Playing Pong with a DQN

Debargha Ganguly
debargha.ganguly_ug20@ashoka.edu.in

Abstract

In this paper, I've used Deep Q Learning to learn control policies to play the game of pong, directly from visual data. The model is based on a Convolutional Neural Network that learns to use the input raw pixel data , to estimate a value function that allows us to approximate the future rewards, for any given action. The exact same model has been trained in different environments, and plays a near perfect game, against a perfect rival bot, as well as against a human. Experiments have also been conducted to better understand and evaluate how our trained agent responds to dynamic changes in the environment.

Problem Statement

For machines to truly interact seamlessly with our everyday environments, it is necessary that they can operate on the basis of high-dimensional understanding of the environment.

A machine might not always be able to access a data pipeline that allows it to optimally learn , i.e. with respect to the environment it is in, it might not know the features that allow it to learn and perform in the best possible way. Feature engineering, however, has seen rapid progress since the introduction of Deep Neural Networks, with convolutional neural networks being the go-to model in approaching a problem with visual inputs.

But combining the paradigms of reinforcement learning and deep learning isn't very simple. A number of issues arise :

1. Usually to attain human level performance, deep learning networks need huge datasets, that are accurately labelled. This data is usually :
    a. Dense in the search space.
    b. Has a direct association between inputs and targets.
2. Reinforcement Learning methods such as the vanilla Q Learning rely on scalar signals, that can be :

   a. Sparse in the search space
   b. There might be noise associated with the environment
   c. Actions and rewards might have thousands of timesteps in between.

3. In deep learning the dataset that it learns on can be independent. Reinforcement learning, however, has causation and correlation based set of actions and rewards that it needs to work with.

4. Deep Learning methods rely on being able to predict and learn the underlying function distribution that's generating the data, therefore works towards optimizing that fit. On the other hand, reinforcement learning is a continuous progress based method, and newer methods of playing the game, as acquired by the agent can result in a changing underlying distribution.

The eventual goal is to be able to create a machine, or an agent that is smart enough to learn how to perform in a variety of situations. The agent that we're creating here should be generalised, such that it can look at a bunch of high dimensional visual inputs, and from the set of rewards, be able to learn in any environment.

Prior Approaches And Their Problems

Reinforcement learning has been around for a long while.

- An agent acts inside an environment for a given time.
- At each of the timesteps t, this agent has a corresponding state S_t.
- There's an action space A containing the legal possible actions.
- The agent chooses what to do on the basis of a policy $\pi(at|st)$

This policy is the function describing how the agent has to behave, and mapping from the state s_t to actions at a_t. Depending on the action, the agent receives a reward r_t from the environment and the entire state transitions to the next with s_{t+1}, according to how the environment behaves.
Therefore, the reward function is given by $R(s, a)$ and the state transition probability is given by $P(s_{t+1}|s_t, a_t)$.

For an episodic environment, this process keeps continuing until it reaches an episodic state, from where it starts again. The return function is given by :

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

With $\gamma \in (0, 1]$, and the agent trying to optimise the long term return from each state.

The value function is introduced as a prediction of the expected reward, discounted for time in the future. The action-value function $Q_\pi(s, a) = E[R_t|s_t = s, a_t = a]$ is the expected return for selecting action a in state s and following policy π afterwards.

An optimal action value function $Q^*(s, a)$ is the maximum action value that can be achieved by any policy for state s and action a . Therefore, we can also define a state value $V^\pi(s)$ function and the optimal state value $V^*(s)$. This will be our target.

With DeepRL applied, there are examples of TD-Gammon, a program that was capable of playing backgammon that learnt the best way to play using just reinforcement learning and playing with itself. It achieved a superhuman level of play. Being one of the earliest examples of this combination of Q Learning and MLP, the agent learnt to approximate a value function through a network with 1 hidden layer.

For a long while, because people found it very hard to replicate the success in TD-Gammon, it was thought that Backgammon was a one-off success case, where the stochasticity of the dice rolls helped the agent explore the state space better. It was also thought that it might have made the value function smoother.

The paper by the DeepMind team, with the Atari environment, describes the first deep learning model to successfully learn control policies from a high dimensional sensory input using reinforcement learning, using a CNN, that outputs a value function that estimates future rewards. They played 7 Atari 2600 games, with the same architecture, and managed to outperform all previous methods on 6 games, and attain superhuman status in three of them.

