Deep Q-Learning with Rubik's Cubes

Thomas Kwashnak

May 7, 2023

Contents

1 Abstract

Deep-Q Learning is a form of Reinforcement Learning that utilizes a Q-Function and a Target Function to make an agent explore and learn within an environment with a set goal. In my project, I am using Neural Networks as the Q-Function, training them to learn how to solve a Rubik's Cube.

This particular project is about a year in the making, and has had many failed attempts. Build a neural network isn't a one and done deal, and takes lots of time and thought to make one work. Thus, this report consists of some discussion, analysis, and thoughts I have with a few of the models I created during my most recent rewrite of the program.

Through these models, I was able to discover mistakes in my code, and make assumptions as to which hyperparameters have more of an impact than others. My conclusion is the fact that there is no conclusion to be made at this point in time for the project, as a successful model has yet to be built.

2 Background

2.1 Neural Networks

Neural Networks are one of the core pieces of machine learning. Neural Networks are complex formulas that you are able to train by providing inputs and expected outputs, and the model will try to best identify patterns in the data. Neural Networks are often used to classify images, text, and many other applications.

In short, a Neural Network is a function approximator. It achieves this through use of gradient descent with its many variables. Neural Networks may have upwards to $100,000$s of variables, consisting of weights and biases. To train these weights and biases, you need to provide pre-labeled data, or data that already has a predetermined answer.

First, the input data is fed into the network. The network will pass the data through the layers, and eventually output what it thinks the answer is. You then compare the result to the expected value, and square the result. The result will be your "loss" for that observation. Loss tracks how badly the network did in this scenario.

With loss, and the loss function, we can calculate the gradient for the entire function. The gradient consists of the partial derivatives for every single weight and bias in the network. Recall that these weights and biases are what determines the output. By finding the partial derivative of each variable as it relates to loss, we can take use the negative gradient to nudge each variable so that the loss decreases.

By performing this sequence multiple times, the network will start figuring out patterns in the data. Eventually, we can throw "unlabeled data" and see what it thinks about it. Additionally, there are some configurations that we can make with this process. We call these parameters "hyperparameters". Hyperparameters are parameters that appear to be arbitrary. In truth, many times we can't fully understand the effects of one hyperparameter on the overall process. As such, very little specific guidance is provided regarding how to set these variables. For a basic neural network, for example, hyperparameters could be the number of layers it has, how many nodes the network has in each layer, or the rate at which it learns[1].

2.2 Deep Q-Learning

Deep Q-Learning is a form of Reinforcement Learning that utilizes a Q-Function to train the model against. The goal of Deep Q-Learning is to explore an unknown environment with a clear goal, learning from prior experience to maximize reward.

Deep Q-Learning requires the user of an environment. In short, an environment contains observations, actions the agent can take, and some form of a reward system that drives the user towards the end goal. For this project, the environment is simply a Rubik's cube[2]. The observations consist of an array of values representing each tile face of the cube[3], the actions

[1] The value we multiple the negative gradient by before applying it to all of the variables

[2] Originally the environment was a standard 3x3 Rubik's Cube, however to simplify things I've recently switched over to a 2x2 Rubik's Cube, with intentions of returning to a 3x3 once I create a model that solves a 2x2 Rubik's Cube

[3] On a 3x3 Rubik's Cube, this would be each of the 9 smaller squares on each side of the cube

are each of the possible moves that can be performed on the cube, and the reward is the number of correct squares on each face of the cube[4].

In Deep Q-Learning, the logic that chooses the answer is called a Q-Function. The Q-Function consumes the current state of the cube,m and returns the predicted reward it thinks each move will give it. For example, if the cube is one move away from being solved, the reward for the move that solves the cube will be higher than all other moves.

In standard Q-Learning practice, the Q-Function is a table that records all states that the agent has seen, what actions they took, and what the resulting reward actually was. However, this is unfeasable for environments such as the Rubik's Cube. It is estimated that the Rubik's Cube has 43 Quintillion possible states. To make sense of that number, if you scramble the cube randomly, it is statistically impossible to reach a state that has already been reached before. Thus, a table isn't effective.

That's where Neural Networks come in! Neural Networks are great at approximating functions, like Q-Functions! In our implementation of Q-Learning we will use 2 Neural Networks. One is called the Q-Network, and the other will be called the Target network.

But why 2 networks? Aren't we only training one? Yes. We are only training one, but one of the problems with reinforcement learning is that there is no "answer" that we can train the network on. Thus, we need to make that answer. The goal with training is to make the Q-Network approximate the predicted reward of taking an action at State A to be equal to the reward of State B plus the Target Network's approximation for the best move from State B.

