Contents

1 Abstract

Reinforcement learning is a machine learning technique that uses inputs from the environment to determine the next optimal steps to take in order to solve the task given. This preliminary report starts out with the theoretical/statistical basis of reinforcement learning techniques on a popular problem 'The One armed bandit'. After these techniques are explored, experiments are performed to compare their performance on a simulated bandit. Finally, Gym is loaded with a game called 'Cartpole' in which we try to balance a pole on a cart. A random search algorithm is used to solve the environment.

2 Aims/Objectives

While the final goal of this project is to *use reinforcement learning techniques to train a computer agent to drive a formula style racing vehicle in a computer simulation*, this preliminary report shall focus on the work done so far in order to achieve this goal:

- An explanation of the theory behind modern reinforcement learning methods
- Basic experiments to test and compare such methods
- Applying these methods on basic simulations like CartPole

Deep Reinforcement learning essentially means using deep neural networks with a feedback loop. This means that the agent will not only learn from examples given to it but will also learn from responses from the environment. In this work, I will investigate the current reinforcement learning techniques and the literature explaining these techniques. Special attention will be given to multi-agent reinforcement techniques i.e. techniques utilized to tie together disparate skills of different agents into one multi-task competent agent and techniques used to design agents that learn to maximize different goals at the same time thus making them multidisciplinary. The end goal would be to drive a formula style racing car in a virtual simulation using an agent trained with deep reinforcement learning methods.

Key objectives of the Project that can be tested at the end: Investigate contemporary literature on reinforcement learning Research into methods of merging learned skills from different agents Drive a formula style racing car in a simulation

3 Introduction

3.1 Deep Learning

Deep learning has been around for long and a lot of the algorithms were developed in the 50s and 60s but has only recently exploded in practical uses as

evidenced by the sharp increase in the use 'Artificial Intelligence' and 'Deep learning' as buzzwords in the everyday journalism parlance. The main reason for this is the advent of cheaper and more powerful computers available today. In 1956 a 5MB IBM server cost tens of thousands of dollars to rent for a few weeks, in 1980 a 10MB Hard drive cost $3500 and in 2018 a 256GB ssd costs about $100 and fits in your pocket.

Processing power has also improved in accordance to Moore's law which states that the number of transistors that can fit into a single chip will double every 18 months.

Geoffery Hinton was one of the early pioneers of AI and he introduced what has become the backbone of the field. The idea is to mimic the operation of the human brain on a conceptual level. The human brain has billions of neurons and trillions of connections between them. The idea behind neural networks is to create an artificial structure with nodes or neurons that can compute and process information in a similar way to the human brain

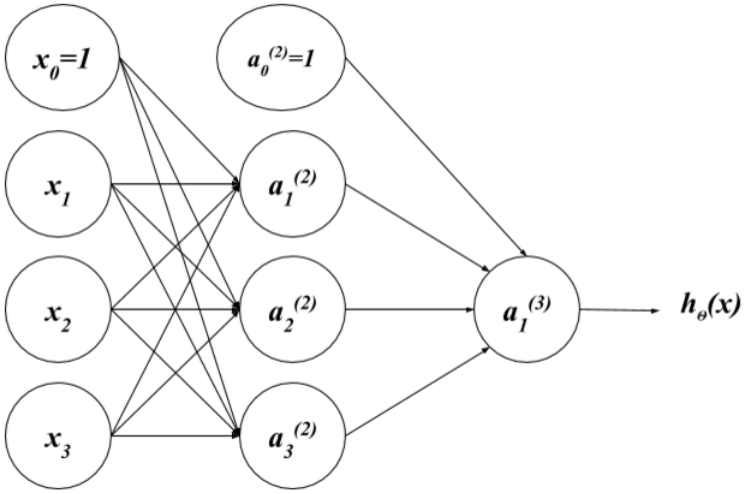

Figure 1: A Neural Network with one hidden Layer

Deep learning is so called because there are many hidden layers which are connected to each other to form th enetwork. More layers usually implies more abstraction

The Neuron

If we are to have a meaningful discussion about the history of neural networks we have to start at the first neural network, the one that functions in the human brain that makes us capable of higher thinking. The neuron is the basic computational unit of the brain. There are in excess of 80 billion $(8 * 10^{10})$neurons in the human nervous system and these are connected to each other by synapses (approximately $10^{14} - 10^{15}$ of them). Each neuron receives inputs from other neurons through its dendrites and produces outputs along

the axon which then connect to the dendrite of other neurons through synapses. It is this basic computation that is modeled (mimicked, if you will) in the mathematical model of the neural network shown in 2

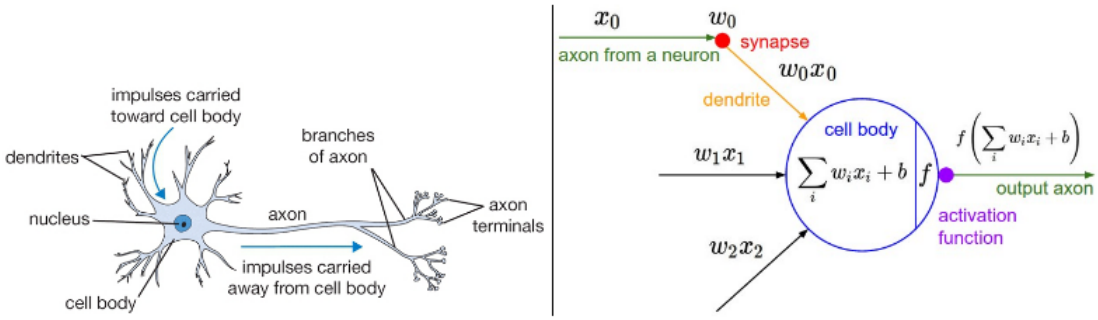

Figure 2: A biological neuron (left) and the mathematical model inspired by it (right)

The Perceptron

The Perceptron was invented by Frank Rosenblatt in 1957. The perceptron is an algorithm that does binary classification (ie it simply classifies a specific input as a 1 or 0). The definition of a perceptron is as follows.

$$f(x) = 1 if w.x > 0, 0 otherwise$$

The perceptron forms the basis for the feedforward neural network.

3.2 Introduction to Reinforcement Learning

Reinforcement Learning is a general term used to describe methods of online decision making where a computer program (refered to as the agent) is given a task by the programmmer and the agent is left to figure out an optimal way to achieve the task. Reinforcement learning is quite unique from supervised and unsupervised machine learning methods because it makes use of feedback loops from the environment (whether simulated or real) such that each action taken by the agent either has a reward or a penalty, after each time step the agent is given the current state (i.e an observation from the environment) and the reward associated with the action taken in the last time step. From this information, the agent figures out it's next step through the learning algorithm used. For example if we create a simulated maze and place an agent in the maze with the goal of solving the maze, the agent might take a long time to solve the maze but if we set rewards based on steps taken towards solving the maze and penalties if it wanders too far from the goal, the agent might be able to find an optimal solution for the maze.

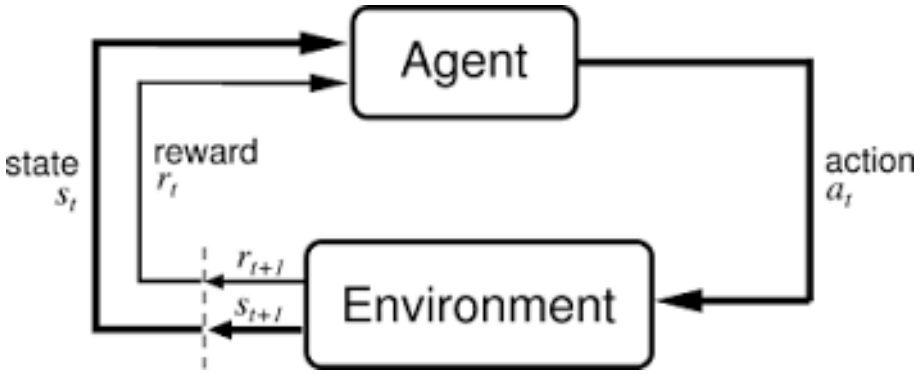

Figure 3: The reinforcement learning Model

One of the very useful features of reinforcement learning is the creation of general purpose agents that can solve multiple environments using the same (or very similar algorithms). This means that reinforcement learning methods have the greatest potential for producing the first true 'Strong' AI (Strong AI being the word in the AI community that encapsulates the goal to build a general purpose artificially intelligent agent that has an intellectual capability that is equally functional to a human [8])

Google's AI Division, Deepmind, was able to program an Agent (named AlphaZero) that mastered Chess, Go and Shogi to the point of being able to defeat word champion programs in each case [**?**]. This is in contrast their previous and well publicized program AlphaGo which used Deep Convolutional Neural Networks and a supervised learning techniques with training weights carefully selected and was programmed for narrow domain knowledge(ie the game of Go) [10]. Alpha Zero was programmed using a Monte Carlo Search Tree algorithm (in order to a deep neural network) and is able to train itself by simulation and self play

We run into a very important dillema in reinforcement learning called the Explore vs. Exploit dillema. Should the agent use a move it knows is likely to work? or try an experimental move in order to potentially gain better insight into optimizing its task. We shall see various methods of solving this dillema in the next section.

3.3 Elements of Reinforcement Learning

Environment - our agent performs actions in the environment

Agent - The agent learns through feedback from the environment (rewards and states) and takes actions to maximize rewards. The agent gets better at navigating the environment (i.e carrying out tasks in it) the more it navigates it. it does this by getting feedback gained on the actions performed.

