# CS229 Project Final Report
# Deep Q-Learning on Arcade Game Assault

Fabian Chan (fabianc), Xueyuan Mei (xmei9), You Guan (you17)
Joint-project with CS221

## 1 Introduction

Atari 2600 Assault is a game environment provided on the OpenAI Gym platform; it is a top-down shoot'em up game where the player gains reward points for destroying enemy ships. The enemy consists of a mothership and smaller vessels that shoot at the player. The player can move and shoot in various directions with a total of 7 actions available. Every time the player shoots, a heat meter keeps track of how 'hot' the engine is; if the player shoots too frequently, the player can lose a life when the heat meter fills up due to overheating. The player can also lose a life upon taking fire from enemy ships. The game ends when the player runs out of lives. We create an AI agent that generates the optimal actions, taking raw pixels as features by feeding them into a convolutional neural network (CNN), also known as deep Q-learning.

## 2 Related Work

The first paper *Playing Atari with Deep Reinforcement Learning* [1] addresses how convolutional neural networks and deep reinforcement learning combine together to accomplish a high performance AI agent that play Atari games. The paper analyzes how Reinforcement learning (RL) provides a good solution to game playing problem and also the challenges in Deep Learning brought about by RL from the data representation perspective. The paper proposes deep reinforcement learning on game playing agents, which is similar to our goal. However, there are differences between our approaches and theirs. One of the differences is that the approach in this paper relies on heavy downsampling images before feeding them into a neural network. Our approach tries to avoid this downsampling procedure in an attempt to produce better data layers for deep Q-learning.

The second related paper *Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning* [2] proposes another solution to game playing. Compared to the first paper, which uses a model-free Q-learning strategy, this paper tackles the problem using a combination of Monte-Carlo techniques and deep Q-learning, ending up with a much more sophisticated algorithm that adds extra assumptions and considerable complexity. Though we choose the model-free learning approach, this paper still provides us with insights about deep learning applied to games, such as the preprocessing of raw data and the architecture of convolutional neural networks.

## 3 Dataset and Features

Put simply, the OpenAI Gym framework gives us the dataset we need in terms of observations at every time step. An observation consists of pixel values of the game screen taken for a window of k consecutive frames. This is a numpy.array of shape (k, 250, 160, 3), where the second value x is the height of the screen, the third value y stands for the width of the screen, and the fourth represents the RGB dimension for each pixel at coordinate (x, y). We have simply chosen the raw pixels processed into black and white pixels in binary as features because they are simple to obtain and we can let the CNN figure out the high level features itself. Other than an observation we also receive an incremental reward score at every timestep.

## 4 Methods

### 4.1 Q-Learning for Game Playing

One common way to deal with the game playing problem is to assume a Markov Decision Process (MDP). This is appropriate for Assault because the enemy agents move randomly. An MDP is a model defined by a set of States, Actions, Transitions, and Rewards. In order to train an AI to tackle game playing tasks, reinforcement learning based on Q-learning is a popular choice. In Q-learning, the MDP recurrence is defined as follows:

$$Q(s^*, a) = E_{s'\ \epsilon}[r + \gamma max_{a'} Q^*(s', a')|s, a] \tag{1}$$

where a is the action it takes, s is the current state, r is the reward, and $\gamma$ is the discount factor. Furthermore, we can use function approximation by parameterizing Q-value. In this way, we can easily adapt linear regression and gradient descent techniques from machine learning. With function approximation, we can calculate the best weights by adapting the update rule:

$$w \leftarrow w - \eta[\hat{Q}_{opt}(s, a; w) - (r + \gamma \hat{V}_{opt}(s'))]\Phi(s, a) \tag{2}$$

where $w$ is a vector containing weights of each feature, and is initialized randomly to avoid getting into the same local optimum in every trial.

### 4.2 Convolutional Neural Networks as Function Approximators

We decided to use deep Q-learning instead of ordinary Q-learning because there are too many possible game states. The size of one frame is 216 by 160 pixels, and for each pixel there are $256^3$ choices of RGB values. Furthermore, a sliding window of k frames leads to $256^{216*160*3*k}$ possible states in total – this is too large for ordinary Q-learning because it will result in too many rows in our imaginary Q-table. Therefore, we decide to use a neural network to learn these Q values instead. In effect, this neural network ends up operating as a function approximator. The network architecture is currently described as follows:

- Preprocess frames: convert RGB pixels to grayscale and threshold to black or white
- Input layer: takes in the preprocessed frames. Size: [k, 160, 250, 1]
- Hidden convolutional layer 1: kernel size [8, 8, k, 32], strides [1, 4, 4, 1]
- Max pooling layer 1: kernel size [1, 2, 2, 1], strides [1, 2, 2, 1]
- Hidden convolutional layer 2: kernel size [4, 4, 32, 64], strides [1, 2, 2, 1]
- Max pooling layer 2: kernel size [1, 2, 2, 1], strides [1, 2, 2, 1]
- Hidden convolutional layer 3: kernel size [3, 3, 64, 64], strides [1, 1, 1, 1]
- Max pooling layer 3: kernel size [1, 2, 2, 1], strides [1, 2, 2, 1]

