Applying Neural Networks to Atari Games Using OpenAI Gym Implementations

Angel Nunez, Christian Ikeokwu, Kendra Lockard, Thibault Irissou

Oberlin College

May 15, 2019

Abstract

We created artificial intelligence (A.I.) agents for the purpose of playing Atari Games using pre-made environments from OpenAI Gym. These games are Breakout, Pong, and Assault. By utilizing neural networks and deep Q-learning, we are able to train our agents very quickly. We implement and compare five different deep Q-learning algorithms within our neural network: dqnfast, fulltrainer, rltrainer, doubletrainer, and dueltrainer. After a training time of about a week, We find that the best performing algorithm varies from game to game, but is usually either dqnfast or fulltrainer.

I. Introduction and Background

Humans have participated in games since B.C.E times[1]. The main goal of games is to determine skill and prowess among competitors. That is, there is a well-defined objective, along with strategies a player can use to achieve the objective. Games of these sort can be modelled well as Markov Decision Problems (MDPs), which we have discussed at length in this class. Whereas sports have a somewhat more violent history[2], games have been an opportunity to compete with one another at a lower set of stakes: there are winners and losers, but loss is not a declaration of death. As cultures have evolved and refined to their current states, so too have the games that those cultures play.

Board games are a good example of a refined game with clear-cut goals and strategies. An example of an ancient (though still relevant) board game is the Chinese Go: as old as it is, the game is well known for its complexity. Like ancient games, a typical modern board game has a well-defined objective, and corresponding strategies used to arrive at that objective. They have the added benefit however, of being more familiar to us given cultural context, making them significantly easier to model as MDPs.

Take the card game Dominion for example: this is a game in which two to four players try to build decks, with the objective of accumulating the most "victory points", an attribute possessed by certain card types. There are other "action" cards that provide various benefits, as well as "treasure" cards which are used to purchase all the other cards in the first place. We can use this a good first example of an MDP:

1) States: Cards every player has either in their deck or hand.
2) Actions: The sequence of cards a player plays in a turn, including the cards they choose to buy.
3) Rewards: The value attributed to purchasing certain types of cards, including the final goal of attaining victory point cards.

Originally, we had planned to use Dominion as the main implementation of our project, but found that we were spending significantly more time building the Dominion environment than actually creating the agent to play it. For this reason, we chose to scrap the project, and instead focused on making an agent that could play basic Atari games instead. In particular, we used a neural network to train agents that could play Assault, Breakout, and Pong. This was made possible through the use of OpenAI Gym's pre-rendered environments for each game. We utilized a total of five different learning algorithms for each game. These were: dqnfast, fulltrainer, rltrainer, doubletrainer, and dueltrainer. We found that for each of the different games, different algorithms learned that fastest. For Assault, fulltrainer was the best; for Breakout, dqnfast performed best at first, but fulltrainer caught up towards the end; and for Pong, dqnfast performed the best.

II. Discussion of the Problem

Building A.I. agents that can compete in Atari games at the level of human play is a problem that can essentially be reduced to a Markov Decision Process (MDP). Each of Assault, Breakout, and Pong has its own set of states, transitions, and possible next states, which are well modeled in the OpenAI Gym's implementation of each game. To build agents that could play these games, we decided to harness the power of neural networks in conjunction with Q-learning. We considered several approaches to Q-learning, including Temporal Difference Learning, Approximate Q-learning, and Deep Q-learning. Ultimately, we chose to approach our problem using Deep Q-learning as this technique has demonstrated superior performance against the other proposed methods[3].

We will now give some background on the techniques we used to implement our solution: neural networks and deep Q-learning. Neural networks are a commonly used artificial intelligence technique that loosely emulate biological neural networks, the systems that govern the function of our brains. Neural networks are comprised of a series of layers of nodes, which are analogous to neurons of the brain. These nodes are connected from layer to layer by weighted edges, which are analogous to synapses of the brain. Each node has an activation value ranging from 0.0 to 1.0, where 0.0 indicates full inactivity and 1.0 indicates full activity.

