Project Part 3
736 Neural Networks and Machine Learning
Nic Manoogian
Robert Bond III
Zach Lauzon

Executive Summary

In the third part of our project, we maintain our original problem and data sets: create an optimal agent for playing Nintendo's *Super Mario Bros.*. In this document, we employ a deep learning model called a *Deep Q Network (DQN)* in an attempt to create an agent which is superior to the one created by the NEAT algorithm.

In this document, we describe deep Q learning and some of its important hyperparameters; we describe our methodology and experiments; we discuss the findings of these experiments; and we compare the performance of the DQN with the neural network produced by the NEAT algorithm.

Overall, we found the following trends:

1. Higher-resolution DQN inputs yield better performance but take far longer to train.

2. A linearly-increasing discount factor yields better performance long-term.

3. A larger minibatch size had little performance benefits but significantly slowed down computation.

4. Although our best NEAT agent has an overall higher score, the agent created by the DQN algorithm seems to have higher potential and is distinct in a somewhat "skillful" way.

Requirements

The goal of this project is to employ a "deep learning" technique to solve a problem. Such methods model high-level abstractions in data by using

multiple neural network layers with complex structures. Relevant examples include *Recurrent Neural Networks (RNNs)* and *Convolutional Neural Networks (CNNs)*.

In this part of the project, we use *Deep Q Reinforcement Learning* to play *Super Mario Bros.* We feel that this type of deep learning will be well-suited to the task and should yield a very effective game-playing agent. DQNs are complex and composed of heterogeneous layers — including several convolutional layers.

In this paper, we describe the hyperparameters of the deep Q learning algorithm; select a subset of these hyperparameters; and we justify our selections in the context of both *Super Mario Bros.* and the original DeepMind paper.

Specification

In a 2015 paper, Mnih and colleagues ("the DeepMind team") use DQN to process raw pixel input and play Atari 2600 games. During training, *experience replay* is used; here, the agent's experiences are stored at each time-step and pooled into a *replay memory*. Randomly-sampled experiences from memory are fed into a convolutional neural network as part of the training process. The original network consists of $84 \times 84 \times 4$ preprocessed inputs followed by three convolutional layers and two fully-connected layers with a single output for each action that the agent could take (Mnih et al., 2015).

Deep Q Learning Overview

Deep Q learning is a multi-step process. The system takes input in the form of a "state," s. This object is composed of h game states. In the game *Breakout*, the game state is one preprocessed image (84×84), so the DQN state is composed of h of these images. The system produces one output for each valid action, a. This output is an approximation of the "utility" of this action, represented by the function Q.

The system uses *epsilon greedy selection* to choose among these Q-scored actions. This proceeds as follows: With probability ϵ, select a random action a. Otherwise, select $a = \text{argmax}_a Q(s, a)$.

Once a is selected, it is executed in the emulator, a new state is generated, which has a new Q value. From this, we calculate the reward r which was associated with this action. This process captures testing with a DQN.

To learn, we also store the transition (s_t, a_t, r_t, s_{t+1}) into D, the replay memory of capacity N. We then sample a random minibatch from D. For each sample j, we compute a target output for the new Q function, $y_j = r_j + \gamma \max_{a\prime} Q(s_{j+1}, a\prime)$. We then perform a gradient descent on $(y_j - Q(s_j, a_j))^2$ with respect to the network parameters θ. This method of learning is designed to propagate rewards (such as beating a level) back to earlier states.

The DeepMind team did indeed release their source code (`https://github.com/kuz/DeepMind-Atari-Deep-Q-Learner`), allowing us to reuse the original DQN implementation. However, this code is designed work only with an Atari emulator to play Atari games. In order to use this implementation, we modify the original code to work with QuickNES — a popular NES emulator.

Reward Function

Similar to the fitness function used in the NEAT algorithm, deep Q learning requires a reward function. Given states s_1 and s_2, the reward function R returns the (positive or negative) instantaneous utility of the transition from s_1 to s_2.

