Reinforcement Learning for Simple Game Play

Christopher Frerichs, Ryan George, and Logan Phillips

***Abstract*—In recent years, Reinforcement Learning (RL) has gained notoriety among the machine learning community, especially with the advent programs like IBM's *Deep Blue* and Google's *DeepMind*. By combining classic RL techniques with Deep Learning, these programs have proven capable of out preforming humans at playing chess, video games, Go, and even solving complex riddles [1], [2], [3], [4].**

In this work, we develop various RL methodologies and investigate their performance on simple games. Specifically, Q-learning and Auto-Encoder dimensionallity reduction is use to train an agent to complete mazes of varying difficulty. Additionally.....

I. Introduction

At its core, the concept of machine learning is the determination of some functional mapping from an input to a desired output. As such, there are a wide array of applications such as classification, clustering, and pattern recognition where some underlying characteristics are discovered.

In the last few decades, the field of Deep Learning has taken off. Due to their universal function approximation capabilities [5], Neural Networks provide robust capabilities for discovering said mapping in a manner consistently out preforming previous state-of-the-art learning methodologies.

One subset of the machine learning domain is called *Reinforcement Learning* (RL). Unlike Supervised Learning, where the system learns by training on specified "correct" input-output pairs, RL has no correct target output. Instead, the agent learns to take actions in some environment such that it maximizes a reward or minimizes a cost. This is analogous to cost-benefit analysis seen in behaviour psychology or various biological systems.

This learning technique can, in many situations, be significantly preferable to that of conventional supervised learning. This is because supervised learning methods require an immense amount of preprocessing to gather and label input-output pairs. Conversely, RL is conducted online and therefore requires far less data acquisition. Some of the more notable accomplishments of RL have been IBM's *Deep Blue* used to defeat Garry Kasparov, the world chess champion, in 1996 [1] and Google's and *AlphaGo* which beet the world champion Go player in 2016 [3].

The purpose of this paper is to better understand various RL techniques and apply them to simple, well known games.

A. Problem Statement: Maze

Mazes provide one of the simplest learning environments in that they provide a relatively small, discrete state space with limited action possibilities (up, down, left, right). Additionally, mazes have a clear cost/reward function - i.e the player wants to get to the goal in as few moves as possible.

For these reasons, we first develop our understanding of various RL techniques under this simple framework. The goal is to developer from scratch algorithms to train an agent to navigate a given maze in as few moves as possible.

B. Problem Statement: Pong

The classic Atari game Pong is a simulated version of table tennis where the player attempts to move their paddle to prevent the ball from entering their goal. A point is awarded to the player if they can get the ball into the opponent's goal.

Though simple by modern videogame standards, Pong is a significantly more complex environment than a Maze. Not only must the player know the position of their paddle, but they also must now the position and trajectory of the ball in order to make decisions. Thus, the available state-space is orders of magnitude greater.

The goal of this problem is to utilize existing tools (such as Keras and pyGym) to train an agent to maximize their score in pong.

II. Reinforcement Learning

In its most basic form, Reinforcement Learning (RL) is simply as Markov Decision Process. An agent observes the current state, $s \in S$ of the environment. Given the current state, the actor then decides to take some action, $a \in A$, from the available action space A. This transitions him to a new state, s', and some reward, r, is given. Based on this reward, the agent updates its decision making process and the sequence continues.

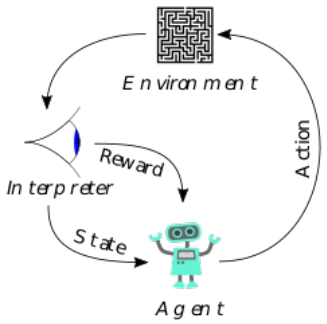


Fig. 1: Typical framework of Reinforcement Learning scenario

The central goal of all RL techniques is to learn how to best impute an optimal policy π for deciding how to take actions. Various methods exist, but here we focus on Q-Learning, dimensionality reduction techniques, and policy gradients.