The DeepMind paper went on to be a landmark paper in this field, and have become the state of the art. This approach has practically solved these environments, and recently released simulation environments such as OpenAI Gym allow researchers and students to experiment with this technology, readily. *These simulators however have some downsides as they're written to be easy to use, but don't allow a researcher to make many changes to how they work. For this purpose, we decided to write our own simulation.*

Adopted Approach

I've decided to use the ***approach defined by the DeepMind team in their paper, proposing Deep Q Networks. Network architecture and hyperparameters were kept constant across all situations,*** for generalisability, just like in the paper.

The loss function is optimised with the help of stochastic gradient descent, and the weights are back propagated, therefore updated after every time step in the environment.

This is a ***model free learning strategy*** because it samples from the simulator, and learns without creating a model to estimate the simulator. It learns off-policy because ***it uses an epsilon greedy strategy*** to be able to act based on exploration to exploitation conundrum. (Multi-armed bandit problem)

The approach can be ***summarized well by the algorithm pseudocode in the original paper, which was used for the implementation.***

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Implementation details

The Simulator

The simulator can be divided into two different distinct parts :

1. **Pong Environment :** Describes the environment for the setup
2. **Pong Game :** Manages the state, and updation of the ball, and paddles based on actions.

Main Subsections :

1. **Paddle 1 :** Controlled by the bot or the human playing, which has :
    a. Paddle Height
    b. Paddle Width
    c. Paddle Buffer : Space between the edge of the window and the paddle.

2. **Paddle 2 :** Controlled by the agent that learns.
    a. Paddle Height
    b. Paddle Width
    c. Paddle Buffer : Space between the edge of the window and the paddle.

3. **Ball :** Makes perfectly elastic collisions against the walls, as well as the agent.
    a. Ball Width
    b. Ball Height
    c. Ball X Velocity
    d. Ball Y Velocity

The **Game class** keeps track of the state and the interacts inherits the elements of the environment to change it's variables as needed.

1. **Paddle 1:**
    a. Paddle 1's Y position
2. **Paddle 2:**
    a. Paddle 2's Y position
3. **Tally :** To keep track of the score
4. **Ball :**
    a. Position along X axis
    b. Position along Y axis
    c. Direction along the X Axis
    d. Direction along the Y Axis

The Agent

The **controller class can interact with the game class, through two specific methods** :

1. **FirstFrame :**
    a. Accepts No Arguments
    b. Returns image data from the first frame
2. **NextFrame :**
    a. Accepts the action taken by the agent
    b. Returns the image data from the next frame and the score of the game.

Therefore, we've just allowed the agent to access high level pixel values, and the score (state) of the simulator. The agent can only interact with the help of the predicted actions. Nothing more.

Network Architecture :
For feature engineering :

1. **1st Convolutional Layer** : Takes a 60x60 frame as input.
2. **Maxpooling Layer** : Reduces dimensionality of feature map
3. **2nd Convolutional Layer** : More feature engineering
4. **3rd Convolutional Layer** : More feature engineering

The result of this feature engineering is flattened into a tensor.

This is followed by :

1. **Fully connected 4th Layer** : Accepts flattened tensor
2. **Fully connected 5th Layer** : Outputs a tensor of the size of Action Space

The result of the fully connected 5th layer is the network output. ReLu activation functions have been used throughout.

Algorithm for training :

(Code was written by following the logic followed inside the pseudocode attached above. I tried writing it down in words, but it just becomes practically incomprehensible.)

It has the following hyper-parameters :

1. ***Action Space*** of 3, which can be [no movement, go up, or go down]
2. ***Gamma*** : This describes the learning rate
3. ***Initial Epsilon :*** The epsilon value we start from (1)
4. ***Final Epsilon :*** The epsilon value we end at (0.05)
5. ***Observation :*** The initial frames for which the agent will just observe.
6. ***Exploration :*** The frames for which the greedy epsilon strategy will reduce the epsilon from the initial epsilon value to the final epsilon value.
7. ***Memory Size :*** The amount of memory provided to the agent to keep track of it's previous experience. (Experience replay)
8. ***Batch Size :*** The number of frames over which training takes place. (48)

Some important things to be noted :

1. Consecutive frames are stacked on top of each other in 4 channels such that the network doesn't just approach it from a spatial perspective, but also can understand the temporal perspective, i.e. which way the ball is moving, speed etc.
2. The experience replay queue is implemented in the form of a Queue structure in Python which can be popped, when the queue is full. This replay queue contains :
    a. Time t Stacked frame
    b. The action taken
    c. Reward received for the action
    d. Time (t+1)'s stacked frame.