Deep Q Network Structure

The deep Q network described in the original paper consists of several convolutional layers and one hidden layer. Specifically,

1. A hidden layer which convolves 32 filters of 8×8 with stride 4 with the input tensor ($84 \times 84 \times 4$) and applies a rectifier nonlinearity
2. A second hidden layer which convolves 64 filters of 4×4 with stride 2, also followed by a rectifier nonlinearity
3. A third hidden layer which convolves 64 filters of 3×3 with stride 1, followed by a rectifier
4. A final hidden layer which is fully-connected and consists of 512 rectifier units

5. An output layer which is a fully-connected linear layer with a single output for each valid action

Implementation

Required Modifications

The original code from DeepMind is designed specifically to work with Atari 2600 games. In order to use the algorithm to generate an agent for *Super Mario Bros.*, we make a number of modifications to both our original feature selection and the DQN code:

1. *Change DQN to use a different emulator.* The original DeepMind code is written in Lua but uses Xitari to simulate the original Atari 2600. We run our modified Lua code in the QuickNES emulator.

2. *Adjust input tensors.* To play *Breakout* on the Atari, the original authors preprocessed display pixels (210×160 with a 128-color palette) to produce an greyscale input (84×84). *Breakout* is a simple game and its features can be easily reduced to greyscale pictures. *Super Mario Bros.*, by contrast, is much more complex. The same enemy may look entirely different from one level to the next. For this reason, we continue to use our sprite-based feature selection but with a larger bounding box and a higher resolution. In this new feature selection system, we create a $n \times n$ input tensor with 0.0 representing empty air, 0.5 represents an enemy, and 1.0 represents a solid block.

3. *Specify a reward function.* In *Breakout*, the reward for going between two states is simply the number of blocks destroyed between s_1 and s_2. In *Super Mario Bros.*, conversely, we use Mario's differential X position as a reward. This encourages movement through the level, discourages backward movement, but does not penalize the agent for halting to allow enemies to move out of an optimal path.

Scoring and Performance Evaluation

During the second project installment, we proposed a fitness function for scoring NEAT agent. At first glance, it seems reasonable to extend this

scoring method to DQN agents. However, on closer inspection, this method of evaluation imposes some unfair criteria on the DQN agent. To improve convergence in our NEAT agents, we penalize networks for "divergent" behaviors such as standing still (taking no action). Because the DQN agent can converge without such penalties, it seems unfair to impose them when evaluating performance.

For this reason, we propose a new function: $S(p, l) = x$. That is, given an agent p and a level l, the performance of that agent is x, the total X distance in pixels that the agent was able to traverse in the level before either victory or defeat.

Source Code and Dependencies

Our code is attached with this submission but requires a relatively powerful machine to train and learn. Specifically, we run all tests on an AWS EC2 g2.2xlarge with 26 ECUs, 8 vCPUs, 2.6 GHz, an Intel Xeon E5-2670, 15 GiB memory, and 60 GiB Storage Capacity.

The code relies on the following dependencies (which can be installed using a script in our code repository):

- Torch `http://torch.ch`
- CUDA `https://github.com/torch/cutorch`
- Headless QuickNES `https://github.com/Bindernews/HeadlessQuickNes`

Experiments

Input Resolution

Rather than providing raw pixel inputs to the DQN, we provide the network with an $n \times n$ tensor of features (i.e. empty air, enemies, or blocks). Thus, each of the n^2 tiles have the dimensions $(s \times s)$, where M is the screen size in pixels and $s = \frac{M}{n}$.

It seems intuitive that higher-resolution inputs yield more precise play, and thus, better performance. But it also follows that such inputs will also yield slower running times due to the growth in search space.

To determine the effects of resolution beyond speculation, we test two values: $s = 4$ and $s = 16$.