That... probably didn't make sense. It's a very complicated subject, and it took me a long time to figure out. Imagine you're playing chess, and you're looking 2 moves ahead. You then associate the first move with the reward you get after the following move. Similarly, we are training the Q-Network to associate an action with the actual reward plus the predicted future reward of that state[5].

In order to keep the target network up to date, every so often we take the target network and update it with the values of the Q-Network. In a perfect world, where the Q-Network perfectly predicts the future target network's result, every time we update the target network, the Q-Network will be looking one more step ahead.

There is a lot I left out in this background section, and a lot that was probably confusing. However, much of the analysis relies on some basic knowledge included here, but shouldn't be too difficult to understand once you figure out the basics.

[4]The reward function has been a tough point to figure out with this project. For now this method will work, but there are surely better reward systems out there

[5]And if there are people who are still confused at this point, I don't know how else to word it.

3 Project Structure

Currently, the project is split up into individual components. The following sections discusses the uses for each component and how they fit into the bigger picture. Originally, this project was just a large project written in python, but I have since split it up during a complete re-write that started at the beginning of the Spring 2023 Semester.

3.1 Rust

In prior iterations of the project, I was never quite satisfied with how efficient the emulation of the Rubik's Cube was in python. Many times it ended up being a bottle-neck for the entire system. Thus, I decided that I would write the environment in Rust.

This project uses the PyO3 library for Rust, which provides the ability to compile Rust into a Python module, allowing me to simply import and use it directly in python. The benefit of this is that all of the logic performed with the cube is done in a memory-save, type-safe, efficient language that runs at the speed of C.

Furthermore, I decided to also implement the replay database in Rust, giving me even more optimizations when it comes to storing and fetching samples to train on.

3.2 SQL

I made some mistakes in the original implementation. Originally, I was using large JSON files to store all of the collected data for analysis. This had a few negative effects. Firstly, the data was unoptimized. Secondly, it required me to load the entire file into ram every time I wanted to add another value[6]. Lastly, there were some additional constraints with the storage of numbers within data file.

Thus, to solve these problems I decided to use SQL to store everything. I set up and configured a Microsoft SQL Server running on a Docker Container. This solved many of the problems above, and also had the additional benefit of being accessible by any of my devices[7].

3.3 Python

The last and most important part: Python. This project uses the Tensorflow library to simplify many of the machine learning and neural network functions. However, I avoided the use of Keras or other all-encompassing modules as I wanted to learn how it works before using someone else's code.

[6]This quickly built up over time, especially when I started leaving the program running overnight and throughout all of my classes

[7]This is achieved through the use of Tailscale, which creates a private virutal network between all of your registered devices

4 Analysis

This analysis will walk through each of the models I created within the last month. I will present and describe graphs discussing the training history, and the conclusions I came to for each model or group of models.

4.1 Model 2043

Model 2043 was the first model after many test attempts that I was able to run. While it wasn't anywhere near successful, it provided me insight into what kinds of data and efficiency I could expect.

