Advancing Deep Learning With Neuromorphic Computing: Review and Survey of Results

Giovanni Michel

***Abstract*—We explore where the deep reinforcement learning revolution began, when Google's DeepMind's deep Q-Networks was able to play Atari games. Before get into the actual deep RL algorithm and framework, let's cover a general control task for RL. For example, let's consider the task to minimize the energy usage at a big data centers. Basically the computers need to be kept cool to function well, hence why large data centers incur significant costs from cooling systems. This task specifically doesn't require any complex machine learning. Essentially we will use the Gridworld game which is part of another family of similar games where they all generally involve a grid board with an agent as the player, and obstacles like walls, pits to make the player interact with the environment to receive points or rewards that can be either negative or positive.**

I. Introduction

CURRENTLY, Reinforcement Learning and Optimality in nature, is built around theories that originated in Psychology from watching living organisms interacting with it's environment and those interactions are measured by the good and bad things the organism experiences. The approach we will be using is using the Q Function to train a neural network to navigate the Gridworld board to some specified goal. The main goal is to use Deep Reinforcement Learning on the agent so that it can play the game with the lease amount of loss so that the rewards are maximized and the agent performs efficiently to without taking losses. Some main parts of the game that our important and you will need some background on are: the state, information that the agent receives and will be using to make decision on what action to take, in our case it will be the Gridworld, a tensor representing positions of all the objects on the grid. A policy denoted by pi, which will be the strategy our agent follows when transitioning from different states. The reward, which is the feedback the agent will receive after taking an action, leading us to a new state.

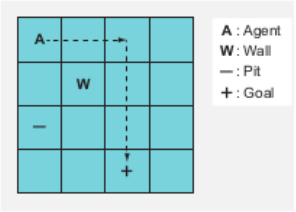

Fig. 1. Gridworld for DQN problem

To begin with, consider the implications of such a system; in which our agent makes a series of actions upon its policy pi and this repeats this process for every episode until the end. This will results in rewards, and we will call the weighted sum of the rewards starting from certain states

$$S_1$$

the value of that state.

$$V_\pi(s) = \sum_{i=1}^{t} w_i R_i = w_1 R_1 + w_2 R_2 + \ldots + w_t R_t$$

Fig. 2. Value Function with the weighted rewards

In the above figure we have coefficients w1, w2, etc, are the weights that will be applied to the rewards before summing them. This weighted sum is also the expected value, which is useful in quantitative fields, and is usually denoted by E[R—pi,s], which is read the "expected rewards given a policy pi and beginning state s". To answer the question, "What is Q-Learning" it is basically using the action=value function Q(s,a) which uses a particular method to learn the the optimal action values for a control task.

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max Q(S_{t+1}, a) - Q(S_t, A_t)]$$

Updated Q value · Current Q value · Observed reward · Max Q value for all actions · Step size · Discount factor

Fig. 3. The Q value at time t is updated to be the current predicted Q value plus the amount of value we expect in the future, given that we play optimally from our current state.

The update rule for Q-Learning is similar to that of the value function V(s) that we used for value iteration with the multi arm bandit algorithm. Essentially we probably could have implemented a NN for optimizing the ad placement of a website.

$$V_\pi(s) = \sum_{i=1}^{t} w_i R_i = w_1 R_1 + w_2 R_2 + \ldots + w_t R_t$$

Fig. 4. Value Function with the weighted rewards

II. Perceptrons and Sigmoid neurons

Lets give some information on what a neural network is, basically it is artificial neuron called a perceptron which was developed in the 1950s and 1960s by Frank Rosenblatt, inspired by earlier work. To be honest the main neuron model used is called a sigmoid neuron. What the perceptron does it takes several binary inputs, x1,x2,..., and produces a single binary output:

Where the neuron's output is either 0 or 1, which is determined by the weighted sum, that is less than or greater

Florida Atlantic University , USA e-mail: giovannimich2020@fau.edu.
Due Date: April 15, 2022

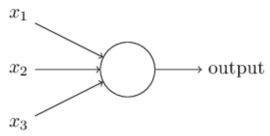

Fig. 5. Perceptron that has three inputs, x1, x2, x3

than some threshold values. This is how a perceptron can weigh up different kinds of factors in order to make decisions. But the perceptron has multiple layers, so to see how learning might work, all we need to do is make a small change to the weight in the network. Each small change in any weight will cause a small change in the output, sometimes it can change the output drastically from either 0 to 1 completely. That flip may then cause the behaviour of the rest of the network to completely change in some very complicated way. The way to solve this problem is by introducing a new type of artificial neuron called a sigmoid neuron. Sigmoid neurons are similar to perceptrons, but are different because small changes in their weights only cause a small change in their output, which makes different compared to traditional perceptrons. Just like the perceptron, sigmoid takes multiple inputs, x1,x2,... .

$$\sigma(w \cdot x + b)$$

Fig. 6. sigmoid function*

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}$$

Fig. 7. sigmoid function definition

To understand the similarity to the perceptron model, suppose zwx+b is a large positive number. Then e-z=0 and so (z)1. In other words, when z=wx+b is large and positive, the output from the sigmoid neuron is approximately 1, just as it would have been for a perceptron. Suppose on the other hand that z=wx+b is very negative.