Annealing Discount-Factor

In the original DeepMind algorithm, the exploration hyperparameter, ϵ, is annealed. That is, $\epsilon = 1$ in the beginning but decreases linearly to $\epsilon = 0.1$ over the course of 1000000 frames. Recall that the agent will ignore the Q values of actions and choose among the set randomly with the probability of ϵ. Intuitively, a high ϵ value is good during the first phase of learning because the agent needs behave randomly to explore different techniques. Once the agent accumulates more experiences, he may start relying on his Q function and converge on the optimal style of play.

The *discount factor* γ, is another hyperparameter in the deep Q learning algorithm. Essentially, it represents the degree to which the reward for transitioning into a state propagates back to intermediate states. For example, consider a game where transitioning into the terminal state yields a reward of 100. During training, a state that transitions into this terminal state would have a reward of $r + \gamma 100$ where r is the intrinsic reward of transitioning into the state. Following a similar line of intuition as for the exploration hyperparameter, an annealed discount factor may be valuable. During the initial phase of the game, propagating reward into intermediate states implicitly discourages exploration by encouraging familiar actions which led to reward. By keeping γ small in the beginning and increasing it linearly over time, we hypothesize that the agent will have a greater opportunity to explore different styles of play. Additionally, some research has shown that annealing the discount factor in this way can produce the same performance as a non-γ-annealed DQN in significantly fewer learning steps (François-Lavet, Fonteneau, & Ernst, 2015).

Larger Minibatch Size

As described in the DQN overview, a "minibatch" of transitions is sampled from D during training. In the original DeepMind implementation, $b = 32$ samples were selected. Without intuition or insightful research to support us, we decided to try a larger minibatch size of $b = 64$.

Experiments Enumerated

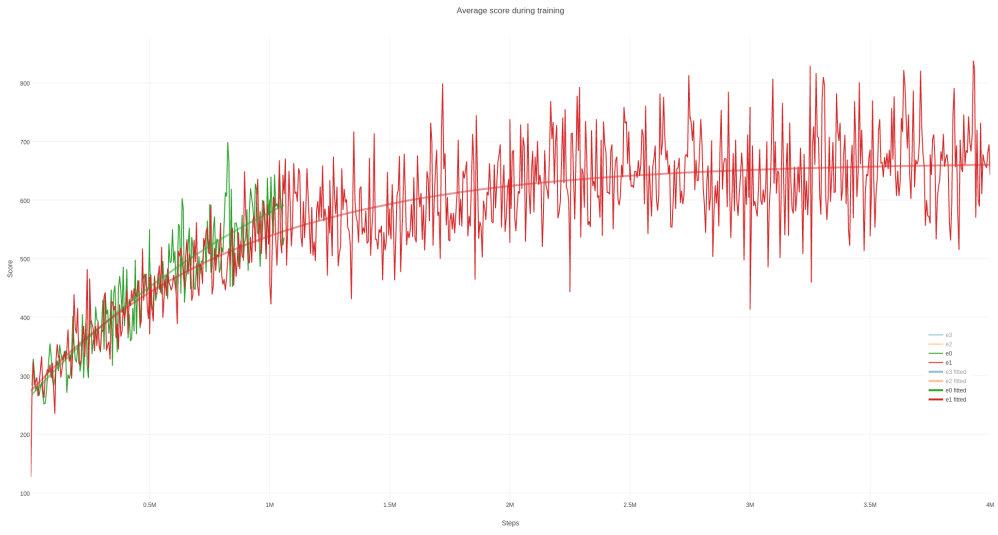
ID	Resolution	Annealing Discount Factor	Minibatch Size	Running Time	Color
e0	4	No	32	8 hours	green
e1	16	No	32	8 hours	red
e2	8	No	64	16 hours	orange
e3	8	Yes	32	40 hours	blue

Results

Experiments 0 and 1

These first experiments are designed to evaluate the value and costs associated with down-sampling the agent's input tensor. As shown below, a finer resolution seems to reach a higher potential but at a higher computational cost. We test $s = 4$ (high resolution, e0) and $s = 16$ (low resolution, e1) and decide that a compromise between the two would be best for further testing. Thus, we settle on $s = 8$.

