2019

Utilizing Various Neural Network Architectures To Play A Game Developed For Human Players

Michael Blake Arender
University of Mississippi

UTILIZING VARIOUS NEURAL NETWORK ARCHITECTURES TO PLAY A GAME DEVELOPED FOR HUMAN PLAYERS

A Thesis

presented in partial fulfillment of requirements

for the degree of Master of Sciences

in the Department of Computer Science

The University of Mississippi

by

Blake Arender

August 2018

ABSTRACT

Neural Networks have received an explosive amount of attention and interest in recent years. Despite the fact that Neural Network algorithms having existed for many decades, it was not until recent advances in computer hardware that they saw widespread use. This is in no small part due to the success these algorithms have had in tasks ranging from image classification, voice recognition, game theory, and many other applications. Thanks to recent strides in hardware development, most importantly in the advancements in Graphics Processor Units including the capabilities of modern GPU Computing, Neural Networks are now capable of solving tasks at a much higher success rate than other machine learning algorithms [3]. With the success of such Artificial Intelligence agents like DeepMind's AlphaGo and its stronger descendant AlphaGo Zero, the capabilities of Deep Neural Nets (DNNs) have received unprecedented mainstream coverage showing people across the world that these algorithms are capable of Superhuman level of play. After learning all of this I wanted to find a way to apply Artificial Intelligence to a game that was quite difficult for Human players: Cuphead. Cuphead is a classic styled game with terrific level design and music. It is a 2d Run N' Gun shooter also featuring continuous boss fights. The unique nature of the game provides clearly defined win and loss conditions for each level making this game a perfect environment for testing various Artificially Intelligent agents. For my Thesis, I used a Deep Convolutional Neural Network (CNN) to develop an agent capable of playing the first Run N' Gun level of the game. I implemented this neural network using keras and supervised learning. In this thesis, I will outline the exact process I used to train this agent. I will also explain the research I have done into how a

Reinforcement Learning agent would be implemented for this game as the Supervised Learning agent is currently only capable of playing the level I trained it on whereas I believe a sufficiently developed Reinforcement Learning agent could learn to play almost any level in the game.

ACKNOWLEDGMENTS

I am thankful for many people for their advice, insight, and most importantly their support of my efforts on this Thesis. Firstly, I would like to thank both Dr. Dawn Wilkins and Carrie Long both of whom without I would have never have been able to follow my passion for programming. I would not have joined and earned a degree from the Department of Computer Science at the University of Mississippi. I would also like to thank Joey Carlisle who was both a friend and Instructor throughout my undergraduate and graduate efforts. Thank you, Dr. Naeemul Hassan, for being a mentor and advisor throughout my graduate thesis. Most importantly I would like to thank my mother, Belinda Arender, who was a constant source of unwavering support and whose dedication I wish I possessed a fraction of.

TABLE OF CONTENTS

LIST OF FIGURES, TABLES, OR IMAGES

1. INTRODUCTION

Games provide an invaluable environment for the development, training, and testing of artificial intelligence algorithms. From simple games such as Tic-Tac-Toe to really complex games such as Go games provide an endless, easily controlled, and readily defined test environment for an artificial intelligence algorithm to learn and improve in. Typically, when an artificial intelligence agent is used to play a game, that game has been coded by the programmer specifically so that the agent can read and interact with the game. Alternatively, some game developers have also built Application Program Interfaces or APIs so that programmers and their artificial intelligence agents can directly interact with the game and get feedback on their inputs. This facilitates a much easier way for an artificial intelligence agent to learn and play a given game. For my Master's Thesis, I wanted to learn what tools were needed for an agent to learn how to play a given game in the same way a human would. I wanted it to play a game without coding the game specifically for the agent to interact with and without an Application Programming Interface for the agent to use to interact with and get feedback from the game. I wanted to learn what an agent needed to play a game developed for humans and not one designed to be played by an artificial intelligence. For this thesis, I attempted to use a combination of multiple neural networks to develop an agent with artificial intelligence capable of playing certain aspects of the game Cuphead. I chose Cuphead because it is a relatively difficult game for human players, yet it has easily defined win and loss conditions. In this project, I used Python 3.6 alongside Keras, a TensorFlow application programming interface, on my personal desktop which has an Intel I5 6600k overclocked to 4.5ghz Central Processing Unit,

Sixteen Gigabytes of Random Access Memory, and an NVIDIA 1080ti Graphics Processor Unit. This problem and project can be separated into two separate and distinct parts: The observer and the agent. The Observer learns to read the health of the player's character and provides the latest health readout to the Agent. The Agent should then take the feedback from the Observer and use that feedback to predict which actions keep it from taking damage in certain states. The agent should learn to do this based on its current health and the moves it has made so far as well as the state of the game (the location of the character and enemies) that resulted from that move. All the agent would need to learn is how to avoid taking damage as it does not need to learn to aim at the enemy, swap weapons, or any other more advanced mechanics. This is because there is an item that can be equipped in the game which causes the shots from the character home in on enemies. Therefore, the longer the agent can learn to stay alive the better its chance of beating the boss is as it will constantly be damaging the enemies. There are two types of fights in Cuphead: Run N' Gun and normal Boss fights. The Run N' Gun levels are pretty scripted and have fairly limited randomness to them and present a good opportunity to test an artificially intelligent agent as you would not necessarily need to code an observer. The Run N' Gun levels typically play out in very similar ways and a sufficient enough agent should be able to learn how to progress through those levels fairly easily. There is significantly more randomness in the boss fights and these cannot be won by simply progressing through the level. The agent needs to learn how to survive and to do this the reinforcement learning agent needs feedback on its health or when it has taken damage. The agent would need to learn a different objective of progressing through the level if it were to be applied to a Run N' Gun level as simply learning to not get hit would cause the agent to learn to never move forward. I attempted to code an agent for Boss fights and a separate agent for the Run N' Gun levels. Throughout this thesis I will outline the process I followed; I will explain

what I tried including what worked and what did not work. I will describe in detail neural networks and what architectures of various neural networks I tested as well as other approaches other researchers have used in applying agents with Artificial Intelligence to games.

2. NEURAL NETWORKS

Neural Networks have received an explosive amount of attention and coverage in recent years despite having existed for many decades. The earliest reference I was able to find that approached combining logical calculus with the architecture of biological neural networks was a paper written in 1943 by Warren S. McCulloch and Walter Pitts titled *A LOGICAL CALCULUS OF THE IDEAS IMMANENT IN NERVOUS ACTIVITY* [1]**.** In this paper, McCulloch and Pitts noted the "all or nothing" nature of biological neurons in which a neuron needs to be provided enough of a stimulus to cause it to fire. They then related this nature of biological neurons to the idea that a response from any neuron is factually equivalent to a logical proposition being satisfied. Neural Networks then made fairly slow progress until the 1990s where various approaches to error backpropagation greatly improved the performance of Artificial Neural Networks [2]. With the combination of error backpropagation and Convolutional Neural Networks, various classification tasks such as image classification, speech recognition, voice recognition, forecasting, and character recognition became fairly easy to solve. Deep Convolutional Neural Networks (CNN) saw great success in these classification tasks often reaching a 98 to 99 percent accuracy on the MNIST database (a database of images containing handwritten digits). This was a vast improvement over other various approaches which had accuracy rates ranging between 88 and 98 percent accuracy. The worst accuracy reported from a Convolutional Neural Network on the MNIST database was higher than most of the average accuracies from other various approaches such as Linear classifiers, K-Nearest Neighbors, SVM, and Non-Convolutional Neural Nets on the same database [3]. With improvements in hardware

most notably in Graphics Processor Units, the training time and accuracy of Neural Networks have steadily improved. With various application programming interfaces such as TensorFlow, Theano, and CNTK and the availability of such powerful Graphics Processor Units almost anyone can develop Neural Networks. This widespread availability has further increased the popularity of various architectures of Neural Networks in recent years. In this section, I will discuss various aspects of these Neural Networks including their structure, training, optimization, layer activation, and the various hyperparameters associated with neural networks. I will also discuss deep neural network layers, convolutional neural network layers, and LSTM (long-short term memory) layers. I will also present the main differences between training a Neural Network on a Central Processor Unit (CPU) versus training one on a Graphics Processor Unit (GPU); As well as the various libraries available to train these Neural Networks such as CUDA, cuDNN, TensorFlow, and Keras.