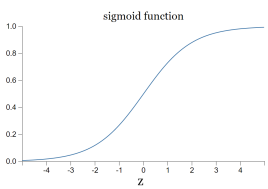

Fig. 8. sigmoid function shape visualized

III. Understanding the Q Network

It is important that we point out the key characteristics that we will be using to make this architecture or network. Luckily since we are using Gridworld we can get away with simply implementing a feedforward neural network with only a few layers, while using the rectified linear activation unit (ReLU). And on top of that we will also need to find a way of representing our input data, and the output layer. Essentially, we want to approach this problem with the Q function because it can tell us the value of taking an action in a state, so we can take the action that will result in the highest value. Just like DeepMind's implementation of deep Q-learning, we want to make the Q function as a vector-value function, where instead of returning a single Q value for a state-action pair it would return all the Q values for all actions possible from the given state-action pair. This might sound difficult but it easy to represent a vectored version of it given that there are only four possible actions (up,down,left,right). Meaning the output layer of the neural network will make the Q function more efficient, since you only need to compute the function once for all the actions.

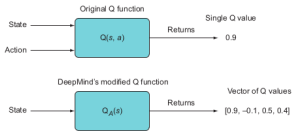

Fig. 9. Original Q function compared to the vectored version DeepMind used.

The above figure shows the original Q function that accepts a state-action pair with its output. And it shows the DeepMind's modified Q function that outputs all the expected Q values for that given state-action pair that it receives. We will use this output of the neural network to decide on what action to take using some action picking procedure, for example we can use a simple epsilon-greedy approach or a softmax selection policy for picking the actions to use.

Fig. 10. Output of the neural network using epsilon greedy for the selection process

For this experiment and the sake of simplicity we will only have a simple neural network with just three layers with widths of 164 (input layer), 150 (hidden layer), 4 (the output layer).

IV. Experiment Parameters and Dimensions

Now we will explain how the gridworld game will be represented. The state is gonna be represented as a 4x4x4 tensor where the first dimension represents a set of four matrices of size 4x4, similar to it having a height by width dimensions. And each matrix is a 4x4 grid of zeros and whenever a 1 is shown indicates the location of a particular object on the grid, like the the player, the goal, the pit, and the wall.

When looking at the figure above you can tell that first matrix encodes the position of the player, the second matrix encodes the position of the goal, the third matrix encodes the position of the pit, and the last matrix encodes the position of the wall.

Fig. 11. DeepMind's vectored output of the Q function with the predicted Q values given a state action pair.

```
array([[[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 1, 0],
        [0, 0, 0, 0]],

       [[1, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]],

       [[0, 1, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]],

       [[0, 0, 0, 0],
        [0, 1, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]]], dtype=uint8)
```

Fig. 12. Matrix representation of the grid game

V. Training of the Neural Network and Experiments

After covering the different parts of the network we will now cover how we will train the model to play the gridgame. The goal is to use a cost function, C(w,b) to quantify how well we are achieving this goal. Here we will cover the results of our experiment and if our model was able to learn to play the gridworld game without taking negative rewards. Here we will compare the results received from the loss graph and monitor the result over the epochs vs loss. We will cover one case, where the agent starts stationary in the map. Here we will show the results received from the agent starting in one of the states and what our action values needed to get to the goal. We will focus on the loss graph over the epochs, and how it converges to zero over time.

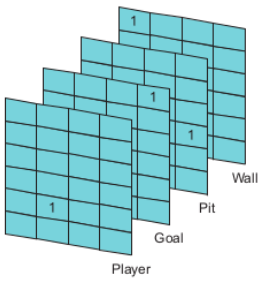

Fig. 13. This is how the Gridworld board is represented as a array. Which is a 4 x 4 x 4 tensor, composed of 4 "slices" of a 4 x 4 grid.

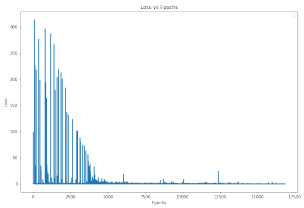

Fig. 14. Value Iteration Algorithm

In regards to the agent dealing with a static environment, the agent actually performed extremely well. It was able to converge after a certain amount of epochs which shows that it was able to learn to navigate to the correct goal, with minimal losses. The output vector of the Q function can be shown as follows where the agent shows the actions to take that will lead it to the goal. value iteration algorithm, the update step is very similar to the update step in the policy iteration algorithm. The only difference is that we take the maximum over all possible actions in the value iteration algorithm. Instead of evaluating and then improving, the value iteration algorithm updates the state value function in a single step.

```
Initial State:
[['+' '-' ' ' 'P']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move #: 0; Taking action: l
[['+' '-' 'P' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move #: 1; Taking action: d
[['+' '-' ' ' ' ']
 [' ' 'W' 'P' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move #: 2; Taking action: d
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' 'P' ' ']
 [' ' ' ' ' ' ' ']]
Move #: 3; Taking action: l
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' 'P' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move #: 4; Taking action: l
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 ['P' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move #: 5; Taking action: u
[['+' '-' ' ' ' ']
 ['P' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move #: 6; Taking action: u
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Game won! Reward: 10
True
```

Fig. 15. Value Iteration Algorithm

VI. Conclusion

After do research on this project, I was able to learn a lot about using tensors in training a model. I had the opportunity to, learn about training the model. And how the Q function must be learned from the current state, and how the neural network takes an input Q function and outputs a vector of all the best moves for that state. Essentially in this case learning the Q function is like training a neural network with backpropagation. To continue this research, we could have a implemented a algorithm that includes experience replay so that we can cover cases of when the state it is not stationary

after every epoch, where instead the environment is random and always changing. We could also have implemented a Q learning algorithm using on-policy learning instead of using the off-policy that we did earlier. Let us just reiterated what a DQN is, simply where we use a deep learning algorithm as the model in Q-learning. And the off-policy learning is when we learn a policy while collecting data using a different policy. Going forward we can also do research on catastrophic forgetting, where the new data being learned erases or corrupts the old information already learned.