- Resize the max pooling outputs to a vector of size [768] and feed to one fully connected layer
- Feed the outputs to a rectified linear activation function
- Collect the final output as 7 Q-values, each corresponding to an action

Though we have 3 max-pooling layers involved in our network architecture, this is not what we had in mind at the beginning. Unlike CNN architectures typically used in computer vision tasks such as image classification, pooling layers in our architecture may not have been desirable for our purposes because we likely do not want to introduce translation invariance since the position of the game entities are important for estimating Q values. However, max-pooling serves as an adequate way to compress our large state space into a vector of size 768, and is the reason why we have been using them. One suggestion to replace these max-pooling layers is to make the strides larger in each hidden convolutional layer – but given that the current strides already have substantial size, we have been hesitant to increase it any further. However, we are not dismissing it as a bad idea and would like to give it a try if we were given an additional month or two to evaluate.

## 4.3 Experience Replay

In general, deep neural networks are difficult to train. In the presence of multiple local optima, gradient descent may end up at a bad local minimum which will lead to poor performance. Initializing the network weights and biases to random values helps a little but is likely not sufficient.

We incorporate a technique called experience replay to encourage the algorithm to find better optima instead of getting stuck at some underperforming local optimum. To be more specific, as we run a game session during training, all experiences $< s, a, r, s' >$ are stored in replay memory. During training, we take random samples from the replay memory instead of always grabbing the most recent transition. By breaking the similarity of subsequent training examples, this trick is likely to prevent the network from diving into some local minimum and will do so in an efficient manner [5]. By using experience replay, the behavior distribution is averaged over many of its previous states, smoothing out learning and avoiding divergence in the parameters.

In our implementation we simply perform a uniform sample from the bank of observed states (the replay memory) to construct a minibatch of size 100 with which we train on each iteration. Storing all past experiences is impossible due to the humongous state space, so we simply retain the most recent 20,000 observations in the replay memory and sample from that.

## 5 Experiments/Results/Discussion

As preliminary evaluation, we first ran our algorithm (without experience replay) for 5 trials, each with a cutoff at the end of 36 hours, using number of consecutive frames k=3. The final scores for each trial is plotted on the top half of Figure 1. From the results we can see that some trials performed pretty badly, and are in fact no better than the baseline Q-learning algorithm. However, other trials performed significantly better than the baseline. We believe these differences in performance among the trials can be explained by the gradient descent approach that we used for training our deep neural network, which is characteristically vulnerable to getting stuck at some under-performing local optimum. We arrived at this explanation because each trial is initialized with random weights and biases, and these trials produce wildly different final scores, so mostly likely each of them ended up in a different local optimum.

We then tweaked our model parameters (aka hyperparameters) such as $\epsilon$ the exploitation-exploration parameter, k the number of consecutive frames in consideration, and $\eta$ the learning rate, in a manner similar to grid search. Since in our situation we do not have a dataset for which to divide into training, validation and testing sets for the reason that out training comes from operating a dynamic game, we simply repeated training on various values of $\epsilon$, k and $\eta$ and find the combination that gives the best scores. This is straightforward compared to the usual hyperparameter optimization process in machine learning.

We found that setting $\eta$ to 0.01, $k$ to 4, and $\epsilon$ to a dynamic one that linearly decreases from 0.8 to 0.05 annealed over 50,000 timesteps gives the best results overall, but we notice that in general varying the hyperparameters does not significantly influence the game agent's performance, so we did not devote too much time trying out different combinations.

