Deep Gaming

Samuel L. Bauza
Electrical and Computer Engineering
University of California, San Diego
California, CA 92120
sbauza@ucsd.edu

Bryan Kerr
Electrical and Computer Engineering
University of California, San Diego
California, CA 92120
bkerr@eng.ucsd.edu

Edward Zamora
Electrical and Computer Engineering
University of California, San Diego
California, CA 92120
e5zamora@eng.ucsd.com

Abstract

This project explores the viability of a using Deep Learning to learn to play a collection of simple video games. In particular, Deep Q Learning is used to merge deep learning techniques with Reinforcement Learning. A network is created and used to play a number of video games using only RGB images as input.

This is of interest due to the difficulty of developing a control solution using images as input. In the interest of training time, all images are pre-processed to decrease the required network size.

1 Description of the Method

1.1 Framework Exploration

Before working on building various networks to play games, a number of different convolutional neural network (CNN) frameworks were examined. Keras and Tensorflow both have implementations in python. Tensorflow comes pre-installed on the target server, and was the most understood by the team, so the decision was made was to develop within the Tensorflow framework.

1.1.1 Keras

Keras is a high-level neural network application programming interface (API) written in Python. It can work as a wrapper around low-level libraries such as TensorFlow, as well as other popular frameworks. Keras allows developers to create prototypes very quickly without worrying about the more technical aspects of neural networks such as the mathematics and optimization methods. In this way the user wouldn't bother dealing with the backend in detail. Because of this Keras does not allow the developer to make low-level changes to their model. Since we may decide to customize our low-level layers we chose to stay away from Keras.

1.1.2 Caffe

In our exploration of deep learning frameworks we ran into Caffe(Convolutional Architecture for Fast Feature Embedding) [JSD+14] and a quick study was done to determine if it was suitable for our needs. Caffe is a deep learning framework, originally developed at UC Berkeley, which supports many different types of deep learning architectures geared towards image classification and segmentation. Our initial study compared the Caffe framework with that of the Tensorflow framework. First we found that Caffe was not a simple installation, it required to be built from source while Tensorflow

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.

was installed with a simple pip command. The second comparison we did was look at pre-trained models; we found that Caffe's model zoo provides access to a large amount of pre-trained models, though most were for image classification and not of much use to use. While Tensorflow did not have as many pre-trained models it did make it extremely simple to import pre-trained models, usually with a single line of code. The third criteria was GPU support and we found Tensorflow's GPU support was easy to use (all done through tf.device()) and had a very flexible architecture (multiple GPUs on a single model), while Caffe was more limiting. The last criteria we judged was layer-wise design. Caffe treats each node in the neural network as a layer whereas in Tensorflow each node is a tensor operation and a layer can be defined as a composition of those operations. Tensorflow was found to be more flexible since we were limited to pre-defined layers in Caffe.

Overall, Caffe was found to be better for image analysis (such as CNNs or RCNNs) and image classification tasks. Tensorflow was found to be easier to use, easier to interface with, and since it inherently uses smaller building blocks it offered more flexibility. The seamless interface, better documentation, and higher flexibility led us to our final decision of choosing Tensorflow over Caffe.

1.1.3 Tensorflow

Tensorflow is an open source machine learning framework that is capable of utilizing a system's GPU to accelerate learning. It allows for very strong control over the various parts of the network. This level of control comes at the cost of being quite a bit more difficult to use.

In general, layers are built one at a time until the entire architecture has been specified. Data can then be passed as input and the entire architecture processed in a single step. Additionally, learning rates, parameter types and structure are all fully specifiable. This enables the testing of various small tweaks to the network to study their effect and relative ability to learn.

1.2 Deep Q Learning

Reinforcement Learning (RL) is an area of Machine Learning in which a software agent makes observations and takes actions within an environment in order to receive rewards. RL can be used by machines to learn which actions to take in a given environment. This technique extends to many tasks, here, it is used to play Atari games. The basic RL structure can be modeled as a Markov Decision Process (MDP). A MDP is represented as a chain of states, and at each state an agent can transition to one of several other possible states. The agent chooses to go from state s to s' based on transition probabilities of the available actions. The goal of the RL agent is to collect as much reward as possible while transitioning through these states. The algorithm used by software for determining which action to take in a given state is referred to as its policy.