A. Q-Learning

The cornerstone of RL is the technique Q-Learning. In this process, the "quality" or "Q value" of each action-state pair is discovered through an integrative process. This

process, described below, is relatively straight forward and easily implemented. Furthermore, it has been shown that for any Finite Markov Decision Process (FMDP), Q-Learning will always eventually converge to the optimal policy [8]. For these reasons, Q-Learning is often viewed as the gold-standard when comparing different RL techniques.

To start, we by thinking of how the state changes in some Δt steps. Obviously, we not only care about the current state but we also care (to some degree) about the steps we took to get there. To that end, we weight each steps reward by $\gamma^{\Delta t}$ where $\gamma \in [0, 1)$ is known as the *discount factor*. This has the effect of valuing rewards earned earlier more than that earned later, emphasizing the importance of a "good start" [6].

Using this concept, we can construct an algorithm to determine our Q-function which calculates the quality of a given state-action pair $Q : S \times A \rightarrow \mathbb{R}$. In other words, a game with only 10 states and 2 actions for each state would have a Q-matrix of size 10×2.

Prior to learning, Q is arbitrarily initialized. Usually, similar to weights in a neural network, these values are randomly assigned. However, other methods for initializing these values can lead to improved convergence [7].

Then, at each time t the agent identifies its current state s_t, chooses the currently perceived (based on the Q values) optimal action a_t, receives a reward r_t, and moves to a new state s_{t+1}. The Q value for that specific action is then updated in the following manner:

$$Q(s_t, a_t) \rightarrow (1-\alpha) \cdot Q(s_t, a_t) + \alpha \cdot \left(r_t + \gamma \cdot \max_a Q(s_{t+1}, a) \right)$$

where $\alpha \in (0, 1]$ is a learning rate. Heuristically, this is the extent to which new information overrides old information. Note that a learning rate of $\alpha = 0$ means that the agent does not learn whereas a learning rate of $\alpha = 1$ means that only the most recent information is considered. Depending on the environment, adaptive learning rates can be used. However, in fully deterministic environments, the optimal learning rate is $\alpha = 1$ [9].

Since its introduction in 1989 by Watkins [10], several variations have developed. However, this simple algorithm plays a fundamental role in providing a framework for most RL algorithms to date.

B. Dimensionality Reduction

As previously discussed, Q-Learning is a relatively straightforward process given that the state-action space is well defined and of a manageable size. However, as RL attempts to tackle more complex tasks, the way forward can be less clear. In continuous environments, it is necessary to discretize the state space so that Q-Learning can be conducted. Additionally, due to the computational requirements of such problems, it is often desirable to conduct some type of dimensionality reduction in order to make the Q-space manageable.

Various methods exist for this reduction, but here we discuss the use Auto-Encoders and Convolutional Neural Networks.

1) Auto-Encoders: An Auto-Encoder (AE) is a type of neural network which determines efficient encodings of data through unsupervised learning. Some input X is propagated through an encoder, having one or more hidden layers, to a code layer. Though other types exist, usually this code layer is of a dimension smaller than that of the input. This is called an *Undercomplete Auto-Encoder*. This code layer is then propagated through a decoder to produce an output $\hat{X}$. The network is then trained to reduce the error between X and $\hat{X}$ such that $\hat{X} \approx X$.