4.1.1 All Epochs

Figure 1: Average Loss vs Epochs for Model 2043

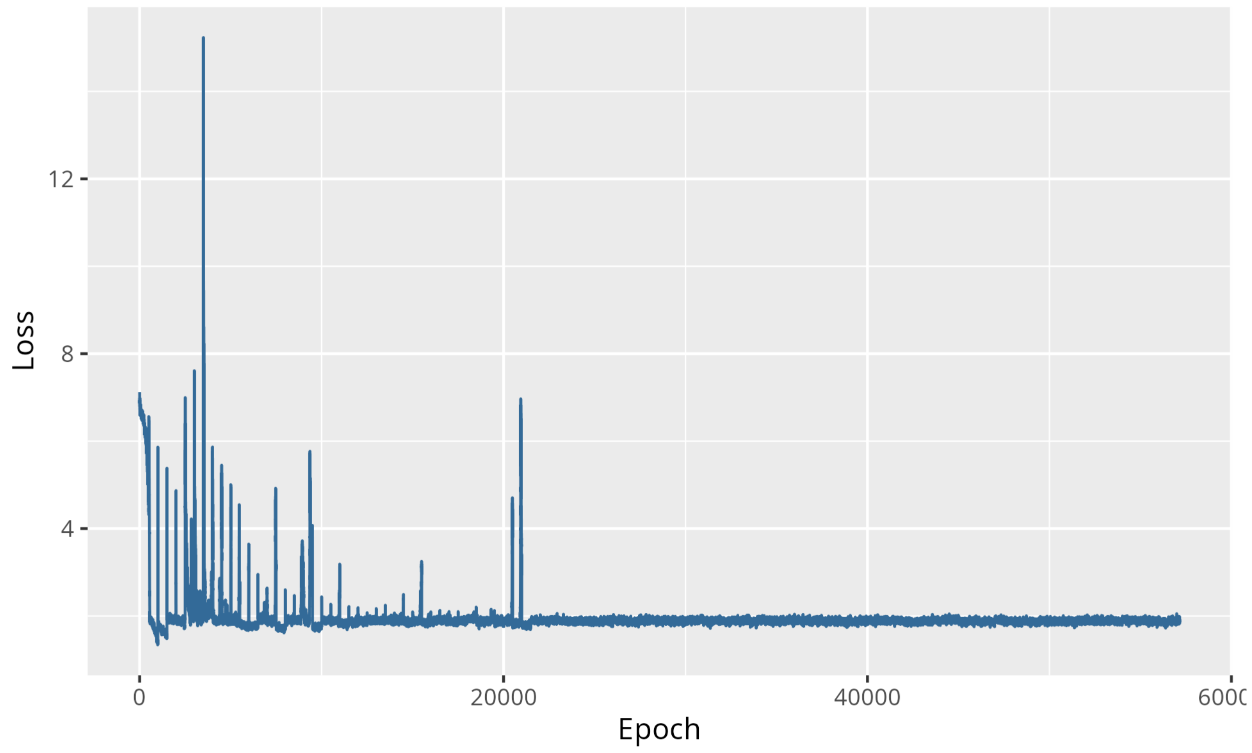


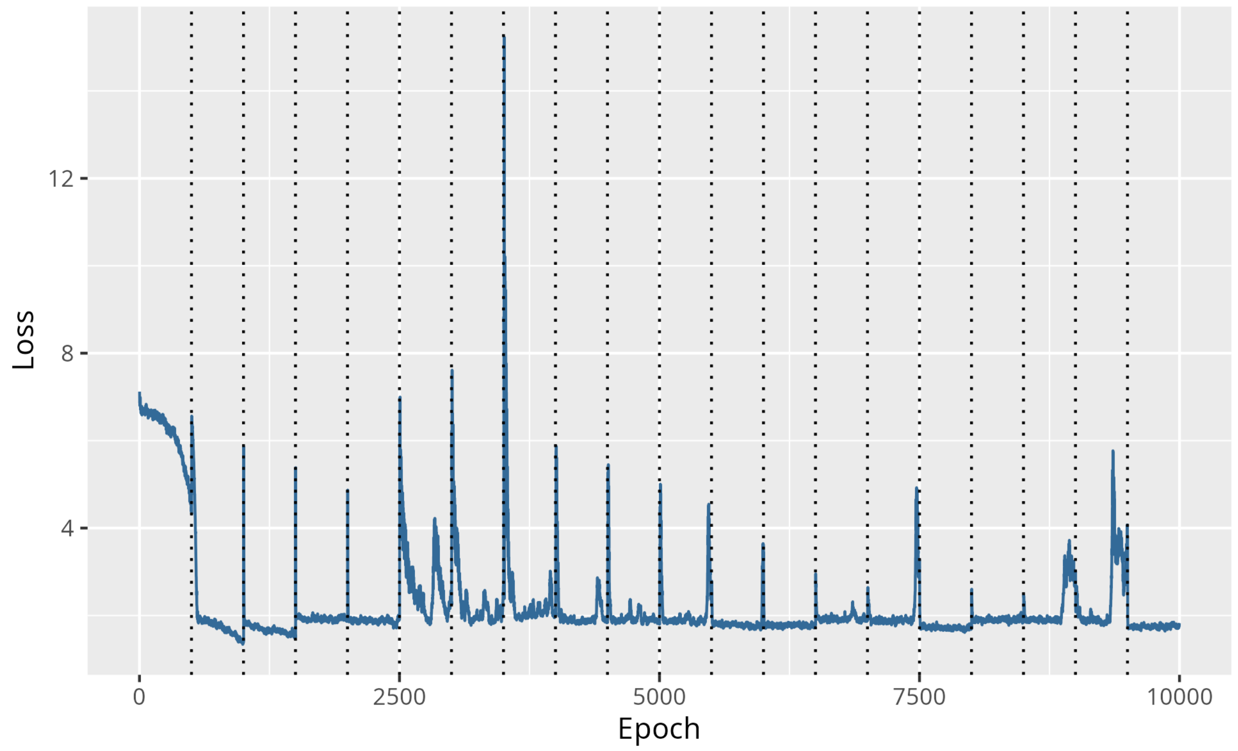
Figure 1 displays a complete graph of the loss of each epoch training cycle for Mode 2043. We can use this to look at overall trends within the training. Once again, the loss is the average difference between what the Q-Function obtained for the current state versus the reward for the next state and the target network's predicted evaluation of the next move.

Note that in Figure 1, there is a lot of spikes towards the beginning, but smooth out later on. This tells us that there is a point where the model cannot learn anything more with it's current configuration.

4.1.2 First 10,000 Epochs

Let's take a look at the first $10,000$ epochs to see what we can learn from the repeating spikes in loss.

