

In this game the agent controls one(or more) traffic lights. The goal is to direct traffic such that cars do not collide with each other. You loose if there is too much traffic clogged up. The cars are not ”intelligent” and if they see a green light they will pass without looking out for other cars. A very important detail is that the agent is being provided incomplete information, it only gets a picture of some cars. It has no way of knowing how fast a car is moving or if is stopped (like the human tester). We provided 5 different levels. The first level is very easy and the rest get harder and harder.



(a) level 3



(b) level 4

Figure 4.7: Two levels of the I love traffic environments

4.6 Go Chicken Go

In this game the agent controls a chicken and must navigate it to the other side of the road without getting hit by the cars or drowning in the river. To decrease the ”sparsity” of the reward, it gets a small reward for reaching the river (after crossing the first road) a small reward after crossing the river and a large reward after crossing the second road safely. We also choose not to seed the environment, in order to test the ability of the algorithms to generalise.
Initially, if the chicken got hit by a car or drowned the agent got a reward of -1. That led to agents becoming ”lazy” and not doing anything. Even after adding a timeout reward of -1 the agents remained inactive. Hence, in the final implementation of the game the agent gets -1 reward for ”stalling” but only -0.7 reward if the chicken loses. That lead to agents exploring the game state to a far greater degree. In fact DQN managed to get past the logs and actually win the game a few times during training. Unfortunately, that exploration was not enough to consistently win the game.

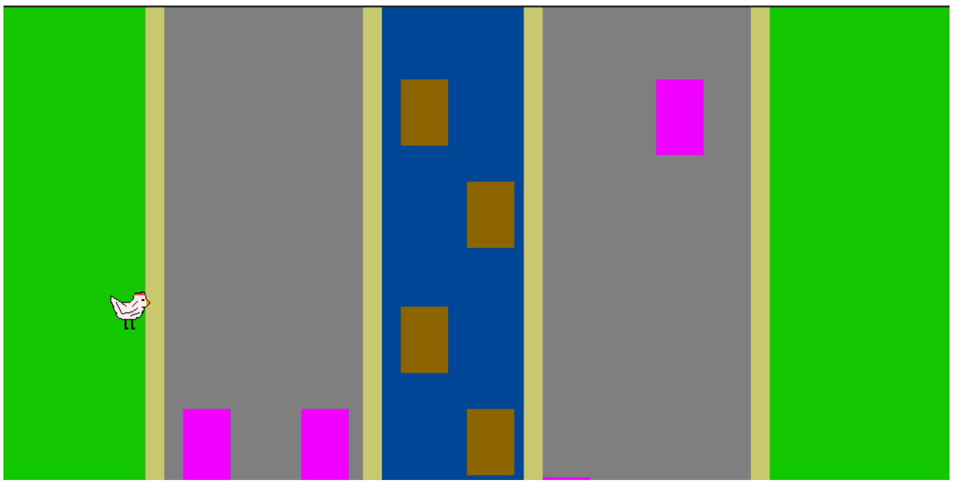


Figure 4.8: Go Chicken Go environment. There are two identical levels,the only difference is the speed of the cars and logs , and the frequency in which they appear.

4.7 Eyecopter Gemland

In this game, the agent controls a helicopter and must navigate it to collect the blue diamond and come back to the base without falling outside of the screen. There are also bonus points for collecting coins. Just like in the Agent Platformer game, small rewards from collecting coins can mislead the agent away from the large reward of bringing the diamond to base.
Initially, the rewards where -1 for losing, 0.1 for collecting the diamond, $\frac{0.3}{\text{number of coins}}$ for collecting a coin and 0.6 for bringing the diamond back to the base. Just like the other games, however we noticed that the agents tend to get lazy in order to avoid losing. After adding a large negative reward of -1 for timing out all agents (especially DQN) explore the state space a lot more effectively .

