Thomas Tuttle

CS 496 Senior Seminar

Prof. Levenick

13 May 2019

Playing *Ms. Pac-Man* with Machine Learning

**Introduction**

The field of Machine Learning is one that has fascinated me ever since I first heard of it. At first, the concept of computers capable of learning, as humans do, to complete complex tasks seemed to me outlandish, more akin to science fiction than reality. Yet in recent times there have been numerous impressive applications of Machine Learning algorithms to several high-profile subjects, with impressive results. For example, the project that first made me aware of the potential of Machine Learning algorithms was the IBM Watson supercomputer, the subject of a massive publicity stunt in 2011. In a televised special episode of the quiz show *Jeopardy*, Watson competed against two of the most successful *Jeopardy* contestants of all time (Ken Jennings and Brad Rutter) as a demonstration of its ability to understand natural language ­— in this case, *Jeopardy* questions — and provide relevant answers. In the game, Watson destroyed its human opponents, earning $77,147 compared to $24,000 and $21,600 from Jennings and Rutter, respectively [1]. When I originally watched Watson play, I was amazed at its speed and accuracy for most of its answers, and entertained by some of its humorous mistakes. Up until that moment, I had never considered that a computer could “understand” written text, but Watson blew that expectation out of the water. As a result, I became very interested in the idea of machines that could perform cognitive tasks traditionally reserved for humans, especially games.

Another major and more recent inspiration for my thesis topic was the thesis work of Carlos Luevanos, Willamette University CLA Class of 2018. The title of Luevanos’s thesis, “Getting a Neural Network to Play Video Games: An Exploration in Reinforcement Learning”, immediately caught my eye, and the work contained within soon became a substantial basis for my project. Luevanos’s initial ideas and goals were very similar to my own; he had become interested in the field of training computers to play video games through exposures to such projects as Google’s AlphaGO and OpenAI’s DOTA 2 player, and was interested in creating a program capable of learning a simple game. To do this, he used multiple tools that I ultimately used in my project as well: the Python language, the TensorFlow machine learning library, and the OpenAI Gym reinforcement learning library. Using these tools, Luevanos created and trained a Deep Q-Network to play the classic Atari video game *Pong* [2]. Thus, my goal in my thesis became to adapt and extend the work done by Luevanos, and to create and train a Deep Q-Network of my own using reinforcement learning to play another classic and more complex Atari video game: *Ms. Pac-Man* for the Atari 2600. My immediate next task was to familiarize myself with the concepts and tools Luevanos used in his work.

One such concept, reinforcement learning, was heavily interwoven with one of the primary tools used by Luevanos, the OpenAI Gym library. OpenAI Gym is a library of reinforcement learning tools for use in Python, with a particular emphasis on built-in learning environments. These environments are some of the most attractive features of OpenAI Gym, especially those contained in the Atari dependency, a set of environments based on Atari 2600 video games. One such game is Ms. Pac-Man, which I chose as the game I wished my learning algorithm to learn. In addition to the Atari environments, there are various basic physics-based environments, useful for learning how to use the software, such as CartPole (which simulates a pole balanced on top of a moving cart) and MountainCar (which simulates a cart sliding back and forth in the valley between two slopes). Gym’s utility goes beyond merely providing environments, however; it also includes an easy-to-implement reinforcement learning framework, revolving around an “agent-environment loop” [3]. This loop is one of the core concepts of reinforcement learning. As explained by Kaelbling et al. in their survey of reinforcement learning: the program consists of an environment and an agent, which learns and interacts with the environment on a step-by-step basis. On each step, the agent receives the status of the environment as an input, and chooses an action to take based on that input. In turn, that action changes the state of the environment, and the changes are sent to the agent via a reinforcement signal. In addition, the agent has behaviors designed to increase the reinforcement signal over time, allowing the agent to “learn” how to best interact with the environment [4]. Gym’s implementation of this loop consists of its aforementioned environments, a step function that returns all the outcomes of an action (an observation of the new state of the environment, the reward generated by the previous action, a boolean signaling whether or not the environment should be reset, and a dictionary of debugging information) [3]. In addition, Gym breaks down the experiences of its agent into episodes. At the start of each episode, an initial state is randomly selected, and operated upon until an ending state is reached; then, the next episode begins. The point of using these episodes is to attempt to maximize the amount of learning within each episode, while achieving best performance in as few episodes as possible. Gym does not create an agent object, instead leaving that to the programmer; this is an explicit design decision, as the point of the Gym is primarily to create an environment to test custom algorithms (agents) on given environments [5]. Lastly, while OpenAI Gym and its Atari environments are primarily designed for Unix-based platforms, they can be made compatible with Windows by installing the “atari-py” library available on GitHub [6].

