IMPLEMENT DEEP Q NETWORK FROM SCRATCH TO TRAIN AN AGENT

Sujay S. Garlanka
University of Southern California
{garlanka}@usc.edu

ABSTRACT

Reinforcement learning is a powerful approach to training agents to solve complex tasks. With the rise of neural networks and inexpensive compute, it has proven to be a particularly promising area. We develop an efficient neural network library and utilize it to train deep neural networks to use within reinforcement learning algorithms. Specifically, we implement and train a Deep Q Network to play Flappy Bird.

1 INTRODUCTION

Deep reinforcement learning is being applied in many ways such as stock trading, energy management, robotic control, etc. with promising results. We harness its promise to develop an agent that can play Flappy Bird. However, in all its applications, RL requires large amounts of data and consequently significant compute to train on all that data. While compute is a bottleneck, so is the time required to train. So in addition to building a neural network to train an agent, we build an efficient neural network library. We implement and test a computational graph approach as well as a layered one. We settle on the layered approach as it proves to be far more efficient and use it to successfully train a deep Q network to play Flappy Bird.

2 BACKGROUND

Reinforcement learning is an area of machine learning where agents/policies are trained to output the appropriate action/value to accomplish a given task. To train an agent, data must be collected. In the case of RL, this is done in an RL loop. This loop consists of an agent first taking in an observation, which is the state of the environment that the agent is in. From this observation, the agent produces an action. This action is then taken by the agent and the environment is updated to reflect the outcome of the action. Finally, a reward and observation are outputted. Rewards are often sparse and are simply large when an action accomplishes a task and negligible otherwise. The observation is fed back into the agent and the loop repeats. This loop is central to producing to producing the observations, actions and rewards from which an agent is trained.

There are multiple options for using this data to train an agent. One such widely used algorithm is Q-learning. This algorithm works with the assumption that states/observations follow the Markov property. This means that the state encapsulates all information about the environment that allows for the agent to find an optimal action.

Q-learning is centered around the Q function. The Q function takes in the state and action and produces a value. The higher the values, the better the state action pair. In Q-learning, the goal is to learn this Q function. From this Q function, an agent can find the optimal action when given a state by finding the action that returns the highest Q value when paired with the state. The Q function is ultimately learned via repeated Bellman updates to the Q function. As the equation below shows, the Q value for the current state-action pair, is the max Q value for the next state multiplied by discount factor (γ), added to the reward that was returned for getting to the next state. Repeatedly running this update with collected returns an optimal Q function. For a DQN, this Q function is represented by a neural network and is updated via a gradient step. This gradient step is taken at the end of each episode on a batch of experiences saved from previous episodes. This buffer of saved experiences is called replay memory. The full algorithm for a DQN is described in figure 1.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

Figure 1: DQN algorithm

$$Q_{i+1}(s, a) = \mathbb{E}_{s' \sim e}[r + \gamma \max_{a'} Q_i(s', a') | s, a] \tag{1}$$

3 Related Work

Within deep RL, the area of research that is applicable to training a RL agent to play Flappy Bird is model-free deep RL. A seminal paper in this area is Playing Atari with Deep Reinforcement Learning by Mnih et al. (2013). It proposed using a convolutional neural network as a Q function approximator. This produced an agent that could play many Atari 2600 games better than humans simply by feeding in RGB frames and a sparse positive reward when the game was won or negative reward when it was lost. Later a paper by van Hasselt et al. (2015) proposed double Q learning. This added a separate neural network to predict the target value in the Q value update. Using a separate target network that is updated less frequently than the policy network adds stability in training. Another paper by Hausknecht & Stone (2015) and Stone proposed a neural network with an LSTM layer. This feature allows for the removal of frame stacking as is done in the DQN method proposed by Mnih et al. (2013). The LSTM holds temporal information and only requires single frame to perform on par with DQNs.

As an improvement to the traditional DQN network, Schaul et al. (2016) proposed prioritized sampling from the replay buffer rather than random sampling. They suggested prioritizing the samples based on the TD error for the samples. The intuition was that these samples contribute more to learning because they are more incorrect. This resulted in better performance on 41 out of the 49 games DQN was tested on. While the above papers show individual improvements to vanilla DQN, Hessel et al. (2017) highlighted how these can be combined to produce the in the most effective DQN implementation. The result was faster learning and increased sample efficiency. Another area outside of deep Q learning are actor-critic methods. Actor critic methods are both value based like q learning, but also policy based. The learning of the actor is done via policy gradients where there is a direct mapping from state to action. The critic is value based where it evaluates the action produced by the actor by computing its value function. The paper Mnih et al. (2016) suggested an actor critic method that allows for asynchronous learning. It proposed a method where multiple agents run in parallel environments and asynchronously update the learner network. This improves training speed by increasing the utilization of a multi core CPU. However, ultimately, we chose to train a vanilla DQN proposed by Mnih et al. (2013) because it works well on discrete action spaces and proves easier to tune.