A policy - A decision making function that describes what action to take in each situation

Reward Function - that defines the short term goals of the agent

Value function - defines what is a good action from the long term perspective

A model of the environment

3.3.1 Preprogrammed agent vs a Reinforcement Learning agent

Preprogrammed If we wanted to preprogram a simple walking robot, we have to think of all the steps that constitute the act of walking, it might involve something like the algorithm below

```
Result: Walk
while True do
    left leg forward;
    right leg forward;
    hold;
    /* contigencies                                    */
    if obstacle is encountered then
        long list of hardcoded contingencies
    end
end
```

Algorithm 1: Making a robot walk

You have to think of every case and exception (or at least quite a number of cases) for the robot to work acceptably

Reinforcement Learning agent No definitive process. Just give it general goals and define the degrees of freedom (like the actions that the agent is allowed to take). Rewards are assigned based on th desired goal or behaviour. The agent then figures out how to accomplish the goal based on these (by trying and experimenting). One of the large obervations encountered when using the reinforcement learning is that the agent usually comes up with novel solutions to solve the problems at hand. At the end of the day, the Reinforcement Learning agent may even be better than a preprogrammed agent

Intuitively, Reinforcement learning is trial and error(variation and selction/search) plus learning (association/memory). It addresses the problem of learning from interaction. Reinforcement Learning is a mathemathical prform of the problem of learning from interaction [13]

3.3.2 Learning from interaction

This means that the learner is very aware of how the environment responsds to its actions, and actively seeks to influence the environment. Example of learning from interaction: An infant playing has a sense/notion and a connection/association with its environment. It learns about cause and effect, consequiences of actions and how to achieve long term goals Learning from interaction

is influenced by animal learning psychology. The theory is of great demand in intelligent systems that operate in dynamic environments Many algortihms have been proposed for solving learning from interaction and the theory can be understood as combinations of a few underlying pricinples.

Learning and planning Reinforcement Learning can be applied to planning problems. Dynamic Programming(more on this in the next section) anticipates future consequences of its present actions ie how the environment will respond to its actions

Imagine trying to boil down the process of frying an egg into an algorithm

Result: Breakfast
while In Kitchen **do**
    Open fridge;
    reach for egg box;
    grab eggs;
    drop eggs on counter;
    go to cupboard;
    open cupboard;
    reach for frying pan;
    grab pan;
    set pan on cooker;
    Switch on cooker;
    Reach for cupboard;
    Grab oil;
    Pour oil on pan;
    drop oil in cupboard;
    for $n < 3$ **do**
        pick up egg from counter;
        Crack into pan;
        drop shell on counter;
    end
end

Algorithm 2: An Algorithm for frying three eggs

The process of frying an egg is a relatively simple process for an adult human but as we can see from a preliminary breakdown of its constituent tasks above it is not as simple as it first seems, because our brain abstracts a lot of the hardwork involved through complex algorithms.

The process involves judgements, order of retrieving objects and performing tasks(we cannot crack the egg before the oil is on the pan), eye movements to obtain information and finally locomotion

Choosing the next action correctly requires taking into account the delayed consequences of actions and this in turn requires foresight and planning

To make an agent useful to humans, we have to make its goals the same as ours (we do this through rewards)

Search and memory The second component of the learning from interaction model: Search and Memory Explore/Exploit dillema Reinforcement Learning combines the whole problem of a goal-directed agent interacting with an uncertain environment. Uncertainty is assumed at the beginning

3.4 Self Driving Cars

In 1989 D.A.Pomerleau of the Carnegie Mellon University Computer Science Dept designed ALVINN, an autonomous land based vehicle in a neural network.

Today, we have such high level examples of self driving cars using similar techniques as Dr Pomerleau pioneered almost 20 years ago.

In a report released by googles self-driving car project team,[3] it was revealed that the software has registered over 2 million miles/ 600 years of an average UK adults driving experience/ 154 years of an average US adults driving experience. (based on an average car mileage of 3300 mi. per person per year for UK drivers [2] and 13000 mi. for US drivers [1])

This is where machine learning (both reinforcement and deep learning techniques) really get to shine. While it is impossible for any human to get 150 years of driving experience there is currently a software system that has gained the equivalent of that in just about seven years of road testing

...our first million miles took six years to drive, but our next million took just 16 months Its important to note that approximately 90% of driving (where youre traveling on freeways, navigating light city-street traffic, or making your way through simple intersections) has already been mastered by Googles self driving algorithm.

There's about 10% left to overcome on the way to implementation of self-driving cars. This consists of:

The algorithm has to master what to do in cases that a lot of drivers experience only once in a lifetime for example seeing horseback riders in a middle of a road, a couple riding unicycles from side to side or seeing multiple cars driving the wrong way on the road.

The social side of driving i.e. understanding other road users and the signals they are giving off. For example if a road user puts on their indicator, he is signalling to slow down while he enters your lane. The software has to be able to take caution and interpret the motions and signals of cyclists and pedestrians in real-time

Advanced driving skills like navigating through construction zones, making way for emergency vehicles with sirens on(eg Ambulances, Police and Fire vehicles), driving through checkpoints and dealing with emergency road/ lane closures and defensive driving skills.

The reason this 10% has to be mastered almost to perfection is because of the high threshold self driving cars have to meet before theyĺl be considered by the public as lawmakers have to be sure that the technology is nearly error-proof

before allowing citizens to ride in them. While this 10% may seem like a lot especially the latter part, it is still being worked on and self driving is still very possible in the near future

The beauty of self driving cars is that it can allow subsets of people that previously couldnt́ go on the road alone (e.g seniors, the visually impaired, physically disabled people) to be able to ride in vehicles without any supervision.

3.5 Tools

A brief description of tools used in the experiments and simulations contained in this report:

Python, Matplotlib and numpy Python 3 was used for all the experiments and simulation. Python is the industry standard for contemporary deep learning applications. Most machine learning tools, from computation libraries like TensorFlow and Theano to deep learning tools like Keras and Testing Environments like Gym and Universe, are built around python. This makes it attractive for this project as there is no need to 'reinvent the wheel' with the multitude of APIs and tools present.

Matplotlib is a library used for statistical representations. It can be run directly for python and can be used to display Graphs, Histograms etc. It was used to plot the values of the simulations carried out below and compare them

Numpy is a very powerful and versatile Python package that allows us to work with matrices much like we would in a dedicated mathematics package (like Matlab).

Gym Gym is a toolkit for developing and comparing reinforcement learning algorithms[11]. It is an open source library consisting of test problems (refered to as environments) that can be used to test out reinforcment learning algorithms. It is especially useful because it aids in the development of general purpose algorithms as the environments have a shared interface. It works with Python code and is used in the second part of the Experimental section 'CartPole'

4 Reinforcement Learning Methods

4.1 Multi Armed Bandits

The multi armed bandit problem (also known as k-armed bandit problem) is a problem where limited resources must be allocated between alternative choices to maximize an expected reward. The name is derived from a thought experiment where a gambler stands in front of a row of k slot machines(aka bandits) and has to decide which machine to play in order to get the best reward. Does he choose the one he knows has a higher chance of winning (exploit) or a random one in order to get more information about the chances of winning(explore).

The above scenario is the explore-exploit dillema and is at the core of the multi-armed bandit problem. Because the gambler is completely clueless about the odds of winning in each machine at the beginining he must begin to collect data on the odds and then balance exploiting the known data with acquiring more data. Some algorithms used to solve this dillema are shown below

4.1.1 Epsilon Greedy

This is one way of solving the explore-exploit dillema In this method, we pick a value known as Epsilon (ϵ) which is usually 5-10%. This value represents how many times we shall explore rather than exploit.

Epsilon Greedy Algorithm

Result: A basic pseudocode representation of the Epsilon-greedy Algorithm

P = random() **if** $\underline{\text{P} < \epsilon}$ **then**
| pull a random arm
end
else
| pull arm with current best mean
end
update sample mean

Algorithm 3: Epsilon Greedy Algorithm

Updating the sample mean

$$\bar{X}_n = \frac{1}{N}\sum_{i=1}^{N} X_i + \frac{1}{N}X_N$$

$$\bar{X}_n = (1 - \frac{1}{N})\bar{X}_{N-1} + \frac{1}{N}X_N$$

Experimenting with different values of ϵ Please note that unless where explicitely stated otherwise, all the plots below are logarithmic plots

4.1.2 Optimistic Initial values

We want to overshoot the actual mean first by initializing it t a high value, then work down from there. In essence, we only explore the 'greedy' part of the epsilon greedy algorithm because we... This encourages the agent to explore early

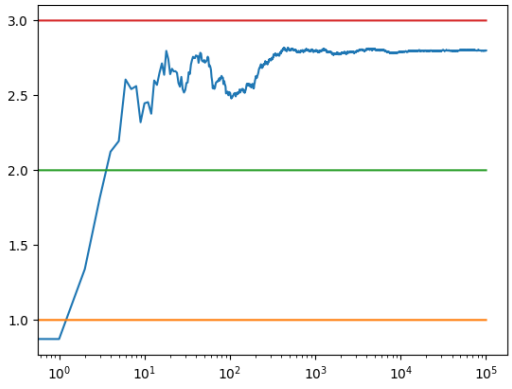

Figure 4: Epsilon Greedy algorithm; $\epsilon = 20\%$

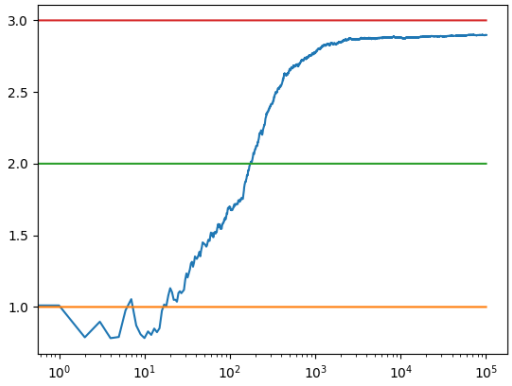

Figure 5: $\epsilon = 10\%$

4.1.3 Upper Confidence Bound (UCB1)

Chernoff-hoeffding bound

$$P\{|\bar{X} - \mu| \geq \epsilon\} \leq 2e^{-2\epsilon^2 N}$$

Confidence bound mean - we know that a mean from n_1=10 arguments is less reliable than one from n_2=1000 arguments hence a smaller confidence bound will be around n_1