Now that I was interested in using OpenAI Gym, I needed to learn more about reinforcement learning, the method that OpenAI Gym is designed to implement. In my further investigations on the topic, I found Kaelbling et al.’s survey on the subject quite comprehensive and readable. In addition to the aforementioned agent-environment loop, the survey described two interchangeable reinforcement learning strategies: one that searches the behavior space to find the best action for a situation, and another that uses statistical techniques to estimate the effectiveness of taking actions in world states. While the former can be accomplished using genetic algorithms and is used in search techniques, the latter takes special advantage of the structure of reinforcement learning algorithms, making it of particular interest. As for which of these strategies I want to consider for my project, I am currently undecided, since I have not finalized the game that I wish to learn. Another important facet covered in the survey was the definition of optimal behavior, and what goals the agent would attempt to reach regarding its learning. The survey mentioned three models of such optimal behavior: the finite-horizon model, which aims to optimize the expected reward for the next h steps via stationary (h decreases to 0 over time) or non-stationary (h remains constant) policy, the infinite-horizon discounted model, which considers the long term reward but discounts the importance of future rewards, and the average reward model, which attempts to optimize the average reward in the long run, without distinguishing between early and late rewards. The survey went ahead to focus on other approaches, specifically those related to Markov Decision Processes [4] but I found the aforementioned items to be most relevant to my research.

Having investigated and found potential in both reinforcement learning and the OpenAI Gym library, I turned my attention to the other notable tool used by Luevanos: TensorFlow. TensorFlow is an open-source library designed for large-scale machine learning, especially training deep neural networks. One of the many interesting features of TensorFlow is that it represents a program as a “unified dataflow graph” of both the computation done by an algorithm and the state that the algorithm operates on. This graph is composed of nodes and edges: the nodes represent computations that update a mutable state, while the edges contain multi-dimensional arrays of data, aptly named tensors, which these edges carry from node to node. TensorFlow is designed with parallel processing in mind, increasing its performance and scalability: to quote its developers, Google Brain, “it efficiently uses hundreds of powerful (GPU-enabled) servers for fast training, and it runs trained models for inference in production on various platforms, ranging from large distributed clusters in a datacenter, down to running locally on mobile devices” [7]. This scalability caught my eye as a particularly useful feature, since I began with little idea how large/small the necessary computations would be for my neural network. In addition, TensorFlow is widely used: as of 2016, more than 150 teams at Google alone have used TensorFlow, and the project is now open-source, greatly increasing its userbase [7]. Thus the focus of my project became not only to design a reinforcement learning algorithm, but also to gain experience working in a widely used software library.

With TensorFlow specializing in deep neural networks, I investigated such networks further, with a specific focus on the “Deep-Q Network” that Luevanos used in his work. First, it is important to distinguish between Q-learning and Deep Q-learning (which refers to the aforementioned Deep-Q Network). Basic Q-learning does not incorporate a neural network, but does utilize the Q-function concept that was later adapted to Deep Q-learning. The central structure of Q-learning is the Q-table, a table in which the columns represent the possible actions that can be taken, the rows represent the possible states of the program in which that action could be taken, and each entry for a given row/column (state/action pair) is the maximum expected future reward, found using the Q-function. This Q-function is often weighted to reduce the rate of learning, since large swings can cause important maxima to be ignored. However, this system does not consider the tradeoff between exploration and exploitation: thus, an exploration rate, called epsilon, is specified as the rate at which the learner will choose a random action in a given state rather than the one with the highest value for that state in the Q-table. Naturally, this epsilon decreases over time as more of the table is filled and all the possible results are explored [8]. However, when considering a real-time game such as *Ms. Pac-Man*, where each frame is a potential state, this method results in the creation of a massive and inefficient Q-table, making regular Q-learning impractical. Thus, Deep Q-learning, a variant of this Q-learning that uses a neural network instead of a Q-table, is a far more effective method of learning in this scenario.