2.1 Neural Network Architectures

There exist many different configurations of Neural Networks. The specific layout of a given Neural Network is dependent on the task it is being developed to solve. In general, Neural Networks are layers of neurons which feed information forward to other hidden layers in the network and use backpropagation methods to calculate the gradients used to update the weights of neurons in the network. This backpropagation is what allows the Neural Network to learn and improve its chance of success at whatever task it is trying to solve. The specific way these gradients are calculated depends on the architecture of the Neural Network most notably in which optimization algorithm is applied and how loss is calculated in each round of training. There are quite a few optimization algorithms such as Root Mean Square Propagation

(RMSProp), Stochastic Gradient Descent (SGD), Adagrad, and Adam. Later on in this paper I will talk more about the optimizer I used in my agent's neural network: Adam. These optimization algorithms are used to either maximize or minimize the loss depending on what the neural network is trying to achieve and have many different hyperparameters that can be used to fine-tune the performance of the network. Typically, the objective of a neural network is to minimize error rates, which leads to lower loss as the network continues to train. Another approach is to consider the loss function to instead be a reward function such as with reinforcement learning. In this approach, the loss from each round of training can be considered to be the difference between the agent's reward from that step and the reward the agent expected to receive. There are again many different ways to calculate loss, but I will present the method I used for my agent, Cross Entropy (specifically Categorical Cross Entropy), as well as Mean Squared Error (MSE) which can be useful in a Deep Q Learning agent.

2.2 Training

Training a neural network can be achieved in any number of ways but typically these approaches fall under three categories: Supervised, Unsupervised, and Reinforcement learning [5]. Of the three learning paradigms Supervised learning is by far the easiest. In supervised learning the neural network is presented a large amount of manually labelled samples of data (images, soundbites, or more simple arrays of data) that the network is trained on to extract features and discern patterns so that it can predict or classify new samples into one of the labels or classes it has been trained on. Supervised learning is quite applicable to classification tasks as it can be fairly simple to generate large sets of training data which have been hand labeled. One potential downside of this paradigm is overfitting; if there are significantly more samples of one

class or label compared to the other classes then it is possible for the neural network to become overfitted to classifying new samples as that class. This is because the network can become more heavily weighted to that one class and when it encounters a sample of data it has never before seen it is likely to mispredict the label of that sample as the label it is overfitted to. This is in part due to probability and the network expecting to see more samples of the class it is overfitted to. Another potential source of mispredictions that supervised learning algorithms are various forms of uncertainty. Since feed-forward neural networks are prone to overfitting and when these neural networks are applied to supervised or reinforcement learning problems they are often incapable of determining the uncertainty of their predictions and are typically overconfident in the classification of new samples [6]. Uncertainty negatively impacts supervised learning approaches more so than reinforcement learning. This is because uncertainty in a reinforcement learning approach simply provides the network with more avenues of exploration and usually ends up benefitting these networks. Of the two types of uncertainty pertinent to neural networks, aleatoric and epistemic, the latter can be detrimental to the performance of a supervised learning agent yet beneficial to a reinforcement learning. Aleatoric uncertainty is due to variation in the samples whereas epistemic uncertainty is due to encountering never before seen samples. These uncertainties can be minimized as the neural network receives more training. For supervised learning approaches, more labeled datasets can help to reduce epistemic uncertainty by exposing the network to new samples of the various labels. Similarly, aleatoric uncertainty can be reduced by providing new samples similar to other previously seen samples but with slight variations in the data. In general, the way to improve the performance of supervised learning is to provide it more samples to train on. This is a potential limitation of the supervised learning approach as there simply may not be enough samples to provide or it may be very time consuming or costly

to label as each sample must be manually categorized into their respective labels. Training a supervised learning network is typically done by letting the network predict the label or class of each sample and comparing that prediction to the true label of the sample. The network is allowed multiple passes through the data, often called epochs, so that it can train over and over on the sample dataset. This allows the network to gradually improve its performance over time by minimizing the loss calculated by its loss function which is usually categorical cross entropy. Below is an image of the output from a supervised learning agent being trained using TensorFlow:

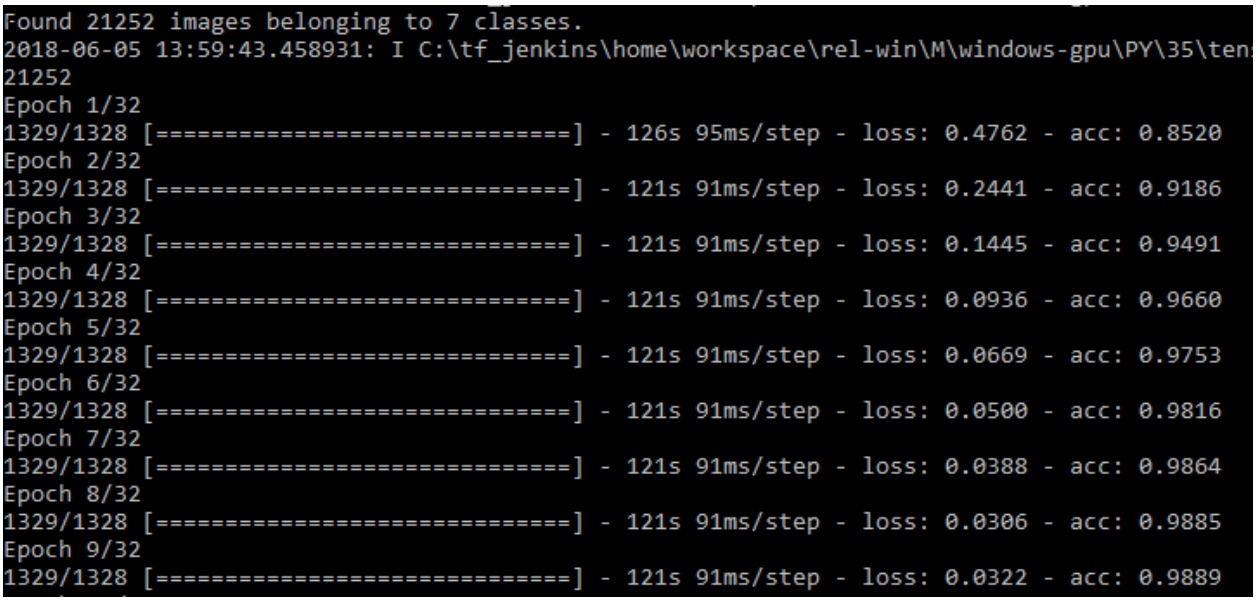

Figure 1: Image of a supervised learning neural network being trained with TensorFlow and Keras

Reinforcement learning is achieved through a somewhat different process. A reinforcement learning approach utilizes a reward function to determine its actions. It starts out with no knowledge of the environment or task it is trying to solve. Initially, it will make randomized decisions until it learns which actions in which states provide the largest and most consistent rewards. Reinforcement learning neural networks do not start off with a large dataset of samples that have previously been manually labeled. Instead, it learns to solve its task by

associating the reward it gets to the decision it makes. A reinforcement learning agent is capable of solving the task assigned to it by learning an optimal policy for the environment with no human input. Due to this nature, a reinforcement learning neural network is capable of learning new policies that a human would be unable to teach a supervised learning neural network. A supervised learning agent is only capable of learning what it is taught, unlike a reinforcement learning agent which learns how to solve a task in the most optimal way without any human input. Later on in this paper I will outline how I used supervised learning to develop an agent capable of beating the first Run N' Gun level of Cuphead and how I think a reinforcement learning agent could be used to beat nearly any level in the game.