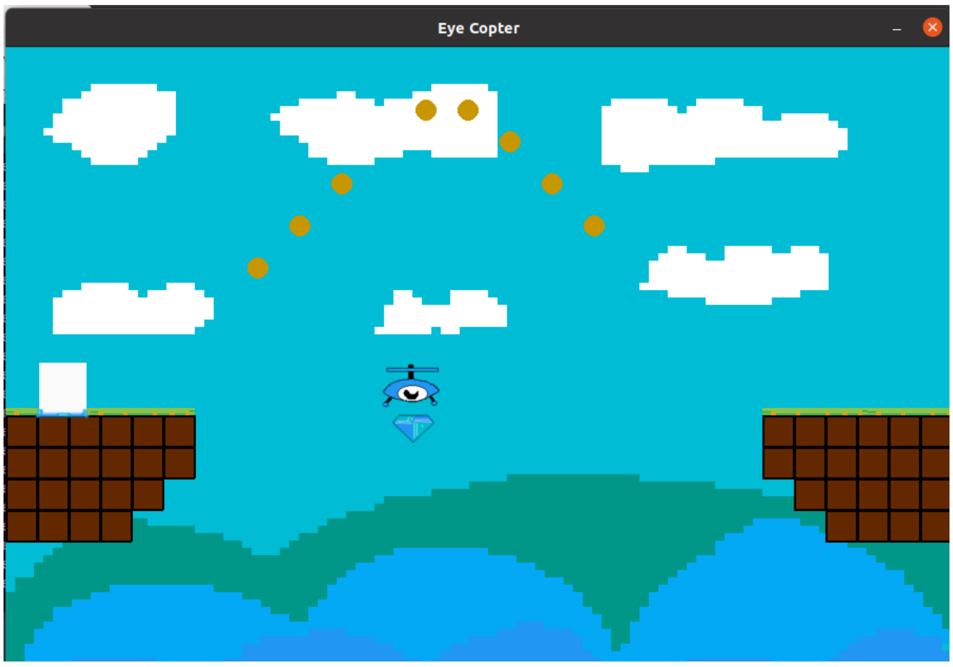
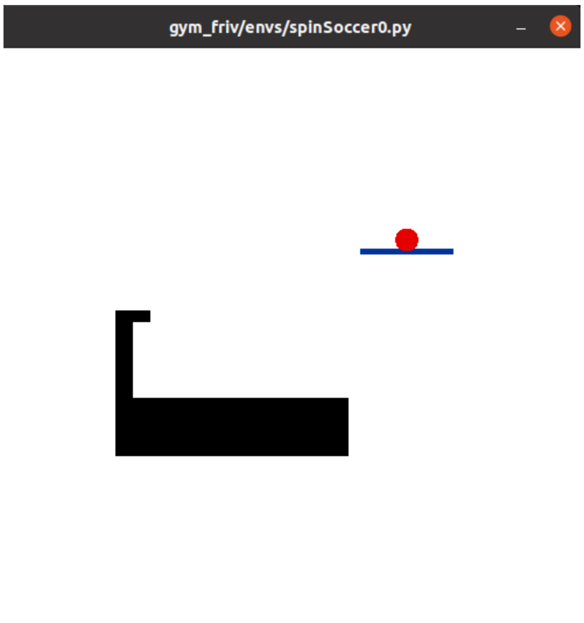