Figure 2: First 10,000 Epochs for Model 2043



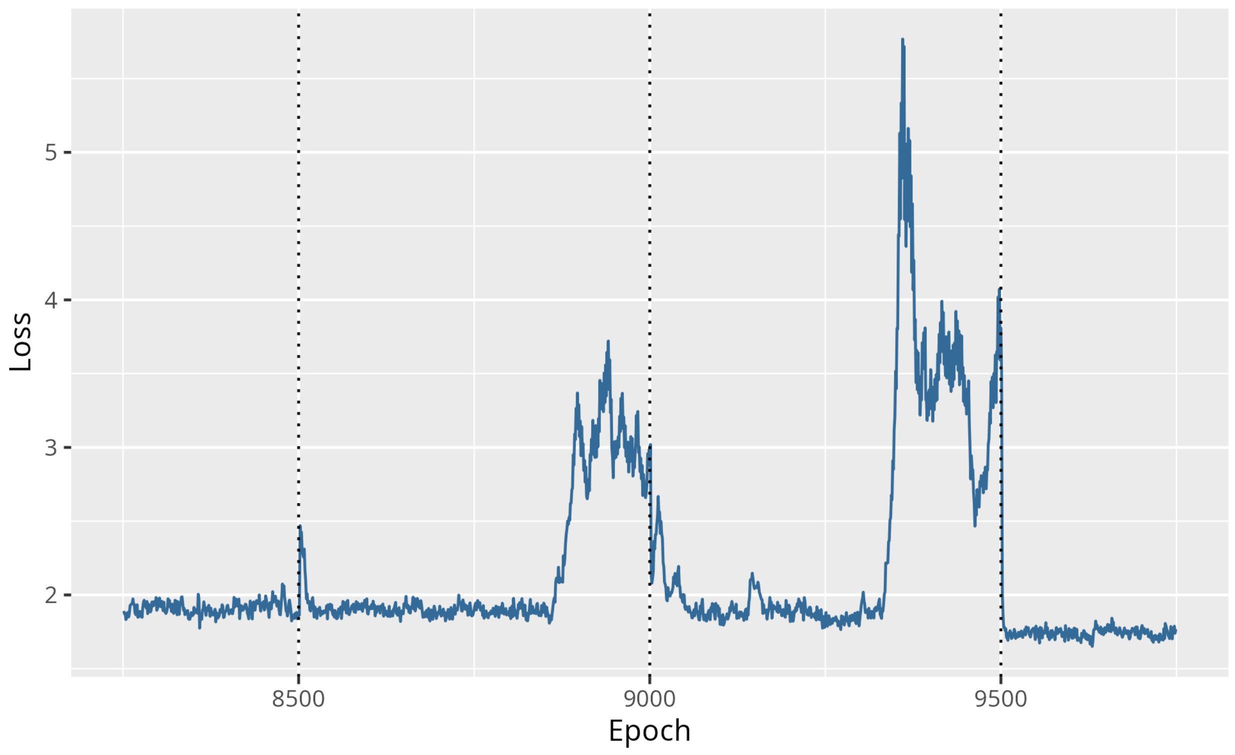
The dotted lines in Figure 2 indicate when the target network is updated. That would explain why we have large spikes in loss righ after each one. By upating the target network, we are changing what the Q-Network is being trained against. This means that the Q-Network needs to start training towards a new goal all over again.

This also makes sense why these spikes eventually dwindle down. As the model trains more and more, we can assume that it will hit a cap for its ability to think about the complex nature of the rubik's cube. This doesn't mean that it has learned how to solve a rubik's cube, but rather that it's learned as much as it possibly can.

4.1.3 Odd Behavior

If you look closely towards the tail end of Figure 2, you might notice that some of the spikes do *not* correspond with the update interval. Let's take a closer look.

Figure 3: Epochs 8250 - 9750 for Model 2043



This is odd behavior indeed. In Figure 3, we see multiple instances where the loss spiked up *before* the target was updated. This shouldn't make sense because the network has no idea when the target is going to be updted, nor have any ability to act on it. Something in the code must be wrong.

4.1.4 Incorrect Code

After some investigation, I found that the code I wrote for calculating Loss was incorrect. For review, our general equation for loss is as follows:

$$\text{Q-Network(State A)}_{\text{Choice}} - \text{Reward} - \gamma \cdot \text{Target(State B)}_{\text{max}}$$

When I looked in the code, however, I found the following snippet:

```
loss_raw = (
        tf.reshape(
                output_2_gathered_scaled,
                 ↪  (output_2_gathered_scaled.shape[0], 1)
        )
        - output_1_gathered
        - reward
)
```

The simple fix that I implemented for the following models is as follows

```
loss_raw = (
        tf.reshape(
                output_2_gathered_scaled,
                 ↪  (output_2_gathered_scaled.shape[0], 1)
        )
        - output_1_gathered
        + reward
)
```

With this discover, we can conclude that Model 2043 is not doing anything correctly, as far as we know.