Upper confidence bound

$$X_{UCB-J} = \bar{X}_J + \sqrt{2\frac{lnN}{N_J}}$$

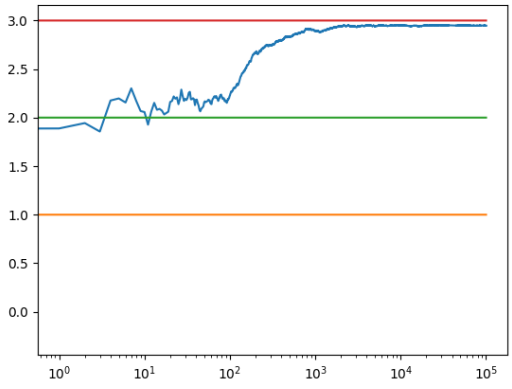

Figure 6: $\epsilon = 5\%$

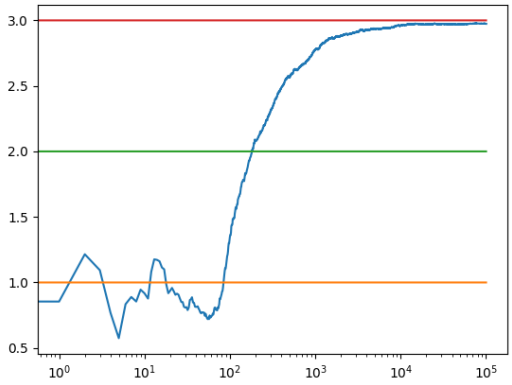

Figure 7: $\epsilon = 2.5\%$

where N is the number of times played N_J is the number of times played on bandit J X_J is the sample mean of the Jth bandit

4.1.4 Bayesian Sampling

We want to find the distribution of the parameters given the data [12]
Posterior distribution $P(\theta|X)$

$$P(\theta|X) = \frac{P(X|\theta)P(\theta)}{\int P(X|\theta)P(\theta)d\theta} = P(X|\theta)P\theta$$

$$likelihood * posterior$$

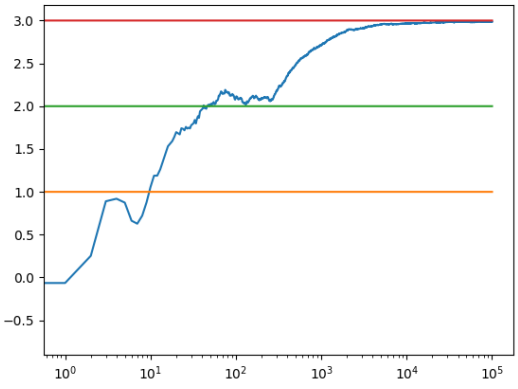

Figure 8: $\epsilon = 1\%$

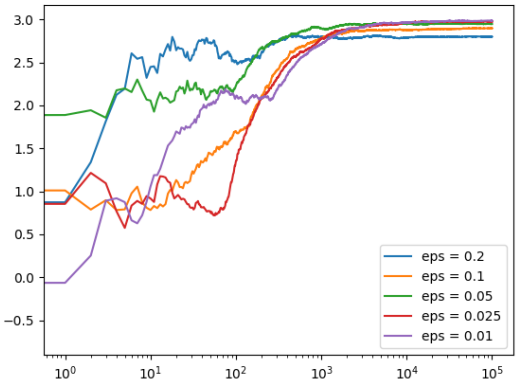

Figure 9: Comparing different values of ϵ

We have a problem here because the integral cannot always be solved. This problem is resolved by using the conjugate pair (a special pair of the likelihood and prior that can easily solve this)

likelihood is bernoulli

$$P(X|\theta) = \sum_{i=1}^{N=1} \theta^{1(x_i=1)}.(1-\theta)^{1(x_i=0)}$$

prior is the beta distribution

$$P(\theta) = \frac{1}{B(a,b)}\theta^{a-1}(1-\theta)^{b-1}$$

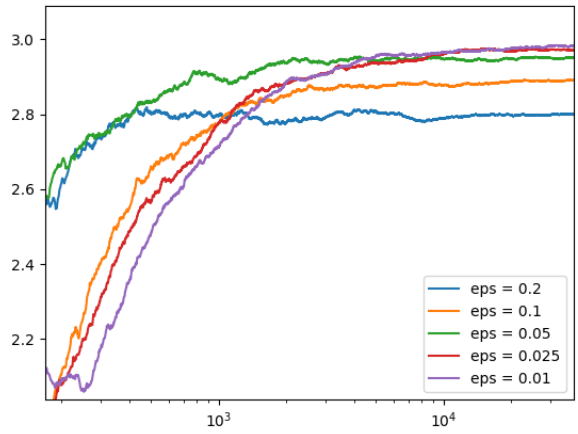

Figure 10: Shows the zoomed in convergence of different values of ϵ to the true mean

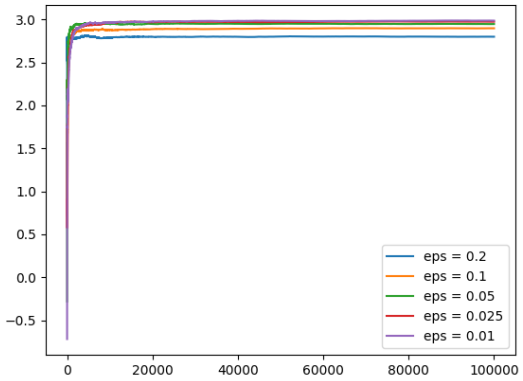

Figure 11: The linear plot of ϵ greedy

We can get the posterior by multiplying the likelihood and the prior and in the end we have:

$$P(\theta|X) = \theta^{\alpha} + \sum_{i=1}^{N} 1(x_i = 1)^{-1}.(1-\theta)^{b} + \sum_{i=1}^{N} ln(x_i = 0)^{-1}$$

Gaussian likelihood and prior

$$\lambda = \lambda_0 + T_N$$

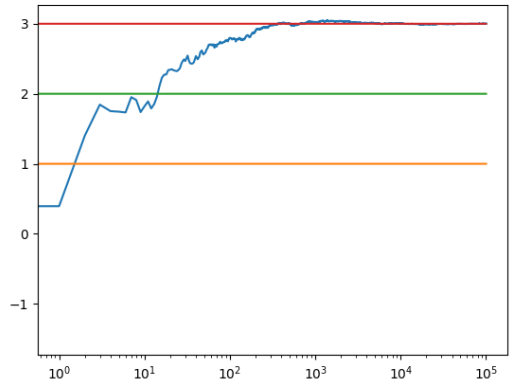

Figure 12: Optimistic Initial Values

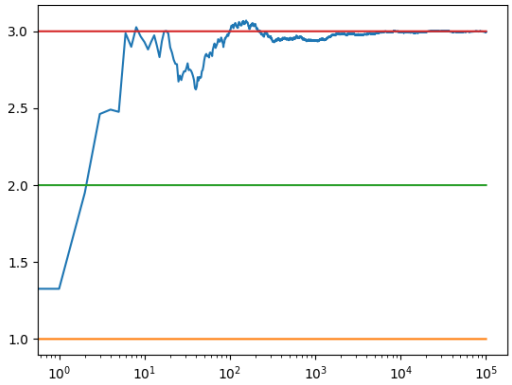

Figure 13: Upper Confidence Bound

$$m.\lambda = m_0 X_0 + \tau(\sum_{n=1}^{N} x_n)$$

therefore,

$$m = \frac{m_0 X_0 + \tau(\sum_{n=1}^{N} x_n)}{\lambda_0 + T_N}$$

Algorithm

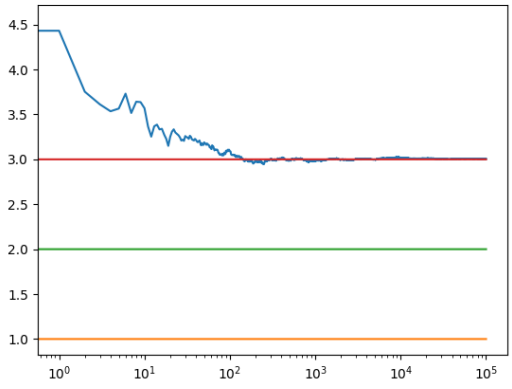

Figure 14: Bayesian Sampling

4.1.5 Decaying Epsilon

Instead of choosing a fixed value for ϵ that the agent will use to allocate explore/exploit resources, we use a decaying value that decreases with each iteration. So our ϵ is always $\frac{i}{i+1}$ where i is the number of iterations so far.

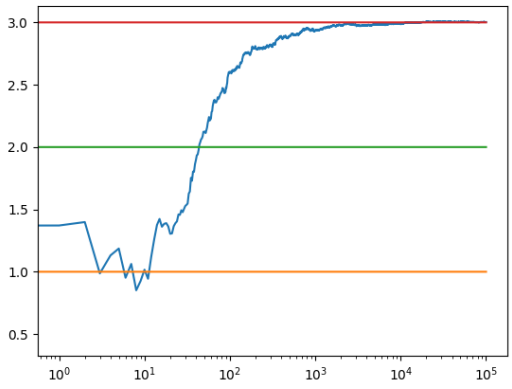

Figure 15: Decaying Epsilon Greedy

Finally, we compare the different algorithms on the same graph

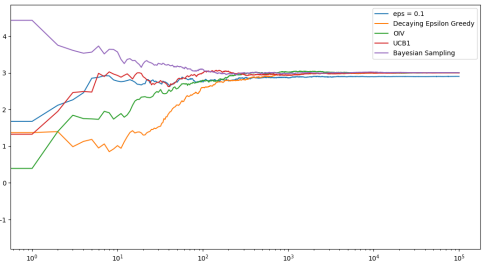

Figure 16: Comparing algorithms for solving the Multi-armed Bandit

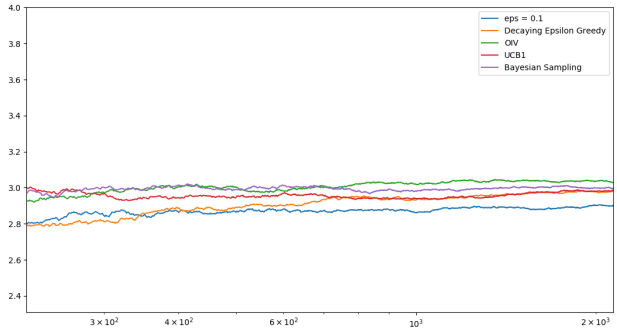

Figure 17: Shows the zoomed in convergence, notice that apart from vanilla epsilon-greedy, their performance is almost identical given enough time (somewhere around 10^2 iterations

4.2 Dynamic Programming

Dynamic programming refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of an environment as a Markov Decision proces [**?**]

A Markov property explains to us how many previous values of x our current probability is dependent on. So a first order Markov of

$$Px_t|x_{t-1}, x_{t-2}, ..., x_1 = P(x_t|x_{t-1})$$

. i.e P is only dependent on the previous value of x. We shall deal mostly with first order Markovs in this research

