Playing Pong with Deep Reinforcement Learning

Bobby Dorward, Ryan Wilson, and Eric Bell

December 16, 2016

1 Introduction and Background

The history between artificial intelligence and gaming is long standing because short term choices give rise to long term sophistication. For example, the moves possible in the game of Go are few, yet it took one of the most sophisticated machine learning algorithms to date to be able to perform sufficiently at this problem. AlphaGo is but one of the few applications of deep neural networks that have. We aim to revisit this relationship between machine learning and gaming in order to gain facility with how machine learning, specifically deep neural networks, are used.

We have trained our neural network through reinforcement learning, specifically through an implementation of deep Q-learning [1]. In Q-learning, we try to approximate the function $Q(s_i, a)$, which represents the expected value of taking action a on state s_i. At optimality, Q^* should obey the Bellman equation

$$Q^*(s_i, a) = r_i + \gamma \max_{a'} Q^*(s_{i+1}, a'),$$

where r_i is the immediate reward from performing action a on state s_i and γ is a discount factor. We can use the Bellman equation to iteratively try to approximate Q. More specifically, when training the neural network, we use the Bellman equation to create our loss function. The pseudocode can be found in Algorithm 1.

We implement a technique called experience replay, which means that we save all the data needed to train the neural network and then randomly sample from these memories to train the network. This gives the advantage that we training on memories that are not correlated and can hopefully get out of local mins. In addition, we can train multiple times on each memory and so we get more use out of each observed result.

Our algorithm interacts with the game through the OpenAI Gym [2], an environment that allows the programmer to interact with Atari games (as well as a few other simulations) such that a machine learning algorithm can be trained to play the game ("solve" the problem). What this entails is that the Gym allows us to extract pixel data from the game, signal actions for the AI to take on a frame-by-frame basis, and collect feedback used to train the algorithm. Our original goal when starting this project was to apply this network to the "DoomBasic-v0" [3] but due to limitations, we decided to start with the simpler "Pong-v0" [4] environment. This environment runs a typical game of pong: a ball bounces between two paddles and one player scores when the ball hits the opposing side of the screen. The two players continue until someone scores 21 points, after which the game ends.

Predict optimal action a by feeding difference frame x_i into neural network
Take action a, observe reward r and set $m = \max_{a'} Q(x_{i+1}, a)$
Save (x_i, a, r, m) as a memory for experience replay
Randomly sample memories and use these to train the neural network on the loss function $(r + \gamma m - Q(x_i, a))^2$.

Algorithm 1: The deep Q-learning algorithm

2 Data

As previously mentioned, the data we used as input into our deep network was the pixel data of the environment. Each pixel is represented as an array of three elements, representing the three channels of an RGB pixel, and these pixels are passed into the neural network as inputs. Neural networks by nature are memory-intensive data structures due to the large amount of nodes and connections they contain. This is especially true when one uses a more complex version of a neural network, such as the deep neural networks we have implemented. Therefore, despite our best efforts to make our implementation generic and applicable to many different environments, we had to make pong-specific alterations to our data to streamline the size of the network and make it runnable on the resources we had available to us.

The first change we made was to grayscale the pixel values, thereby cutting the data we need to put into our network by a factor of three. Essentially what this means is that we have averaged the three values of the RGB channel into one value for each pixel. Next, we cropped the data being given to us by the OpenAI Gym in order to exclude all the pixels on the screen that weren't directly relevant to the frame-by-frame progression of the game. The scoreboard, the white edges, and the padding behind each paddle was cropped to minimize the input count of the network. Through this preprocessing, we were able to reduce the amount of inputs being taken for each frame to 20,800.

Finally, since the game of pong does not consist of still images, but rather gains meaning through the motion of the game objects (two paddles and the ball), we needed to capture this motion using the pixel data. We achieve this by extrapolating a difference frame (suggested in [8]), shown in Figure 1, between each subsequent pair of images, in which we subtract the first image from the second image. Through this, we can train the algorithm with data that represents where the game objects have been and where they're going, zeroing out any unmoving (and thus irrelevant) pixels in the process. It is because of these difference frames that we have limited the action that our AI can take; instead of the 5 actions that are available by default to the algorithm, we've limited it to only moving left and right. This way, the algorithm is always able to see its own paddle because it's constantly in motion and therefore creating data on the difference frame.