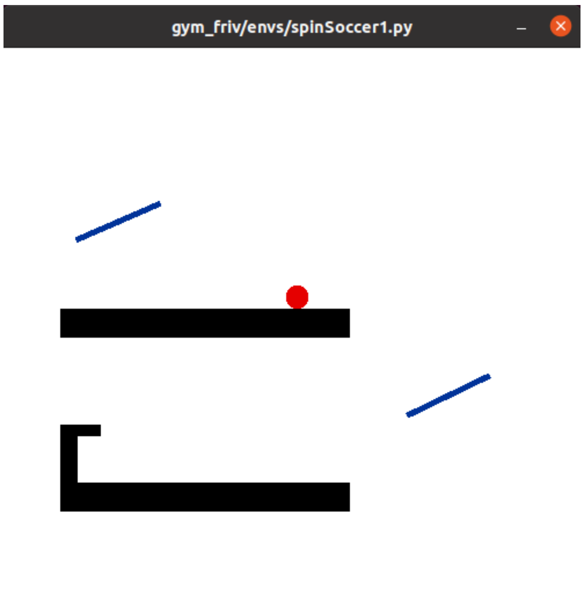
Figure 4.9: The EyeCopter Gemland game

4.8 Spin soccer

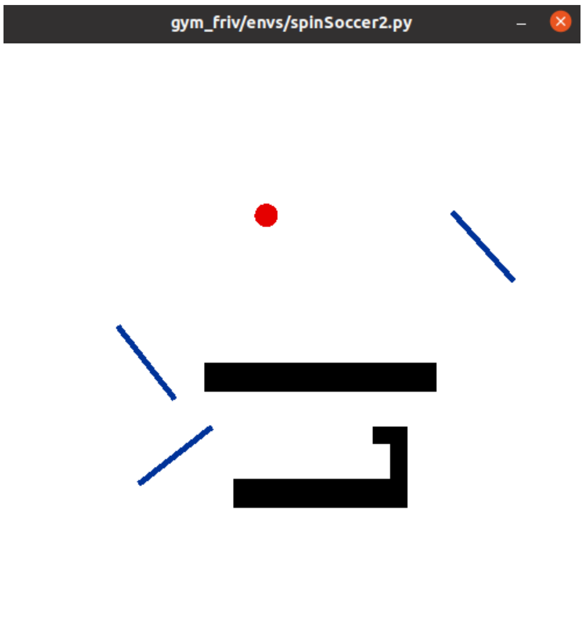
In this game, the agent controls all the platforms and must navigate a soccer ball to the goalpost. All platforms move simultaneously. We used the box2d physics engine to make sure the simulations are realistic. There are a total of 4 tasks of progressive difficulty. All of them are relatively tough for humans as a small mistake can send the ball very far from the goalpost.