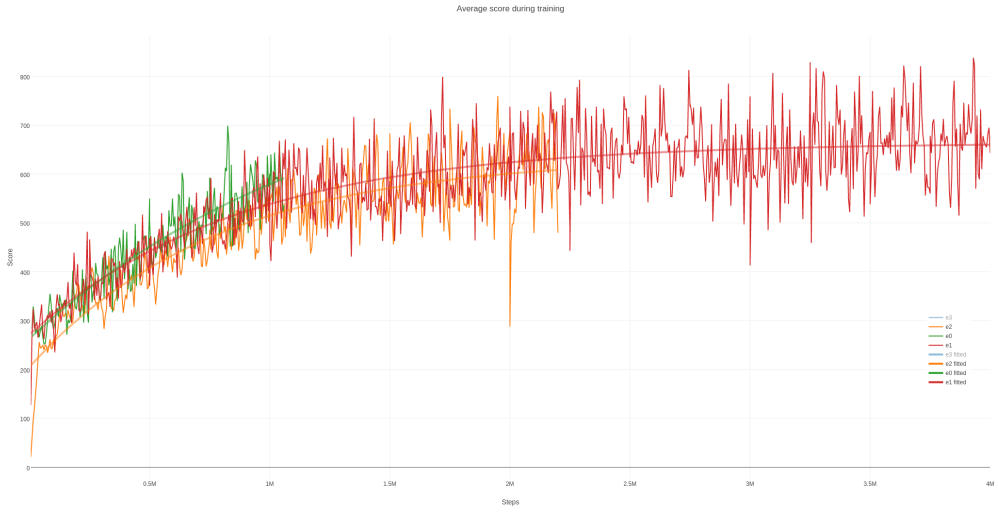
Figure 1: Experiments 0 and 1



Experiment 2

In this experiment, we test the effects of a larger minibatch size. We test $b = 64$ and compare it to all other experiments which run $b = 32$. As shown below, a higher minibatch size does not seem to affect game performance but does slow down the system significantly. Thus, we proceed with a minibatch size of $b = 32$.

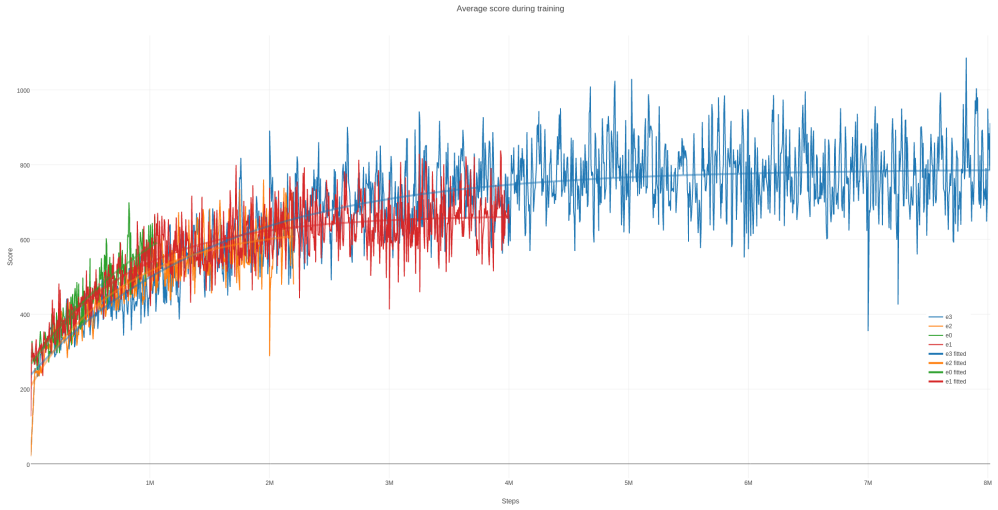
Figure 2: Experiments 0, 1 and 2



Experiment 3

In this final experiment, we test an annealed discount factor and compare it to all other experiments with a constant discount factor. As shown below, the annealed discount factor significantly improves performance with no significant computational cost. As expected, the networks produces lower scores during the annealing period (the first million steps), and then rapidly achieves higher scores.