Deep Q-Learning is a variant of Q-learning that can be used for learning in environments with a large number of states, such as *Ms. Pac-Man*. Instead of being centered around a Q-table, Deep Q-learning is centered around creating a neural network that, given a state, can approximate the Q-value for an action. The network used varies dependent on the problem: if the state input is, for example, a frame from a video game, and the output action is which action the player should immediately take, a Deep Convolutional Neural Network is ideal [9]. This is quite relevant to my project, as Luevanos used images of the game screen as input in his project, and the Atari Learning Environment represents the game environment by using such an image of the game screen [2]. Convolutional networks are particularly effective at object recognition, which is essential for recognizing a game state from an image of the game screen. When a convolutional neural network is given an image as input, it separates the image into several component “tiles”, feeds each “tile” into a small neural network (each tile is processed by a network with identical weights), unusual patterns will be catalogued, and the result of each “tile’s” network is sent to a two-dimensional array proportional to the original image. This array is then “downsampled” by creating a smaller array of the largest values in the previous array. For example, the array could be broken into two by two squares, with the largest value in each square being put in the square’s respective location in the output array. Finally, this small array is fed into a final neural network, which produces the final output. This process comprises a single convolution layer, and a deep convolutional network may have several [10]. Another important choice when implementing a convolutional or any neural network is choosing an activation function, which determines whether or not a neuron has been “activated” by an input. A simple example of an activation function is to simply sum the values of all the inputs of the neuron and compare them to a threshold, activating only when the threshold is met or exceeded. One such activation function is ELU, or “Exponential Linear Units”. The equation for ELU is as follows:

In this equation, *α* is a hyperparameter, a value set by the user before learning, to control the behavior of the ELU. Furthermore, it is assumed that *α* > 0. This function is of particular note because it is particularly effective when used as an activation function in convolutional and deep convolutional networks [11]. In general, this Deep Q Convolutional Neural Network has been used to learn multiple games, such as *Space Invaders* [9] and, most relevant to my paper, *Ms. Pac-Man* [12].

After exploring and learning about the tools that Luevanos used in his thesis, I explored other approaches to the topic of learning games with reinforcement learning. In this, I discovered a paper by Mnih et al. that was also very similar to Luevanos and my planned project: they had also used Deep Q-Learning to learn Atari games via the ALE emulator in OpenAI Gym. An interesting feature of this project was the concept of experience replay: each episode, the agent’s experience at each time step was stored in a dataset. This dataset was termed the replay memory. In the middle of the loop of the algorithm, random samples would be drawn from the replay memory, and Q-learning would be performed on those samples to determine their reward. Thus, the algorithm was able to compare its performance at that step with decisions from past episodes [13]. Their algorithm, titled “Deep Q-Learning with Experience Replay”, is described below:

* Initialize replay memory to capacity
* Initialize action-value function with random weights
* **for** episode = **do**
  + Initialize sequence and preprocessed sequenced
  + **for**  **do**
    - With probability select a random action
    - Otherwise select
    - Execute action in emulator and observe reward and image
    - Set and preprocess
    - Store transition in
    - Sample random minibatch of transitions from
    - Set
    - Perform a gradient descent step on according to gradient equation
  + **end for**
* **end for** [13]

While Mnih et al. did not apply this algorithm to *Ms. Pac-Man*, they did apply it to seven other Atari emulators, all from ALE: *Beam Rider*, *Breakout*, *Enduro*, *Pong*, *Q\*Bert*, *Seaquest*, and *Space Invaders*, with the same network architecture, learning algorithm and hyperparameters across all seven. Their results were mixed, but nonetheless impressive: their method had superhuman performance in *Breakout*, *Enduro*, and *Pong*, and almost human performance on *Beam Rider*, but subhuman performance on *Q\*bert*, *Seaquest*,and *Space Invaders*. The fact they were able to apply this algorithm with identical settings across several games with great results from many speaks to the robustness of their algorithm, and its potential when applied to *Ms. Pac-Man*. However, the games that their approach faltered on likely failed due to their networks inability to find a long-time-scale strategy, a weakness that could be exposed when dealing with *Ms. Pac-Man* and its long-lasting levels [13]. That said, their approach is one I must consider when testing different networks on the problem of learning *Ms. Pac-Man*.