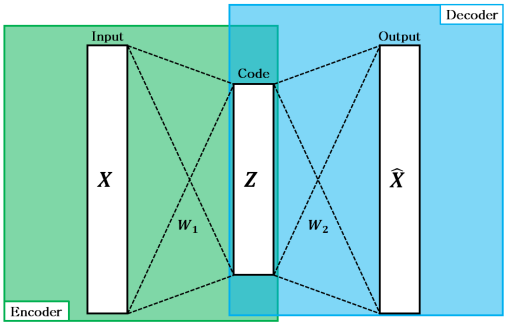


Fig. 2: Undercomplete single-layer Auto-Encoder

As an example, lets use the undercompete AE depicted in Figure 2. First, the input X is passed through an encoder to a code layer Z such that:

$$Z = \sigma(W_1^T X + b_1)$$

where σ is some activation function, W_1 are the weights, and b_1 are the biases. This code is then passed through a decoder so that the output is

$$\begin{aligned} \hat{X} &= \sigma(W_2^T Z + b_2) \\ &= \sigma(W_2^T \sigma(W_1^T X + b_1) + b_2) \end{aligned}$$

Now, we can conduct back propogation in the stardard way by computing the gradients:

$$\delta^L = \nabla_a C \circ \sigma'(z^L) \tag{1}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \circ \sigma'(z^l) \tag{2}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{3}$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \tag{4}$$

$$w_{j,k}^l \leftarrow w_{j,k}^l + \eta \cdot \frac{\partial C}{\partial w_{jk}^l} \tag{5}$$

$$b_j^l \leftarrow b_j^l + \eta \cdot \frac{\partial C}{\partial b_j^l} \tag{6}$$

where C is our cost function, $z^l = W_l^T X_l + b_l$ (not to be confused with code layer Z), a^l is the activation of the l^{th} layer, δ^l is the error of the l^{th} layer, and $\eta \in (0, 1]$ is the learning rate. Also recall that $\circ$ denotes the Hadamard product.

AEs are used in several applications to include denoising images and generative models, but here we are using it simply for dimensionallity reduction. First, we train the AE to reproduce all possible states is S. Then, at any state s_t we can then extract the code layer Z to represent our state-space s_t' in a lower dimension.

C. Policy Learning

In contrast to Q-learning, policy learning seeks a function of the form:

$$\pi : S \rightarrow A$$

for maximizing some reward. The rewards take the form:

$$V^{\pi}(s) = E[\sum_{i=1}^{T} \gamma^{i-1} r_i] \forall s \epsilon S \tag{7}$$

where r_i is the numerical reward assigned at a given time and γ is a discount factor. In this paper, we use $\gamma = .99$, since a win is about equally valuable at any point in time. We do not implement a complete policy-learning algorithm here and so omit theory about Bellman equation updates.

1) RMSProp: RMSProp is an alteration to standard gradient descent for highly non-convex situations. It exponentially weights past gradients with the following two update steps:

$$r \leftarrow \rho r + (1 - \rho) \nabla C \nabla C$$
$$w \leftarrow w - \frac{\eta}{\delta + \sqrt{r} nabla C}$$

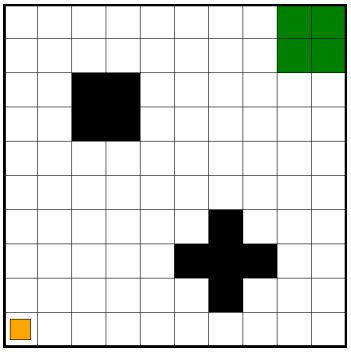
In practice, this update step is performed only after some number of batches.

III. Implementation and Results

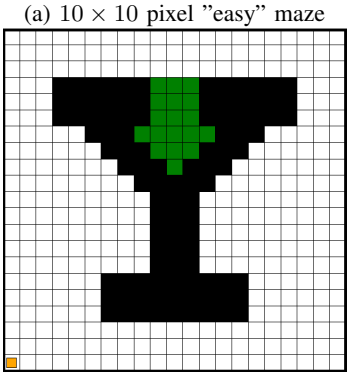
Using the above described methods, a RL agent was trained to first complete a Maze an then to play the game of Pong. The maze RL agents were constructed in python using only the `numpy` library in order to better understand (and appreciate) the inner workings of the algorithms being used. The Pong RL agent was trained using existing tool-kits such as Keras and pyGym. The specific implementation and findings are presented below.