A Markov Decision Process is any reinforcement learning task that follows the markov property. It usually consists of a five-tuple (set of states, set of actions, set of rewards, state transition probability, discount factor)

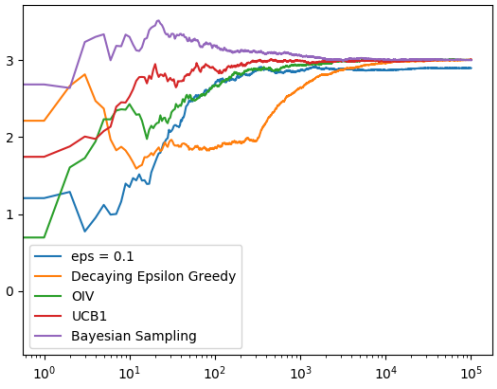

Figure 18: Simulation #2

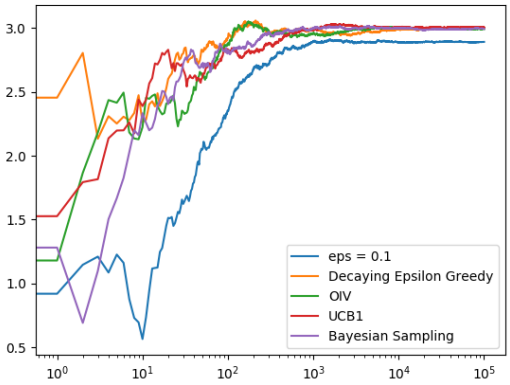

Figure 19: Simulation #3

A Policy(represented by π) is the algorithm that the agent is using to navigate the environment

The state transition probablility - We think of agents as probabilistic because the agent may not always have a perfect knowledge of the state because it can only estimate it by what it senses. An example would be in a real-time 3d game, if an agent controlling the player character judges that flicking the left analog stick forward will put it in a certain state where the character would have moved a certain number of units forward from its current point. If something else that the agent cannot sense interferes with the movement (a very strong

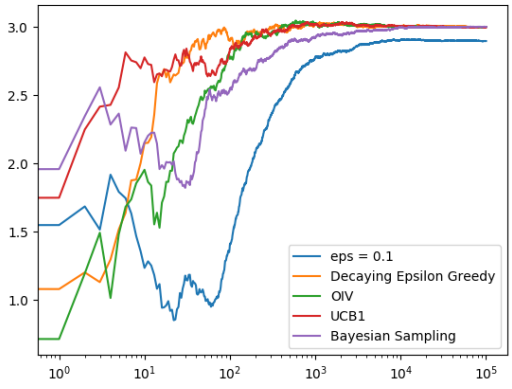

Figure 20: Simulation #4

wind for example), the character would not move into the exact state intended hence the State transition probability. It is represented by $P(s', r|s, a)$

Future Rewards : A function that represents the sum of all rewards gotten over time i.e total reward

$$G(t) = \sum_{\tau-1}^{\infty} R(t+\tau)$$

Discount Factor (γ) This is used to tune the greed of the agent ie how it judges future rewards vs immediate rewards. If $\gamma = 0$ thene we have a totally greedy algorithm that maximizes the immediate reward of the next action without regard to the total goal. On the other hand, if $\gamma = 1$ the agent weighs all actions equally no matter how far in the future the rewards will come. In practice, γ is usually set to a high value 0.9. The practical reason we discount future rewards is because the further the reward is in the future, the harder it is to predict. Adding, the discount factor, our new function G(t) of future rewards can be rewritten as follows:

$$G(t) = \sum_{\tau-1}^{\infty} \gamma^T * R(t+\tau+1)$$

Richard Earnest Bellman (1953) The Bellman Equation State, action, γ Discount, R - reward

V=1 is the percieved value of being in this state as once we are in this state, we are just one step away from victory

We can put all the states in our path with V=1 but imagine the agent is put in another starting positino

$$V(s) = max_a(R(s,a) + \gamma V(s'))$$

where R(s,a) is the reward for the enxt action V A state is more valuable the coloser you are to the finish line. We can replace the values with arrows creating something akin to a treasure map for our agent

Theory of Dynamic Programming (Bellman)

Markov Decision Process A Deterministic process is one in which we have a 100% probability of our action translating into the desired next state ANon-deterministic (stochastic) process is one in which there is a possibility that our action will not result in the desired next state

A markov process depends on how our environment is designed MDP - framework to navigate environment

A MDP provides a mathematical framework for modelling decision making when outcomes are partially random or partially under our control It is a framework that the agent usies in an uncertain event

$$V(s) = max_a(R(s,a) + \gamma \sum_{s'} P(s,a,s')V(s'))$$

the probability of getting into our desired next state is represented by psasvs' This is the Bellman equation

Policy vs Plan A plan assumes a deterministic (ie non-stochastic) environment

Living penalty Negative rewards can act as an incentive for an agent to solve the environment quickly In our gridworld example, an incentive to exit the game quickly

We can see from the above that the optimal policy changes with different reward policies

The Value function and Bellman Equation Expected value - tells us the mean of . (Ironically, it is a value that can never be expected as an outcome). $E(X) = \sum_x P(x).x$ The Value function V(s) = E(G—s) - (what is the sum of rewards I can expect in the future given my current state s)

$$V(s_1) = r_2 + r_3 + r_4 + ... + r_N$$

$$V(s_2) = r_3 + r_4 + ... + r_N$$

Therefore, $V(s_1)$ can be written as $r_2 + V(s_2)$ Adding discount -

$$V(s_1) = r_2 + \gamma V(s_2)$$

Bellman's equation:

$$V(s) = E[r + \gamma V(s')]$$

using example of gridworld

The Value function is dependent only on the future staes; V(s) of the terminal state is 0

$$V_\pi(s) = E_\pi[G(t)|s_t = s]$$

$$E_\pi[R(t+1)|s_t = s] = \sum_a \pi(a|s) \sum_r rp(r|s,a)$$

$$= \sum_a \pi(a|s) \sum_{s'} sum_r p(s', r|s, a)$$

so,

$$V_\pi(s) = E_\pi[R(t+1) + \gamma \sum_{\tau=0}^{\infty} \gamma^T R(t+\tau+2)|S_t = s]$$

$$V_\pi(s) = \sum_a \pi(a,s) \sum_{s'} \sum_r P(s', r|s, a)\{r + \gamma E_\pi[\sum_{\tau=0}^{\infty} \gamma^T R(t+\tau+2)|S_t + 1 = s']\}$$

$$V_\pi(s) = \sum_a \pi(a,s) \sum_{s'} \sum_r P(s', r|s, a)\{r + \gamma V_\pi(s')\}$$

The State value function is $V_\pi(s) = E_\pi[G(t)|s_t = s]$ while the action value function, also refered to as the Q value is $Q(s,a) = V_\pi(s) = E_\pi[G(t)|s_t = s, A_t = a]$

Optimal policies and Optimal Value Functions A policy

$$\pi_1 \geq \pi_2 \quad if\ V_{\pi_1}(s) \geq V_{\pi_2}(s) \qquad \forall_s \epsilon S$$

So, for the best policy π_*

$$V_*(s) = max_\pi(V_\pi(s)) \qquad \forall_s \epsilon S$$

The best policy is the one which leads to the greatest value function, our goal is technically not to find the optimal policy but he optimal value function as multiple policies can result in the same optimal value function

Optimal action value function

$$Q_*(s,a) = max_\pi\{Q_\pi(s,a)\} \qquad \forall s \epsilon s, a = A$$

$$= E[R(t+1) + \gamma V_*(S_{t+1}|S_t = s, A_t = a]$$

Q gives us a definite best next step to take given the current state. We simply choose the *argmax* If we only had V, we would have to try out all possible next steps first and then look for the best action to maximize the value function

Bellman Optimality Equation

$$V_*(s) = max_a E[R(t+1) + \gamma V_*(S_{t+1}|S_t = s, A_t = a]$$

$$= max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V_*(s')]$$

$$Q_*(s, a0 = E[R(t+1) + \gamma max_{a'} Q_*(S_{t+1}, a')|S_t = s, A_t = a]$$

$$Q_*(s,a) = \sum_{s',r} p(s', r|s, a)[r + \gamma max_{a'} Q_*(s', a)]$$

Result: iterative_policy(π)
initialize $V(s) = 0$ for all $s \in S$;
while True **do**
  $\Delta = 0$;
  for each $s \in S$ **do**
    old_v=V(s);
    V(s)=$\sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\{r + \gamma V(s')\}$;
    $\Delta = max(\Delta; |V(s) - old\_v|)$;
  end
end
if $\Delta <$ treshold_value **then**
  break;
end
return V(s);

Algorithm 4: Iterative Policy Evaluation

Dynamic Programming Define Two problems Prediction problem: Find V(s) for a given policy Control problem: finding the optimal policy

The algorithm above solves the prediction problem using Dynamic programming

Using the current policy, we can get the state-value function. So can we change just one action of s; $a! = \pi(s)$

$$find \qquad a \epsilon A \quad suchthat \quad Q_\pi(s, a) > Q_\pi(s, \pi(s))$$

Is there any action that gives us a bigger Q? if so, change policy for current state to new action

We want a new policy $V'_\pi(s)$; where $V_\pi(s) < V'_\pi(s)$ If we already have Q, then

$$V'_\pi(s) = argmax_a Q_\pi(s, a)$$

$$= argmax_a \sum_{s',r} p(s', r|s, a)[r + \gamma V_\pi(s')]$$

This is called a lookahead search

Policy improvement is greedy, not global. The optimal policy will generate a value fuction that will not be able to be improved further

Policy Iteration How do we fix an out-of-date value function given a new policy We recalculate it So Policy iteration comprises of two steps: policy evaluation then policy improvement

Result: Perform Policy iteration
randomly initialize V(s) and $\pi(s)$;
V(s)=iterative_policy(π);
policy_changed=False;
for $s \in$ all_states **do**
  old_a=policy(s);
  policy(s)=argmax[a]$\{\sum_{s'r}\{p(s', r|s, a)[r + \gamma * V(s)]\}\}$;
  if policy(s) != old_a **then**
    policy_changed=True;
  end
  if policy_changed=True **then**
    recalculate value function V(s);
  end
end