2.3 Optimization Algorithms

There exist many different optimization algorithms that can be applied to neural networks most of which have a host of hyperparameters that can be used to fine-tune the performance of the network. Currently, the most popular optimization algorithm is Adam. Adam was introduced in a conference paper from the International Conference on Learning Representations in 2015. The name Adam is derived from adaptive moment estimation [7]. Adam is the result of attempting to combine the aspects of two other popular optimization functions: Adagrad and RMSProp [7]. Adam computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients [7]. Adam has certain advantages over other optimization algorithms such as: the magnitudes of parameter updates are invariant to the rescaling of the gradient, it works well with sparse gradients, and it naturally performs some step size annealing [7]. Adam is different from standard Stochastic Gradient Descent which has a

fixed learning rate compared to Adam which has a decaying learning rate to help prevent bias. The following is pseudocode for the Adam algorithm from the 2015 conference paper:

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

Figure 2: Adam pseudocode from 2015 ICLR conference paper [7]

As seen above in the pseudocode Adam has several hyperparameters which are used to fine-tune its performance: Alpha, Beta1, Beta2, and Epsilon. Alpha is also called the learning rate or step size and it is the rate at which weights in the neural network are updated. Higher learning rates can lead to faster learning initially but can ultimately cause the neural network to be unable find optimal weights for the neurons in the network. However, a low learning rate can cause the network to take significantly longer to train and may cause it to never be able to optimally solve its task in the number of passes through the data it is allowed. Beta1 is the exponential decay rate for the first moment estimates and similarly, Beta2 is the exponential decay rate for the second-moment estimates [8]. Epsilon is a very small number used to prevent division by zero in certain cases. The suggested or default settings for each of these parameters are as follows: Alpha = 0.001, Beta1 = 0.9, Beta2 = 0.999, Epsilon = 1e-08.

In general, Adam performs very well. It tends to perform somewhat better than other optimization algorithms at classification tasks requiring fewer passes through the training dataset. This is part of why Adam has been such a popular optimization algorithm recently as it achieves optimal results fairly quickly. The figure below shows the performance of Adam compared to several other popular optimization algorithms: AdaGrad, RMSProp, SGD Nesterov, and AdaDelta on the MNIST database of handwritten digits.

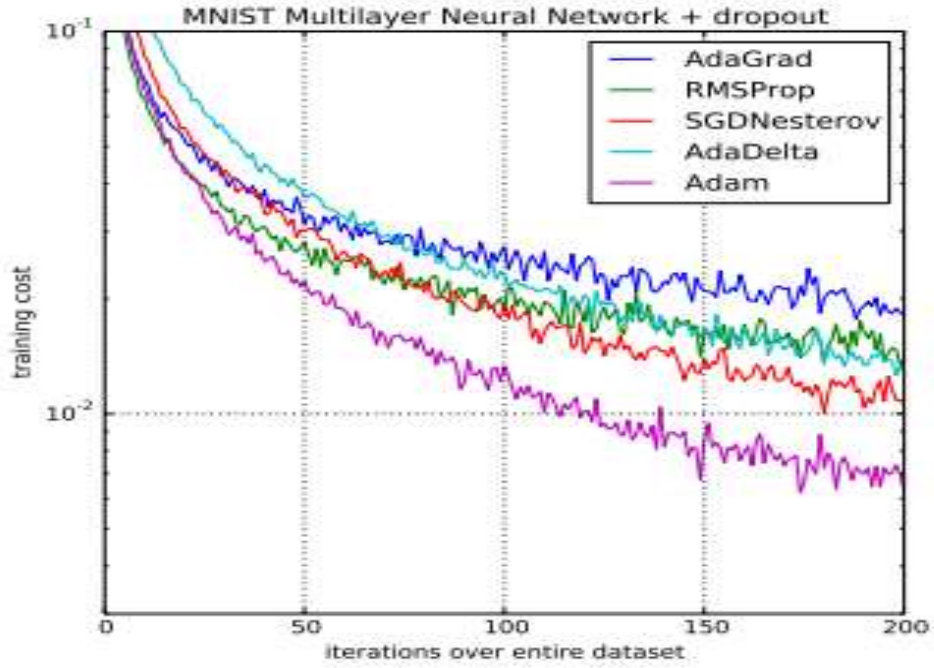

Figure 3: The performance of Adam compared to other popular optimization algorithms [8]

2.4 Calculating Loss Or Reward

The specific way loss is calculated is highly dependent on the task that the network trying to solve. Typically, for classification problems with a supervised learning approach, loss is calculated using a form of cross entropy, Categorical Cross Entropy. Cross entropy is very similar to Log Loss and measures the performance of a neural network whose output is a probability value that lies between zero and one [9]. Cross Entropy loss increases as the predicted probability of a given sample's label diverges from the true label of the sample [9]. If

the neural network were to predict a low probability of a sample being a certain label when that label is the true classification, that would result in a high loss value for that prediction. As the predicted value approaches one for the true label of the sample, the loss decreases. The graph below shows the range of possible loss values for a true observation [9]:

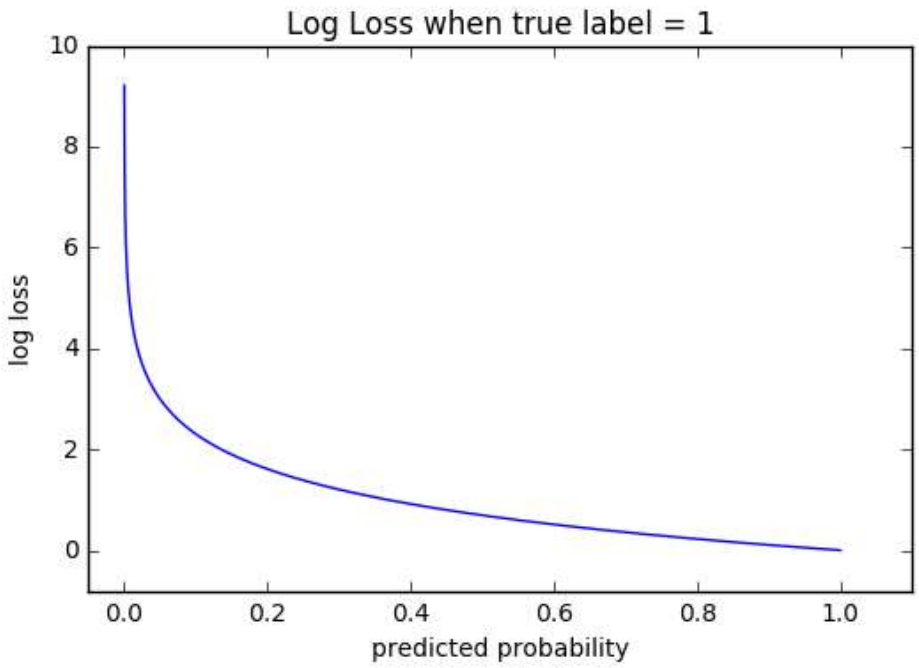

Figure 4: Example Log loss for a true value of 1.0 [16]

Since reinforcement learning is such a vastly different approach than supervised learning, loss is also calculated differently. For some reinforcement learning approaches such as Deep Q Learning, loss can be considered to be a reward function. In each round of training, the agent will predict which decision results in the highest reward value. Mean Squared Error can then be applied to determine the difference between the predicted reward and the reward the agent actually received. This allows the agent to learn the best action to take in a given state by minimizing its loss. This loss is the difference between the reward the agent expected to get and the reward it actually got. By minimizing this difference, the agent is capable of learning how to

accrue the largest amount of rewards by accurately predicting which rewards result from which decisions in a given state.