A. Maze

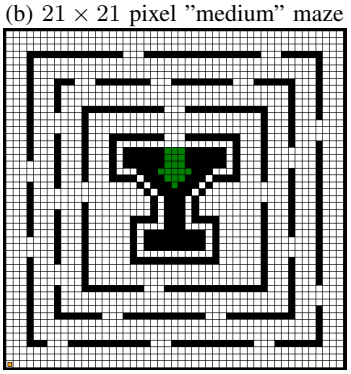
The first task was to develop the maze game environment for the agent to operate. Mazes were drawn in Microsoft `Paint` and .png images were imported into the program as arrays with each pixel representing a state. Black pixels represented "walls" that the agent cannot access, white pixels indicate available spaces, and green pixels represent the "goal".



(a) 10×10 pixel "easy" maze



(b) 21×21 pixel "medium" maze



(c) 49×49 pixel "hard" maze

Fig. 3: Maze environments in increasing levels of difficulty

The game then positions the player, marked by an orange square, initially at the bottom left most position. Based on using input, either by keystrokes or directly for the agent, the player moves about the maze. Once the player reaches a goal, the total number of moves is displayed/recorded and the game restarts.

1) Q-Learning: Using the method described above, Q-Learning was implemented to have an agent learn the optimal policy to beat the maze in the least number of moves. In doing so, several parameters needed to be set.

The objective of the game is to complete the maze in the fewest moves possible. Therefore, the "reward" is selected to be negative. Heuristically, it is easier to think of this as minimizing a penalty instead of maximizing a reward under the framework of this problem, but to be consistent with the literature we'll keep with the stated terminology. While complex reward assignments have been shown to provide increase performance in some cases [10], here we simply set the reward for each action to $r = -.05$. This selection, besides

being negative, is somewhat arbitrary since all actions have the same reward. Several different values were selected but with little change in performance in the small maze (figure 3a). However, in larger mazes, where the number of moves can grow significantly, large negative rewards created computational problems. For this reason, this small but negative reward was chosen. The only exception to this is for the goals, which had a reward of $r = 10$.

The next hyper-parameter to adjust was the discount factor γ. As alluded to above, γ is a measure of the importance placed of future rewards. A factor close to 0 means that the agent is only concerned with maximizing the current reward, whereas a factor close to 1 considers long term effects. With this in mind, we tested a range of discount factors to determine convergence averaged over 10 trials (figure 4). It as been shown that $\gamma \leq 1$ can diverge [6], so we did not consider it.

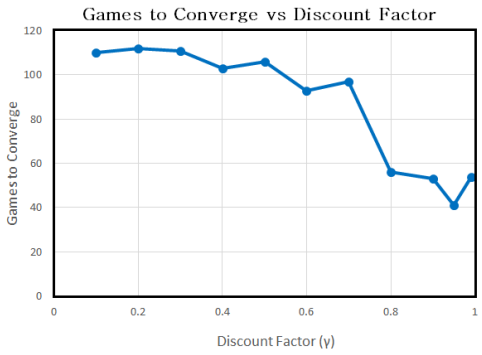


Fig. 4: Effect of Discount Rate γ on moves to convergence of "easy" maze

Though slight variations occurred due to the random initialization of Q values, it is evident that the optimal performance occurred at at values close to $\gamma = 1$. This confirmed our original hypothesis and $\gamma = 0.9$ was selected for the remainder of the experiments. It is important to note that it is possible that in other mazes this parameter would need to be optimized again. However, for comparison purposes, we kept it constant.

The final hyper-parameter was the learning rate α. As discussed above, for FMDMPs the optimal learning rate is $\alpha = 1$. However, we tested both smaller values and adaptive learning rates. In all cases, this decreased performance so we maintaned $\alpha = 1$ for all experiments.