4 Neural Network Library

To implement and train the neural network as a function approximator in Q learning, a neural network library had to be developed. The are many approaches to developing a neural network library, to two we investigated were the computational graph approach and the layered approach. Both have their strengths and weaknesses. We implemented both and benchmarked them by training a DQN to successfully solve the Cart Pole problem.

4.1 COMPUTATIONAL GRAPH APPROACH

The computational graph approach, as the name suggests, is building a graph that represents all the computations taking place in a neural network, then implementing backpropagation over the graph to compute the gradients for all values involved in the computations.

A computation is simply a mathematical operation between two values. In our case, these values are all floats. To represent these computations, we build a Value class in python to wrap values. The Value class contains a property for the float value, but also stores the operation between the two values that produced the float value. With this, we have stored half the information about the computation. The other half is the values that are operated on for the computation. In addition, we store references to the Values classes of the two values that produced the float value. So we end up with a Value class that has a float value, an operation, and the two Values that produced the float value. With this, every Value encapsulates a computation 1.

To create the computational graph to represent a series of computations, we overwrite the basic operators in the Value class. The operators we overwrite are add, subtract, multiply, divide, negation and exponentiation. We overwrite these operators, so when a Value is operated on with the operators listed above with another Value, the output is a new Value that points to these two parent Values. Repeatedly doing this will create a graph of Values where each Value is the child of the two parent Values that were involved in a computation.

The last feature to implement is backpropagation. It is the method to compute the gradients for each Value in the computational graph. Backpropagation is implemented by iterating from the final output Value to the inputs. While iterating, each Value computes the gradients for its parents. The child Value has the information of the parent Values, the operation applied between the Values, and its own gradient. It uses this information to find the gradient of each parent with respect to the operation and the Value of the other parent and multiplies it with its own gradient. This produces the gradient for each parent with respect to the one computation that produced this child Value. However, a problem remains because the parents may be involved in multiple computations. To solve this issue, the calculated gradient for each parent is simply added to its current gradient. This ensures that for every computation a Value is involved in, its gradient is not overwritten, but accumulated to find the total change (gradient) a Value can affect on the neural network output.

When implementing backpropagation as described above in the computational graph, the graph must be traversed in a specific order. Since the gradient of each Value depends on the gradient of its child Values, the gradient of the child Values must be completely calculated before moving to the parent. This is only possible in directed acyclic graphs, which neural networks are. To traverse the graph with the restriction imposed above, a topological sort of the directed acyclic computational graph must be found. In our implementation, this is done via a recursive method. We traverse over the nodes and compute the gradient in the order returned by sort.

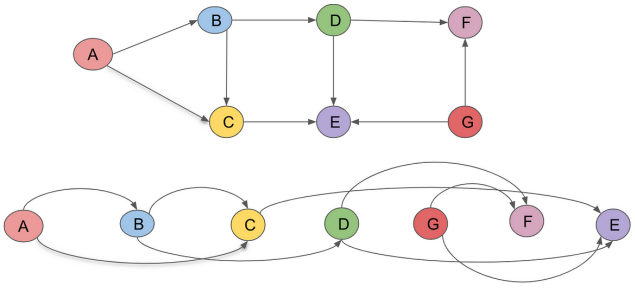

Figure 2: Topological sort

4.2 LAYERED APPROACH

The layered approach is the second approach we undertook. We did this with the expectation that this design would allow us to write the library with matrix operations to allow for parallelization and the utilization of efficient computational libraries such as numpy and cudapy. Numpy is a

Table 1: Value Class

Property	Description
val	A float value
op	A string describing the operation executed to produce the value
parents	The references to the two values that the operation was applied between to produce the float value
grad	The gradient for this value

scientific computing library that runs the matrix operations efficiently on the CPU. CudaPy is an implementation of numpy that runs on the GPU.

Our layered approach consists of a separate class for each type of layer. This varies from the computational graph approach in that each layer consists of a forward pass function to compute the output of the layer and a backward pass to compute the gradients. The computational graph had all functionality (i.e. forward pass and backward pass) at the individual value level rather than at the layer level. Doing this pass at the layer level allows us to compute all outputs, weight gradients and input gradients for each layer via matrix operations. This approach additionally allows for efficiency gains across another dimension, the batch size. The computational graph required passing in each input independently through the network. However, the layered approach allows for the all calculations to be done in parallel across all inputs that make up a batch.