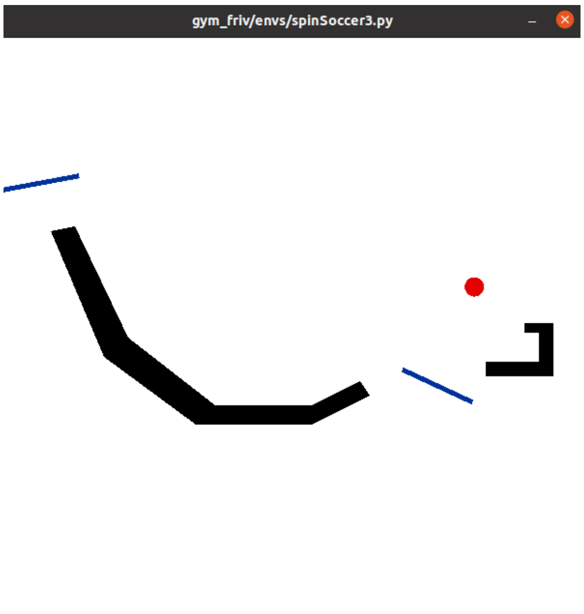
(a) level 1



(b) level 2



(c) level 3



(d) level 4

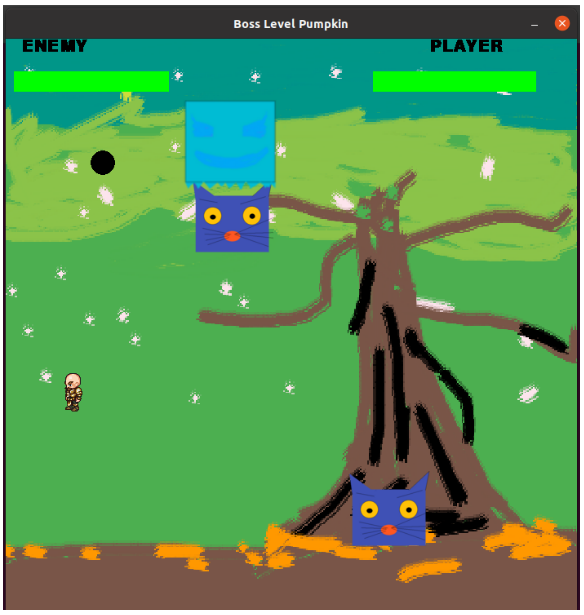
Figure 4.10: Spin Soccer

While the RL agents can learn pretty quickly, letting the training process go for slightly too long (even just a few hundred timesteps more) can lead to serious catastrophic forgetting. We also noticed that all agents ”learn” to keep the platforms steady and do nothing. For that reason we added a large negative reward after some timesteps. The reward is -0.7 for losing , 1 for winning and -1 in case of a time out. The time out penalty led to tremendous improvements in all agents

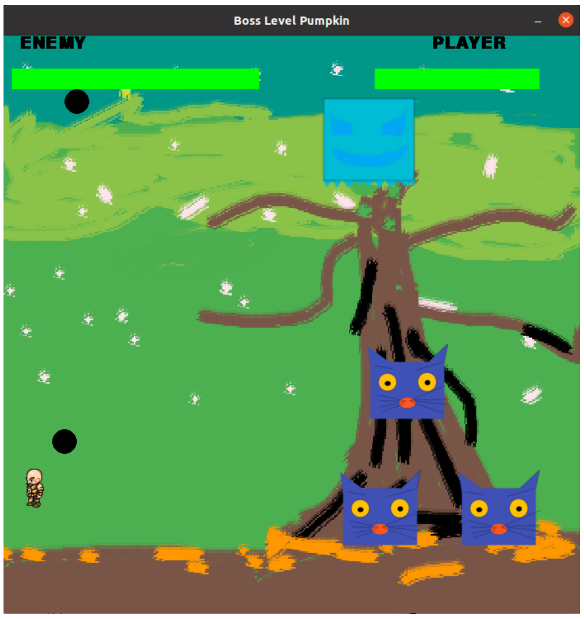
4.9 Boss Level Pumpkin

This game was introduced by Friv on Halloween. The player, controls a person that moves left right, jumps and shoots. Above the ground there is a large ghost that tries to hit the player throwing scary objects. On the first level the objects are not moving, in the latter levels the objects can move, and the ghost might throw different objects at the same time. The goal is to shoot the ghost in order to kill it. The player can also buy update packages to increase their health or damage.
We have implemented 3 levels. In the first and the second level, the ghost throws the same static objects. The only difference is that in the second level, the ghost can take more bullets without dying, it moves a little faster, and the objects fall downwards slightly faster as well. In the third level, the ghost throws moving objects, that detect where the player is positioned and try to chase them.
We have performed the following simplifications

- In the original game, the player can touch some boxes which then throw grenades at the ghost. That made the game too easy to conduct interesting experiments so we did not implement them.
- We removed the player's ability to buy updates