Algorithm 5: Policy Iteration

4.3 Q-learning

4.3.1 Intuition

Q-represents the quality of the next action Bellman equation - $V(s) = max_a(R(s, a)+ \gamma \sum_{s'} P(s, a, s')V(s'))$

$$Q(s, a) = R(s, a) + \gamma \sum_{s'}(P(s, a, s')V(s'))$$

$$therefore V(s) = max_a Q(s, a)$$

The value of a particular state is te maximum Q value. The Q value is the expeted reward of performing a certain action.

$$Q(s, a) = R(s, a) + \gamma \sum_{s'}(P(s, a, s')max_{a'}Q(s', a'))$$

4.3.2 Temporal difference

Remember that all values V(s) are recursive, ie you can calculate one from the others Deterministic bellman form of Q:

$$Q(s, a) = R(s, a) + \gamma max_{a'}Q(s', a')$$

The agent recalculates the value of the state after taking a certain action . Temporal difference is the Q value minus the expected Q-value from the previoius experience

$$TD(a, s) = R(s, a) + \gamma max_{a'}Q(s', a') - Q(s, a)$$

$$therefore Q(s, a) = Q(s, a) + \alpha TD(a, s)$$

α is the learning rate. It determines how fast we want our agent to learn If the situation that the agent just arrived at occures just 1% of the dime, we do not want our agent to ditch the entire policy in favour of this new value)

$$TD(a,s) = R(s,a) + \gamma max_{a'} Q(s',a') - Q_{t-1}(s,a)$$
$$Q_t(s,a) = Q_{t-1}(s,a) + \alpha TD_t(a,s)$$
$$Q_t(s,a) = Q_{t-1}(s,a) + \alpha[R(s,a) + \gamma max_{a'} Q(s',a') - Q_{t-1}(s,a)]$$

4.3.3 Deep Q networks

Q-values optimized through a neural network.

The Q-values are passed through a softmax function to select the best possible action

Softmax function

$$f_j(z) = \frac{e^{zj}}{\sum_k e^{zk}}$$

In softmax, we take the best action the number of times of its probablity. So if a certain action is 91% certain to be right, we take that action 91% of the time and explore other actions the remaining 9% of the time. So unlike ϵ-greedy, our ϵ here is based on the Q-values not just a preprogrammed value

4.3.4 Experience Replay

Learning: Update weights slowly to get better at solving environments The weights are update and backpropagation is done for every state. The interdependency of states will make the network overfit to a particular example or section of the example and this is undesireable. See **??**

We can mitigate this by using Experience replay - We store the previous states in batches and at a certain treshold, the agent updates it neural network with a uniformly distributed sample of the batch experiences.

The advantage of experience replay: because we are slicing up experiences in batches, the agent can learn from the same experience multiple times. We use a rolling window to do this

Priorirtized Experience replay Prioritized experience replay allows the agent to learn more efficiently by replaying more significant experiences mroe frequently. Experience replay can reduce the amount of experience needed to learn and relace it with more computation and memory (both resources are cheaper from the agent's point of view)

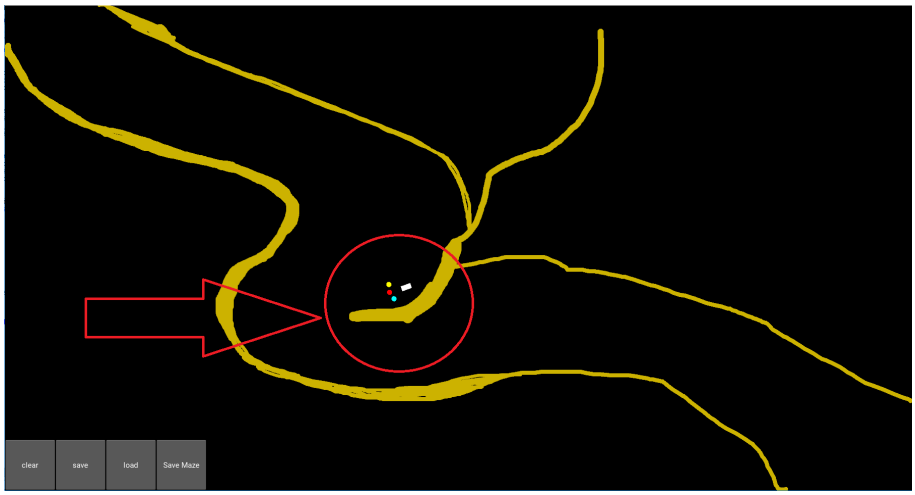

Figure 21: The car could get stuck around a corner like the one highlighted

Action Selection policies

- ϵ-greedy - take the best Q(ie Q_{max}, explore for ϵ% of the time and exploit for the remaining $(1-\epsilon)$% of the time
- ϵ-soft - exploit $(1-\epsilon)$% of the time and explore ϵ% of the time
- Softmax

The reason we don't just choose the best action is because of the Explore-exploit dillema

4.4 Monte Carlo Methods

Any algorithm that involves a significant random component (in this case our return is random) Instead of calculating E(G), we calculate the sample mean Only episodic tasks as we can only calculate G after every episode Every state is a multi-armed bandit problem

We run into two problems:

- Prediction problem (finding the Value function given a policy), solved by Policy evaluation
- Control Problem (finding the optimal policy), solved with Policy iteration

$$V_\pi(s) = E[G(s)|s_t = s]$$

$$V_\pi(s) = \frac{1}{N}\sum_{i=1}^{N} G_{i,s}$$

where i is the episode and s is the state

We play a bunch of episodes and log states and reward sequences

$$G(t) = r(t+1) + \gamma * G(t+1)$$

We calculate G by going through the states in reverse order; since G depends on the future values. After multimple episodes, we can take sample mean of states (s) ffrom the set of G

First visit or every visit monte carlo If you get multiple states with the same reward

4.4.1 Monte Carlo Algorithm

Result: firstVisitMonteCarloPrediction(π,N)
V=rand_int;
all_returns={} **while** i¡ N **do**
  States,returns = play_episode **for** s,g in zip(states,returns) **do**
    if s is not seen in episode yet **then**
      all_returns[s].append(g) V(s)=sanple_mean(all_returns[s])
    end
  end
  i++;
end
return V

Result: Calulating returns from rewards
s=grid.current_state() States_and_rewards =[(s,r)] **while** !game_over **do**
  a=policy(s) r=grid.move(a) s=grid.current_state(s)
  States_and_rewards.append((s,r))
end
G=0 States_and_returns=[] **for** s,r in reverse(States_and_returns) **do**
  States_and_returns.append((s,G)) G=r+gamma+G
end
States_and_rewards

Unlike Dynamic programming, we don't have to loop through all possible states just the states that we actually visit

Monte Carlo in a 'windy' environment (i.e. there is a State transition probablility ¡1) after a=policy[s];

add a= random action[a]

This way, the wind chooses another action

4.4.2 Solving the Control problem - finding the optimal policy with Monte Carlo

Since we only have the value function $V(s)$ for a given policy, we don't know what will lead to a better V(s) without a lookahead search. We solve this by playing multiple episodes and getting back the states and returns

Use Q(s,a)
then argmax[a]Q(s,a)
so instead of just returning s,G
we return s,a ,G

Problem: If we follow a fixed policy, we can only perform one action per state, so we can only fill in $\frac{|S|}{|S|*|A|} = \frac{1}{A}$ values in Q We solve this by using exploring starts method A state and initial action are randomly chosens, then the policy is followed from there

$$Q_\pi(s,a) = E_\pi[G(t)|s_t = s, A_t = a]$$

We run into another problem when we implement Monte Carlo with exporing Starts for policy evaluation and $argmax[a]Q(s,a)$ for policy iteration because there is an iterative algorithm within an iterative algorithm. We remedy this by doing the policy implementation after every episode instead of after every round

4.4.3 Monte Carlo with Exploring starts

Q=random, π=random WhileTrue s,a=random selection from S and A
states,actions,returns = playgame(start=(s,a)) Fors,a,G in
states,actions,returns returns(s,a).append(G)
Q(s,a)=average(returns(s,a)) For$s \in$ states pi(s)=argmax[a]{Q(s,a)}

Interestingly, this method converges even though the samples for Q are from different policies. If Q is subobtimal, then the policy will change causing Q to change until we get to a stable state where the value and policy converge to the optimum value and optimum policy.

4.4.4 Monte Carlo summary

In Dynamic programming, we know all the state transition probablilities (because they are constant) and do not actually explore teh game environment. In Monte Carlo, we are able to learn from experience

Monte Carlo can be more efficient than Dynamic programming because we do not have to loop through all the possible game states. We mioght not get the "full" value function using Monte Carlo but this is OK because we may never reach all of the states Since Monte Carlo results in each state being visited a different amount of times (depending on the policy), we use exploring starts

smethod to make sure that w have adequate data for each stare All Markov Decision Processes are like having different multi-armed bandit problems at each state

4.5 n step Q learning

In Gridworld, we get information about the environment as a vector this is called "God-mode" where we can see every possible state of the game. This does not accurately simulate a real life problem. The agent (eg. A self driving car) will usually get inputs from the environment as a matrix on pixels and has to interprete the meaning of these pixels

Eligibility Trace In Q learning, the agent chooses an action, takes it, computes the reward and learns from this action Eligibility traces (n-step Q-learning) chooses n actions, takes them, calculates the cumulative rewards and learns from this batch of actions

Eligibility trace can be thought of as a middle ground between temporal difference and Monte-Carlo methods.

Eligibility traces illustrate that a learning algorithm can gain computational advantages by batching the actions

4.6 Primer on Artificial Neural Networks

Since we are going to design agents that use Reinforcemnt learning techniques connected to Artificial Neural Network "brains" (to maximize the Q-value), we shall briefly explore Neural Networks and Convolutional Neural Networks

Inputs are independent variables of a single observation. Variables are usually standardized (eg by normalization).

The output can either be - continuous (used for prediction, eg based on the information provided, what is the estimated value of this house? - binary(used for single class classification, eg does this xray show cancerous cells or not?) - categorical (used for multi class classification, eg which of these ten animals is shown in the image above)

Weights - This is how Neural Networks learn, they get adjusted during the learning process.