4.2 Models 2044-2046

Models 2044, 2045, and 2046 all have similar traits. The only differences between them were small changes with the hyperparameters. Therefore, we will combine them and look at their collective information to try and pull some data out.

4.2.1 Inter-Model Changes

The following table shows us some of the changes made to hyperparameters between each model, giving us a sense of what differences might arise from.

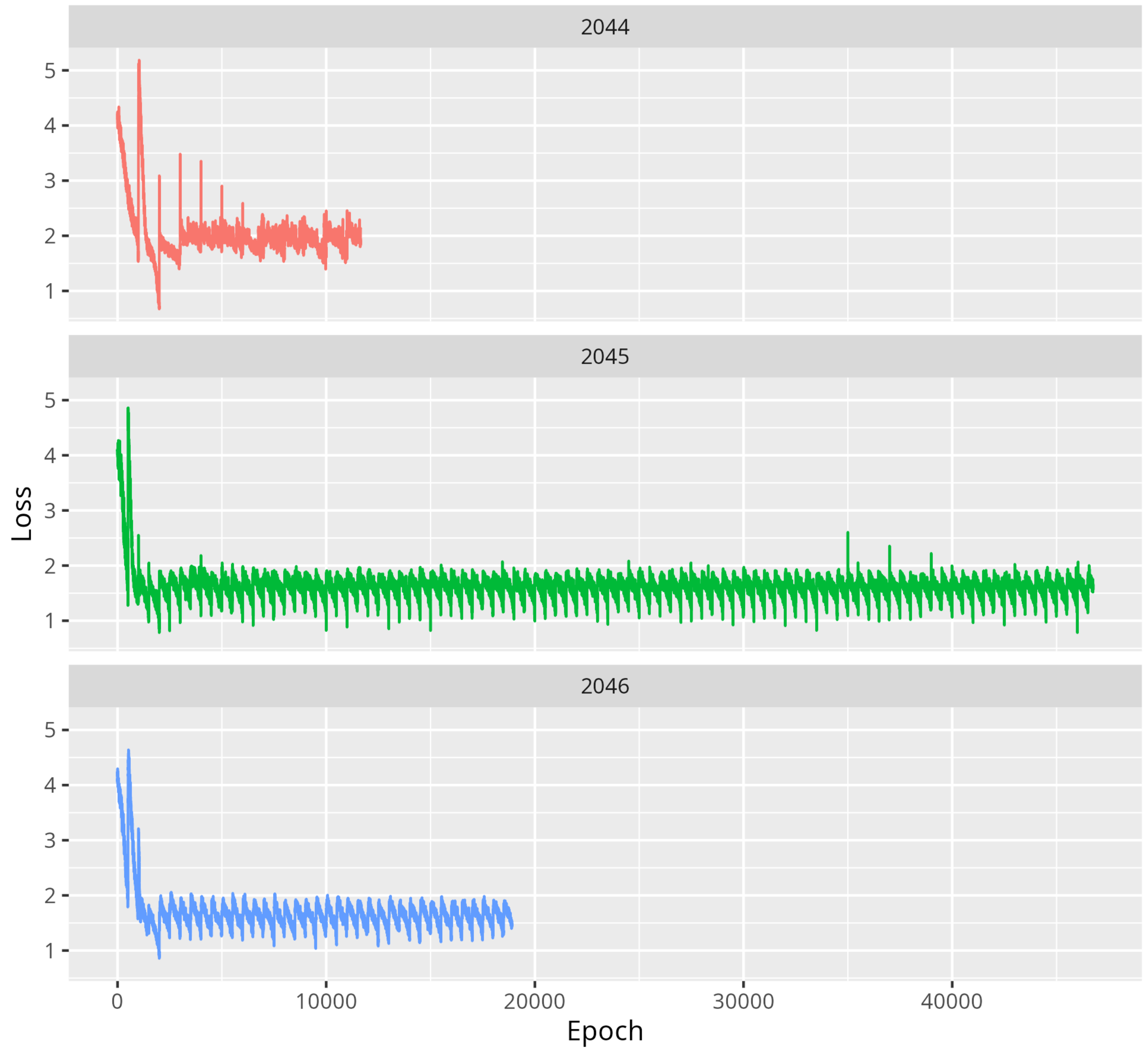
	Model 2044	Model 2045	Model 2046
Hidden Layer Sizes	$300, 250, 200, 100, 50$	$300, 300, 300$	$288, 144, 100, 50$
Replay Size	$10,000$	$2,000$	$10,000$
Gamma	0.9	0.7	0.7
Update Interval	1000	500	500
Learning Rate	No Change	No Change	Decreases slower

We will be referencing changes noted in this table as we compare each model in the following graphs.