2.5 Deep Neural Networks

Deep Neural Networks differ from Neural Networks only by the number of layers present in its architecture. Typically, any neural network with three or more layers is considered to be "deep" whereas any neural network with two or fewer layers is considered "shallow". The specific number and types of layers present in a deep neural network are highly dependent on the task it is trying to solve. Typically, shallow neural networks are only used for supervised learning tasks such as classification or prediction whereas deep neural networks can be applied to supervised, unsupervised, and reinforcement learning tasks. There are several different types of layers that can be combined to form a deep neural network such as Convolutional layers, Pooling layers, Fully Connected layers, Activation layers, and Dropout layers. When these layers are combined and have their information fed into the subsequent layers they are capable of incredible success.

2.6 Convolutional Neural Network Layers

Convolutional neural network layers are exceptional at extracting features from input images. This allows convolutional neural networks to perform exceptionally well at image classification. Convolutional layers are a digital approximation of vision. In biological vision, a receptive field is the specific region of the retina in which a stimulus will cause certain neurons to fire. This biological nature of vision is approximated in convolutional neural networks by gradually building filters that the network slides across the input image in order to extract a

feature map. Each convolutional layer in a neural network applies a specific number of filters from the input image and extracts a feature map for each filter and at the end, these maps are combined as the output of the layer. The output of the layer then typically has a nonlinear activation applied to it such as ReLU as well as a pooling layer after ReLU has been applied. Without a nonlinear activation function, the neural network would only perform roughly as well as a single layer neural network no matter how many layers the deep neural network has. Pooling layers are used to help reduce the parameter size and computational cost of the convolutional layers. The most common architecture for a Deep Convolutional Neural Network is three convolutional layers each followed by a ReLU activation and pooling layer followed by one or more dense or fully connected layers and a dropout level before one final dense layer which has as many neurons as labels in the dataset and an activation layer using either a sigmoid function or more commonly the softmax function; The convolutional layers are used for feature extraction for the input image, the dense layers are used for decision making on the resulting feature map provided by the convolutional layers, dropout helps to prevent overfitting, and the final activation layer is used to measure how likely it is the given input sample falls into each class. In the next two sections of this paper I will present dense neural network layers and dropout layers. Below is an example of a deep convolutional neural network layout:

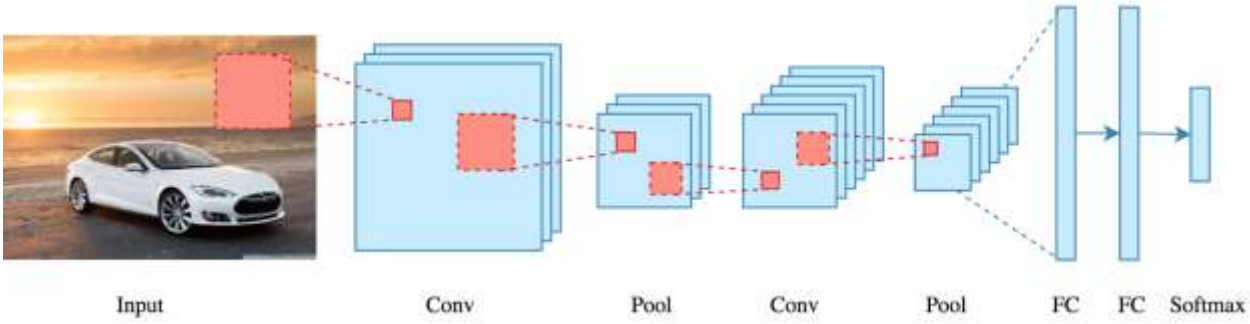

Figure 5: Example CNN architecture [14]

2.7 Dense Neural Network Layers

Dense neural network layers are layers in which each neuron in the layer is connected to every neuron in the previous layer. When used in a deep convolutional neural network the dense layers are used for classification after receiving input from the convolutional layers. Before receiving input from the convolutional layers, the tensor is flattened so that the input only has two dimensions: batch size and features. The fully connected layers in a neural network allow it to learn nonlinear combinations of the features extracted by the convolutional layers. These combinations are what allows the network to make predictions on the classification of the input. From what I understand since the final dense layer in a deep convolutional neural network has one neuron for each label in the dataset and each of these neurons is connected to every neuron in the previous dense layer, the way the neural network decides the percentage chance for the input sample to be each of the classes is based on how many neurons from the penultimate dense layer fire and activate each of the neurons in the final dense layer. So, if eighty percent of the neurons in the penultimate dense layer activate one neuron in the final dense layer it indicates that the neural network is predicting the sample to belong to the class or label associated with that one neuron.

2.8 Dropout Layers

Dropout layers are used to prevent overfitting when training a neural network. Dropout layers function by turning off a percentage of neurons during training time. This helps to prevent the neural network from getting too overconfident and too set in its current ways by forcing it to use different neurons to solve the task. Dropout is one approach to regularization in machine

learning and regularization helps to prevent overfitting by adding a penalty to the loss function [10]. This penalty prevents the model from learning interdependent sets of feature weights [10]. Overall dropout sacrifices training cost for a more generalized model. A neural network with dropout enabled can take longer to train as it is initially more prone to making classification errors but generally eventually performs better than a comparable neural network without dropout. Below is a figure that shows the performance of different neural networks with and without dropout enabled.

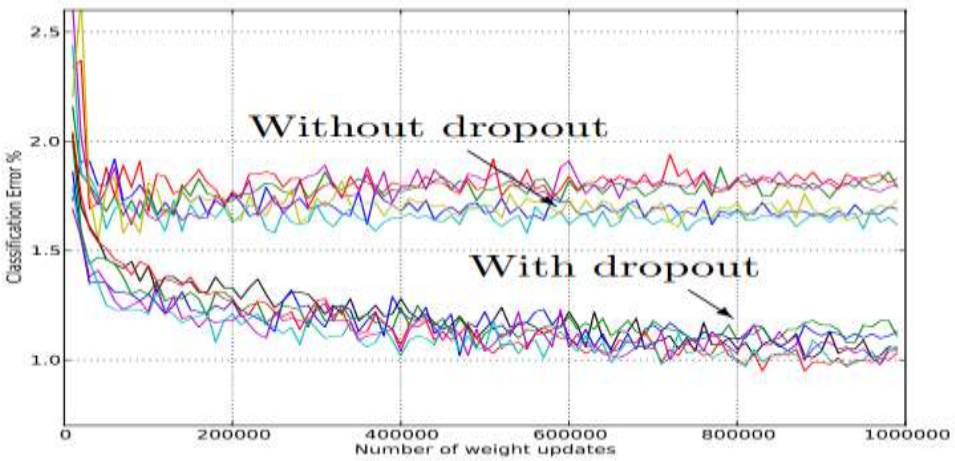

Figure 6: Performance of various neural networks with and without dropout [14

2.9 Training on a CPU vs GPU

In almost all cases training a neural network on a Graphics Processor Unit is significantly faster than training one on the Central Processor Unit. A CPU is a general-purpose processor and is a good choice for operations with random or non-uniform memory accesses. A GPU is a highly specialized processor unit and is exceedingly fast at performing the same operations over and over. Deep Neural Networks are structured in such a manner that in each layer of the network numerous artificial neurons are performing the same calculations. GPUs are designed to perform the same instructions many different times in parallel. GPUs also have other advantages

over CPUs for training a neural network such as having more computational units or cores and having a higher bandwidth for memory access to the RAM located on the GPU. One potential advantage CPUs have over GPUs for training a neural network is the amount of memory available. The amount of RAM available on GPUs is typically eight to twelve gigabytes whereas a CPU can have access to up to one terabyte of RAM depending on the hardware and operating system used. Another potential limitation is that a CPU is required to transfer data to the GPU and that the clockspeed of GPUs are typically 1/4th to 1/3rd less than the clockspeed a modern CPU is capable of achieving. GPUs however still vastly outperform CPUs when training a neural network due to the sheer number of cores available to the GPU and the parallelizability of the computations present in a neural network. The image below shows the difference in training speed between two different CPUs and two different GPUs:

Device	Speed of training, examples/sec
2 x AMD Opteron 6168	440
i7-7500U	415
GeForce 940MX	1190
GeForce 1070	6500

Figure 7: The difference in training speed between CPUs and GPUs [15]

2.10 TensorFlow

TensorFlow is an open source library for rapid calculation of tensor mathematics. Most modern neural networks use tensors to operate. Images are input as tensor information and the outputs are usually vectors, one-dimensional tensors. TensorFlow was developed by Google and released under an open source library in 2015. TensorFlow is capable of running on most modern CPUs and GPUs. In May of 2016 Google released a new unique type of processor called a TPU or Tensor Processor Unit. As its name suggests TPUs were developed specifically for machine learning and tensor processing. For comparison, an NVIDIA 1080ti is capable of 11.3 teraflops whereas a TPU is capable of up to 180 teraflops. TensorFlow is available on multiple different

platforms such as Windows, MacOS, Linux. It is even available as a lite version on Android and in web browsers through cloud computation. TensorFlow is also written for Python, C++, and CUDA with several APIs for other languages. For my project, I used the python implementation of TensorFlow. It requires Python 3.6.x for the CPU only version. TensorFlow also requires CUDA 9.0 and cuDNN 7.0 for the GPU enabled version.

2.11 Keras

Keras is also an open source library for the development of neural networks. It is an API that runs on top of popular machine learning libraries. Keras is capable of using TensorFlow, Theano, and Microsoft's Cognitive Toolkit or CNTK [12]. Keras was developed with a focus on ease of use for experimenters. It is very useful for quickly prototyping various neural networks. Keras allows the developer to define each layer of their neural network with one line of code per layer. Keras makes the process of supervised learning incredibly easy as it has all the necessary tools to do so included in it. Keras has several predefined layers such as convolutional, dense, lstm, pooling, and dropout layers. Developers can however use the lambda wrapper layer to define their own custom layers. Keras also provides predefined loss and optimization functions including cross entropy, mean squared error, stochastic gradient descent, rmsprop, adagrad, and adam. Keras also provides an ImageDataGenerator class which provides a simple means of processing and feeding image data into a neural network. The ImageDataGenerator class allows for real-time data augmentation on the CPU whilst the neural network is being trained on the GPU. This means that images can be augmented or changed such as resizing or rescaling the image at the same time the agent is being trained on the GPU

3. CUPHEAD ARTIFICIAL INTELLIGENCE AGENT

Note: In this section, I will outline the approach I took to develop an artificially intelligent agent capable of playing some level in Cuphead as well as describe one potential reinforcement learning approach.

3.1 Supervised Learning Agent

Note: In this section, I will discuss the way I approached using a supervised learning agent to play the first Run N' Gun level of Cuphead

After a few months of attempting to develop a reinforcement learning agent to play Cuphead, I decided to change my approach and apply what I actually knew how to do. I decided to apply a supervised learning approach to the first Run N' Gun level in the game. To achieve this, I implemented a deep convolutional neural net using Keras and TensorFlow. This project was run on my personal desktop which has an NVIDIA 1080ti, an Intel I5 6600k running at 4.5GHZ, and 16GB of 2400MHz Ram. I used the PyAutoGui Library to input the keystrokes that the agent wanted to use. The next few sections of this paper will cover the process I took and the programs I used to develop the supervised learning agent. This agent controls the Cuphead character to play the level it has been trained on. In the Figure 8 shown below you can see Cuphead and a small part of the level I trained the agent on.

Figure 8: An image of Cuphead on part of the first Run N' Gun level

3.1.1 Screenshot taking process

I attempted to write the screenshot taking process in Python, but it was simply too slow so instead I wrote it in Java. The screenshot program takes an 800x600p size screenshot, since that it the lowest resolution that I could lower my screen to. Originally, I was taking full 2560x1440p sized screenshots, but this was a slow process even for Java and it produced a lot of overhead for the neural network to resize this large image to a size more appropriate for convolutional layers. By reducing the size of the screenshot from 2560x1440p to 800x600p, it greatly improved the speed of the screenshot taking process and the time between predictions as the network had to do less work to resize the image to a smaller size suitable for the

convolutional layers. By reducing the initial size of the screenshot, I was able to increase the number of screenshots the program was capable of taking per second from six screenshots per second to about thirty-two screenshots per second. This is much faster than the neural network is able to process and predict input images meaning more screenshots will be taken than can be read by the neural network. The neural network only reads the most recent screenshot available to it. The name of each screenshot is a number and the way the neural network determines the name of the most recent screenshot is to calculate how many screenshots are currently in its training directory and how many new screenshots have been taken since. It then adds those numbers together which results in the number that is the name of the most recent screenshot taken or the screenshot that is currently being taken. The screenshot program actually makes two copies of the image. One is saved in a directory containing all of the new screenshots that have been made so that it is easy to look through all of the new screenshots and manually label them. The other copy of the screenshot is placed in a folder labeled unknown inside another folder that is numbered the same as the screenshot. Since I used Keras to implement this neural network I used the ImageDataGenerator class. This class makes it easy to process images to make them suitable for convolutions. The ImageDataGenerator class expects the images it is reading to reside inside a directory that is located inside another directory. It will iterate through every image in every directory inside the given directory in batches performing various transformations on the image such as resizing or rescaling. Resizing the image to a significantly smaller size greatly reduces the computational cost of the convolutional layers. Rescaling converts the pixel values for the image to another value which is usually smaller. This is done by multiplying each pixel value by a given fraction. Typically, the pixel values for the image will be rescaled from a 0 to 255 value to a value between 0 and 1 by multiplying the pixel value by the fraction 1/255.

This converts each 0-255 pixel value to a decimal value between 0-1. For example, a value of 100 would be converted to 0.39216. These smaller decimal values make it easier for the neural network to learn. If the ImageDataGenerator class is being used during training the name of the directory containing the images will be used as the label for those images. This makes labeling samples as easy as moving the images into their respective class's directory.