The layers of the network can be broken down into three segments: the input layer, some number of hidden layers, and the output layer. Input data to a neural network typically needs to be preprocessed before it is learnable by an agent. Signal values from the input layer then propagate through the hidden layer, transforming the input into output depending on the weights of the connecting edges within the network.

The process of training a neural network is facilitated through use of back-propagation. The desired outputs are compared against the output actually produced by the network, and the difference (or *error value*) between these results is propagated backwards through the neural network, adjusting the edge weights along the way.

Two phenomena are commonly observed during the training of neural networks: *generalization* and *overfitting*. Generalization is a beneficial emergent behavior of neural networks that occurs when a network has been trained and is able to extrapolate patterns from the data it has previously seen. Neural networks that are able to generalize can be given new input data that they have never seen before, and reliably produce the desired outputs. On the other side of this coin is overfitting, which is a negative result of trained neural networks that have an excessive number of hidden layers. Neural networks that overfit map too directly between the input layer and the output layer, and thus are unable to detect patterns and effectively learn from the past data they've seen. A common solution to overfitting is simply to reduce the number of hidden layers so that the network is forced towards generalization.

Neural networks and deep Q-learning are frequently used in tandem for artificial intelligence research. Both techniques give agents the capacity to learn for themselves and equip agents with the tools – namely, the *reward func-*

tions – that they need to be able to find patterns on their own. What makes these techniques so powerful is that the underlying structure of their environment, and the actions that lead to their success in that environment, may be too subtle for their human programmers to notice and implement themselves. The human is no longer in charge of telling the agent how to behave – instead, this job is left up to the agent itself.

At a high level, deep learning is a method in which the agent trains itself to process and learn from data, rather than relying on hard and fast rules to follow. Related to this is Q-learning, an algorithm that produces a Q-table which enables an agent to choose the best action in a given state. The combination of these methods results in the technique we used to build our Atari game-playing agent: deep Q-learning. Deep Q-learning makes use of deep Q-networks; instead of creating definitive Q-tables to decide actions, deep Q-networks implement a neural network that takes a state as input and approximates Q-values for each possible action. Input to a deep Q-network in our Atari environment might look like this: a series of frames of the game, about four or five of them, are given to the network. From these frames, the network can make a conjecture about what is going on during one half-second or so of the game. These four frames pass through the network as a single "state", and the network outputs a vector of Q-values for each possible action that could be taken next. The agent takes the largest of these Q-values to choose its next action[4].

In recent years, researchers in the field of computer science have made leaps and bounds in the technique of deep reinforcement learning, and have shown that deep learning is one of the most successful methods for training agents in domains with very large state-spaces[5]. In one of the earliest papers on deep reinforcement learning, Mnih et al. [2015] used image and performance data within the Arcade Learning Environment (ALE) to build a method for learning to play the Atari 2600 games[6]. This paper presented results showing that deep Q-networks outperformed all previous reinforcement learning methods, and Roderick et al. [2017] went on to build a method for playing the Atari games using an adapted version of the deep Q-network. We build upon this research in our own implementation of agents that can play Atari Assault, Breakout, and Pong.

III. Solution and Implementation

Training an agent to play an Atari game can take some time, anywhere from a few hours to several days. We chose to make an agent to play a simpler game, CartPole, using the same ideas used in the paper of deep Q-learning. CartPole is one of the simplest environments in OpenAI gym; the goal of CartPole is to balance a pole connected with one joint on top of a moving cart. Instead of pixel information, there are four kinds of information given by the state, such as the angle of the pole and position of the cart. An agent can move the cart by performing a series of actions of 0 or 1 to the cart, pushing it left or right.

We used the the library Keras with a TensorFlow backend to implement a simple a neural network to play the game. We had an input layer corresponding to all the states in CartPole, along with two 24-node hidden layers with a relu activation function and an output layer with two nodes corresponding to the actions one can take in CartPole. We used Adam optimizer to compile this network with a learning rate of 0.00025. We then put this neural net in a deep Q-network (DQN) agent, where we had it interact using an epsilon-greedy approach to exploitation vs exploration. We stored all the experiences of the agent in a Python deque, and then sampled 32 experiences after every episode to retrain the model. This is the technique known as *experience replay*, pioneered by DeepMind. Our DQN agent was able to solve CartPole and stay up for 200+ steps after about 10 minutes of training.