Formulating the perfect opponent bot :

Since we cannot keep playing the game, for the bot that our agent trains against, I had to create the perfect agent.

This is very simple because you can just set ***the bot's center to always follow the center of the ball***, therefore making sure that it always hits. (Important information for later test results)

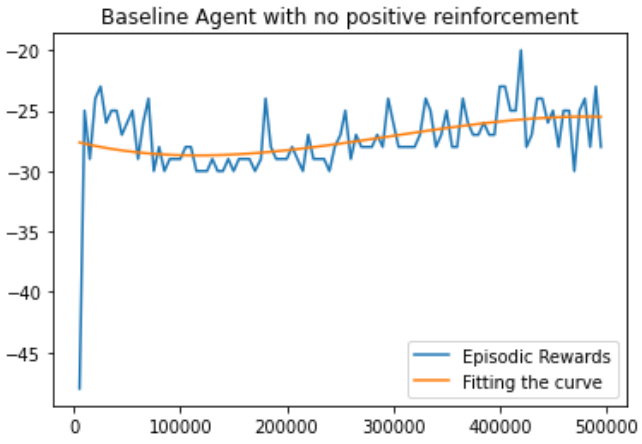
Results

The lot of experiments were run under a varying set of parameters.

In this section, ***the original colours refer to black agents playing on a white screen***, and ***flipped colours mean the reverse.*** The headers being talked about are scores that were being displayed on the screen. Experiments suggested that the agent didn't learn well because of the confusion created by the scores inside the frame, therefore they were removed.

Based on formulation of the reward function :

Initially, ***the reward function was set in such a way that only if our agent beats the bot, we get a positive reward.*** If it doesn't beat the bot, we get a negative reward. Now, keeping in mind that our bot is a perfect agent, meaning it's unbeatable, therefore the agent would realistically never get a positive reward.

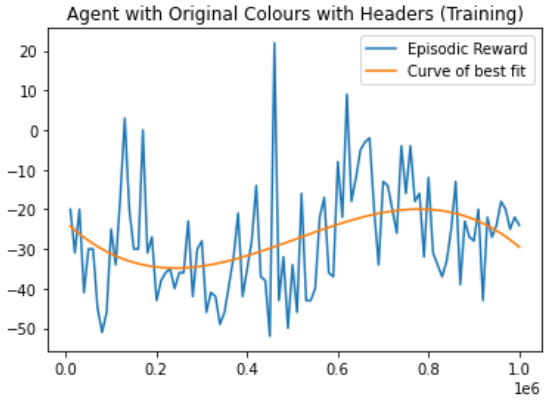


Learning in this context could be ***associated with reducing the number of negative penalties inside a given frame of time (episode).***

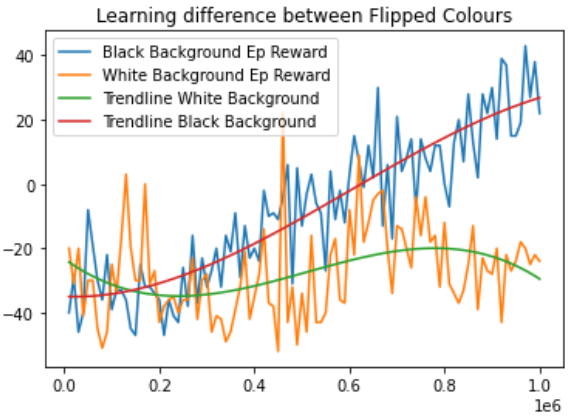
This agent, when tested practically learned nothing over half a million training steps, therefore was abandoned.