(a) level 1



(b) level 2

Figure 4.11: First Levels of Boss Level Pumpkin

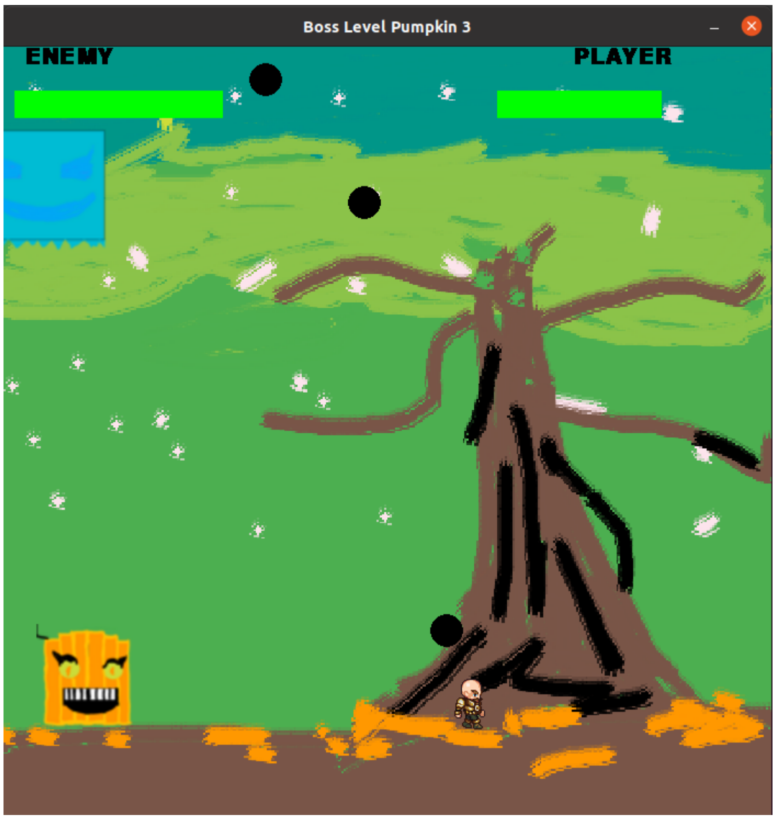


Figure 4.12: The third level of Boss Level Pumpkin

5. RESULTS

Table 5.1: The performance of different agents in FLE

Game	PPO	A2C	DQN	Human	Random	Constant	Perturb
SpinSoccer 0	1	1	1	1	-0.2	-1	0.6
SpinSoccer 1	1	-1	-1	0.2	-0.7	-0.7	-0.7
SpinSoccer 2	1	-1	1	-0.6	-0.7	-0.7	-0.7
SpinSoccer 3	1	-1	-1	-0.6	-0.36	-1	-0.7
Platformer 1	-0.8	-0.8	-0.67	1	-1	-1	-1
Platformer 2	-1	-1	-1	1	-1	-1	-1
Platformer 3	-1	-1	-1	1	-1	-1	-1
Zombie Easy	1	-0.462	1	1	-0.9	-1	-0.8
Zombie Hard	1	-0.74	-0.203	-0.272	-0.94	-1	-0.93
Traffic 0	1	1	1	1	1	1	1
Traffic 1	1	1	1	1	-0.96	-1	-0.9
Traffic 2	1	1	1	1	-0.96	-1	-0.92
Traffic 3	1	1	1	-0.22	-1	-1	-1
I love Traffic	1	-0.85	1	0.42	-0.92	-1	-0.86
EyeCoper	-1	-1	-0.6	1	-0.985	-1	-1
OnionBoy	-1	-1	-0.994	-0.829	-1	-1	-1
CarParking 1	-1	-1	-1	1	-1	-1	-1
CarParking 2	-1	-1	-1	0.2	-1	-1	-1
Go Chicken Go Slow	-0.7	-0.7	-0.2	-0.2	-0.7	-1	-0.76
Go Chicken Go	-0.7	-0.7	-0.52	-0.32	-0.7	-1	-0.82
Boss Level Pumpkin 1	0.47	-1	0.7	-0.244	-0.64	-0.52	-0.55
Boss Level Pumpkin 2	-0.707	-0.81	-0.45	-0.674	-0.867	-0.77	-0.8
Boss Level Pumpkin 3	-0.69	-1	0.30	-0.33	-0.88	-1	-0.96

The results are completely expected. When the agent can sufficiently explore the state space and observe frequent reward signals, it performs comparably or even better than humans. When the results are sparse or misleading such as in agent platformer and car parking it performs really poorly. We should keep in mind that both PPO and A2C require a lot less computational resources (both memory and CPU) than DQN.
We assume that if the reward difference between two agents is less than 0.2 they perform comparably well. As you can see,in 16 out of 23 games, at least one RL agent performs comparably well to the human agent. In 8 out of 20 games at least on RL agent outperforms the human with more than 0.2 reward.