Figure 3: Experiments 0, 1, 2, and 3 - `https://plot.ly/~ZachLauzon/109/average-score-during-training/`



DQN vs. NEAT

Our best NEAT agent reaches an average S score (across all tested levels) of 2136. Our best DQN agent (the result of Experiment 3) reaches an average S score of 1068. It should be noted that the NEAT agent was allowed to train for far longer than the DQN agent.

The DQN agent also exhibits a few interesting patterns of behavior, that in many ways, is skillful. In particular, the agent:

- seems to "wait" to allow enemies to pass through critical areas of the level,
- will backtrack though parts of a level if no progress can be made on the agent's current path,
- and seems to learn techniques such as jumping on Koopas and kicking their shells to clear lines of enemies.

It should be noted that these behaviors were never exhibited by our NEAT agent. In other words, our NEAT agents appear to achieve goals "with luck" while the DQN agent appears to achieve goals "with skill." It is our overall

sense that the DQN agent, although not currently the highest-scoring, has far greater potential than NEAT.

Resource Consumption

The following table summarizes the resources consumed during training by both NEAT and DQN:

Method	CPU Utilization	RAM	GPU Utilization	GPU Memory
NEAT	100%	100MB	None	None
DQN	100%	1.6GB	80%	700MB

During testing, DQN uses the same amount of resources, while NEAT uses 25% CPU and 5MB of RAM. It is clear that the DQN approach requires far more resources than NEAT, but we find the use reasonable given the nature of image-based problems. DQN's RAM usage is likely due to the *replay history* feature of DQN.

Further Research

We see several very interesting avenues in further developing our DQN agent:

1. *Annealing the learning rate.* Our tests reveal that an annealed discount factor is effective in increasing the performance of an agent. Perhaps annealing the learning rate will also be effective.

2. *Using actual pixel values rather than feature-selected inputs.* By imposing our own feature selection, we may be restricting the algorithm from creating the optimal player.

3. *Including the score of the game as a factor in the reward function.* Using the game score as a reward in deep Q learning would encourage behaviors such as jumping on enemies, collecting coins, and getting power-ups.

4. *Including Up and Down buttons into the agent actions.* Although *Super Mario Bros.* can be beaten without pressing *Up* or *Down*, there are some more advanced techniques that can only achieved by pressing these buttons. It would be very interesting to see what kinds of

techniques are learned when the agent has the capacity to make more complex and advanced decisions.

5. *Implementing negative rewards.* By only using Mario's differential x position in the level, we do not negatively reward falling to pits or making contact with enemies. It is merely the impossibility of further rewards that makes such actions undesirable to the current agent. By levying a cost for these actions, we may see faster convergence when it comes to avoiding negative outcomes. An unfortunate property of traditional Q-learning is that negative rewards are not propagated backwards. Research in Q-learning by (Fuchida, Aung, & Sakuragi, 2010) that considers negative rewards could solve this issue.

References

François-Lavet, V., Fonteneau, R., & Ernst, D. (2015). How to discount deep reinforcement learning: Towards new dynamic strategies. *CoRR*, *abs/1512.02011*. Retrieved from http://arxiv.org/abs/1512.02011

Fuchida, T., Aung, K. T., & Sakuragi, A. (2010). A study of q-learning considering negative rewards. *Artificial Life and Robotics*, *15*(3), 351–354. Retrieved from http://dx.doi.org/10.1007/s10015-010-0822-7 doi: 10.1007/s10015-010-0822-7

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... Hassabis, D. (2015, Feb 26). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529-533. Retrieved from http://dx.doi.org/10.1038/nature14236 (Letter)