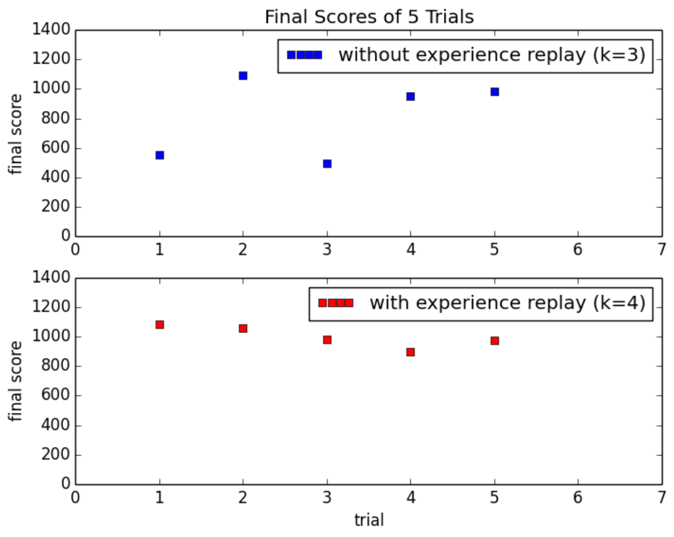

Figure 1: Comparison: performance with and without experience replay

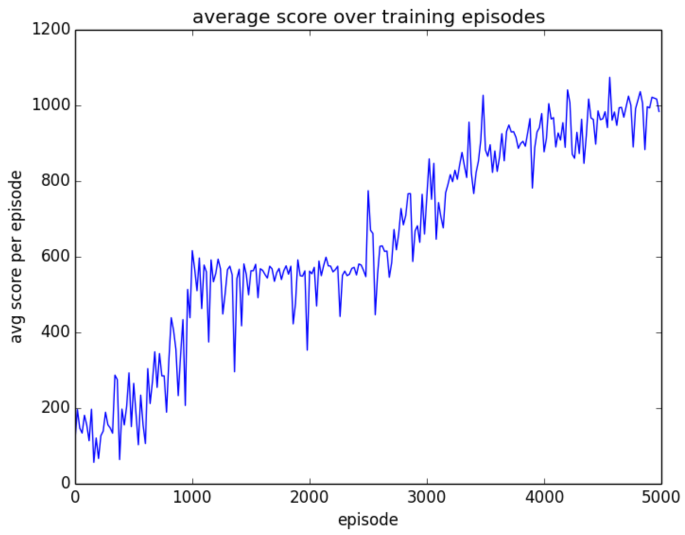

Figure 2: Average scores over training episodes

We obtained our final results by running the agent using the model weights computed at the end of training and noting down the final score repeated for a total of 5 trials, and instead of doing the cutoff based on time, we now terminate training after performing 50,000 iterations. We switched from time-based cutoff to iteration-based cutoff because the time it takes to train a model is mainly a function of the hardware used to train it. Furthermore, reporting the scores obtained from a certain number of episodes makes it more robust against the noise/stochasticity of the OpenAI gym environment. This time we also incorporated experience replay into the training procedure. The scores we obtained are plotted on the bottom half of Figure 1. Comparing these results with those above, we can see that experience replay produced results that are more consistent and stable. This is because this technique was able to effectively prevent the algorithm from getting stuck at some bad local optimum, and thus help achieve (slightly) higher and more consistent results.

To aid with our analysis we plot the average score per 20 consecutive training episodes over one complete trial in Figure 2. We can see that in the beginning, the score rises rapidly to around 600 because the game mode stays the same up to this point and so far it is quite easy to get there. After which it appears to get stuck at around 600 for over 1500 episodes, and this is because starting from this point, the game jumps in difficulty – new enemy entities known as ”crawlers” appear on the left and right sides of the agent, in addition to the enemy ships already hovering above. As if it encountered a roadblock, the agent was unable to make much progress past 600 for quite some time, but it did eventually learn to overcome this obstacle. So we conclude that even with this change of difficulty, our agent simply needed some time to adjust and continue to learn. From 600 onwards the agent improves at a rate slower than it did from the start to 600 because the game is no longer as easy as it was in the beginning. Finally, it saturated at around 1000 and this is roughly the final score it was able to achieve.

## 6 Conclusion/Future Work

In this project, we have implemented a game playing agent for Atari Assault using deep Q-learning. We first implemented ordinary Q-learning to obtain a baseline of score 670.7, then we implemented deep Q-learning by constructing a convolutional neural network using Tensorflow. We obtained promising results after experimenting with and without experience replay. For us, experience replay worked well in helping the agent avoid getting stuck at some bad local optimum. Some of our improvements also came about by extending the training time and tweaking hyperparameters. Our deep Q-learning agent managed to significantly out perform the baseline: the average score it obtained was 980, while the ordinary Q-learning baseline has an average score of 670.7.

Although our agent does significantly better than the baseline, it still does not come close to the oracle. We believe that there are still many places we can try to improve, such as revising the neural network architecture, adding customized feature extractors, and experimenting with more fully connected layers.

## 7 Division of Work Between CS229 and CS221

Although there exists a lot of overlap, we decided to divide the work as follows. For 229, we used standard machine learning techniques like mini-batch gradient descent for optimizing the weights and biases of the neural network, cross validation via grid search and basic concepts from reinforcement/Q-learning. For 221, we modelled the MDP states and actions, designed efficient feature extractors, and were responsible for the deep learning architecture.

## 8 References

1. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Wierstra, D., & Riedmiller, M. (2016). Playing Atari with Deep Reinforcement Learning. University of Toronto.

2. Guo, X., Singh, S., Lee, H., Lewis, R., & Wang, X. (n.d.). Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning. Retrieved 2016, from http://papers.nips.cc/paper/5421-deep-learning-for-real-time-atari-game-play-using-offline-monte-carlo-tree-search-planning.pdf

3. Assault (1983, Bomb) - Atari 2600 - Score 4153. (n.d.). Retrieved November 16, 2016, from https://www.youtube.com/watch?v=Wxio1_ytTTo

4. Assault-v0. (n.d.). Retrieved from https://gym.openai.com/envs/Assault-v0

5. Matiisen, B. T. (n.d.). Demystifying Deep Reinforcement Learning. Retrieved November 16, 2016, from http://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/