Deep Q-Learning is a method that has been used to apply deep learning to reinforcement learning challenges. Q-Learning is an approach to reinforcement learning in which the algorithm develops a Q-Table, specifying the expected total reward when taking a particular action. This table is filled for each state in the environment. For each possible state-action pair the algorithm keeps track of a running average of possible rewards upon leaving state s with action a. To optimize performance, the agent simply takes the action with the highest Q-Value estimate for its current state.

A major problem with Q-Learning is that it can only work if the exploration policy explores the Markov Decision Process thoroughly enough. As state and actions spaces become larger and larger, then the algorithm will need to keep track of a Q-Table that is way too large for any practical application. Exploring each of these states, all of their possible actions, and all state-actions that follow, becomes impossible to compute in a reasonable time-frame, even with exceptional computational hardware. To solve this problem, a policy must be found that approximates the Q-Table by exploring a manageable number of state/action pairs. Deep neural networks (DNN) have been shown to work very well for this task and using a DNN to estimate Q-Values to approximate the Q-Learning is what Deep Q-Learning is all about.

The estimated Q-Values should be as close as possible to the reward that is actually observed when transitioning from Q(s,a) to Q'(s,a), plus the discounted value of future rewards henceforth. Future rewards are estimated by running the Deep Q-Network (DQN) on the next state for all possible actions. After picking the highest value and discounting, the system obtains an estimate of the future discounted value. This way of approximating the target Q-Value in the Q-Table is referred to as the Bellman equation (see equation 1).

$$Q(s, a) = r + \lambda max_{a'} Q(s', a') \tag{1}$$

The algorithm can be trained using Stochastic Gradient Descent (SGD) by first defining a loss function. Performing SGD on the loss function with respect to the weights will direct the back-propagation to adjust the weights in a manner that will best minimize the loss. Generally, the loss function used here is the standard squared error loss between the estimated Q-Value and the target Q-Value (see equation 2).

$$L = \frac{1}{2}[(r + \lambda max_{a'} Q(s', a') - Q(s, a))^2] \tag{2}$$

1.3 Neural Networks

Images in particular are of interest due to the important of space within the image. While the state of a particular pixel may hold little to no value, groups of pixels are often much more informative to the actual state of the image. To capture spatial information made up by groups of adjacent pixels, convolutional layers are often added to the neural network, making them describable as Convolutional Neural Networks, or CNNs. Back-propagation on these layers actually updates the kernels that are used during each convolution. Over time, the layers tend to build up kernels that represent increasingly complex features. Lower levels may function as simple edge detectors, while higher levels can represent large structures like faces or vehicles.

Pooling layers often follow convolution layers to decrease the spatial size of the image. These layers use some metric to represent a block of pixels by just one decreasing the size of the output based on the kernel size of the pooling layer. The use of convolutional layers and pooling layers in conjunction, help to decrease the necessary network size, while maintaining spatial information about the original image.

Due to the nature of standard RGB images, using the image itself as an input to a standard Neural Network would require significant amounts of computation. A single 8-bit RGB pixel, can represent as many as 1.67 million different states (2^{8^3}). Many of these states will be extremely similar, off by only a fraction of a hue. Pre-processing the images can reduce the required network size, and in some cases can highlight important features.

The most common form of pre-processing used in gray-scaling, or taking the RGB pixels and making them scale from black to white. Also extremely common is decimating the data, or decreasing the image size. The specific pre-processing function used for each game is discussed in the results section. By gray-scaling and decimating by a factor of 2,the number of inputs is decreased by a factor of 12.

We have shown that we can approximate the Q-Values using a CNN and back-propagating the error. To force the target network to better represent the true target q value, previous states are stored and sampled from the reply memory at each training iteration. This will break the similarity of subsequent training samples and help us to avoid falling into a local minimum.

2 Implementation Details

This section will discuss the various components and tools used by our program. This section is not meant to explain the usage of the tool, as it will be described in the README.rst file.

2.1 Network Generator

As mentioned before, Tensorflow is used to implement the Convolutional Neural Network. We were initially interested in loading a variety of existing neural networks. Because many of these networks have caffe specifications, we implemented a .prototxt to Tensorflow network parser.