4.2.2 All Epochs

Let's take a look at all of each model's data.

Figure 4: Epochs vs Loss for Models 2044, 2045, and 2046

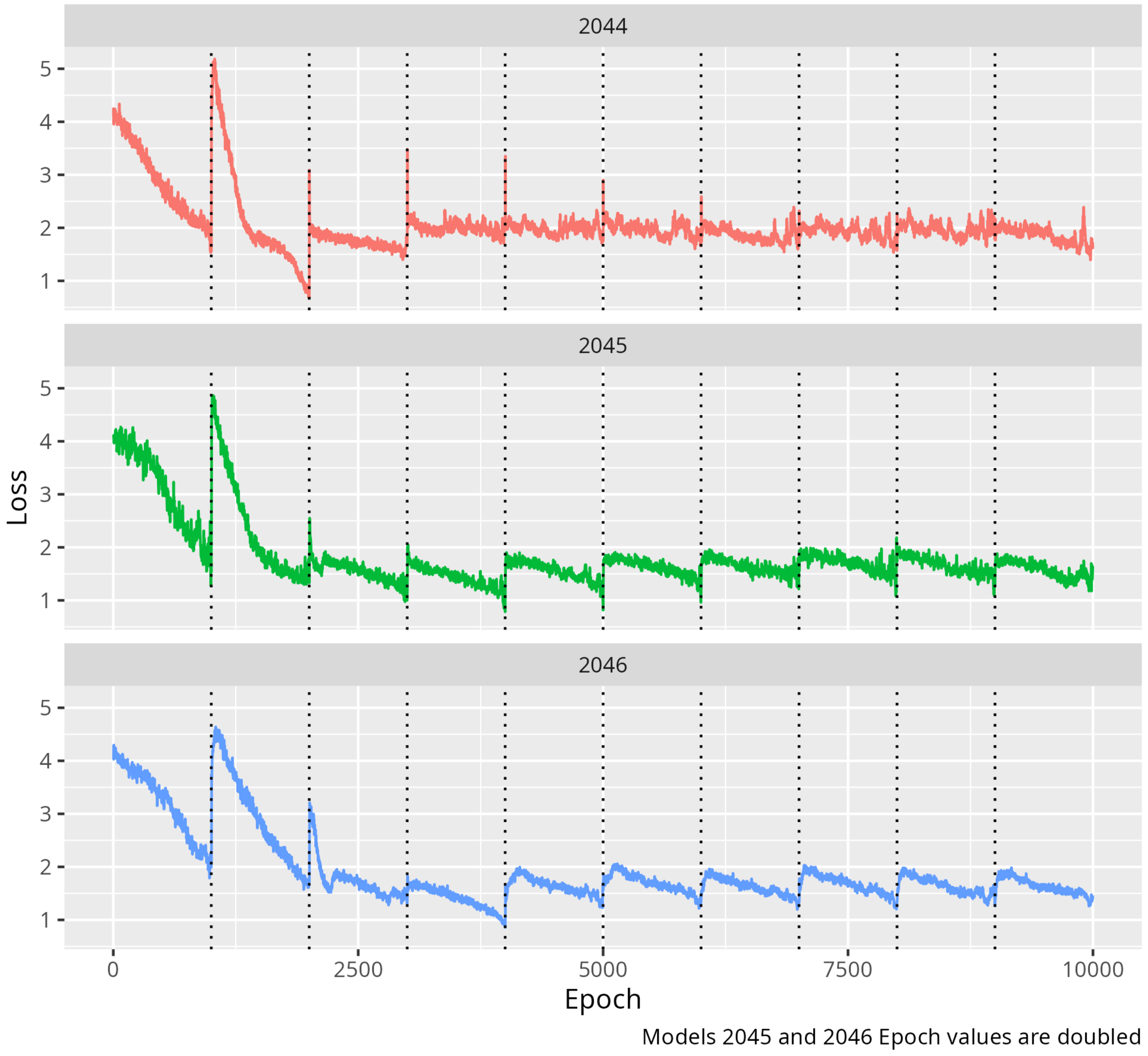


Some initial observations that can be made is that all three models eventually started repeating some cycle. This cycle is most likely related to the target update intervals. Apart from that, not much insight can be drawn from this information.

4.2.3 First 10,000 Epochs

Now, let's take just the first 10,000 Epochs of each model so we can better see the similarities and differences. The only caveat is that not all of the models have the same target update frequency. Model 2044 updates its target every 1000 epochs, while Models 2045 and 2046 update every 500. In order to better relate the cycles, Models 2045 and 2046 both have their Epochs doubled so that their cycle lines up with 2044.