6. CONCLUSIONS AND FUTURE WORK

All in all, we provided an open source environment for evaluating reinforcement learning agents based on the website FRIV and hope that other people can use it in their work. It is very easy to use and maintain. We also provided some performance baselines both regarding human performance as well as state of the art algorithms. In addition, we mentioned some interesting observations regarding the training process and the construction of the reward signals.
The most important outcome of this thesis is the following:
When dealing with a task with infinite or very long horizon and sparse rewards, the agent tends to stay inactive in order to avoid large negative rewards. Penalising inactivity with a negative reward larger(in absolute value) than that of losing can improve the agents performance tremendously.
FLE provides a lot of interesting opportunities. For starters, experimenting with transfer learning from easier to harder levels of the same game is a very interesting domain. Moreover, we could experiment with advanced exploration strategies because most of the games have very sparse reward signals. Last but not least, we plan on implementing more games from FRIV since the website has over 50 different games and it occasionally adds more.

BIBLIOGRAPHY

[1] Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning, 2017.

[2] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, Vol. 47:253–279, 2012. cite arxiv:1207.4708.

[3] D.P. Bertsekas. Rollout algorithms: an overview. In *Proceedings of the 38th IEEE Conference on Decision and Control (Cat. No.99CH36304)*, volume 1, pages 448–449 vol.1, 1999.

[4] D.P. Bertsekas. *Dynamic Programming and Optimal Control*, volume 1 and 2. Athena Scientific, 2 edition, 2001.

[5] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

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

[7] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy networks for exploration, 2019.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[9] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning, 2017.

[10] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[11] Richard E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(2):189–211, 1990.

[12] 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.

[13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. 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(7540):529–533, February 2015.

[14] Thomas M. Moerland, Joost Broekens, and Catholijn M. Jonker. Model-based reinforcement learning: A survey, 2021.

[15] OpenAI, :, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique P. d. O. Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning, 2019.

[16] Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. Stable baselines3. `https://github.com/DLR-RM/stable-baselines3`, 2019.

[17] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1889–1897, Lille, France, 07–09 Jul 2015. PMLR.

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

[19] Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, and Dongbin Zhao. A survey of deep reinforcement learning in video games, 2019.

[20] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[21] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, October 2017.

[22] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[23] Norman Tasfi. Pygame learning environment. `https://github.com/ntasfi/PyGame-Learning-Environment`, 2016.

[24] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[25] John Tsitsiklis and Benjamin Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Trans. on Automatic Control*, 42(5):674–690, 1997.

[26] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. 2015. cite arxiv:1509.06461Comment: AAAI 2016.

[27] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2016.

ABBREVIATIONS - ACRONYMS

RL	Reinforcement Learning
MDP	Markov Decision Process
DP	Dynamic Programming
DL	Deep Learning
NN	Neural Network
MLP	Multi Layer Perceptron
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
LSTM	Long Short Term Memory
MC	Monte Carlo
TD	Temporal Difference
MCTS	Monte Carlo Tree Search
PGM	Policy Gradient Methods
DQN	Deep Q Network
KL	Kullback Leibler
TRPO	Trust Region Policy Optimisation
PPO	Proximal Policy Optimization
AC	Actor Critic
A2C	Advantage Actor Critic
A3C	Asynchronous Advantage Actor Critic
ALE	Arcade Learning Environment
PLE	Pygame Learning Environment
FLE	Friv Learning Environment
AABB	Axis Aligned Bounding Boxes