Deep Q-Learning with Rubik's Cubes

Thomas Kwashnak

May 7, 2023

Contents

1 Abstract

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.

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

3 Analysis

3.1 Historical Attempts

This project has been a work in progress for about a year, and I have collected data from prior models I tried. Before stepping into the models I created recently, it helps to look at what it used to be like.

For obvious reasons, only the last few historical models will be accounted for, as they are the closest to the most recent models.

3.1.1 Model 31

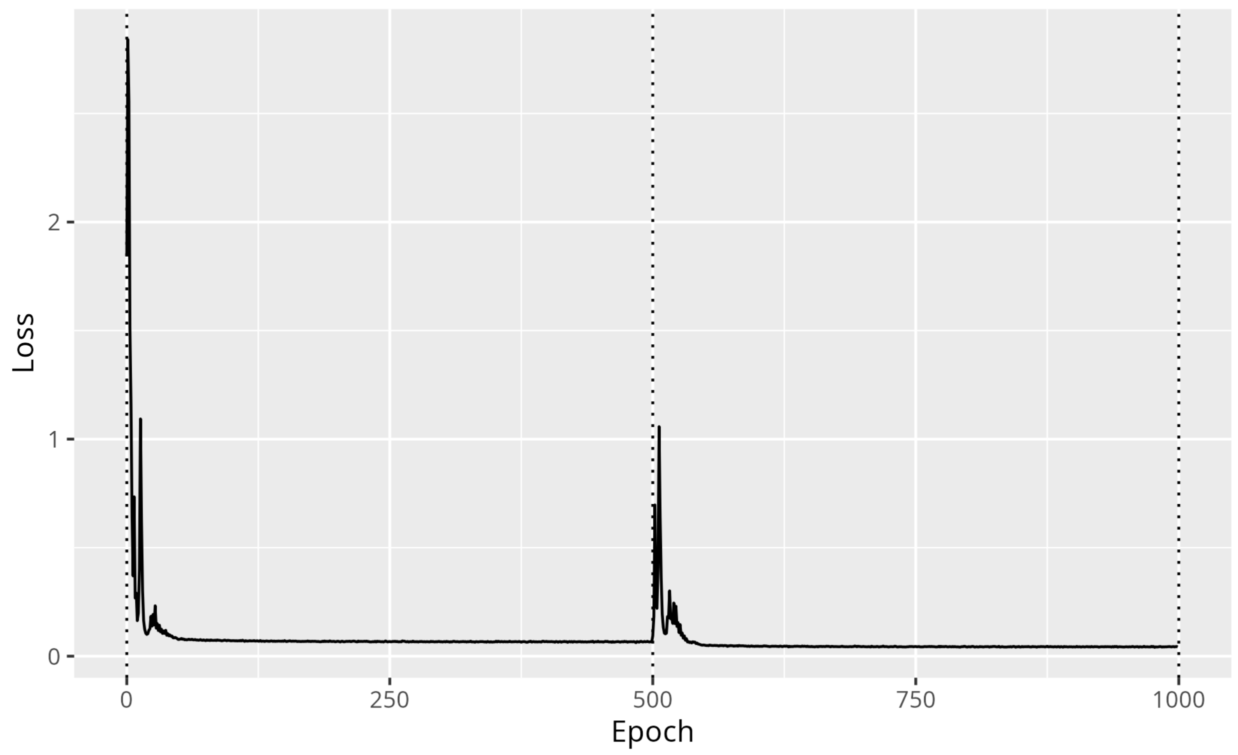

Figure 1: The Loss graph of main-6

This graph represents the average loss of the model at each epoch that it was trained on. The lower the loss, the better the result is. As described in the background, the model is being tested against the target's predicted reward of the following move. What this means is that while the graph at points may be very close to 0, it does not mean that the model is anywhere close to being able to solve a rubik's cube, but simply that it has gotten good at predicting what it would evaluate the next move in the sequence.

One of the first things to note is that the dotted lines indicate model updates. That is, each dotted line states that the target network was updated with values in the Q-Network at that point. The spikes confirm this, as whenever the target gets updated, the model is no longer being tested against the same thing.

However, what we see is that this particular model doesn't tend to update. It quickly reduces its loss to very close to 0, and then stagnates for the rest of the cycle. This type of behavior is indifferent to the overall success. It just means that the target must be so simple that it takes almost no time to learn how to replicate it. One possible reason for this could be that it assumes the reward of any move will be 0, so each move might be equally as good.

3.1.2 Model 39

Going forward a few iterations, I was trying out sequential models instead of trying to parallelize everything I could. In doing so, I got only slight differences from prior models.