3.1.2 Neural Network Structure

The agent's neural network consisted of three convolutional layers with each convolution followed by a ReLU activation layer. I did not apply any max pooling after the convolutions because I did not want the network to lose any information about how close together or far apart objects are on the screen. The output from the final convolutional layer is then flattened and passed to a dense fully connected layer. A ReLU activation and dropout are applied and the output from that dense layer is passed to another larger dense layer. Another ReLU activation and dropout are applied before the output is passed to the final dense layer which has as many neurons as labels in the dataset. These labels in the dataset are all of the moves that the agent can have the character perform. The output from this final dense layer has a softmax activation applied to it which forces the sum of the chances for the sample to belong to each class to equal one hundred percent. The first two convolutional layers have thirty-two filters each with a stride of (3,3). The final convolutional layer applies sixty-four filters. The first convolutional layer takes the Input shape keyword. The input shape for this neural network is the size of the input image which is a three-parameter` tuple containing the height, width, and the number of channels for the input image. The input shape tuple for my neural network is (84,84,3) meaning the size of the image is 84x84p and has 3 color channels. The image data generator class from Keras will

handle resizing the 800x600p to 84x84p. I originally tried a size of 256x256p and then 128x128p, but these resulted in too much computational cost for the convolutional layers and caused the decision-making process on which move to make to take a couple fractions of a second too long. I settled on a size of 84x84p as this was the size used by Deep Mind in their Deep Q learning Neural Network to play any Atari game. It also seemed to allow the network to perform optimally when deciding which move to make. This caused the agent to sometimes take unnecessary damage such as running off an edge where it would normally know to jump. The input shapes for the rest of the layers are also handled by Keras. After the layers of the neural network have been defined, the model is compiled with its loss function, optimizer, and the metrics it is being trained on (usually accuracy). Below is the code used to initialize this neural network with Keras:

```
1  resize = (84,84)
2  model= Sequential()
3
4  model.add(Conv2D(32,(3,3), input_shape = (resize[0],resize[1],3)))
5  model.add(Activation('relu'))
6  #model.add(MaxPooling2D(pool_size=(2,2)))
7
8  model.add(Conv2D(32,(3,3)))
9  model.add(Activation('relu'))
10 #model.add(MaxPooling2D(pool_size=(2,2)))
11
12 model.add(Conv2D(64,(3,3)))
13 model.add(Activation('relu'))
14 #model.add(MaxPooling2D(pool_size=(2,2)))
15
16 model.add(Flatten())
17
18 model.add(Dense(32))
19 model.add(Activation('relu'))
20 model.add(Dropout(0.25))
```

```
21
22  model.add(Dense(64))
23  model.add(Activation('relu'))
24  model.add(Dropout(0.25))
25
26  model.add(Dense(len(moveSet)))
27  model.add(Activation('softmax'))
28
29  model.compile(loss = 'categorical_crossentropy',
30                                optimizer = adam,
31                                metrics=['accuracy'])
```

The lines preceded by a # are commented out and are the pooling layers I chose not to include in the neural network. Since my approach for the supervised learning agent is a classification task I used the categorical cross entropy loss function which is already implemented in Keras. I also used Adam as the optimizer. When using Keras you can either use the default parameters for various optimization algorithms or you can define the parameters yourself by accessing the Keras API. Below is how I defined my parameters for the Adam optimization algorithms using Keras:

```
adam = keras.optimizers.Adam(lr=0.00056, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.00001)
```

In this implementation, I set the learning rate to 0.00056 from a default value of 0.001. I left beta_1, beta_2, and epsilon as their default values. I also added a small amount of decay to the learning rate. I chose a learning rate of 0.00056 because I wanted to give the neural network a better opportunity to learn optimal weights at a slight cost to training time. I tried an even lower learning rate of 0.00012 but the network did not perform very well. I settled on the 0.00056 learning rate because it allowed the neural network to perform optimally while still lowering the learning rate.

3.1.3 Training Of Supervised Learning Agent

I trained this supervised learning agent by providing it a large amount of manually labeled images. I sorted the images into their respective labels based on the move I felt the agent should make immediately after that frame occurs. For this agent, there are seven potential moves the agent can make. The neural network predicts a value of 0-6 and the number it predicts is associated with one move. A prediction of 0 causes the agent to turn left and jump. A prediction of 1 is an empty move which causes the agent to stand still. A predicted value of 2 causes the agent to stand still and aim upwards. A predicted value of 3 makes the agent move right and progress through the level. A value of 4 lets the agent move right and perform a dash move. A predicted value of 5 causes the agent to jump right and dash while in the air. A value of 6 will cause the agent to jump to the right. In all cases, the agent is constantly shooting an equipped item that causes its shots to home in on enemies. For a value of 3,4,5, or 6 the agent continues to hold down the D key until the next move is decided. This gives the agent an easier time and meant that I did not have to teach the agent how to aim. I also equipped the agent's character with an item that gives it one more health at a slight cost to its damage output. This gives the agent more of a chance to beat the level because it means it can take one more hit throughout the level. Each time the agent takes damage it loses one health and gains invincibility for a few seconds. Next, I will show some screenshots taken while the agent was playing and describe its label as well as why the agent should perform the move associated with that label.

Figure 9: A screenshot from Cuphead Labeled 0

Figure 9 was labeled as 0 because the agent needs to jump up and to the left. In this image, the agent is currently running to the right but it gets stuck here on the wall. If it were unable to jump to the left it would be unable to progress through the level. The agent very rarely will use this label as there are only a few instances where it needs to move to the left. This part of the level is the most prominent time this move gets made but later on in the level when fighting what can be considered the boss it may need to move left if it gets too close. Without this label, the agent would have a difficult time progressing through the level as it would not be able to make it over this specific cliff.

Figure 10: A screenshot from Cuphead Labeled 1

In Figure 10 the agent needs to stand still and fire until there are no more enemies in front of it. This image was labeled 1 to achieve this. This label occurs pretty frequently and happens whenever there is an enemy on screen in front of the character. Another label is used when enemies are above the character. I chose to use this label whenever enemies are on screen because it allows the agent to clear out the path ahead of it safely and avoid unnecessary damage. This action of standing still is not without risk because the agent can get overwhelmed if too many enemies show up on screen at once such as in the screenshot shown in Figure 8. Enemies can also spawn behind the character and can cause damage to it if the agent stands still for too long. Generally however standing still and clearing out the enemies in front of the agent works

out pretty well as it prevents the agent from simply running into an enemy and taking unnecessary damage.

Figure 11: A screenshot from Cuphead Labeled 2

The screenshot in Figure 11 was labeled as 2 because there is an enemy above the character that will drop onto the character and deal damage if the agent does not move or kill it. The label of 2 causes the agent to stand still and aim upwards. I used this label whenever there was an enemy above the character that needed to be dealt with before the enemies in front of the agent. By aiming up it helps to ensure the shots fired by the character are homing in on the target above instead of an enemy in front of it. This label was used more often when these acorns are spawning and flying in above the character. Once the agent gets to the "boss" of this level it

simply stands still and aims upwards because the boss is what spawns these acorns and otherwise does not attack.

Figure 12: A screenshot from Cuphead Labeled 3

In Figure 12 the agent has cleared out all of the enemies in front of it and needs to continue progressing through the level. This image was labeled as 3 because it is clear of enemies and the agent needs to move to the right. This label is used any time the agent needs to move to the right and the path in front of the agent is clear of enemies. This label is also sometimes used when a flying enemy is about to drop on top of the character and it does not have time to kill it before taking damage. This label had the highest number of samples because

it was the most used label and is the action that allows the agent to move and progress through the level.

Figure 13: A screenshot from Cuphead Labeled 4

In the image shown in Figure 13, the agent needs to dash to the right to avoid taking damage. This image was labeled as a 4 to make this happen. These pink spiked balls float up and down. The agent could simply run to the right in this specific instance, but I have found that teaching the agent to dash under these spiked balls gives it a significantly larger window for passing by them without taking damage. This move can also be used if the agent needs to cross a gap, but the next label is more suited for this.

Figure 14: A screenshot from Cuphead Labeled 5

In Figure 14 the agent needs to move to the right, jump, and dash over the gap to avoid taking unnecessary damage by falling into it. A label of 5 will cause this to happen. This move is used typically when there is a gap that can be crossed by jumping and dashing over it. This move got used extensively at the end of the level because there are a lot of gaps that need to be crossed. There are a couple gaps throughout the level that can be crossed in this manner. There are also some instances where the agent can use this move to jump over enemies specifically when there is an enemy that does not take damage until it pops up to shoot back at the character and an enemy has spawned behind the agent. In a case like this, the agent may decide to jump and dash over the temporarily invulnerable enemy to avoid taking damage from the enemy behind it.