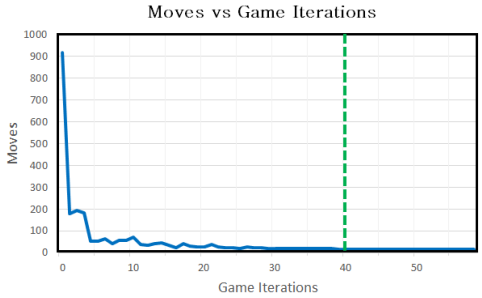


Fig. 5: Number of moves for each game on "easy" maze

Using these parameters, we tested our Q-learning agent on the three mazes. Below are the number of games needed to achieve convergence averaged over 10 trials.

TABLE I: Number of Games to Converge

Maze	Available States	Number of Games
Easy	87	41
Medium	322	98
Hard	1864	387

Note that despite the significant increase in the complexity of the "hard" maze the number of games needed to converge still scales relative to the available states.

2) Auto-Encoder: Having successfully implemented a simple Q-learning algorithm, we now implement an undercomplete AE to reduce the available state-space for S to some smaller S' and evaluate performance. First, outside of the game loop an AE is constructed to accurately reproduce every possible state. This is accomplished in the manner prescribed above. Weights are initialized with a centered Gaussian distribution such that $W_1(i,j) = \texttt{randn()}$ and $W_2 = W_1^T$. The network is then trained until all possible states are accurately reproduced.

Each possible state (i.e. the player's current position on the board) is represented by an $[n \cdot m \times 1]$ dimensional vector where n and m the the dimensions of the maze. The current position is represented by a 1 and all others are a zero. Thus, each state is incredibly sparse, especially for larger mazes.

Given the sparsity of these states, sigmoidal activations resulted in unsatisfactory properties in the code layer (which we will discuss later). To alleviate this, ReLu activation functions were used.

A learning rate of $\eta = .1$ provided optimal results in all three maze environments. Adaptive learning rates were attempted, but added complexity without an increase in performance so it was not included. We also chose not to include dropout due to the already sparse nature of the input.

The size of the code layer, which will become a relevant parameter in later discussion, was adjusted to find a threshold at which accurate reproduction breaks down. We discovered that it was possible to reduce the code size and still decode accurately to a dimension (about 15%) far lower than was effective for its implementation for Q-learning. In other words, as we'll discuss in a moment, the size of the code was limited by the efficiency of Q-learning and not by the accuracy of the encode-decode process.

Having trained the AE, we extract the weights W_1 for use in the encoder used in the game-loop. At every state in the game, the agent inputs its current state S into the encoder. Then the maximum value of the code layer argmax $Z = 1$ and all others are set to zero. This is now some lower dimension S'. Q-learning is then conducted in the same manner as described above but now using this S'.

The agent was then evaluated on its ability to complete the "easy" maze using the encoder to reduce the state-space. The size of the code layer Z was varied between 70-100% of the

original state-space. The number of games needed to converge (averaged over 10 trials) is presented in Figure 6

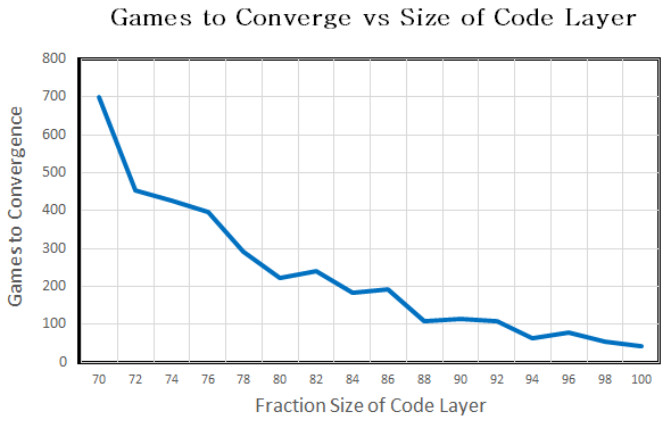


Fig. 6: Average number of games needed to converge on "easy" maze with different code sizes

As one might expect, with a fractional code size of 100% (i.e the same size as the input) the number of games need to converge is almost identical (43 vs 41) to the regular Q-learner. As the size is decreased, performance remains about the same up to 90%. Sizes in the 80-90% region, while slower, still converge in a reasonable amount of time. However, sizes less than 80% rapidly increase the games needed to converge.