The next stage was to build a double-DQN. In the temporal difference version of Q-learning, one has to estimate what the final reward is using the Bellman equation. However, the final reward is unknown. We used the same model for generating the Q-values to estimate the target values that we use to generate our TD-loss for fitting our models. Therefore, at every step of training, our Q-values shift, but also the target value shifts. So we progress to our target, but the target is also moving – it's like chasing a moving target! This leads to a big oscillation in training. The idea behind the double-DQN is to use another model for the target estimation with fixed weights, and at regular predetermined intervals, copy the weights from the online model which has been getting trained to the offline model. An extension of this idea is to also decouple the Q-value estimation from the action selection. Thus, our online DQN network only learns how to pick the best action to take while the offline target network focuses on Q-value estimation. We implemented this by having two copies of our neural net and during replay using the target network to predict the rewards instead of the regular network. For CartPole, we copied the targets after every episode. This gave us only a slight improvement in training time for CartPole.

At this stage, we decided to use a library called keras-rl to help implement all of the reinforcement learning algorithms. One example of this is the provided data structures, like a memory class, that we could use instead of Python deques. This class was optimized for random sampling and provided lots of convenient methods for reinforcement learning, making it easier for us to focus on hyperparameters, experimentation and neural nets to get faster results. Thus, they provided a way for us to easily use dueling deep Q-networks without having to implement them from scratch. The idea behind a dueling-DQN (DDQN) is motivated from the fact that Q-values correspond to how beneficial it is to be at that state and taking an action at that state $Q(s,a)$. So we can decompose $Q(s,a)$ as the sum of $V(s)$: the value of being at that state; and $A(s,a)$: the advantage of taking that action at that state. With the DDQN, we want to separate the estimator of these two elements, using two new streams one that estimates the state value $V(s)$ and one that estimates the advantage for each action $A(s,a)$. We then combine these two streams through a special aggregation layer to get an estimate of $Q(s,a)$. Intuitively, our DDQN can learn which states are (or are not) valuable without having to learn the effect of each action at each state, since it also calculates $V(s)$. Since we were using keras-rl, we didn't need to implement this from scratch, so we used their premade DDQN class to create our agent.

IV. Experimental Setup

The first part of our implementation was to preprocess the input: we grayscaled and resized the input images from the open AI gym atari environments. At first, we were cropping the inputs to potentially remove parts of the screen that didn't contribute more information but since we would have to make that specific to each game and we wanted to keep our code as general as possible we opted to just resize the images. This probably limited the performance of our algorithm compared to the deepmind implementation but this was sufficient for our purposes.

We then implemented two neural networks to train our agents. The first was a similar network to the one in the deepmind paper with an input layer, a convolutional layer with filter size 32, a kernel of (8,8) and stride of (4,4) and relu activation function , then another convolutional layer with filter size 64, a kernel of (4,4) and stride of (2,2) and relu activation function , then another convolutional layer with filter size 64, a kernel of (3,3) and stride of (1,1) and relu activation function then we flatten and add a dense layer with 512 nodes and relu activation function and then one more dense output layer with size corresponding to the action size of the game.

We also implemented a smaller network that would train faster and we believed would be useful for debugging purposes. This network has an input layer, a convolutional layer with filter size 16, a kernel of (8,8) and stride of

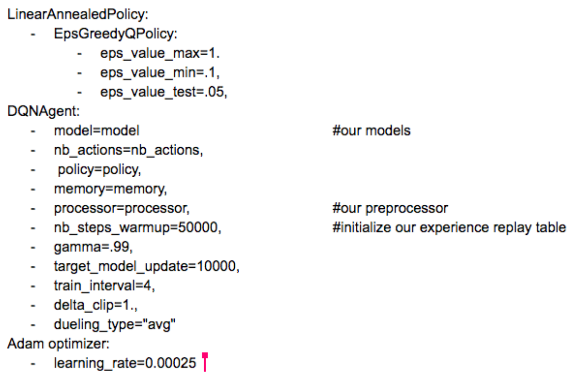
(4,4) and relu activation function , then another convolutional layer with filter size 32, a kernel of (4,4) and stride of (2,2) and relu activation function , then we flatten and add a dense layer with 256 nodes and relu activation function and then one more dense output layer with size corresponding to the action size of the game.