Modifying the Reward Function

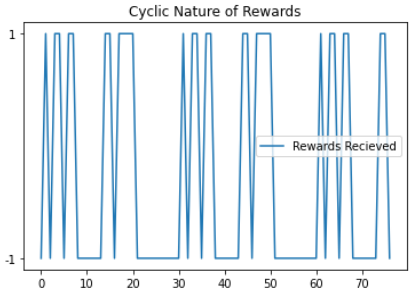
To counter the problems arising from the lack of positive reinforcement, ***the reward function was changed to such that the agent would get positive reinforcement whenever it hit the ball.***



It was observed that this made the agent actually start learning. It was observed the environment the agent was learning in was slightly rectangular, therefore, the ball would make cyclic patterns through the end of the agent. It's important to note that the agent played the balls well, that were coming towards the center, but as the ball kept moving towards the



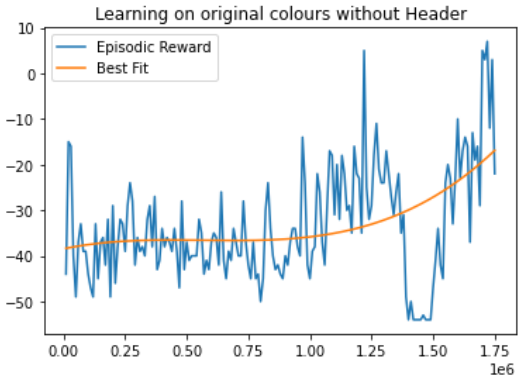
extremities periodically, it couldn't hit them at all. Therefore a ***cyclic nature of rewards*** was observed.



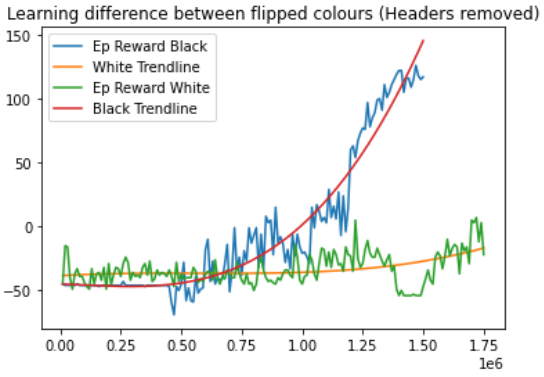
Yet the agent was not learning well, so ***I flipped the colours in the environment, and it started learning much better, even with the headers being present.***

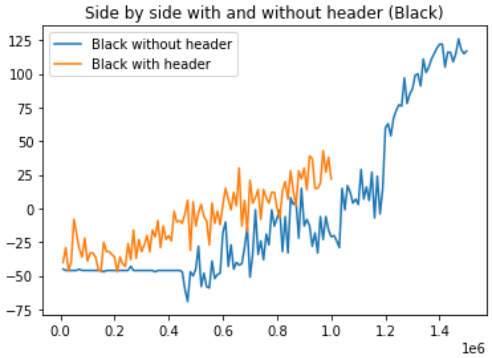
Removing the headers from the frame

Even with the flipped colours, it was evident that the ***headers were causing the agent to get confused because everytime the ball would go near the headers,*** the trained agent wasn't able to play it. Therefore the agent was trained again, on both the environments with the header removed.



It's still unexplainable, why just flipping the colours of the environment led to the agent to start learning however there are two hypotheses. Either the ***network was getting saturated*** earlier, with the previous network that got fixed when the colours were flipped, or the network just had ***a few dead neurons from the ReLu activation function receiving high gradients.***





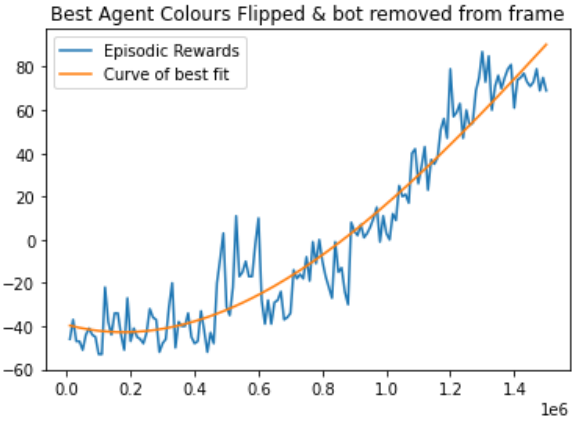
Anyway, without the header, past about 1.3Mn timesteps, our agent becomes virtually perfect. Outperforming the agent with the headers.