Inside the neuron First, a weighted sum of all the input values is found:

$$\sum_{i=1}^{m} w_i x_i$$

Then, an activation function is applied to the weighted sum (which determines which signals will be propagated through the network

4.6.1 Activation Function

Treshold Function

$$\phi(x) = 1 if x \geq 0, 0 if x < 0$$

Sigmoid Function

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

It is used in the output layer to predict the probability in multi-class classification problems

Rectifier function

$$\phi(x) = max(x, a)$$

Hyperbolic Tangent Function

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

How NNs work Imagine a Neural Network created for predicting house prices, it has already been trained and the weights adjusted accordingly

Hidden layers correlate the findings from the data to a methodology making it possible to predict the solutions of examples that the neural network has not seen previously Each hidden layer is weighted to calculate the price of the property

Perceptron

$$Costfunction = \frac{1}{2}(y - \hat{y})^2$$

Gradient Descent A brute force approach to minimize the cost function will not work, especially with current technology. Imagine a simple fully connected neural netowrk with one hidden layer consisting of five nodes, this network will have 25 weights to compute. Now imagine using the current world's fastest supercomputer the Sunway Taihulight**??** with a compute capacity of $93 * 10^{15}$ FLOPS to train this neural network with a thousand examples (which is a small sample size in contemporary machine learning), if each weight uses one FLOP for calculation(which is an underestimation) we will need $100^{25} = 10^{75}$ FLOPS to minimize the cost function and this will take $1.08 * 10^{58}$ seconds which is equivalent to 34 million years! This is the reason gradient descent is used. Gradient descent algorithms minimize the cost function by using the slope instead of simulating all possible combinations of the data.

Stochastic Gradient Descent One drawback of gradient descent is that it always requires the cost function to be convex. If Gradient descent is used in the figure above, the algorithm might get stuck at a local minimum which is undesireable as we want to find the global minimum.

$$c = \sum \frac{1}{2}(y - \hat{y})$$

In SGD, we adjust the weights after every data entry and the advantage is that we are able to always find the global maximum

Back propagation
The backpropagation algorithm was introduced in the 1970s but it's importance was first identified in 1986 by Rumelhart, Hinton and Williams [20]. Their paper described neural networks designed to continually change weights and minimize errors through backpropagation. It made neural networks a viable alternative for solving problems which were hitherto considered insoluble. Today, the backpropagation algorithm is at the core of learning in most neural networks.

4.7 Primer on Convolutional Neural Networks

When a person sees an image, the human brain looks for features in order to classify the image. This why optical illusions seem very confusing as the brain sees features from two seperate classes. The brain ties to rectify the two different object hence the illusion. An example is the image before considered the first published optical illusion. Is this a duck? or a rabbit?

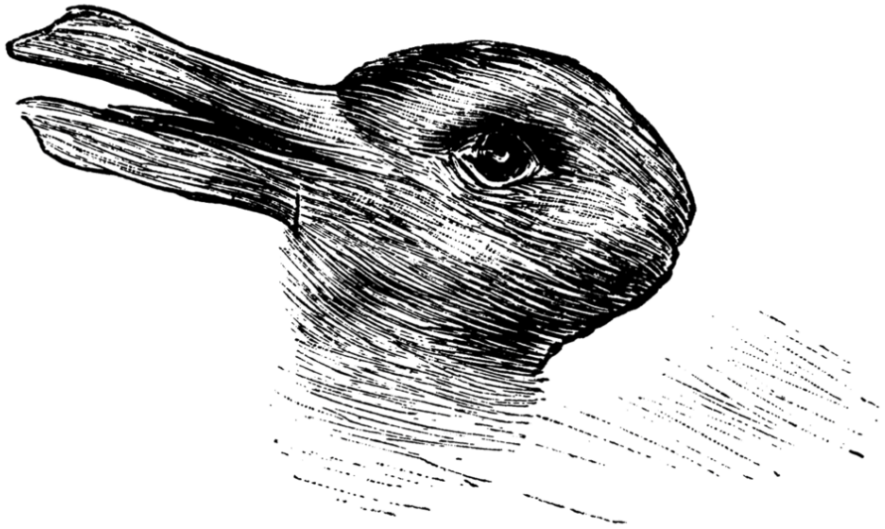

Figure 22: An optical Illusion: is this a rabbit? or A duck?[**?**]

Convoluted Neural Networks are based on the idea of replicated features. maps detect features. When a feature is detected, its exact location is less important than its approximate location in relation to other features. This is especially useful because we can have zoomed in, flipped, and distorted variations of the same image that will increase the algorithm's accuracy. Also, a lot of object eg a cat or a human face have features that are usually roughly in the same positions relative to other features. Using the human face, we know that the two ears would be at the edges of the face and the eyes a little inward, the

nose below the eyes, the mouth below the nose and so on. Having more of these features in the 'traditional' position increases the confidence of the algorithm when predicting the image to be a certain class. Also of note and importance is the fact that features do not necessarily have to be precisely identified as this is irrelevant and potentially harmful to our goal. This is because the positions and structures of different features are likely to vary even among objects in the same class. It is for this reason we use multiple subsampling layers (also known as pooling layers) to reduce the resolutions of the feature maps. Pooling helps us reduce the sensitivity of our output to shifts and distortions. MatConvNet uses stochastic gradient descent, an algorithm similar to fminunc discussed above.

How do CNNs recognize images Any black/white image can be represented digitally as a 2-dimensional array with each cell having an intensity value between 0 and 255. Condequently, any coloured image can be represented as a 3-dimensional array with each pixel having 3values (for the Red, Green and Blue channels)

4.7.1 Convolution Layer

The convolution layer is so named because it uses the convolution operation

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$

We convolve the feature detector(sometimes called a filter) with the input image and keep moving the feature detector over the image in steps of s(called the *stride*). This creates the feature map - A matrix that shows how each section of the input image matches with the chosen filter. A higher value in any cell of the feature map means there are more matches to the feature in that section of the image. These features could be things as simple as curves to whole body parts (e.g floppy ears to recognize a dog) where a 'body part' will be defined by the Convolutional neural network as a combination of various simple features of curves and lines close to each other in the image. The stride amount reduces the image detail so we might lose some information by doing this but this is usually acceptable as

Feature maps help us get rid of the unnecesary details and focus on the features we need to recognize the image The network, through its training, determines which features are important to recognize each particular image and it creates different filters to recognize these different features of the image. So, there are usually a lot of feature maps in the convolutional layer depending on how many feature maps are needed to recognize the image

A simple example of a filter/feature detector is the edge detect

The Convolutional Neural network creates its own features based on what it decides is neccesary to recognize the image. These features may be unrecognizable to the human eye

4.7.2 ReLU

ReLU stands for rectified linear unit. It is usually applied element-wise to the output of some other function (i.e it is usually not the first layer of a Convoluted Neural Network).

A ReLU is defined as

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

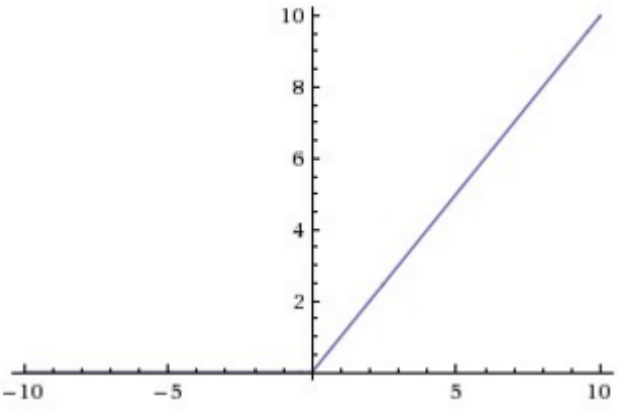

Figure 23: the ReLU activation function, which is zero when x¡0 and then linear with a slope of 1 when x¿0

ReLU units are used in Convolutional Neural Networks as a way to introduce non-linearity into the network since our data from the real world (e.g images, text, audio) is mostly non-linear, and the Convolution operation is a linear one (i.e element wise matrix multiplication and addition). Other non-linear functions can be used in place of ReLU are the sigmoid function (discussed above) and the tanh function. ReLU operations have however proven to perform better in most situations. ReLUs improve networks because they speed up training, this occurs because they are computationally simple (as opposed to sigmoid and tanh which involve exponentials). The result of this is that they are also faster. Another advantage of ReLU over sigmoid functions is that ReLU has a stable gradient value when $x > 0$ whereas the sigmoid function becomes increasingly small as the x increases. ReLU units are not without their disadvantages though, one common problem is that a large gradient flow through a ReLU neuron could cause a change in weight that effectively 'kills' the neuron. That is, the neuron will never activate from that point on and the gradient flowing through that neuron will be zero from that point on. A good counter to this problem is to monitor the learning rate to make sure it isn't set too high and killing the neurons.