The basic architecture of each layer is a forward pass method that takes in an batch of inputs and produces a batch of outputs, with each entry in the output batch corresponding to its entry in the inputs. The backward pass does two important calculations. The first is computing the gradients with respect to all the parameters (weights and biases) in the layer. The second is computing the gradients with respect to all the inputs. The input gradients are then returned by the backward pass and passed to the previous layer, so the previous layer has the gradients with respect to its outputs (outputs of previous layer are inputs of current layer). With this architecture, the backward pass is simply iterating over each layer, calling the backward method, getting the output and passing it to the next layer.

The type of layers we implemented in this approach are fully connected, convolution layer, max pooling, mean squared error and flatten. The implementation details for most of these layers are standard except for convolution and max pooling. These two layers have windowing, which is traditionally a serially implemented with for loops. However, the main benefit of this approach is parallelization via matrix operations, so we aimed to find a matrix approach. Luckily, both numpy and cudapy have an **as strided** function that allows for parallelized windowing by accessing the underlying bytes.

The **as strided** method works by requiring the final output shape after windowing across the inputs and the byte length of each dimension. So for windowing across a 3x3 matrix input of floats with a 2x2 filter of stride 1, the output shape would be 2x2x2x2. From right to left, the first 2x2 is the dimension of the filter, which is the window created by the filter. The third and fourth 2 describe the number of times the filter windows horizontally and vertically, respectively. For each dimension, the length in bytes must be specified. Again from right to left, the every value in the first dimension is 8 bytes apart because in numpy a float is 8 bytes (64 bits). In the second dimension, every row is 24 bytes apart since the matrix is has width of 3 floats. The third dimension is the byte difference between strides horizontally. In our example, the stride is 1 so the byte length is 8 for this dimension. The last dimension is the difference between windows that are vertically shifted. Again, since the stride is 1, the byte difference between vertical windows is size of a row, which is 24 bytes. For this example, the **as strided** parameters for the output shape are (2, 2, 2, 2) and the byte lengths for each dimension are (24, 8, 24, 8).

4.3 Comparison

We used both implementations to train a DQN to solve the Cart Pole problem. We used the neural network architecture below with a mean squared loss function. In addition, we used a discount factor

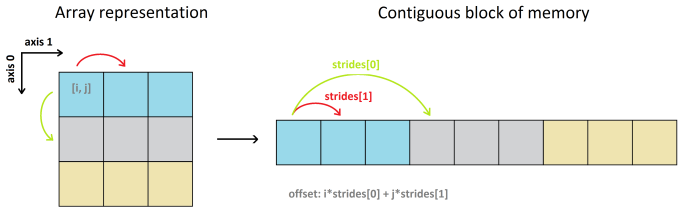

Figure 3: Striding for numpy arrays

Table 2: Neural Network Library Computational Speeds

Method	Forward Pass (ms)	Backward Pass (ms)	Total Time - 1000 iterations (s)
Computational Graph	2.47	0.66	99.95
Layered (numpy/CPU)	0.011	0.048	1.57
Layered (cupy/GPU)	0.31	0.57	28.35

of (γ) of 0.95, a batch size of 32, a learning rate of 0.001 and epsilon decay rate of 0.995 and train for 1000 episodes.

Neural Network Architecture:

Fully Connected (in: 2, out: 24) $\rightarrow$ ReLU $\rightarrow$ Fully Connected (in: 24, out: 24) $\rightarrow$ ReLU $\rightarrow$ Fully Connected (in: 24, out: 2)

The results of the training are below. Running the layered approach on the CPU increased computational speeds by over 100x while the running it on the GPU was only 3x faster than the computational graph approach. From this, we decided to use the layered library to train our Flappy Bird agent.

5 METHOD

Flappy Bird is a game that requires the player to have a bird looking sprite navigate through gaps in pipes. The sprite has a constant x velocity as it moves towards the pipes. The player has the ability to apply a force in the y direction. The goal is for the player to time the application of the force to navigate through the gaps between the pipes for as long as possible.

For Flappy Bird, we used an environment following the gym API to train and evaluate our agent. This environment's observation space is the x and y difference between the bird and the gap in the next pipe (2 dimensional vector). The action space is one dimensional, with 0 applying no force and 1 applying a set force. For our neural network, our output space is 2, with one dimension for each value. The reward was very simple with a value of 1 returned every time step the bird did not crash. We had to modify the environment slightly for more efficient training by considering going above the screen as a crash. This allowed for more quickly learning to get passed the first pipe and exploring more varied scenarios. One thing worth pointing out is that the Markov assumption does not fully hold for the state because the velocity of the bird is not included. We only have positional information. However, this proves to be sufficient, but may reduces the efficieny of our training.