We later discovered, that the Caffe and Tensorflow Frameworks do not fully overlap in their capabilities. Because of this, we adapted the format to better fit Tensorflow layer generation. Though this means it will no longer be fully compatible with existing networks, it does give us an easy means to update and test different networks with various games.

Now, the network is assembled layer by layer in a .prototxt configuration file. Here, the layer type, parameters, and activation functions are all specified in this file. This allows us to build various architectures and interchange them using simple command-line arguments.

2.2 Training Framework

The learning process can be repeated indefinitely, but here is repeated only for a pre-set number of learning steps. The current state is input to the online network, and the output values are used to select an optimal action. There will always be some probability that a random action is taken. This helps ensure that the system does not get stuck repeating the same actions. This probability begins high, but decreases as the number of training iterations increases, meaning the system will initially tend to explore the space, while eventually beginning to trust its own estimates.

The environment's state is updated using the selected action, producing the new state, a reward for that action, and whether or not the current game has terminated. This set of information is loaded into memory and stored for later reference.

A batch of previous state are pulled from memory and passed through the target network, and the same values mentioned above are collected from each. The states and actions taken are passed into the loss function. The rewards are added to a weighted amount of the expected final reward for the optimal action. This is the y value passed into the loss function discussed in the next section.

The loss function is then calculated and used to update the online network parameters. Occasionally, the target network is updated to match the current online values. This allows the target network to improve in tandem with the online network, keeping its loss estimates accurate. At a regular interval, the weights of the network are saved, allowing the learning to be paused and the current performance examined.

2.3 OpenAI Gym

OpenAI Gym is a toolkit that provides various environments in which to develop and compare reinforcement learning algorithms. These environments range from simple physics simulators, such as inverted pendulum balancing, to old Atari-style games, to advanced robotic simulation environments. These environments provide information about the current state, and simulate the results of various actions upon the environment.

Our usage will largely focus on the subsets of gym environments under the Atari collection. These are emulations of old, recognizable games, such as MsPacman, Space Invaders, and Asteroids. These will be the environments in which our architecture gains experience.

3 Experimental Setting

3.1 Loss function

Due to time constraints and concerns about limiting our test space, we selected and implemented a single loss function. This loss function will be common to all environments and networks we test. This loss is minimized at each learning step, and used to power the back-propagation to update the online network values.

The error is the different between the optimal reward expected by the target network, and the reward from the online network when taking the selected action. The loss function used is the same used by [Age18]. The equations representing these are below and a graph of this function is represented in Figure 1

$loss = mean(error^2)$ if $error \in (0.0, 1.0)$

$loss = mean(2 * error) - 1$ if $error \in (1.0, \infty)$

There is no dataset for Deep Q Learning. Instead, the dataset is generated during processing. The agents acts upon a simulated environments and observes the effects within that environment. There concept of "epoch" is not used here, instead, training is split into episodes, or complete runs of the game.

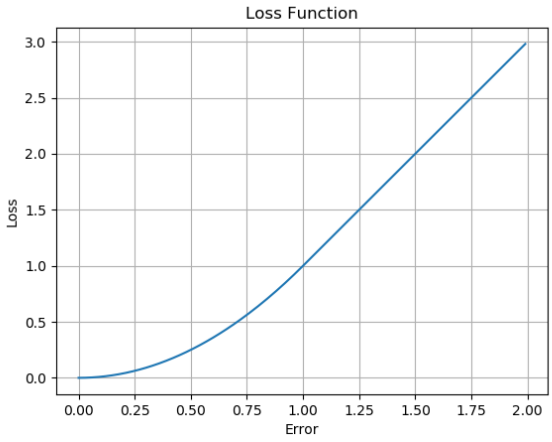
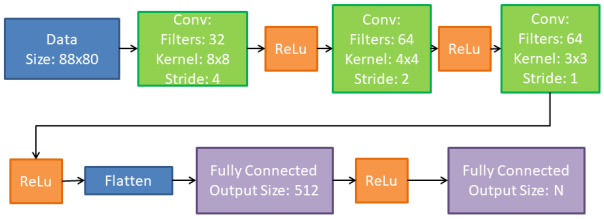
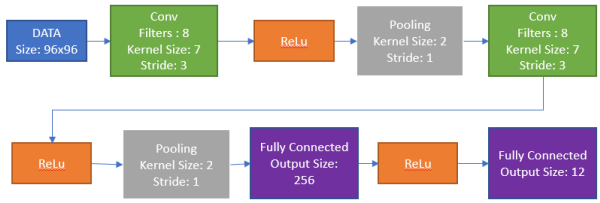


Figure 1: Loss Function

The training algorithm used is a heavily modified version of the one used by [Age18]. The online agent takes a single step in the environment. The target network then looks at a set number of previous states and produces an estimate of the target q-values. This estimate is directs the back-propagation of the online network, and the cycle continues.