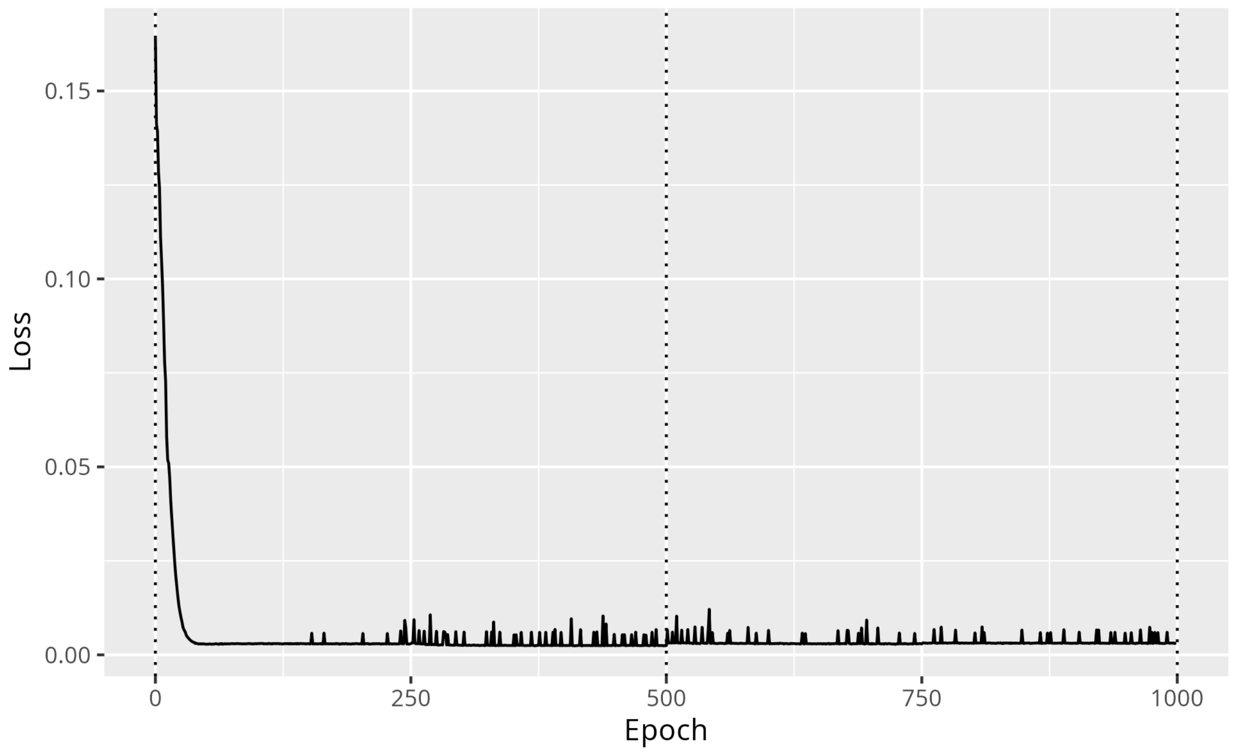

Figure 2: Loss Graph of sequential-6

There was definitely some variation, but now we see a lack of major spikes whenever the model was updated. Now, I don't remember what the configuration was at that point[2], but with the way the trend looks, it wasn't doing much.

3.1.3 Overall Conclusions of Historical Data

There isn't much to conclude with my historical attempts. The only important parts to note is that the code wasn't set up in the most optimized way. I found that python had a difficult time emulating a rubik's cube, and dumping all of the data into JSON files was not the best use of system ram.

[2]I had some very interesting ways of tracking models. This is totally not the reason that there are many branches on the GitHub repository.

3.2 The great rewrite of Spring 2023

In the spring semester of 2023, I decided to try and tackle this project again. I couldn't bear the old code, so I decided to scrap everything and start from scratch. In doing so, I made the choice to include different languages for what they do best.

For emulating the rubik's cube, I used the Rust Programming Language, particulary using py03 to compile Rust code into a Python module. This allows me to emulate the cube using memory-efficient code.

For storing data, I decided to use a Microsoft SQL Server database run from a Docker Container. Using SQL allowed me to not worry about managing JSON files, and also gave me the ability to access the data from anywhere[3].

Because I used Microsoft SQL Server, I was able to use R for analyzing, as it has better graphing and reporting features than many other languages like Python. All of the graphs and data tables in this report are made using R and R libraries.

Lastly, I used python for the model itself. I am used to working with the Tensorflow library, and to change things up would not be the best idea. I am using pyodbc to connect to the database, and the Rust library mentioned earlier to emulate the cube.

3.2.1 The Replay Database

The rust implementation not only emulates the cube, but also stores the history to be used as a replay database. As opposed to prior implementations in python, using Rust for the replay database made sense, as Rust handles memory efficiency much better than python.