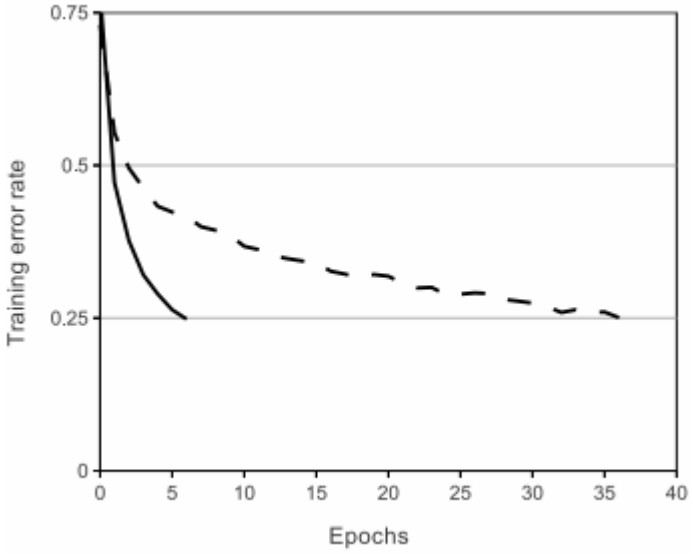

Figure 24: Krizhevsky et al. [15] found the ReLU operation was able to reach a 25% error rate on the CIFAR-10 database six times faster than an equivalent network with tanh neurons

4.7.3 Max)Pooling (also known as downsampling)

We want the Neural Network to recognize the image even when it is somehow distorted (eg the subject is looking in a different direction, lighting conditions are different, different image orientation) or when we have different variations (eg closeups, wide shots, full body or face shot) of the same subject. We want the Neural network to recognize the subject irrespective of where certain features are absolutely located in the image. This is called spatial invariance

From the above, we can see how the MAX pooling operation works in a Convolutional Network. A filter is placed in a layer(in our case 2*2) and that section of the network is downsampled to a 1*1 matrix using the highest value in the filter, this is done throughout the network using the stride specified (2 in this case). In 26, this reduces the 4*4 layer to a 2*2 layer. The intuition behind max pooling: If you had 4 districts in a city and wanted to know whether it was raining in that city, you don't add up all the values (true or false) of rain in the 4 district but simply look out for a true value as you cycle through the status of each district. Max pooling usually outperforms other downsampling operations in Neural networks [**?**] hence it is used as the default method of pooling in most

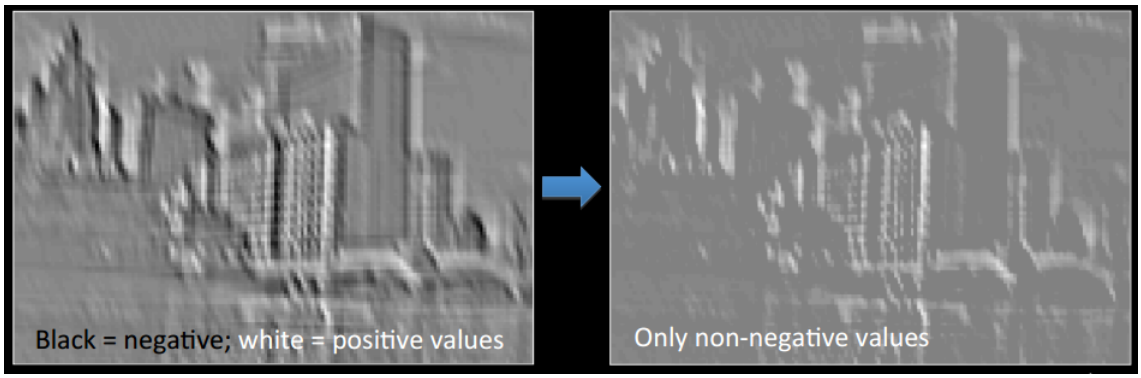

Figure 25: A ReLU operation applied to a feature map. The output map is the 'rectified' source map[16]

ConvNet architectures. The reason why maxpooling works so well is this: A feature usually doesn't appear twice in a small neighorhood (for example if a particular filter is looking for the pointy ears to identify a dog, there would not usually be another pointy ear in very close proximity). Therefore, if a feature is detected at the first cell we usually do not need to look at the other three cells because it is less likely to be present in any of them. If we took the average of all the features in the neighborhood we would get a result that is lower than the actual value (i.e we would get a result that is more computationally expensive and less accurate).

By undergoing pooling, we further compress the image and this is an important step in preventing overfitting.

Many object recognition architectures are based on the model of the mamalian visual cortex. Visual area V1 has simple and complex cells[7][6]. Simple cells perform feature extractions while complex cells combine several such local features from a small spatial neighbourhood. This spatial pooling is cruicial for forming invariant features. The subasmpling technique was the previously favoured method for pooling, it propagates the average of a set of local features to the next layer but in 2010, Dominik Scherer[6] was able to demonstrate the superiority of max pooling over subsampling and consequently most contemporary convolutional neural network architectures use max pooling.

$$a_j = tanh(\beta \sum_{NxN} a_i^{nxn} + b$$

Subsampling takes an average over the inputs, multiplies it with a trainable scalar β and adds a trainable bias b

$$a_j = max_{NxN}(a_i^{nxn} u(n,n))$$

Max pooling applies a filter (or window function) $u(n,n)$ to the input patch and computes the maximum value in the neighbourhood

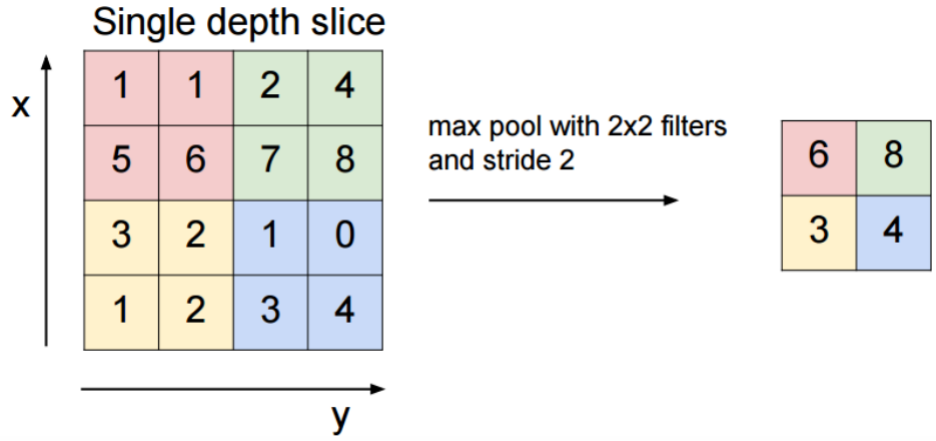

Figure 26: Single depth max pool with 2*2 filters and stride of 2

4.7.4 Flattening

The objective of this process is to make the pooled image into a column vector in order to feed this into the input of the neural nework

4.7.5 Full Connection

Finally we have a fully connected layer before the output. To train the Convolutional Neural Net, forward propagation is used and the errors are computed. Backpropagation is used to adjust the weights of the neural networks and finally the features detectors are also adjusted. Through multiple iterations the final fully connected layer gets certain features that correspond to certain classes in the output value, this forms a voting mechanism where a high correlation of certain features means the image will be classified as the desired class

5 Experiments

5.1 CartPole

The principles learned from the multi armed bandit can be applied to other environments Markov Decision Processes (MDPs) with similar results, we shall attempt to do that here: Cartpole is a simple physics game, it involves training an agent to balance a pole on a cart by using the left and right key, the environment being the pixels on the screen

The gym environment provides us with a good API so we can focus on programming our agent (refer to explanation above)

A simple random search algorithm was implemented and the simulation was run 200 times

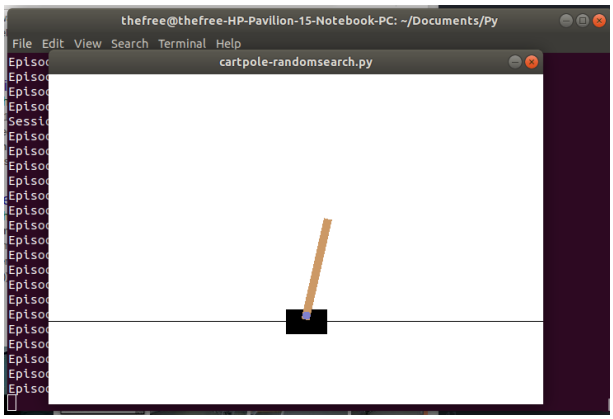

Figure 27: The GUI of the cartpole game with the terminal running in the background

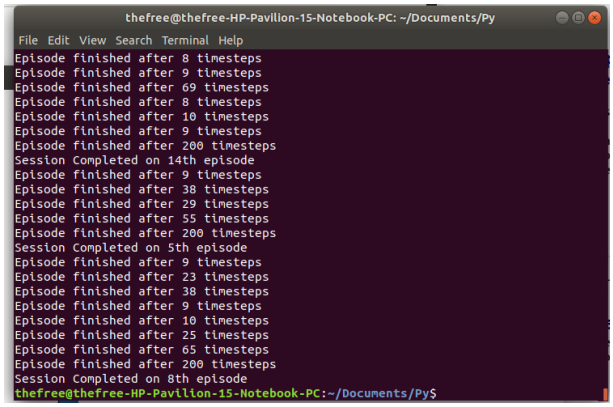

Figure 28: The Outputs of episodes and sessions. An episode fails if the Agent is unable to keep the cart between +/-2.4 units from the center and the pole between +/-12^0 from the center. An episode suceeds if the agent meets the above balancing requirements for 200 consecutive timesteps

The histogram gives us the right part of a normal distribution *ie log normal*, this is expected because we plugged in random numbers into the random search algorithm and our histogram only details speed of the successful experiments.

5.2 Self Driving Car

A simple artificial Neural network was built to solve an envrionment (designed by [22]). The environment provides a simple representation of a car (a box and two circles) with sensors and drawing tools so a road can be constructed in real time.The goal of the agent in this environment is move between the upper-left

```
run session initialize best_params initialize best_reward for i in range
 1000 do
    parameters = random() initialize total_reward as zero for i in range
     200 do
        Ifparameters*observation¡0 action=0 else
            action=1 take next step
        end
        return reward
    end
    Ifreward ¿ best_reward best_reward = reward best_params =
     parameters  if reward = 200 then
        break
    end
end
return session_complete
```

Algorithm 6: The random search algorithm

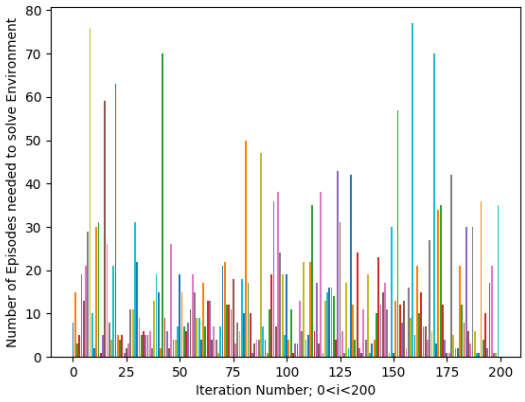

Figure 29: Bar chart showing the time of solution after 200 sessions

of the screen and the lower-right of the screen making continuous round trips. The obstacles on the road incur a penalty (negative reward) so the agent is incentivized to avoid them. Four such roads were constructed and a Neural Network was built to solve them

Neural Network Specification To solve this environment, a three layer artificial neural network was constructed with two fully connected layers activated by the ReLU function