(a) Diagram of PacNet Architecture, used by MsPacman, Asteroids, Snake



(b) Diagram of CarNet Architecture

Figure 2: Diagrams of architectures used for various games

4 Results

4.1 Snake

The Snake environment, shown in Figure 3 was the first game examined, and was implemented using custom code that closely matched the gym environment API. This allows for full control over the size of the board and rules of play. The reward function was chosen to be 0 if the snake simply moved, and 1 if the snake found food. The game ends when the snake collides with its own body. In order to facilitate fast network evaluation, and to increase the probability of the snake finding food (speeding up the game), a small game board of 25x25 was used, and required no pre-processing.

Unfortunately, even on small game boards, scoring occurs far too infrequently for learning. The game often requires pixel-accurate information, which is lost when using convolutional neural networks.

These problems ultimately lead to the total failure of the system to learn the game. The highest score achieved was a 6, which is unimpressive as the minimum score possible is 4. Figures 4a and 4b describe the score and q sum over time respectively. Though one might say the increased incidence of scores of 5 may be indicative of learning, this is actually due to epsilon decreasing. The initial exploration has a relatively high probability ($p = .33^3$) of causing a game loss, regardless of network input. In later stages, the network learns to simply move in mostly straight lines.

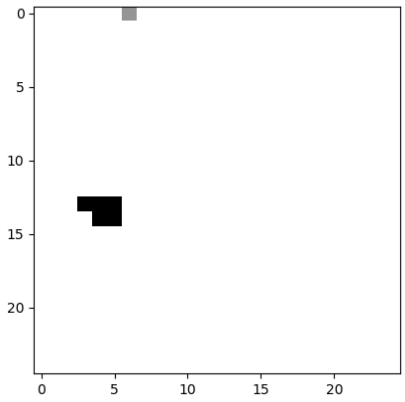
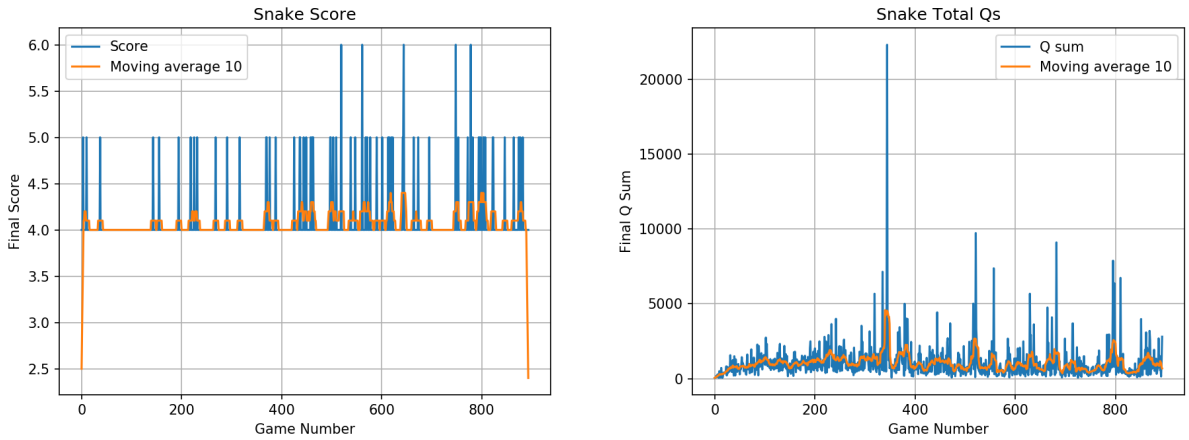