We compiled both nets with Adam optimizer and at first we used a learning rate of 0.01 then ultimately used a learning rate of 0.0025 for our final implementation. We used these models to create 5 different kinds of DQN agents.

dqnfast: used the smaller neural net for its prediction but it was a double and dueling network.
rltrainer: used the big net for training but was a simple dqn.
doubletrainer: used the big net but was a double q network
dueltrainer: used the big net but was a dueling q network
fulltrainer: used the big net but was a double and a dueling neural network.

Those were the only differences between the networks and all the hyper parameters were the same. They all used a linearly decaying epsilon-greedy policy and used the Adam optimizer. Below is a summary of parameters we used.

Figure 1: *Parameters*



```
LinearAnnealedPolicy:
    -   EpsGreedyQPolicy:
            -   eps_value_max=1.
            -   eps_value_min=.1,
            -   eps_value_test=.05,
DQNAgent:
    -   model=model                             #our models
    -   nb_actions=nb_actions,
    -    policy=policy,
    -   memory=memory,
    -   processor=processor,                    #our preprocessor
    -   nb_steps_warmup=50000,                  #initialize our experience replay table
    -   gamma=.99,
    -   target_model_update=10000,
    -   train_interval=4,
    -   delta_clip=1.,
    -   dueling_type="avg"
Adam optimizer:
    -   learning_rate=0.00025
```

In order to try out a variety of algorithms on a handful of games, we obtained access to high performance computing machine at Oberlin, SCIURUS, on which we could train all of our instances in parallel. This was facilitated by the use of both Anaconda and Keras which make full utilization of high performance cluster processors when available. We simply had to learn PBS scripting and how to interact with the HPC task manager in order to quickly run these tests on the supercomputer. We also wrote a quick versioning script which just ensured that we don't overwrite previous data and such that each algorithm stores its weights and logs in an appropriate file structure.

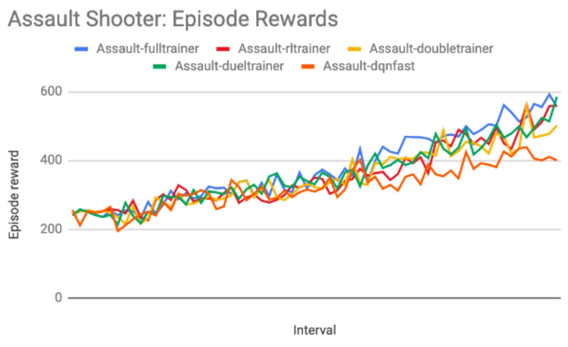
It should not be possible to train our algorithms and get the exact same weight models due to the nature of OpenAI gym as randomly sampling from a library of experiential data. However, the final weights that we have obtained can be loaded into the algorithms combined with the "–test" tag so as to test the best results that any of our algorithms had by the end of their time training. We also made it possible to load weights into the algorithm in order to not have to train from scratch, which means that improvements with preprocessing and other such fine optimizations can be implemented midway through the training without having to train from square 0.

Most of our algorithms were only run once due to not wanting to overuse the HPC. We found it more important to be able to compare the relative performance of the algorithms on three games: Breakout, Pong, and Assault, which leads us to what we can draw from our results.

V. Results and Interpretation

We deployed them to the Oberlin supercomputer and let them train. On Assault, *fulltrainer* was by far the best performing, followed by *dqnfast*. The lowest performing model was *rltrainer*. This is likely because *fulltrainer* used all the double and dueling networks, along with the bigger model. The fact that *dqnfast* outperformed the other models suggests that the learning algorithm is more important than the size of the network. Overall, in Assault, we didn't reach state of the art performance, but we did reach human-level play with average scores in the 900s.