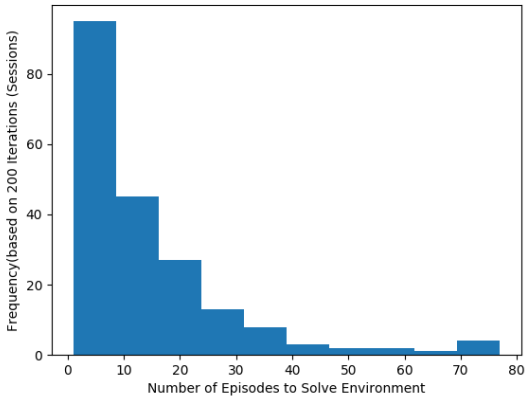

Figure 30: Histogram showing the speed of success of the Algorithm over many iterations. Note that it is able to solve the environment in less than 10 episodes most of the time

5.2.1 Road 1

The first obstacle was to get the agent moving between the two points. A temp variable (0-100) was used to control the confidence with which the agent takes an action. It was set to 80 for this experiment. The maximum reward that can be gotten by the 0.10

5.2.2 Road 2

5.2.3 Road 3

5.2.4 Road 4

```python
class Network(nn.Module):

    def __init__(self, input_size, nb_action):
        super(Network, self).__init__()
        self.input_size = input_size
        self.nb_action = nb_action
        self.fc1 = nn.Linear(input_size, 100)
        self.fc2 = nn.Linear(100,100)
        self.fc3 = nn.Linear(100, nb_action)

    def forward(self, state):
        y = F.relu(self.fc1(state))
        x = F.relu(self.fc2(y))
        q_values = self.fc3(x)
        return q_values
```

Figure 31: Neural Network Architecture

```python
def select_action(self, state):
    probs = F.softmax(self.model(Variable(state, volatile = True))*80) # T=100
    action = probs.multinomial()
    return action.data[0,0]
```

Figure 32: Confidence with which the agent takes an action

Figure 33: The First road

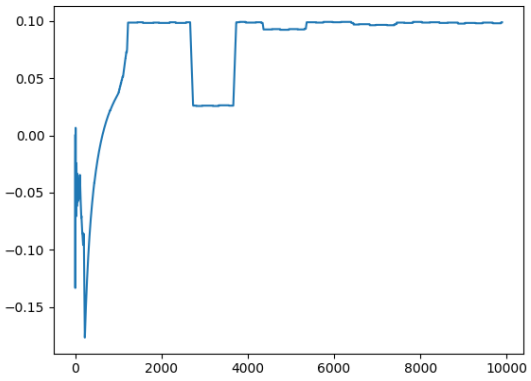

Figure 34: Solving the First roadl

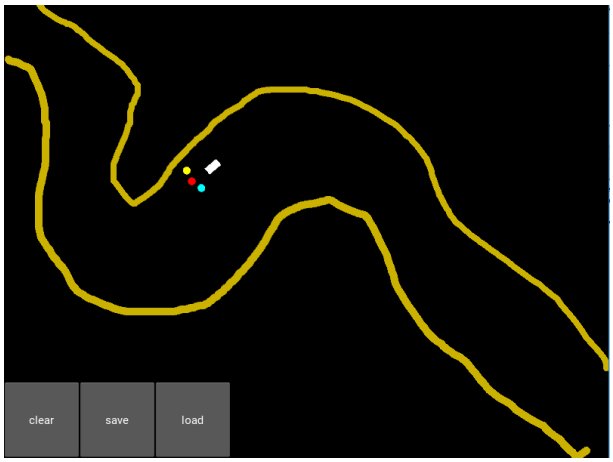

Figure 35: The Second road

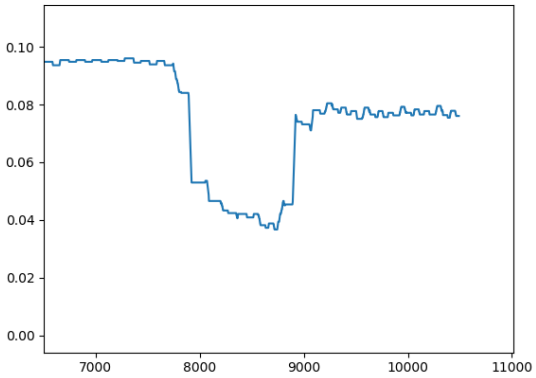

Figure 36: Solving the Second roadl

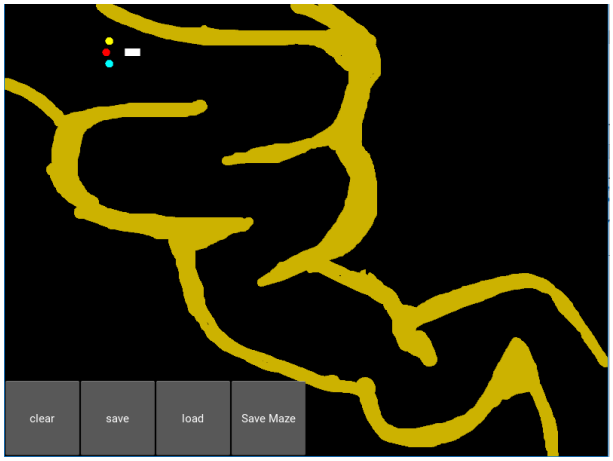

Figure 37: The Third road

Figure 38: Solving the Third road

Figure 39: The Fourth road

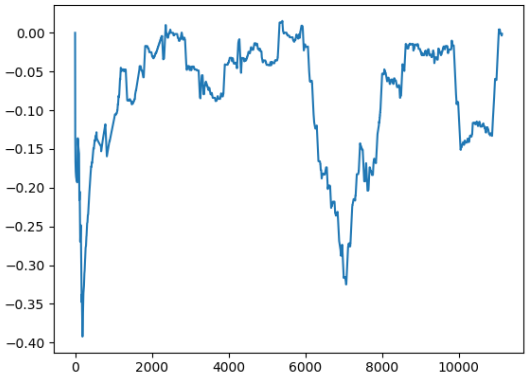

Figure 40: Solving the Fourth road

6 Conclusions

6.1 Possible Future work

- More advanced techniques such as the Hill Climbing algorithm will be studied and experiments perfomed
- Techniques learned (like decaying epsilon and Bayesian Sampling) will be applied to test environments like Cartpole
- Neural Networks will be created to retain information between sessions
- More complex environments will be explored (as stepping stones to the ultimate goal of controlling a racing game with an Agent trained with reinforcement learning environments)

7 References

References

[1] "Average Annual Miles per Driver by Age Group." Average Annual Miles per Driver by Age Group. Accessed October 11, 2016. url https://www.fhwa.dot.gov/ohim/onh00/bar8.htm.

[2] "Average Distance Travelled by Age, Gender and Mode: England, 2015." September 8, 2016. Accessed October 11, 2016. url$https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/457748/nts0605.xl$

[3] Dolgov, Dmitri. "Google Self-Driving Car Project Monthly Report." Google. Accessed October 11, 2016. Available: url https://static.googleusercontent.com/media/www.google.com/en//selfdrivingcar/files/reports/report-0916.pdf.

[4] Fliegende Bltter, 1892.

[5] "Sunway Taihulight", Nsccwx.cn, 2018. [Online]. Available: http://www.nsccwx.cn/wxcyw/. [Accessed: 08- May- 2018].

[6] [3]D. Scherer, A. Muller and S. Behnke, "Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition", 20th International Conference on Artificial Neural Networks (ICANN), Thessaloniki, Greece, September 2010, 2018.

[7] DH Hubel and TN Wiesel. Receptive fields of single neurones in the cats striate cortex. The Journal of Physiology, 148(3):574, 1959

[8] Ocf.berkeley.edu. (2018). Strong AI. [online] Available at: https://www.ocf.berkeley.edu/ arihuang/academic/research/strongai3.html [Accessed 19 Feb. 2018].

[9] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K. and Hassabis, D. (2018). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. [online] Arxiv.org. Available at: https://arxiv.org/abs/1712.01815 [Accessed 19 Feb. 2018].

[10] Silver, D., Huang, A., Maddison, C., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T. and Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. Nature, 529(7587), pp.484-489.

[11] Gym.openai.com. (2018). OpenAI Gym. [online] Available at: https://gym.openai.com/docs/ [Accessed 19 Feb. 2018].

[12] Murphy, K. (2007). Conjugate Bayesian analysis of the Gaussian distribution. [online] Cs.ubc.ca. Available at: https://www.cs.ubc.ca/ murphyk/Papers/bayesGauss.pdf [Accessed 19 Feb. 2018].

[13] sbbook R. Sutton and A. Barto, Reinforcement learning. Cambridge, Mass.: MIT Press, 1998.

[14] CS231n Convolutional Neural Networks for Visual Recognition. [Online]. Available: http://cs231n.github.io/convolutional-networks/. [Accessed: 02-May-2017].

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, ImageNet Classification with Deep Convolutional Neural Networks. [Online]. Available: http://www.cs.toronto.edu/ fritz/absps/imagenet.pdf. [Accessed: 02-May-2017].

[16] R. Fergus, Neural Networks, 2015. [Online]. Available: http://mlss.tuebingen.mpg.de/2015/slides/fergus/Fergus_1.pdf. [Accessed: 02-May-2017].

[17] U., An Intuitive Explanation of Convolutional Neural Networks, the data science blog, 02-Apr-2017. [Online]. Available: https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/. [Accessed: 02-May-2017].

[18] ConvnetJS demo: toy 2d classification with 2-layer neural network, ConvNetJS demo: Classify toy 2D data. [Online]. Available: http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html. [Accessed: 02-May-2017].

[19] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, Going deeper with convolutions, 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015.

[20] D. E. Rumelhart, G. E. Hinton, and R. Williams, Learning Representations by Back-Propagating Errors. [Online]. Available: https://www.iro.umontreal.ca/ṽincentp/ift3395/lectures/backprop_old.pdf. [Accessed: 02-May-2017].

[21] LeCun, Y. (1998). Efficient Backpropagation. [online] Yann.lecun.com. Available at: http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf [Accessed 8 May 2018].

[22] https://www.superdatascience.com/

[23] R. Bellman, "The theory of dynamic programming", Bulletin of the American Mathematical Society, vol. 60, no. 6, pp. 503-516, 1954.

[24] A. Barto, "Temporal difference learning", Scholarpedia, vol. 2, no. 11, p. 1604, 2007.