Figure 3: Snake State



(a) Snake final score over 4,000,000 timesteps

(b) Snake final Q sum over 4,000,000 timesteps

Figure 4: Snake Score and Q sum over time

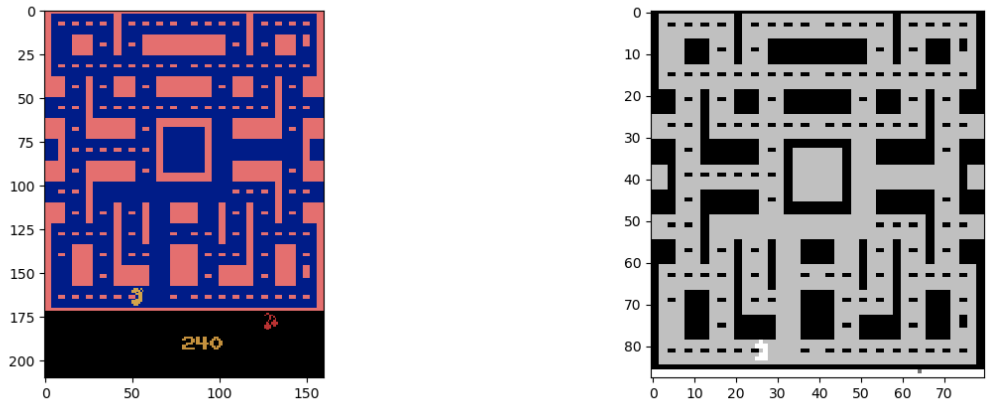
4.2 MsPacman

The goal of Ms. Pac-Man is to collect pellets across the map while avoiding ghosts. Each pellet provides points, as do collecting special items and eating ghosts when possible. The game ends when 3 lives have been lost.

In this environment there are nine discrete actions available. These correspond to the 4 cardinal directions, 4 diagonals, and the stationary position. The observations are screen-shots of the game represented as an array of 210x160 RGB pixels. The pre-processing function crops, decimates, and gray-scales the image into an 88x80 grayscale image. These are represented in Figures 5a and 5b respectively.

This game was learned by far the most successfully, and can consistently clear over half the board with less than a day of training. The most notable problem with this game is the frequent flickering of enemies. Because the system does not keep track of previous frames when testing, it will often direct itself into enemies that are invisible for a few frames. This often results in runs where over 1000 points are achieved in a single life, only for the agent to instantly kill itself until game over.

Figures 6a and 6b display an example growth of score over time when previous frames are not considered during training, while Figures 7a and 7b demonstrate the score when they are. Attempting

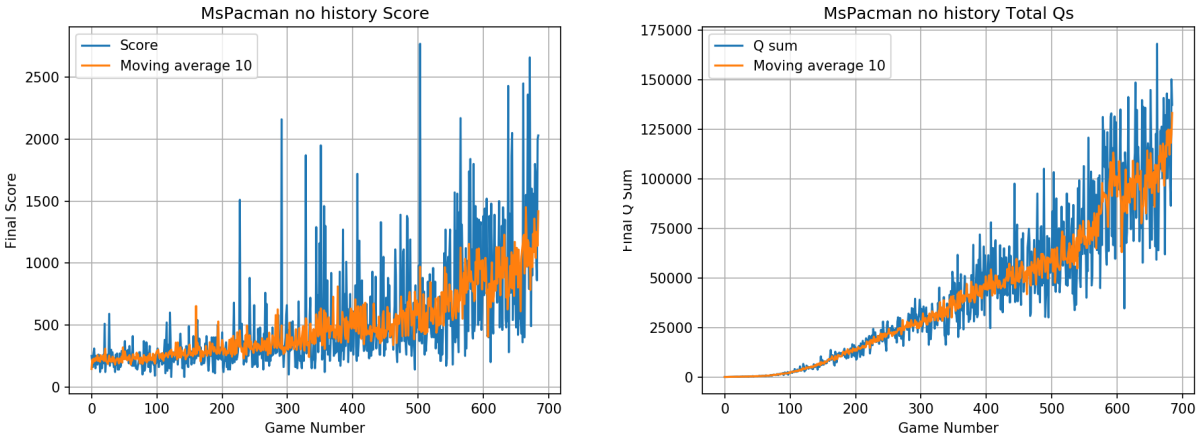


(a) MsPacman environment

(b) MsPacman Processed frame

Figure 5: MsPacman Environment before and after pre-processing

to include older frames using the method discussed above were largely unsuccessful. The agent initially appears to learn at an acceptable rate, however it later loses some of this performance, leveling out well below its highest average score rate. Even here, however, its score is improved over randomly exploring the space.