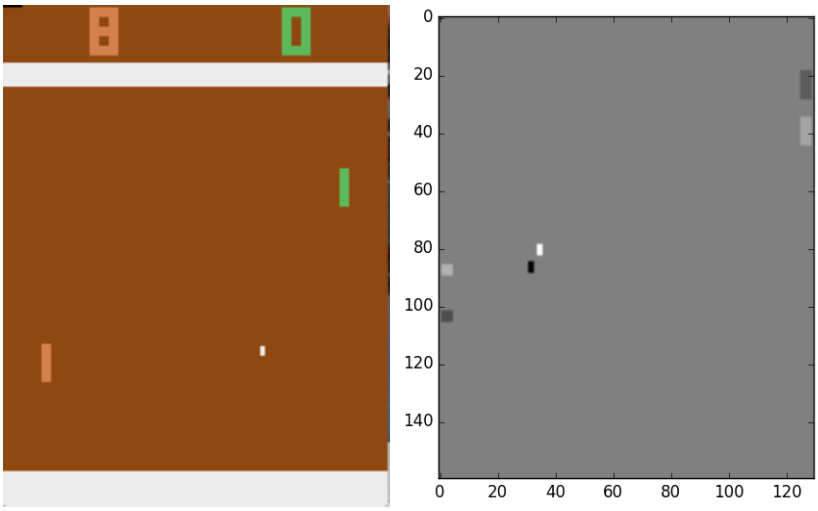

Figure 1: Raw data (left) and preprocessed difference frame (right) used as input into the neural network

3 Implementation

Our deep network is implemented in a python 3 program called "deep_pong.py", which is dependent on the OpenAI Gym module, numpy, and Tensorflow [5], a package developed by Google that constructs and trains neural networks. We used the code in [9, 10] as a jumping-off point to try to understand Q-learning, though we made extensive changes to the code. The program starts by defining the topology and hyperparameters used to train the neural network. The learning rate starts at 1e-7 and decays to 1e-8 by multiplying the current rate by .999 every training session. We then define epsilon as 1, a probability that a random action will be taken, which ensures that the AI will receive positive reward simply by pure chance, and define epsilon's decay as .999992, which is multiplied by the current epsilon during every training session. The next value is the discount multiplier for expected future reward (γ in Q-learning). We introduce two discounted factors to take advantage of previous knowledge about pong. We know that we will only get a nonzero reward after one of the sides scores. Therefore we decided not to immediately store memories after observing them, but to wait until someone had scored, at which point we could use one of the discount factors to ensure that we were always training on nonzero rewards. The other discount factor is used for the actual Q-learning algorithm.

We then define the maximum episode count (20000, a count we'd have to run for over a week to reach), and the batch size, which corresponds to the amount of instances being sampled to train the network during each session of training. After establishing a mechanism for saving the neural network and defining a few functions that will help us construct the model and preprocess the data, we then construct the neural network using Tensorflow.

Our deep neural network consists of 20800 input nodes feeding into two fully connected hidden layers, the first layer having 100 nodes and the second layer having 50 nodes, which then feed into two output nodes corresponding to the two actions (right and left). The

weights are initialized corresponding to a normal distribution with a standard deviation of 0.1, and the bias is initialized to 0.3. We define our loss function as minimization of the sum of squares difference between the Q value given by the neural network and the Q value predicted by the Bellman equation. We then multiply this by a one-hot vector of the action taken to ensure weights are only altered for the action taken. We use stochastic gradient descent as the optimizer. Since the model for the neural network is now established, we can then initialize the model and start the OpenAI Gym environment.

The first thing we do when running the Tensorflow session is try to load a previously made model and restore the weights if possible (code found in [11]). After that, we create the good and bad memory lists, where the memories collected in a point will be stored depending on whether they lead to a positive or negative reward. This was done because the original Q-learning algorithm had the model training on almost exclusively negative rewards, which caused the weights of the network to diverge to negative infinity. On the start of a new game, a first random action is chosen to allow the creation of a first difference frame, the first frame is preprocessed and the memory lists for the current game are created. For each frame following, an action is chosen either based on randomness if a random float is higher than epsilon, or otherwise is based on what action leads to the highest Q. The state derived from taking that action is then saved so that future expected reward can be calculated later. If the frame resulted in the scoring of a point or the game being over, then the memories collected during the run are assigned target Q values based on the outcome of the point and stored in either the good or bad memories lists based on the sign of the reward. Old memories are overwritten if the list size exceeds the memory size defined previously (40000 in our case). Finally, after all the memories have been stored, we can begin training. In order to train, the memories (instances) are sampled at random from the list of good and bad memories according to the specified batch size. Two times the amount of memory that just happened (good if we got a point, bad if they got a point) is passed into the network, which then results in the adjustments of the network's weights through minimization of the loss function defined earlier. This was done to try to break out of local mins caused by training equally on good and bad memories, i.e., the vast majority of memories are bad and so sampling equally on good and bad memories was leading to the network overvaluing the experiences from the good memories and getting stuck in the local mins of wiggling around one of the boundaries.