Playing against Human Subjects

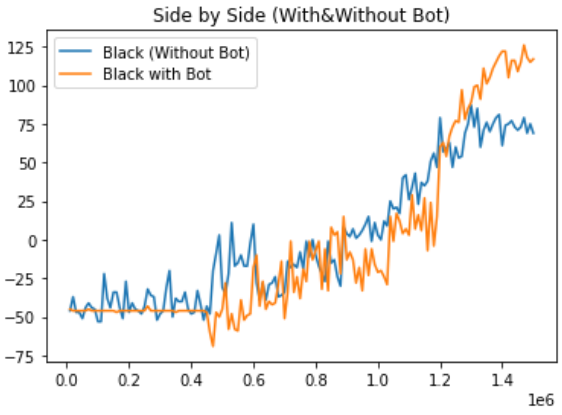
The last model, near perfect model we created performed very well inside the simulator. However, ***once we swapped out the simulator bot, with a real human playing, it's performance suddenly dropped.*** There was no way to reliably show this drop in graphs hence I can't show it.

However the reason for this drop in performance was the following :

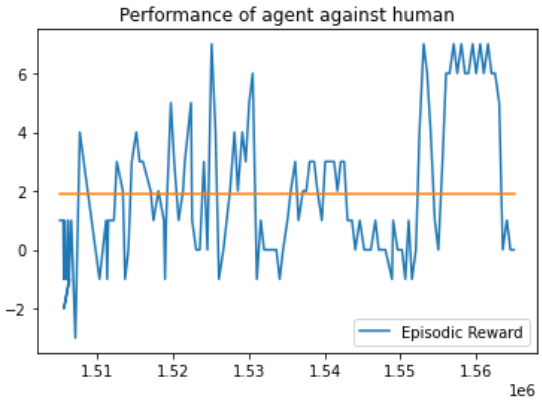
The bot that was playing the game was constantly following the ball, irrespective of whether the ball was close to it or not. Since it was programmed inside the simulator, the bot's center was always at the same Y Position as the ball's center. ***Therefore, during the process of training, for no fault of its own, the agent had learned to follow the other paddle (Controlled by the bot).*** Once the human took over the paddle, it was no longer following the ball all the time. Our agent however kept somewhat mirroring the other paddle, therefore it kept losing points.



The solution to this was simple, ***the part of the frame that had the bot was cropped, and the resultant frame/ stack of frames were fed to the network and retrained.*** This fixed the problem, leading to a perfect agent too. The learning curve for it was just slightly different.



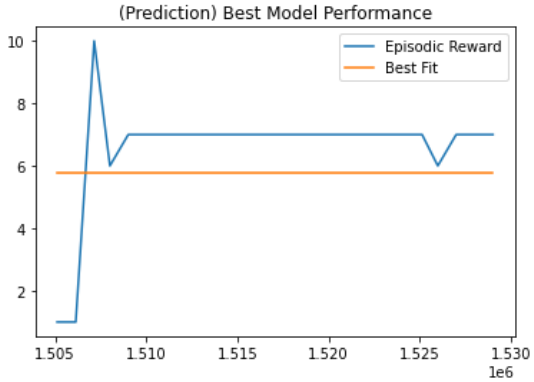
The final performance of the agent against the human is graphed below. This graph doesn't mean much as the humans playing kept getting frustrated from losing and kept stopping playing, while the agent racked up it's points.