A very notable achievement involving machine learning applied to *Ms. Pac-Man* was another interesting study that I first learned about from Luevanos’s paper and decided to investigate further. A team of researchers from Microsoft developed a method named Hybrid Reward Architecture that was able to learn to play *Ms. Pac-Man* with superhuman performance. Their method decomposes the reward function for their reinforcement learner into a number of different reward functions, each assigned a different agent. All these agents learn in parallel using “off-policy” learning. Each agent gives the action-value function result of its current state to a special aggregator, which combines them into a single value for each action; this value is used to select the next action. Their method is based on the existing Horde architecture, which also utilized off-policy parallel learning [14]. The success of this method when it was applied to my target game, *Ms. Pac-Man*, makes it very relevant to my research, and I may choose to attempt to replicate their findings in my thesis project.

While I have yet to decide on the various details of implementing my project, it seems clear to me that creating a reinforcement learning algorithm to play *Ms. Pac-Man* is within my grasp. In the research portion of my project, I’ve discovered several useful tools that will be greatly beneficial to my project, handling low-level details so that I can focus on high-level concepts. In particular, using TensorFlow will not only provide a method to implement neural networks in my learning algorithm, it will also allow me to gain experience using a widely-used software library specializing in one of the hottest subjects in modern computer science. In addition, OpenAI Gym will allow me to take advantage of already-made emulators for *Ms. Pac-Man*, other Atari games, and simple physics simulations, in a framework specialized for reinforcement learning. Finally, the articles I’ve found of researchers using similar tools to my own with similar purposes is encouraging regarding the possibility of my goals. I am very optimistic about the potential outcome of this project, and I am excited to learn how to implement the sorts of programs I have been reading about.

**Conclusion**

This project was an extremely educational, if occasionally frustrating, experience. I spent the vast majority of my time simply learning the basics of the various tools I was interested in using, as well as understanding the theory behind using them. Learning Tensorflow, even when using the very human-readable Keras API, was difficult due to both needing to learn all the necessary formatting terminology to prepare proper inputs, outputs, and models, but also due to needing to learn enough about the underlying theory of neural networks to determine which parameters were significant and needed to be specified. I also spent a lot of time struggling with the process of implementing a Deep-Q Learning scheme using Tensorflow and Python, a struggle that was only surpassed by careful rereading of Mnih’s paper on the subject [13] and many additional online tutorials, some of which were very helpful [15] [16]. Ultimately, through this project I gained invaluable experience using Tensorflow’s Keras API, the OpenAI Gym, and the Python language, alongside the basics of reinforcement learning, neural networks, and the Deep-Q Learning algorithm. In Keras, I learned how to construct neural networks, with a focus on convolutional networks, and the importance of selecting the correct parameters and layers when creating such networks. While OpenAI Gym provided a useful tool in that it supplied the *Ms. Pac-Man* emulator in an environment conducive to testing, it also provided a challenging learning experience in coding with limitations – specifically, the limitation that the emulator would only provide images in a specific format as output, alongside the occasional notification that a game had ended. This required me to design a preprocessing function in order to more efficiently work with the high-dimensional images provided by the environment, and it guided my work in the direction of generalized, modeless learning using the Deep-Q Network on image inputs. In the Python language, in which I already had some familiarity, I learned to use several new packages that proved essential to my work on this project, namely pickle (for preserving score and memory data for use in future learning) and numpy arrays, which I used extensively in my interactions with Keras. Ultimately, all this learning culminated in a Deep-Q Learning algorithm which is capable of learning to play *Ms. Pac-Man* based on past play, although its learning is slow and its play still very flawed after hundreds of games. In the end, though, I feel that the effort I put into this project was substantial, and that while its results did not meet my initially lofty aspirations, the results were substantial as well.