(a) MsPacman final score over 4,000,000 timesteps

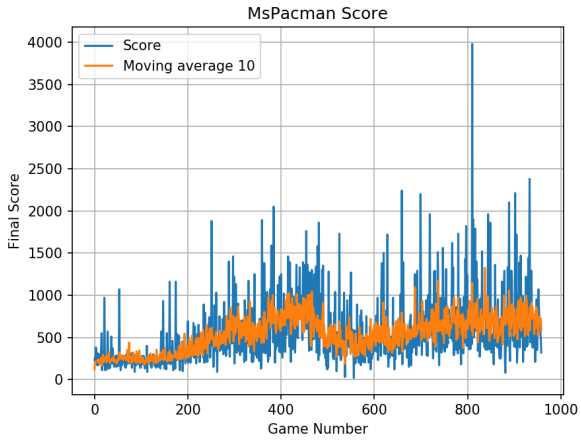
(b) MsPacman final Q sum over 4,000,000 timesteps

Figure 6: MsPacman Score and Q sum over time when only considering current frame

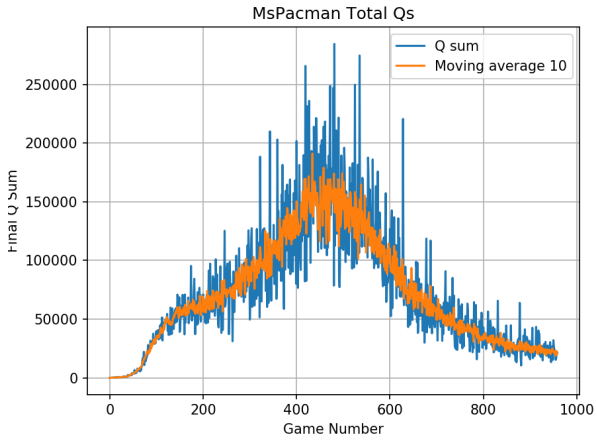
4.3 Car Racing

The third game we tried was CarRacing, based on the Box2D module from the Open AI gym environment. The observation states are 96x96 pixels with 3 channels for each pixel and 8-bit values for each color, representing a view from the top Figure 8a. Due to this top-down racing environment, CarRacing is said to be the easiest continuous control task to learn from pixels; this was the reason CarRacing was selected as one of our games. The default environment provides no penalty for going off the track other than the frame cost, if the car drives off the track it does not collect rewards for visiting track tiles but rather collects negative rewards for each frame.

In this environment there are 3 continuous values that the agent can choose: steering (direction of wheel) which the values range from -1 (left) to 1 (right), acceleration which it can choose a real value

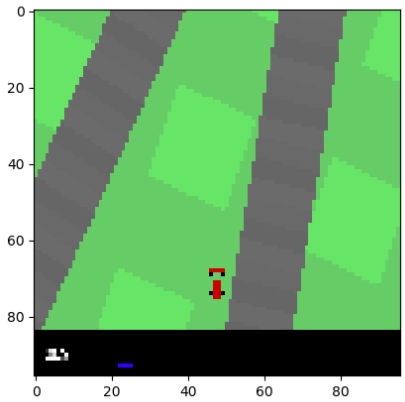


(a) MsPacman final score over 4,000,000 timesteps

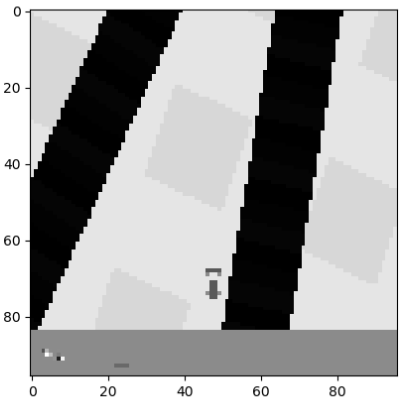


(b) MsPacman final Q sum over 4,000,000 timesteps

Figure 7: MsPacman Score and Q sum over time when previous frames are merged



(a) CarRacing environment



(b) CarRacing Processed frame

Figure 8: CarRacing Environment before and after pre-processing

from 0 to 1 and brake which it can also choose a value from 0 to 1. The reward -0.1 every frame and +1000/N for every track tile it visited, where N is the total number of tiles in a track. The episode finishes when all tiles are visited.