Even the best gamers were losing to it by a score of 3-21, which goes on to speak about how good the machine was. I think **it has attained superhuman capabilities, matching up to DeepMind's results.**

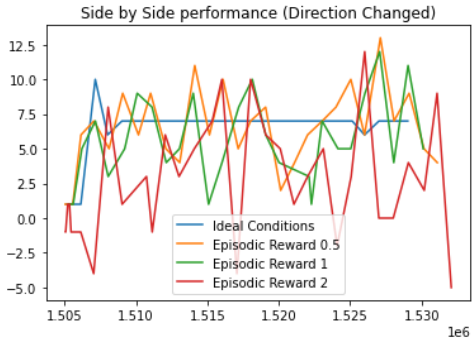
Adding Entropy to the System

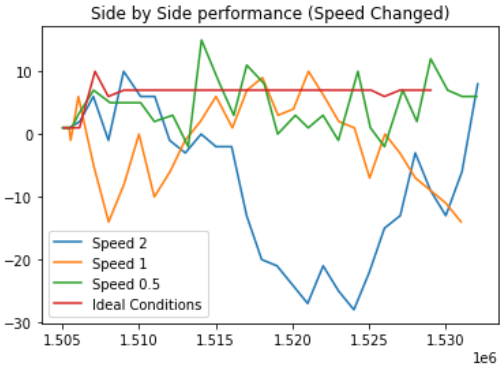
We've already understood how the agent behaves in the same environment against the bot as well as against the human. When predictions are run over 30,000 timesteps, we see that it's a near perfect model, only occasionally losing a point in edge cases against the bot.



In order to test out how our model reacts to a change in the environment's condition, we decided to add obstacles in between.

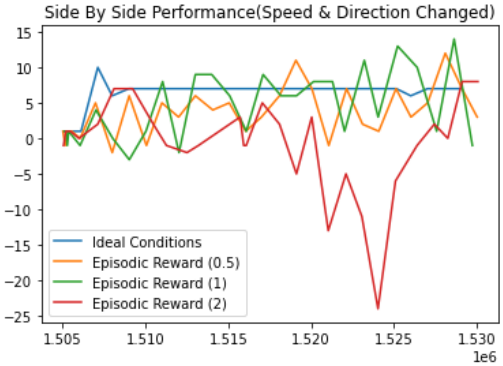
Considering that adding visual obstacles was messing up our model's behaviour, because it had not been trained to function under all that visual noise, we decided to simulate it by adding noise every time the ball collided with the bot paddle, either by changing the direction, or speed or both. ***A change in direction is linked to spatial perception, because the ball is going to be in a different position. A change in speed is linked to the temporal perception of the agent.***





How was the noise generated?

In the simulator section the ball had directions, therefore every time the ball hit the paddle on the side of the bot, I programmed it to choose a random number from a uniform distribution of ***(-x.+x)***. The ***xε{0.5 ,1 , 2),*** and add it to the direction, speed or both. The speed and direction, both take values in the range ***(-1,1)*** therefore this noise is significant.



When only the direction is changed, we observe that the variance of the reward curve goes up with the increase in noise. This is also accompanied by a drop in average scores. This means the agent behaves more unpredictably, and is worse off.

When just the speed is changed, we observe that the variance of the reward curve goes up too, with the increase in noise, as with the change in just direction. The change in speed, however, has a much more profound effect on the average rewards. (It has reduced much more.)

This increased drop can be attributed to the fact that ***our model doesn't really account for understanding temporal change,*** in any way apart from the stacking of our frames, through which, it again converts it into a spatial problem. The ***model just isn't equipped to handle a change in temporal dynamics*** of the system, as much as spatial.

When adding both the noises, direction and speed, to the same environment, we don't see a complete failure by the agent to play. The average score drops, and the variance of the rewards received curve go up, but the model doesn't fail entirely. This is a very good sign as now ***we're sure we've got a moderately generalisable model, even under severe noise***.

Conclusions

We have created an agent that can l**earn from high dimensional sensory inputs**, and after interacting with an environment. It can **play pong, with superhuman capabilities, beating humans in every game played.** It also **generalises somewhat well**, with a change in the environment's entropy.

Citations & Acknowledgements

Tesauro, Gerald. "TD-Gammon, a self-teaching backgammon program, achieves master-level play." *Neural computation* 6.2 (1994): 215-219.

Sutton, Richard S., and Andrew G. Barto. *Introduction to reinforcement learning*. Vol. 135. Cambridge: MIT press, 1998.

Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." *arXiv preprint arXiv:1312.5602* (2013).

A lot of PyGame tutorials that actually taught me how to make the simulation. (LocalTrain)

'Machine learning with Phil' has good videos showing how to implement papers in Tensorflow Code and Reinforcement learning tutorial videos in general.

Google for Collab, without who's GPU access this project would have taken a few years to test.