Were I to redo this project with the knowledge I have now gained, I likely would have chosen a game more suitable to this form of learning. *Ms. Pac-Man*, despite being a very old video game, is very complex in terms of the features that a modeless algorithm must “discover” from image data in order to succeed. Unlike other Atari games like *Pong* or *Breakout*, which feature one-dimensional player movement and a single, very important rule to prevent failure (to stop the ball from passing the player), *Ms. Pac-Man* contains two-dimensional movement in 8 directions, in a maze filled with small yet significant figures (pellets) and many enemies with seemingly-random movements (ghosts), made worse by the technical limitations of the *Atari 2600* system that caused the ghosts to “flicker” and disappear often during gameplay. With more time on a simpler game, I could have created a more robust model that could surpass my original goal of decent-human performance, which my agent was unfortunately unable to meet. Also, with this additional time I would also consider building my own *Ms. Pac-Man* simulation, instead of relying on the one provided by OpenAI Gym. Doing so would have given me more access to the inner meaning of the game’s logic, allowing me to learn based on features such as player and ghost position and distance rather than the images provided by a “black-box” program like OpenAI Gym. Clearly this would create additional challenges (most significantly the coding of the ghost AI), but it would afford me many more tools to tinker with in my learning process.

With all the difficulties I had working on this project, there were many problems I had to solve. A persistent issue was that of computational performance: during early runs with both the *Ms. Pac-Man* emulator and the Deep-Q Network running, the emulator would run at a snail’s pace, or not at all, rendering it impossible for me to generate results. However, using Python’s time library to measure runtimes and recursively printing the runtime of code snippets of decreasing size, I was able to discover the culprit: the 32 model updates based on image input happening every timestep. By reducing the number of model updates down to 1, the program’s speed was dramatically increased to the point that it was playing faster than human speed. (I ultimately increased the number of model updates to 8 by my final tests, which caused a performance hit but still ran at a reasonable pace.) Another difficulty I had was working effectively with preexisting libraries and their idiosyncrasies, particularly Keras. One common issue was the dimensions of model inputs. Keras expects each image input to a convolutional layer to have a specific format that its error messages fail to specify, in which a “batch dimension” representing the number of images being input must precede the actual image’s dimensions. I initially had issues with this when trying to input multiple images into the model at once, without meeting these requirements; however, I was able to determine the problem and solve the bug by reducing the scope of the issue and attempting to input a single image (which I was able to do by adding single dimensions on each side of the two-dimensional image to create a four-dimensional model input).

Given more time, there are various avenues I would take to extend this project. The simplest and most obvious would be to continue testing variations of my training model, in order to find optimal parameters for playing *Ms. Pac-Man*. Additionally, I would be able to run the most successful variations for longer, hopefully increasing the skill of the agent as more training occurs. Another method I would attempt to implement would be a homemade simulation of *Ms. Pac-Man*, not reliant on OpenAI Gym. With such an emulator I could gain direct access to values and features in the game state, such as ghost locations and death timings of individual lives, as well as the ability to alter the game scenario itself to attempt to train on a simpler version—for instance, one with no ghosts, where the player attempts to collect as many pellets as possible in a fixed time period. Such a scenario would remove the random elements that complicate training (ghost movements), and would provide insight into the effectiveness of Deep-Q Learning for agent maze navigation, an important subproblem of learning *Ms. Pac-Man*. Finally, in Mnih’s research [13], he utilized a list of the four most recent (preprocessed) frames as inputs for his model, to give the agent a sense of the temporal progression of the game; however, my agent only supports a single input frame each timestep. Given more time, I would like to support Mnih’s method in a scalable fashion, allowing users to specify how many past frames they would like to use as prediction input each timestep. These additions would both add more depth to the results of this project and provide a new direction for future testing.

To any students beginning major projects in the years to come, particularly those involving machine learning, I would greatly advise patience. Over the course of this project I frequently found myself frustrated because I would become too impatient, write large portions of complex code without testing its individual segments in the process, and then be disappointed at the lack of instant results. A semester-long project is a far larger undertaking than many students have attempted in their undergraduate careers, so the scope can seem daunting, especially when a rushed first attempt doesn’t pan out. Patience, then, allows you to respect both the project and the time. The best thing about a multiple-month long project is that you have so much time at the start to work slowly. Note that this is different from working infrequently: you should still be working the same amount on the project from start to finish, but the first month or two is a valuable opportunity to slowly but deliberately build a stable groundwork for your project. By the time the project is about to be completed, that solid foundation will have paid off in its ability to minimize the odds of a catastrophic error or program failure. This is the best piece of advice I can give an inexperienced student working on a long-term project, as it will save incalculable hours of debugging and frustration which I myself experienced by occasionally neglecting this ideal.