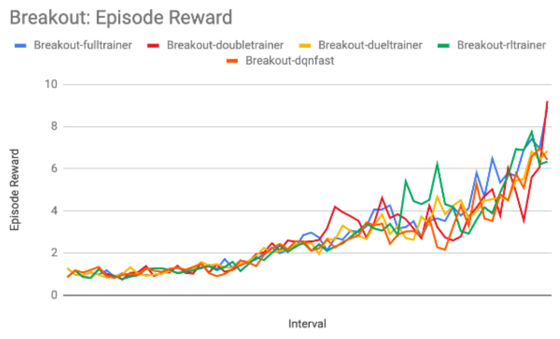
Figure 2: *Episode reward data for Assault.*



For most of the models, Breakout was slow to train at first, but then quickly started learning. In comparing the graphs of performance in Breakout vs Assault, notice that the shape of the graph looks close to exponential/quadratic, whereas Assault looks more linear with time. This is because it takes a long time for the agent to learn how to start the game. To fix this, we hardcoded that the first action of the game should be to start the game. This greatly improved our training. As for the models, *dqnfast* trained faster and performed better at first. Eventually, all the other models caught up, and *fulltrainer* surpassed *dqnfast*. We reached human level play in Breakout, but our agent was unable to completely win the game. We believe that with more training time this would have been possible. We ultimately got to a high score of 68.

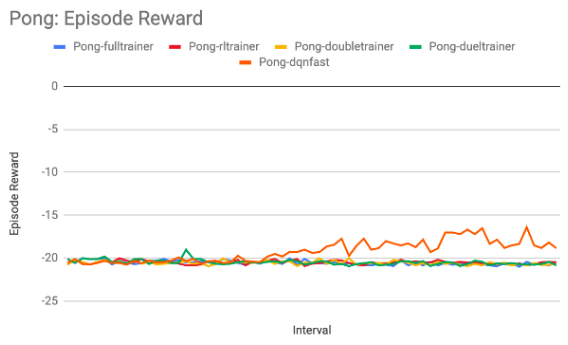
For Pong, while all models made improvements continually in performance over time, the only *dqnfast* reached a level where it learned how to consistently win the game against its opponent. We don't fully understand why the other algorithms never learned how to win the game. We believe that this is due to the size of the networks since the computational complexity of training is a quadratic function of the size of the network, which means we may simply not have trained it enough that we learned how to win the game. Once again, we believe

Figure 3: *Episode reward data for Breakout.*



the proposed solution to this problem is to allow the algorithm to train for longer. Our high score was 12.

Figure 4: *Episode reward data for Pong.*



VI. Conclusion

Through the use of neural networks and machine learning algorithms we were able to construct A.I agents that were capable of playing Atari games near or past the level of human skill. This in and of itself is impressive, but is even more so when considering that the time spent training these agents was under a week. We find that for Assault and Breakout, fulltrainer produced the best results. However, for Pong, we find that dqnfast had the best results. We believe that if given more time to train the Pong agent, fulltrainer would eventually converge with and surpass the results of dqnfast.

We affirm we have adhered to the honor code on this assignment.

VII. References

[1] https://www.britgo.org/intro/history

[2] https://www.ancient-origins.net/history-ancient-traditions/pankration-deadly-martial-art-form-ancient-greece-005221

[3] Oppermann, A. (2018, November 04). Deep (Double) Q-Learning. Retrieved April 4, 2019, from https://towardsdatascience.com/deep-double-q-learning-7fca410b193a

[4] Simonini, T. (2018, April 11). An introduction to Deep Q-Learning: Let's play Doom. Retrieved from https://medium.freecodecamp.org/an-introduction-to-deep-q-learning-lets-play-doom-54d02d8017d8

[5] Roderick, M. (2017, November 20). Implementing the Deep Q-Network. Retrieved April 4, 2019, from https://arxiv.org/pdf/1711.07478.pdf

[6] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., . . . Hassabis, D. (2015, February 25). Human-level control through deep reinforcement learning. Retrieved April 4, 2019, from https://www.nature.com/articles/nature14236