4 Results and Discussion

Despite much effort, our algorithm is not performing as expected: the large majority of games played by our algorithm end in scoreless losses. This could partially be due to a lack of exploration of the hyperparameter landscape: due to technical limitations, training time took too long to adequately evaluate the performance of a neural network with each possible configuration. In addition, the technical limitations put a cap on how large of a network we could generate without running out of memory, and our topology reflects this upper limit. It's possible that even though we have a large number of nodes, it's still insufficient to handle the massive amounts of input generated by the environment.

However, our difficulties are also partially due to the realization of pong in OpenAI Gym's

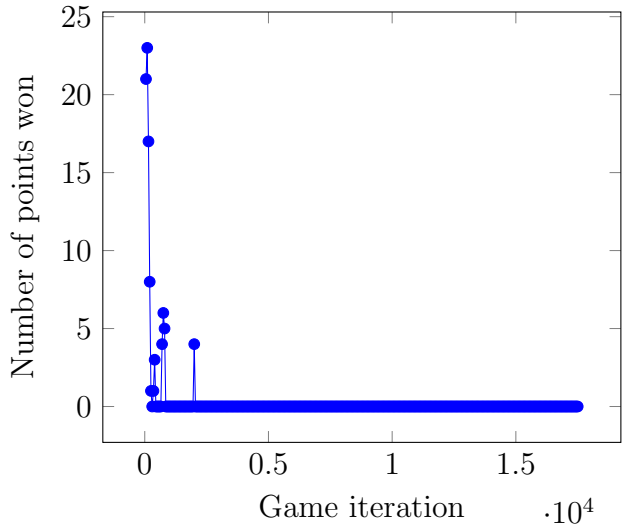

Figure 2: A graph of the number of points won every group of fifty games

environment. The environment has your AI faced against a premade AI that is quite good at returning shots. One particular facet of the AI that has posed problems for learning is its service pattern, which alternates between one of two spots (one on top, one on bottom) depending on whether the player returned the serve or not. Due to this pattern, we could see our algorithm learning to start by moving to either the very top and very bottom of the screen and then struggle to learn to move to the other side after several points of missing the ball, which was being served in the same spot on the other side of the screen. We tried to alleviate this by suppressing the learning rate and implementing a decaying learning rate that would hopefully minimize the effect these service problems had on the weights it learned during earlier training sessions. Ideally, if we had the ability to alter the environment provided to us by the OpenAI Gym we would have pitted our algorithm against a more fair opponent: itself. At the very least, altering the service pattern so that it served in a random direction rather than in the same direction every time would have potentially allowed for our algorithm to learn more easily.

The "Pong-v0" environment is not an impossible problem, and as a matter of fact, there exist several implementations that work well on the webpage of the problem [4], but the difference between these implementations and our own is that the neural networks they used were more sophisticated than what our computational resources could handle. For example, one of the solutions uses a convolutional neural network [6, 7], and while we attempted to train one of these networks, training time was incredibly slow (1.5 minutes for 1 iterations, which translates to 2.5 hours for 100 iterations), and creating a network of any significant size resulted in running out of memory. In summary, in order for success to be achieved on this project, either we needed to alter the problem (opposing player AI), or gain access to more computational resources, such as the supercomputer.

5 Conclusion

In order to teach a computer to play pong, we created a deep neural network that involved two fully connected hidden layers as well as learning rate decay. However, due to technical and temporal limitations, we were not able to train the algorithm to be able to win the game. Given more time and more resources, we could have done more hyperparameter experimentation and potentially implement more sophisticated algorithms such as convolutional neural networks that might offer more definitive results.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller, Playing Atari with Deep Reinforcement Learning, https://www.cs.toronto.edu/ vmnih/docs/dqn.pdf

[2] https://gym.openai.com

[3] https://gym.openai.com/envs/DoomBasic-v0

[4] https://gym.openai.com/envs/Pong-v0

[5] https://www.tensorflow.org

[6] https://gym.openai.com/evaluations/eval_Ce7x6ryRrCMB7JlpbQ

[7] http://cs231n.github.io/convolutional-networks/

[8] http://karpathy.github.io/2016/05/31/rl/

[9] https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-0-q-learning-with-tables-and-neural-networks-d195264329d0#.1u6k2rpib

[10] https://www.tensorflow.org/tutorials/mnist/beginners/

[11] https://gym.openai.com/evaluations/eval_qbonzReT72KkjMRBcziwQ