**User Guide to Project**

This project was created using Python 3.6 in the PyCharm IDE, available from https://www.jetbrains.com/pycharm/ in the form of a free Community Edition or a paid Professional Edition. I specifically used the Professional Edition under a free student license, although projects in this edition should still be compatible with the Community Edition or any other Python IDE. I’ve made my code available in the following form: a .zip file containing the project code with saved models, memories, and scores from previous runs in their respective folders, which can be run using any Python compiler with the correct packages installed. This file is available on the Willamette University Academic Commons, alongside this paper.

Before attempting to run the project code, the following information is very important. This code is designed to work with Python 3.6, and support on other versions is not guaranteed. Furthermore, the following packages must be installed using pip or another Python package manager: *gym*, *tensorflow*, *matplotlib*, and *numpy* (if not already installed by a previous package). The *tensorflow* package may be replaced by the *tensorflow-gpu* package, which utilizes a computer’s CUDA-enabled GPU instead of CPU for processing; this is the package included in the Virtual Environment, and as such that code will not function without a properly set-up CUDA-enabled GPU. For more information on setting up the *tensorflow-gpu* package, visit the official Tensorflow documentation at https://www.tensorflow.org/install/gpu. To install the *Ms. Pac-Man* emulator used by *gym* on Mac OSX or Linux, simply run the following pip command in the Terminal wherever pip is located:

pip install gym[atari]

To install the *Ms. Pac-Man* emulator used by *gym* on Windows, locate the Python and pip installation locations using the command line and run the following commands:

pip install git+https://github.com/Kojoley/atari-py.git

pip install gym[atari]

To make sure all packages are installed correctly, run *test.py* in the project folder; ideally, it should play a short game of *Ms. Pac-Man* and generate no errors.

This project contains 5 code files, 3 of which are to be run at a given time in the course of testing. The aforementioned *test.py* serves no purpose outside of making sure all necessary packages are installed correctly. The file *deep\_q\_agent.py* contains the code that describes a Machine Learning agent (named DeepQAgent) for learning *Ms. Pac-Man*, and does not need to be run directly or edited during testing, although users interested in the structure of the project should investigate it as its code is essential the project’s function. The most important feature for any users in this code is the agent’s constructor (found in the function “\_\_init\_\_”), since it shows the parameters used by the model and their default values when not specified. In addition to the code files, there are three folders – “models”, “memories”, and “scores” – that contain saved models, memories, and scores (respectively) from previous runs. These folders also act as the destination for any future saved models, memories, and scores generated during runtime. The models, memories, and scores (respectively) currently included come from my most successful runs. These models were all created using the default parameters, save their names and the number of trials run. They can be found in the following files:

* pacman\_dq3\_norm.h5, None [no memory file was generated for this run], pickled\_scores\_norm.m
* pacman\_dq3\_norm\_cont.h5, pickled\_memories\_norm\_cont.m, pickled\_scores\_norm\_cont.m
* pacman\_dq3\_norm\_cont\_2.h5, pickled\_memories\_norm\_cont\_2.m, pickled\_scores\_norm\_cont\_2.m