For our agent, we used the neural network architecture below with a mean squared loss function. The discount factor (γ) was 0.95, the batch size was 32 and the learning rate was 0.001 like our Cart Pole agent. The replay memory size was 2000 and the epsilon decay rate was set according to the equation below, so our chance of taking a random action would decay appropriately as we train for various number of episodes. In addition, we modified our sampling of random actions to be weighted 75 percent towards applying no force and 25 percent towards applying a force. This allowed for more efficient exploration of the space because apply a force as often as not applying one quickly results in crashes in the beginning.

Epsilon Decay Equation:

$$epsilon\_decay = \sqrt[num\_episodes]{\frac{0.01}{\epsilon}} \tag{2}$$

MLP Architecture:

Fully Connected (in: 2, out: 64) → ReLU → Fully Connected (in: 64, out: 128) → ReLU → Fully Connected (in: 128, out: 2)

6 RESULTS

After training for 576,000 episodes (2 hours on a personal PC), the network converged at a solution for an agent that averaged over 2000 time steps when playing Flappy Bird. This results in an average score of navigating through 30 pipes. While this is not super human performance, it shows that the agent can play the game. With more training time, the agent would be able to play at super human levels. The graph below (figure 4) plots the average episode length of running the agent for 10 episodes after every 1000 episodes of training. The agent can be found and run here (`https://github.com/sujaygarlanka/567-final-project`). All code for the Cart Pole experiments and neural network implementations can also be found at the preceding link.

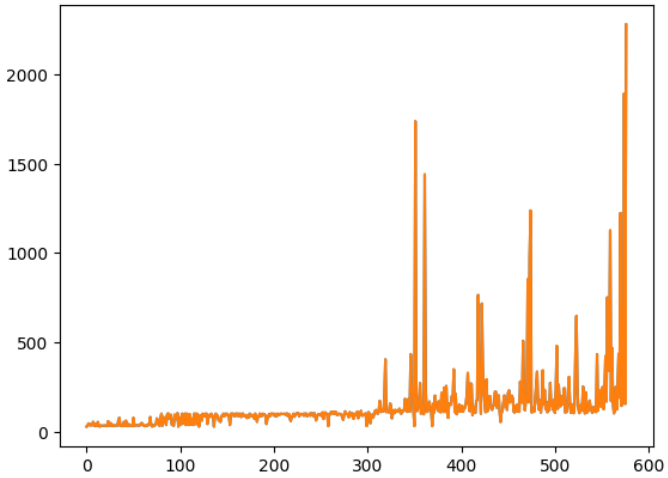

Figure 4: Episodic evaluation of Flappy Bird agent

7 FUTURE WORK

In the future, we hope to get an agent that plays Flappy Bird from training only on the RGB information of the environment. Our progress on this was fully implementing efficient convolutional layers and the network architecture below for doing so. We did not use max pooling layers because it can cause translation invariance where the position of objects like the bird is lost. In addition, we implemented the pre-processing of RGB inputs where we would take in 288 x 512 images over four actions, downsample, crop to 84 x 84, convert them to grayscale and concatenate them as done by Mnih et al. (2013). However, do to time limitations and the efficiency of our library, we were not able to train a sufficiently capable agent. The code for this can be found here (`https://github.com/sujaygarlanka/567-final-project`). Our numpy implementation of a neural network library was not sufficiently efficient and neither was the cupy drop in version that ran things on the GPU to train in a reasonable time frame. We need a library that utilizes the GPU like our cupy version, but keeps the weights on the GPU more often so time isn't lost shuffling them back forth between the CPU.

CNN Architecture:

Conv (in: 84x84x4, filter: 8x8, stride: 4, num filters: 32) → ReLU → Conv (in: 20x20x32, filter: 4x4, stride: 2, num filters: 64) → ReLU → Conv (in: 9x9x64, filter: 3x3, stride: 1, num filters: 64) → ReLU → → Flatten(in: 7x7x64, out: 3136) → Fully Connected (in: 3136, out: 512) → ReLU → Fully Connected (in: 512, out: 2)

REFERENCES

Matthew J. Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. *CoRR*, abs/1507.06527, 2015. URL http://arxiv.org/abs/1507.06527.

Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning, 2017.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. 2013.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2016.

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015. URL http://arxiv.org/abs/1509.06461.