A few modifications and enhancements were made to facilitate optimize training; limitations, modifications, and assumptions are discussed below. First, we discretized the controller inputs to 12 distinct actions:[-1,+1,0.2], [-1,+1,0], [-1,0,0.2], [-1,0,0], [0,+1,0.2], [0,+1,0], [0,0,0.2], [0,0,0], [+1,+1,0.2], [+1,+1,0], [-1,0,0.2], [-1,0,0] (as mentioned before, these represent [steer, gas, brake]). The value of 0.2 for braking was chosen as to prevent the car from fully locking the tires while braking, this inherently provides greater stability. Second, the 3-channel RGB observation underwent preprocessing in order to convert it into a grayscale image (Figure (8b). This allowed the observation to be reduced into a single frame while not losing important information, as long as there is a distinction between track tiles and grass tiles the color does not necessarily have importance. Third, random actions gave preference to gas over brake; this reduced the chance that a random action would cause the car to not move under the assumption that this would lead to quicker training. Fourth, early stopping due to consecutive negative rewards (the max was set as a parameter) was added and would cause the episode to end. If The total reward was lower than a pre-defined threshold this early stopping was punished, if it was greater than no additional punishment was added. Lastly, our design was based off

of [MKS+13] but we modified the neural network in order to reduce processing intensity. We aimed for a network which would train in a reasonable amount of time while not requiring a GPU. Figure 2b shows the layer-by-layer description of the neural net.

Figure 9) shows the training score over 500,000 steps. We can see a growth start at about 40k steps and leveling out at about 180k steps. Checkpoints were saved every 250 episodes which allowed us to load in saved weights and replay the game those games. Loading in the checkpoint closest to 40k steps showed that at this point the agent began to understand the rules of the game, mainly that the goal was to stay on track tiles. It was beginning to attempt to stay on track though it was not very successful. The checkpoint at 80k steps showed the agent successfully staying on the track during straight sections and mild curve, though it still struggled with sharp turns. Checkpoints at 180k and 400k steps caused very similar performance to each other; the agent was able to reliably complete episodes and we witnessed that the controller was purposely cutting corners (either for turning stability or for minimizing time spent, we are unsure).

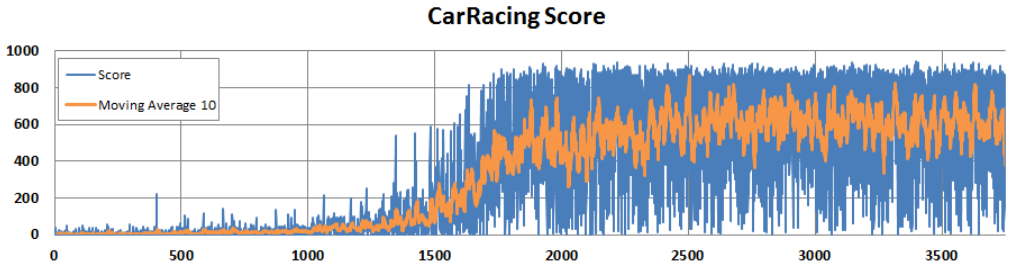
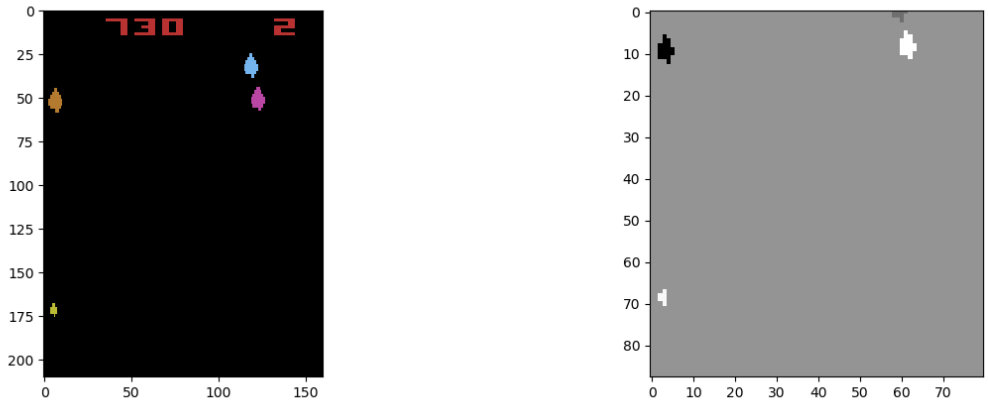