There are three runnable Python files in this project with main functions. In order to utilize them properly, the user must access them using an IDE or other code editor in order to specify parameters such as input and output model names. The simplest of these is *show\_pacman\_scores.py*, which displays a chart of scores over time (in terms of episodes) contained in a specified .m file, which is selected by entering its name as the value of the “filename” variable. The scores displayed are the scores as used by the learner, not the rewards generated by the *Ms. Pac-Man* emulator; thus, they are normalized. The next runnable file, *pacman\_model\_testing.py*, is used to run a saved model without implementing learning. Note that while this file does use a DeepQAgent, it does not fully use its functions and thus many of its parameters do not affect performance. (In particular, output files are not generated or saved.) A model in the “models” folder can be run using this file by specifying its file name (without path) in the “in\_model” argument of the DeepQAgent constructor near the top of the file. Other important parameters to include in the DeepQAgent constructor include “eps\_min” and “eps\_start” (which both must be set to 0 to test a model without random movements, or 1 to test a fully random agent), and “trials”, which will determine the number of episodes run. This file, when run, displays the emulator in real-time, and ultimately outputs the average score (as viewed by the emulator, not by the model, i.e. not normalized) over all episodes specified. Finally, the last runnable file, *pacman\_deep\_q\_main.py*, is used to train a DeepQAgent. All specifications of the training process and agent trained can be customized using the DeepQAgent constructor near the top of the file. The most important of these are the input and output models, scores, and memories. When creating an entirely new model without building off an existing file, the values for “in\_model”, “in\_scores”, and “in\_mems” should all be set to None. Additionally, None should be used for one these values in any scenario in which it is missing; for instance, when loading a model without memory detail, “in\_mems” should be set to None. It is recommended that the output file names (specified in “out\_model”, “out\_scores”, and “out\_mems”) should be original, as of yet unused file names, to avoid overwriting existing files. As in all other runnable files, the input and output fields require only the filename with suffix, and not any path; they will automatically search or be placed in the “models”, “scores”, or “memories” folders. There are many customizable numerical arguments that can be specified in the constructor, all with default values when left unspecified. These are all numerical except for the aforementioned filenames and the “model” field, which takes one of two strings in order to choose a specific convolutional neural network architecture: “mnih”, which consists of two convolutional layers and a fully connected layer with 256 nodes, and “dq3”, which consists of three convolutional layers and a fully connected layer with 512 nodes. If left unspecified, “dq3” is selected by default. When run, this file will train either a new or already-specified DeepQAgent for a specified number of trials (500 by default), periodically saving its model, score, and memories with every target model update. While running, the program will also print which trial is currently being run, and will print the value of epsilon whenever the target model is updated. This file takes a very long time to run: in my testing, training a model for 500 trials would require a model to run for an entire night, up to 12 hours, using GPU-based Tensorflow with a CUDA-enabled Nvidia Geforce GTX 1050 GPU. This runtime could change based on the computational power of the computer running the program, and the version of Tensorflow used (CPU or GPU).

**Works Cited**

1. Markoff, J. (2011, February 16). Computer Wins on ‘Jeopardy!’: Trivial, It’s Not. *The New York Times.* P. A1. Retrieved from https://www.nytimes.com
2. Luevanos, C.E. (2018). *Getting a Neural Network to Play Video Games: an Exploration in Reinforcement Learning.* Retrieved from Willamette University Academic Commons.
3. *Getting Started with Gym.* Retrieved from https://gym.openai.com/docs
4. Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4, 237-285.
5. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). Openai gym. *arXiv preprint arXiv:1606.01540*.
6. Kniazev, N. (2017). Atari-py. Github Repository, https://github.com/Kojoley/atari-py
7. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ... & Kudlur, M. (2016, November). Tensorflow: a system for large-scale machine learning. In *OSDI* (Vol. 16, pp. 265-283).
8. Simonini, T. (2018, April 10). Diving deeper into Reinforcement Learning with Q-Learning. *freeCodeCamp*. Retrieved from https://medium.freecodecamp.org
9. Simonini, T. (2018, April 11). An introduction to Deep Q-Learning: let’s play Doom. *freeCodeCamp*. Retrieved from https://medium.freecodecamp.org
10. Geitgey, A. (2016, June 13). Machine Learning is Fun! Part 3: Deep Learning and Convolutional Neural Networks. *Medium*. Retrieved from https://medium.com
11. Clevert, D. A., Unterthiner, T., & Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (elus). arXiv preprint arXiv:1511.07289.
12. Van Hasselt, H., Guez, A., & Silver, D. (2016, February). Deep Reinforcement Learning with Double Q-Learning. In *AAAI* (Vol. 2, p. 5).
13. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602.*
14. Van Seijen, H., Fatemi, M., Romoff, J., Laroche, R., Barnes, T., & Tsang, J. (2017). Hybrid reward architecture for reinforcement learning. In *Advances in Neural Information Processing Systems* (pp. 5392-5402).
15. Grigsby, J. (2018, June 29). Advanced DQNs: Playing Pac-man with Deep Reinforcement Learning. *Towards Data Science*. Retrieved from https://towardsdatascience.com
16. Hinzman, L. (2019, February 15). Beating Video Games with Deep-Q-Networks. *Towards Data Science*. Retrieved from https://towardsdatascience.com