More interesting is the qualitative nature by which this method converges. Note that with the standard Q-learner depicted in Figure 5, there is a relatively smooth decrease in the number of moves per game iteration. However, using the AE with a code size of 94% (Fig. 7) this is not the case. Instead, there appears to be almost no correlation between game iterations initially. Then, the agent abruptly collapses to the optimal path and remains there. This same qualitative effect is seen throughout all reduced code sizes just with a difference in scale.

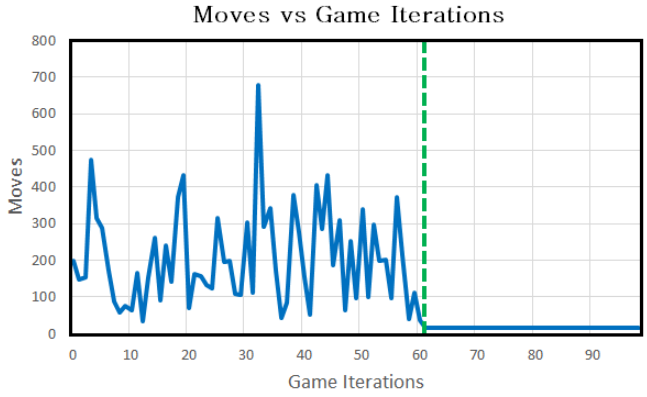


Fig. 7: Moves for each game iteration with code size of 94%

While it is not precisely clear why the agent preforms in this manner, we present the following heuristic argument based on watching how the games evolve.

By reducing the state space to some smaller dimension, states no longer refer to individual pixels. Rather, each state refers to some groupings of pixels, or sections of the graph. To test this, we observed the code layer for to adjacent pixels, call them a and b. Indeed, in many cases argmax Z_a = argmax Z_b meaning that $S'_a = S'_b$. We can also see how this effects the game play. The agent will explore (and seems to get lost in) various sections of the maze in a chaotic manner. Eventually, the agent learns to ignore whole sections of the maze. As these sections are "closed off" we see the agent quickly collapse to the optimal route.

IV. Pong

The second application of reinforcement learning we explored was teaching an agent to play Atari Pong. The pong movements are relatively simple; they consist of moving the paddle up or down, or remaining still.

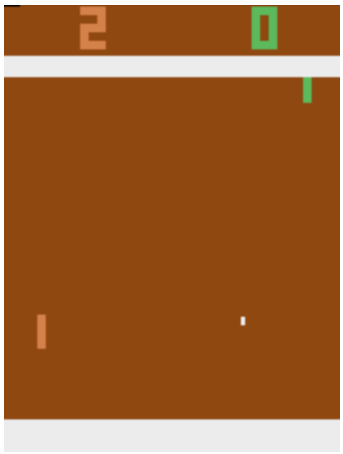


Fig. 8: Classic Atari pong run via Gym.ai

A. Gym.ai

In order to train in an appropriate environment, we use the open-source reinforcement training API Gym.ai[11]. "Gym" aims to be a training framework for applying Reinforcement learning to problems as simple as the cart-pole problem or as complex as the classic video game Doom. We use the Pong-V0 environment to train our network.

The advantage of using Gym.ai is that it already implements the classic Atari computer player against which we train our agent. The Atari agent is not so good as to be perfect, but it is difficult enough to play against as a human that an agent trained to defeat the Atari AI would almost certainly defeat a human player.