Figure 9: CarRacing final score over 500,000 timesteps

4.4 Asteroids



(a) Asteroids environment

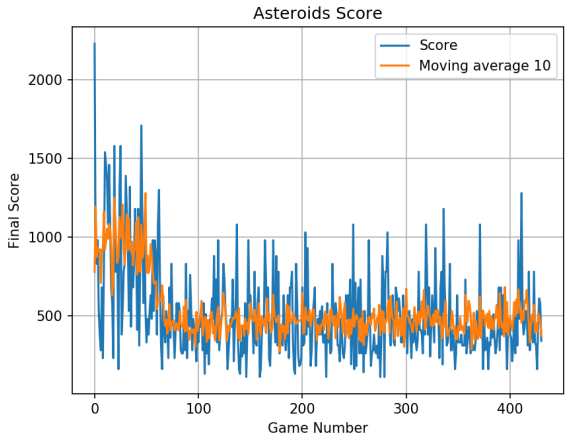
(b) Asteroids Processed frame

Figure 10: Asteroids Environment before and after pre-processing

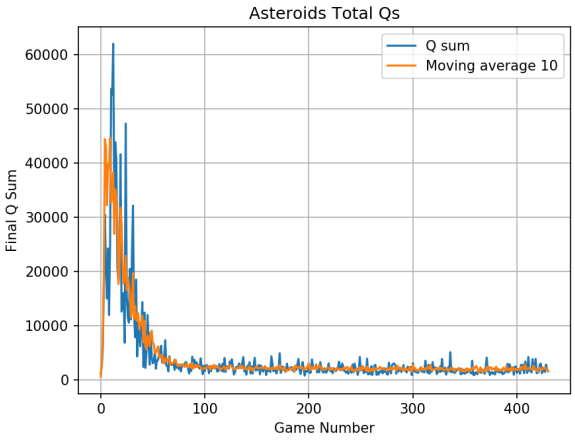
The goal of asteroids is to float about a frictionless environment in a space-ship, shooting asteroid while trying to avoid being crushed. Here, points are given for shooting an asteroid, and the game ends after 3 lives have been lost. Figures 10a and 10b represent the pre and post-processed images.

Flickering is very present in this game, making it difficult for a human to play, let alone a program making decisions one frame at a time. Many of the observed frames lack either the asteroids, or the ship, or even both, making it impossible to aim at or avoid collisions.

The score results for this game were quite unexpected, initially garnering an average of 1000 points, but eventually averaging only half of that. The change is quite sudden and would have been indicative of other issues, had the game not been played manually during testing. Figures 11a and 11b have the score and q sum values for the training session



(a) Asteroids final score over 4,000,000 timesteps



(b) Asteroids final Q sum over 4,000,000 timesteps

Figure 11: Asteroids Score and Q sum over time

5 Discussion

A notable problem that was present during some of the games was sprite flicker. In some games, enemies or even the character sprite would become invisible for a few frames at random intervals. Because the network makes decisions based on a single frame, this often led it to not realize an enemy or obstacle was present, causing it to blunder right into it. This problem is noticeable in MsPacman, though the nature of the game makes it less of a problem. In Asteroids, however, it makes the game nearly unplayable, even for a person.

The examined solution of averaging older frames together was unsuccessful, and did not work for a single game it was tested on. Another solution would be to concatenate images, though this would significantly increase the network's input size. This solution may also have issues as related pixels in different frames would be far apart, making it difficult for a CNN to relate them, though this may be solved by adding dense layers.

Finally, it may be reasonable to interpolate frames, placing adjacent pixels near to each other. This would make the image difficult to interpret by a person, but would perhaps maintain spatial information between frames when using strided convolutions.

References

[Age18] Ageron. handson-ml. `https://github.com/ageron/handson-ml/blob/master/16_reinforcement_learning.ipynb`, 2018.

[JSD+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. Technical report, 2013.