Figure 5: First 10,000 Epochs of Models 2044, 2045, and 2046



All three of the models have very similar initial patterns. They all have significant downward momentum until the first epoch. This process repeats every target update, but the range shortens every time. Notice how Model 2044 doesn't even seem to have the spikes in loss after 5 target updates.

Model 2044 had more layers and a higher γ than 2045 and 2046. Therefore, we could assume that those two attributes might have an impact on how it ended up learning. I predict that the change in γ had the most profound effect, as γ dictates how much the network weighs the "future expected reward" returned from the target network.

5 Conclusions

This[8] project isn't the kind that a final conclusion can be had at any point in the middle. While I discussed a lot of my thoughts about what things *could* mean, there is absolutely no way of knowing for sure if my conclusions are correct. Neural networks are black-box items, so there is no definitive way to crack open the box and understand what it's doing. However, we can learn about the impact levels that different hyperparameters have, such as gamma (γ), or the size of the neural network[9].

This summer I plan to continue my research into hyper-parameters, trying out different things such as larger network sizes or different setups to see if I could try to inch my way closer to having a network that can solve a 2x2 Rubik's Cube.

[8]Some models have been omitted from this report in the interest of not spending many more hours dissecting the information, and also to keep this report from being 30-40 pages. You're welcome.

[9]This summer I am going to literally crank this up till my computer cries to see what will happen

A SQL Code

A.1 Tables

A.1.1 Model

This table is used to differentiate between different models, as each model will have different configurations, sizes, and other features that need to be kept separate. This table consists of both historical records parsed from JSON and new models that are used in current analysis.

```sql
CREATE TABLE Model
(
    ModelId   int identity primary key,
    ModelName VARCHAR(100),
    GitHash   VARCHAR(40),
    CubeType  VARCHAR(50)
)
```

A.1.2 Epoch

The Epoch table stores all of the epoch data collected during training

```sql
CREATE TABLE Epoch
(
    EpochId int identity primary key,
    ModelId int not null foreign key references Model (ModelId),
    Epoch   int not null,
    Loss    float(53),
    Reward  float(53)
)
```

A.1.3 Evaluation

The Evaluation table stores related data for evaluation runs by the model. The primary function of the Evaluation table is to keep a record of when the model starts making varied moves (instead of repeating the same move over and over).

```sql
CREATE TABLE Evaluation
(
    EvaluationId int identity primary key,
    ModelId      int not null foreign key references Model (ModelId),
    Epoch        int not null,
    Solved       bit,
    MoveCount    int,
    Seed         bigint,
)
```

A.1.4 EvaluationMove

The Evaluation Move table contains each individual move made during evaluations. Each move is tied to an evaluation, a move index, and stores the move and its reward.

```sql
CREATE TABLE EvaluationMove
(
    EvaluationId INT NOT NULL FOREIGN KEY REFERENCES Evaluation
 ↪  (EvaluationId),
    MoveIndex    INT NOT NULL,
    MoveName     VARCHAR(10),
    Reward       FLOAT(53),
    PRIMARY KEY (EvaluationId, MoveIndex)
)
```

A.1.5 Network

Acts as an identification item for each unique network stored in the Bias and Weight tables.

```sql
CREATE TABLE Network
(
    NetworkId INT IDENTITY PRIMARY KEY,
    ModelId   INT FOREIGN KEY REFERENCES Model (ModelId),
    Epoch     INT,
    IsTarget  BIT NOT NULL DEFAULT 0
)
```

A.1.6 Bias

Stores each of the biases within a network.

```sql
CREATE TABLE Bias
(
    NetworkId INT NOT NULL FOREIGN KEY REFERENCES Network (NetworkId),
    Layer     INT NOT NULL,
    X         INT NOT NULL,
    Bias      FLOAT(53),
    PRIMARY KEY (NetworkId, Layer, X)
)
```

A.1.7 Weight

Stores each of the weights within a network.

```sql
CREATE TABLE Weight
(
    NetworkId INT NOT NULL FOREIGN KEY REFERENCES Network (NetworkId),
    Layer     INT NOT NULL,
    X         INT NOT NULL,
    Y         INT NOT NULL,
    Weight    FLOAT(53),
    PRIMARY KEY (NetworkId, Layer, X, Y)
)
```