B. Policy Networks

In order to train our Pong network, we implement a policy-based network. For our paradigm, we define the following MDP elements:

- State S: The network state is the screen output of the Atari game. We preprocess this in a number of ways described below.
- Action: There are two actions available to the agent: go up or down. We denote this tiny space of possible actions as $a \in A = 0, 1$

- Reward: The reward for a pong agent is 1 if if wins a round, and -1 if it loses.

We seek a policy of the form $\pi : S \rightarrow A$ for maximizing rewards. Since the game of Atari pong is somewhat stochastic (the bounce of the ball isn't perfectly predictable, we implement a non-deterministic policy algorithm. Given that there are only two possible actions, we can make this policy as simple as $\pi(S) = \alpha$, where α is the probability of choosing to go "up". Thus our function is $\pi : R^N \rightarrow R$. In the following sections, we refer to p as the probability that the paddle will go up.

C. Preprocessing and Feature Engineering

We perform some pre-processing on the Atari screen output before feeding it into the network. In particular, we:

- flatten the three colors into a single dimension
- crop the bottom and the upper portion (score)
- reduce background pixels to 0
- downsample by factor of 2

Finally, and most importantly, we take the difference between the previous frame and the current frame. This allows the network to be aware of temporal changes such as the movement of the paddles and the ball.

The actual input into the network consists of a 1x6400 vector denoting this difference. An example of the vector, reshaped into a square, can be found in Figure 9. Since most pixels don't change between frames, the vector is very sparse.

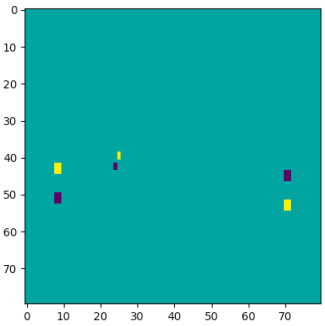


Fig. 9: Difference vector input into network

D. Network Architecture

In order to create the policy function π, we implement a fully-connected with one hidden layer of 300 neurons. These neurons are ReLU-activated and do not include a bias term. These neurons map to a single output term, mediated by a sigmoid function, that determines the probability p that the paddle will go up (as opposed to down). Network architecture is shown in Figure 10.

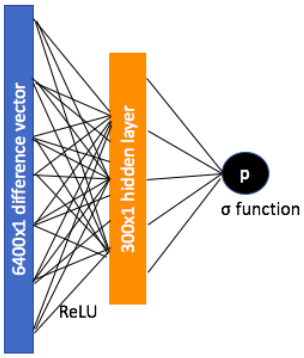


Fig. 10: Network architecture

E. Training the Network

We train the network through gradient descent using the simple backpropogation (recall equations (1)-(6)). The loss function is simply the difference between the action taken ($a \in 0, 1$) and the probability that the paddle will go up:

$$L = a - p$$

We modulate this loss function by the summed rewards of the episode. The reward amount is calculated using the discounted values of the various games played (see equation 7). These summed rewards are unit normalized before being applied to the loss.

In order to stabilize learning over time, we implement RMSProp. We use a decay rate of $\rho = 0.99$ and a learning rate of $\eta = .002$. We apply RMSProp updates every 20 batches (i.e. every 600 games).

V. Areas For Future Research

We believe that the current work certainly fulfills our stated purpose in that we have researched various reinforcement learning techniques, implemented their usage in simple games, and analyzed the results. However, in the course of this project we have discovered several avenues that we believe to be worth pursuing in the future. The following are some of possible topics

A. Maze with Human input

This research was inspired by the Two different attempts were made to improve upon classical Q-Learning by having the agent observe as the user for some number of iterations.

The first method was to charge the reward weights for the states chosen by the user to a value closer to zero than that of the other states. The idea was that the agent would then see these states as more preferable and travel along that trajectory. However, initial tests proved to be unsuccesful and these efforts were halted in order pursue other research.