Figure 15: A screenshot from Cuphead Labeled 6

The screenshot shown in Figure 15 is labeled as 6 because there is an obstacle in front of it that it needs to jump over in order to progress through the level. This label is used whenever there is an obstacle or projectile that the agent needs to jump over to progress through the level or to avoid taking damage. There are also some gaps in the level that can simply be jumped over instead of jumping and dashing.

To generate a training data set for the neural network I allowed the agent to play through the game for several attempts at a time and manually sorted each image into the label for the move I felt the agent needed to make immediately after that frame in the screenshot. After being

trained on the current dataset, the model was saved to a file which could be loaded at any time. Keras makes this process very simple:

```
model.save(directory_name)
```

The code above saves the model to the specified file and to load a model you can use:

```
model = keras.models.load_model(directory_name)
```

Which loads the specified model. With these two methods you can iteratively improve the model by loading a previously trained agent to train on newly labeled samples. While training the agent I went through about eighteen iterations of training an agent, letting it play through the level, sorting the resulting screenshots from the agent playing, and then training a new agent on the newly labeled samples. When letting a new agent be trained I left old samples in the training directory so that it had more opportunities to train on those samples as well as on the newly labeled samples. Initially, I allowed the agent 56 epochs or rounds of training on the data but steadily decreased it to 32 and then 24 as the training time increased because of the number of samples in the training dataset. Also, as the training time increased I would remove images from the training data set that the agent had already been trained on multiple times in an attempt to reduce the training time of each round. I tried to keep each round of training between an hour to two hours; so I would say in total the agent had 25-36 hours of training before it had its first successful run after 18 iterations of training. By the time the agent made it to the end of the level, it had been trained on roughly 60,000 images. Typically, by the end of each training session, the agent had achieved a greater than 98% accuracy on the training data set. If the agent ever learned bad habits such as running off the edge of gaps instead of jumping them or trying to jump over the pink spiked balls instead of dashing underneath them I attempted to remove or

relabel the screenshots I thought were causing those bad habits. I then rolled back the agent to a previous iteration that had not yet learned those bad habits and started the training over from there. Below is an image showing one full round of training:

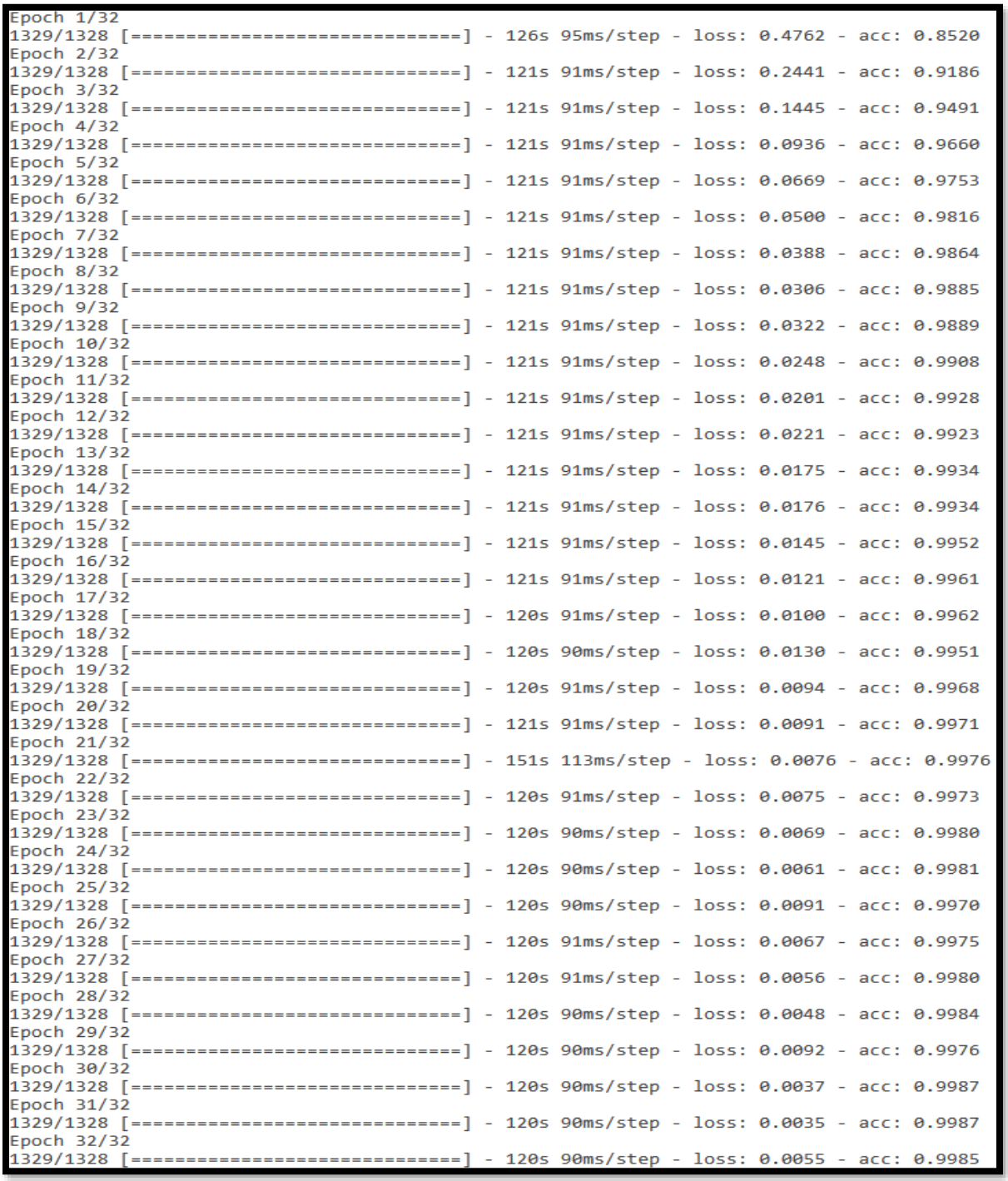

Figure 16: Full round of training for the supervised learning agent

3.1.4 Performance

Overall, I was incredibly surprised at how well this supervised learning approach worked. I fully expected this neural network to take well over 100,000 samples before it was capable of consistently making it halfway through the level however its first successful run occurred after only being trained on 60,000 images. In fact, the Agent was consistently beating half the level after being trained on 25,000 images. As expected the performance of the agent increased consistently as it was trained on more images. The agent program first loads the last saved model and calculates the total number of screenshots in the training directory which is used to calculate the name of the most recent screenshot taken. It then sets the Pause value for PyAutoGui to 0.045 seconds from a default value of 0.1. This pause value is how long the program sleeps in between keystrokes and took quite a lot of fine-tuning. I settled on the 0.045 second pause value after some trial and error. The default value of the pause in PyAutoGui is slightly too long for most moves but perfect for trying to jump. When moving right the default pause length can sometimes cause the agent to move too far and run off the edge of a gap when it is trying to move right and jump. The 0.045 second pause is too short however to get the full height out of a jump. To get around this I set the pause to 0.1 seconds before it jumps and reset the value back to 0.045 after as this provided the overall best performance of the agent. The performance of the agent increased steadily as the number of images it had been trained on increased. The best run from the final iteration, which was a successful run, had a lower time than the time of the best run from the previous iteration which indicates it was progressing through the level faster than previous iterations. While the time of the best run (the run in which it completed the largest amount of the level) from each iteration is not a complete indicator of the performance of the agent during that iteration of training it does provide some insight to the progression of the agent

over iterations. The time of the best run from each iteration is an indication of how well the agent is performing in that indication but this can be affected by how many enemies the agent has to deal with in that run and in fact a lower time is considered to be better by the game. Below is a chart that graphs the best run time from some of the iterations of the agent:

Figure 17: Best Run Time for some iterations of the agent

The performance of the agent was somewhat dependent on luck. If the agent got overwhelmed by enemies in front of it and had enemies spawn behind its character, it almost always took damage. Luck however is not a factor unique to the artificial intelligence agent as luck certainly plays its part when humans are playing the game. The performance of the agent did degrade as it continued to play more and more rounds and I believe the reason for this is the way the name of the most recent screenshot gets calculated. The piece of code that achieves this is:

```
len([name for name in os.listdir(ss_data_dir) if os.path.isfile(os.path.join(ss_data_dir, name))]))
```

This allows the program to count how many new images have been created. As the number of new images gets to seven or ten thousand the time it takes for the agent to make a new move increases by a fraction of a second. This causes the agent to sometimes take unnecessary damage by either running into enemies where it would otherwise stop or run off the edge of a gap where it would otherwise jump. Since the line of code used to calculate the name of the most recent screenshot occurs each time the agent is deciding its next move it has to count how many new images have been created and it takes slightly longer each and every loop. Ultimately, I was very happy with how well the agent ended up performing. I was surprised at how few images it took to get a successful run as I fully expected it to take many more samples. I thought there would be too much variation in the images for the network to reliably predict the move I wanted it to make in a given frame. I simply thought there would be too much misprediction for the agent to reliably progress through the level. Given that the agent is using keyboard inputs it plays better than I could if I had to use a keyboard to play this game. A video of the agent getting progressively better at playing the level can be found at the following link: https://www.youtube.com/watch?v=VSwnuM7c9qo&feature=youtu.be\

3.1.5 Potential future improvements

There are many avenues for potential future improvements that could be made to this agent. The easiest and most obvious one is simply providing more labeled samples for the agent to be trained on. The agent did not get much of a chance to train on the last ten percent or so of the end of the Run N' Gun level so there is room for improvement when the agent gets to that part of the level. Another potential way to improve the performance of the agent may be to

program it to use controller inputs instead of keyboard inputs. Cuphead was designed to be played with a console controller. It is significantly easier to control the agent with a console controller compared to trying to use a keyboard and could allow the agent finer-tuned control over the character. Applying max pooling after the convolutional layers could potentially help improve the performance of the agent by reducing the computational cost of predicting a move from an input image. It could also prove to have a negative impact on the performance of the agent. The main reason I chose not to include max pooling because I did not want the neural network to lose any information about how close together or far apart objects in the screenshot are. The most promising avenue for improving this project would be to completely start over with the agent and implement a reinforcement learning agent.

3.2 Reinforcement Learning Agent

Currently, the supervised learning agent is only capable of playing the level it has been trained on. I believe a reinforcement learning approach could be applied to almost any level in the game. The approach would need to be completely different from the supervised learning approach as the agent would need to learn what to do in certain situations on its own. I think the best approach to this would be to let the agent learn to not take damage. If it can simply learn to avoid damage and survive it should be able to beat most of the levels in the game. This should be achievable by applying Deep Q Learning and letting the network predict which action has the greatest chance of preventing the agent from taking damage. This approach would take two separate neural networks working in tandem: one observer and one agent. The observer neural network would simply read the health of the character and provide it to the agent. The agent could then determine the reward it got from the action it took. Over time the agent should be able

to minimize the difference between the reward it predicts it will get and the reward it actually got from an action in a given state. This will let the agent predict which action in which state results in the best reward. The agent should receive a small reward for each frame that it did not lose health in and potentially lose some reward if it detects that it lost health. I said that this approach would require two neural networks and that is because it needs one network to learn to play the game and one network to learn to read the game. Although these networks are separate they could be considered to be two parts of the same artificial brain. All the observer really needs to be able to read is the health of the character. This is always located in the same spot on the screen and is simply a number; therefore the observer could be a supervised learning neural network designed to read numbers. A deep convolutional neural network would work exceedingly well at this as it could be provided with small screenshots just containing the health of the character. The observer could then be trained on numerous samples of the various numbers that represent the character's health. The observer would then be designed to determine which actions in which states give it the best chance of not taking damage or give it the best reward. It would then calculate the reward it actually received based on the current health of the character provided by the observer. As the agent progressively gets better at predicting the reward it will get, the agent should also get better at staying alive by not taking damage because it should choose the action it predicts will result in receiving a reward.

LIST OF REFERENCES

1. Mcculloch, Warren S., and Pitts, Walter. “A Logical Calculus of the Ideas Immanent in Nervous Activity.” *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, 1943, pp. 115–133., doi:10.1007/bf02478259.

2. LeCun, Yann.. “A theoretical framework for back-propagation”. In D. Touretzky, G. Hinton, and T. Sejnowski, editors, *Proceedings of the 1988 Connectionist Models Summer School*, pages 21-28, CMU, Pittsburgh, Pa, 1988. Morgan Kaufmann.

3. THE MNIST DATABASE of handwritten digits. Available at: http://yann.lecun.com/exdb/mnist/

4. Ruder, Sebastian. “An overview of gradient descent optimization algorithms”. *arXiv:1609.04747v2* (2017)

5. Sathya, R., and Annamma Abraham. “Comparison of Supervised and Unsupervised Learning Algorithms for Pattern Classification”. *International Journal of Advanced Research in Artificial Intelligence*, vol. 2, no. 2, 2013, doi:10.14569/ijarai.2013.020206.

6. Blundell, Charles., Cornbise, Julien., Kavukcuoglu, Koray., and Wierstra, Daan. “Weight Uncertainty in Neural Networks”. *arXiv:1505.05424v2* (2015)

7. Kingma, Diederik P., and Ba, Jimmy., “Adam: A Method for Stochastic Optimization”. *arXiv:1412.6980v9* (2017)

8. Gentle Introduction to the Adam Optimization Algorithm for Deep Learning. Available at: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

9. Loss Functions. Available at: http://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html

10. Dropout in (Deep) Machine learning. Available at: https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5

11. Mnih, Volodymyr, et al. “Human-Level Control through Deep Reinforcement Learning.” *Nature*, vol. 518, no. 7540, 2015, pp. 529–533., doi:10.1038/nature14236.

12. Keras Documentation. Available at: https://keras.io/

13. Dropout Layers. Available at: https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/dropout_layer.html

14. Applied Deep Learning - Part 4: Convolutional Neural Networks. Available at: https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2

15. Tensorflow performance test. Available at: https://medium.com/@andriylazorenko/tensorflow-performance-test-cpu-vs-gpu-79fcd39170c

16. Log Loss. Available at: http://scikit-learn.org/stable/modules/generated/sklearn.metrics.log_loss.html

17. TensorFlow Documentation. Available at: www.tensorflow.org

18. Silver, David, et al. "Mastering the Game of Go with Deep Neural Networks and Tree Search." *Nature*, vol. 529, no. 7587, 28 Jan. 2016, pp. 484–489., doi:10.1038/nature16961.

19. Goodfellow, Ian, et al. *Deep Learning*. MIT Press, 2017.

20. Raschka, Sebastian, and Vahid Mirajalili. *Python Machine Learning: Machine Learning and Deep Learning with Python, Scikit-Learn, and TensorFlow*. Packt Publishing, 2017.

VITA

MICHAEL BLAKE ARENDER

EDUCATION

M.S., Engineering Science, University of Mississippi, August 2018

Concentration: Computer Science

Thesis: Utilizing Various Neural Network Architectures To Play A Game Developed For Human Players

B.S., Computer Science, University of Mississippi, May 2016

TEACHING EXPERIENCE

Graduate Instructor, August 2017 – May 2018
University of Mississippi
Course: CSCI 103 Survey of Computing

Graduate Assistant, August 2016 – May 2017
University of Mississippi
Courses: Matlab, Java I, Java II, Multimedia

Lab Assistant, August 2014 – August 2016
University of Mississippi
Adler Labs, Computer Science Department