A.2 Users

A.2.1 Model

The user used when running and training the model itself. The Model user must be able to insert and update data, as well as remove old networks to preserve space.

```
CREATE LOGIN Agent WITH PASSWORD = 'MlCubeAgentPass1234';

CREATE USER Agent FOR LOGIN Agent;
GRANT CONNECT TO Agent;

GRANT INSERT ON Model TO Agent;
GRANT SELECT ON Model TO Agent;

GRANT SELECT ON Network TO Agent;
GRANT INSERT ON Network TO Agent;

GRANT SELECT ON Weight TO Agent;
GRANT INSERT ON Weight TO Agent;

GRANT SELECT ON Bias TO Agent;
GRANT INSERT ON Bias TO Agent;


GRANT INSERT ON Evaluation TO Agent;
GRANT INSERT ON EvaluationMove TO Agent;

GRANT INSERT ON Epoch TO Agent;

GRANT EXECUTE ON get_current_epoch TO Agent;
GRANT EXECUTE ON delete_network TO Agent;
```

A.2.2 Reports

The user used when building reports and graphs. This user only needs to be able to select and fetch data from various tables.

```
CREATE LOGIN Reports WITH PASSWORD = 'mlcube-reporting123'
CREATE USER Reports FOR LOGIN Reports;
GRANT CONNECT TO Reports;

GRANT SELECT ON Model TO Reports
GRANT SELECT ON Epoch TO Reports
GRANT SELECT ON Evaluation TO Reports
GRANT SELECT ON EvaluationMove TO Reports
GRANT SELECT ON GroupedEpoch TO Reports
GRANT SELECT ON EvaluationData TO Reports
```

A.3 Views

A.3.1 EvaluationData

A collected table from both the Evaluation and EvaluationMove tables. Used for fetching data for reports and analysis.

```
CREATE VIEW EvaluationData AS
SELECT ModelId, Epoch, Evaluation.EvaluationId Id, Solved,  Seed,
↪  MoveIndex, MoveName, Reward
FROM Evaluation
         LEFT JOIN EvaluationMove ON Evaluation.EvaluationId =
↪  EvaluationMove.EvaluationId
```

A.4 Procedures

A.4.1 get_current_epoch

Returns the current epoch for a specified model. Calculates by grabbing the last epoch recorded for that model.

```
create procedure get_current_epoch(@ModelId int)
AS
begin
    SELECT E.Epoch
    FROM Epoch E
    WHERE E.ModelId = @ModelId
      AND Epoch = (SELECT MAX(Epoch)
                   FROM Epoch
                   WHERE Epoch.ModelId = @ModelId)
end
```

B Report Graph Source Code

The following is the source code used to create the graphs found within this report.

```r
library(tidyverse)
library(ggplot2)
source("src/database.R")
source("src/functions.R")

file_name <- function(name) {
  return(paste("../SER-300-Report/assets/", name))
}

save_gg <- function(name, height = 4) {
  ggsave(file_name(name), width = 6.5, height = height, units = "in")
}

get_epochs(2043) %>%
  mutate(Loss = sqrt(Loss)) %>%
  ggplot(aes(x = Epoch, y = Loss, color = ModelId)) +
  geom_line() +
  theme(legend.position = "none")
save_gg("2043_0.png")

get_epochs(2043) %>%
  filter(Epoch < 10000) %>%
  mutate(Loss = sqrt(Loss)) %>%
  ggplot(aes(x = Epoch, y = Loss, color = ModelId)) +
  geom_line() +
  geom_vline(xintercept = seq.int(500, 9500, 500), linetype = "dotted") +
  theme(legend.position = "none")
save_gg("2043_1.png")

get_epochs(2043) %>%
  filter(Epoch > 8250 & Epoch < 9750) %>%
  mutate(Loss = sqrt(Loss)) %>%
  ggplot(aes(x = Epoch, y = Loss, color = ModelId)) +
  geom_line() +
  geom_vline(xintercept = seq.int(8500,9500,500), linetype = "dotted") +
  theme(legend.position = "none")
save_gg("2043_2.png")

get_epochs(2044,2045,2046) %>%
  mutate(Model = factor(ModelId)) %>%
  ggplot(aes(x = Epoch, y = Loss, color = Model)) +
  theme(legend.position = "none") +
  facet_wrap(~Model, ncol=1) +
  geom_line()
save_gg("2044-2046_0.png", height = 6)

get_epochs(2044,2045,2046) %>%
  mutate(Model = factor(ModelId)) %>%
  mutate(Epoch = case_when(Model == 2044 ~ Epoch, Model != 2044 ~ Epoch *
↪  2)) %>%
  filter(Epoch < 10000) %>%
  ggplot(aes(x = Epoch, y = Loss, color = Model)) +
  theme(legend.position = "none") +
  facet_wrap(~Model, ncol=1) +
  geom_line() +
  geom_vline(xintercept = seq.int(1000,9000,1000), linetype="dotted") +
  labs(
    caption = "Models 2045 and 2046 Epoch values are doubled"
  )
save_gg("2044-2046_1.png", height = 6)
```

C GitHub Repository

All source files, including the ones found in prior appendixes, can be found on the GitHub Repository. `https://www.github.com/LittleTealeaf/mlcube`