The replay database implementation consists of a list of entries. Each entry contains the initial cube state, the chosen move, the next cube state, and the reward of the next cube state. When the model builds up the replay database, it populates this list until it reaches its capacity, where it will then start randomly replacing values already in the set. When the model needs to train, a random sample of data points will be returned from the set. No items are physically removed from the replay, so it is possible for an entry to be pulled more than once for a training session.

[3]This was achieved through the use of Tailscale, which creates a private virtual network between your devices

3.3 Model 2043

Git Commit 33486dc0e1b074dd91d75fec95c8f1bcea5e8b59

Model 2043[4] is the first actual model that I ran after rewriting the entire program. One of the things I forgot to do when building this model was to square-root the average loss values when collecting and storing data. For the graphs below, average loss has been square-rooted in order to keep consistency with the other model graphs.

3.3.1 First 10,000 Epochs

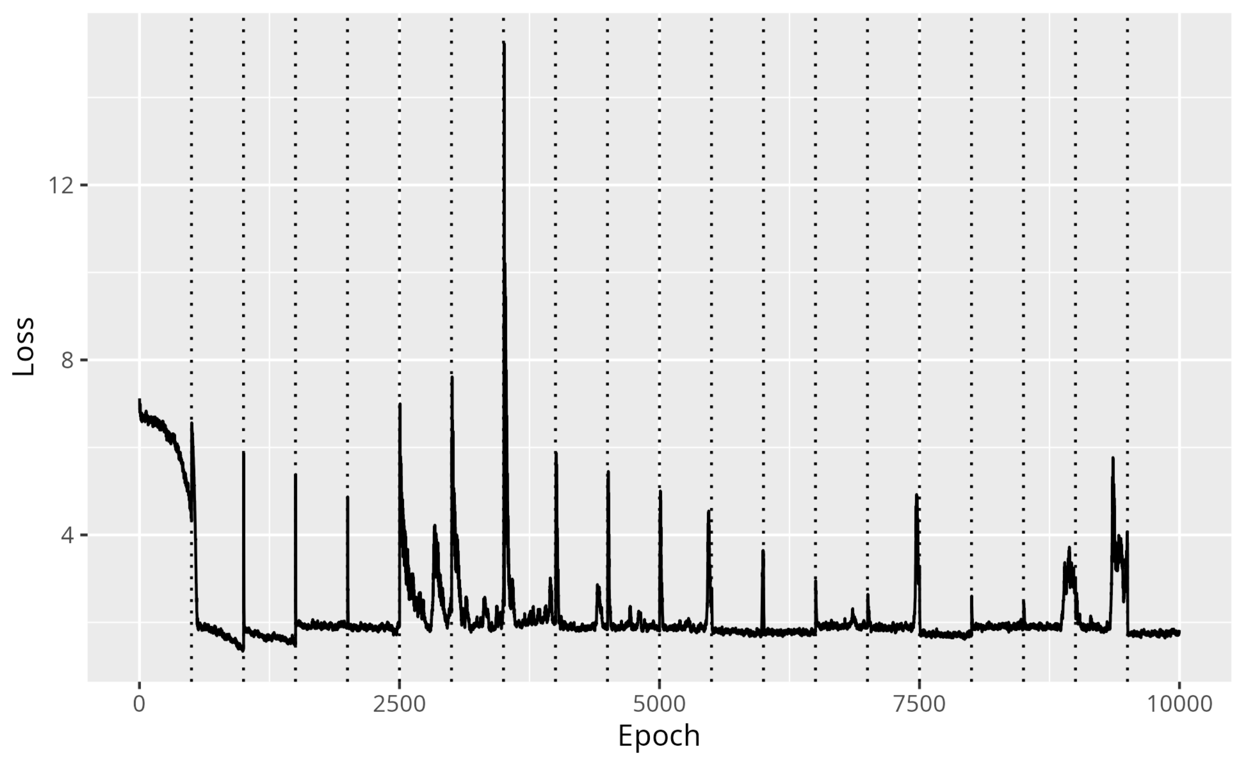

Figure 3: First 10,000 epoch entries of Model 2043

Notice that many of the spikes are accurately seen as a result of the target updating. These are good signs, meaning that the difference between models for each cycle are different.

One of the major differences between historical data and the data in Model 2043 is the reward system. Calculating reward can be one of many different methods, and it is unclear the results of that choice. For these models, I decided to define the reward of a state as the total sum of correct tiles on the cube. That is, a fully solved 2x2 Rubik's Cube will have a reward of 24. In further implementations, this is slightly changed so that the minimum possible reward is 0.0, but it doesn't fundamentally change the meaning behind reward.

When looking deeper into the first 10,000 epochs, I noticed that some of the spikes happened *before* the target update.

[4]Note that the number jump does not mean there were 2000 models. Many of the indexes in the database were used for testing, and actual model tests did not start until index 2032

3.3.2 Odd Behaviour

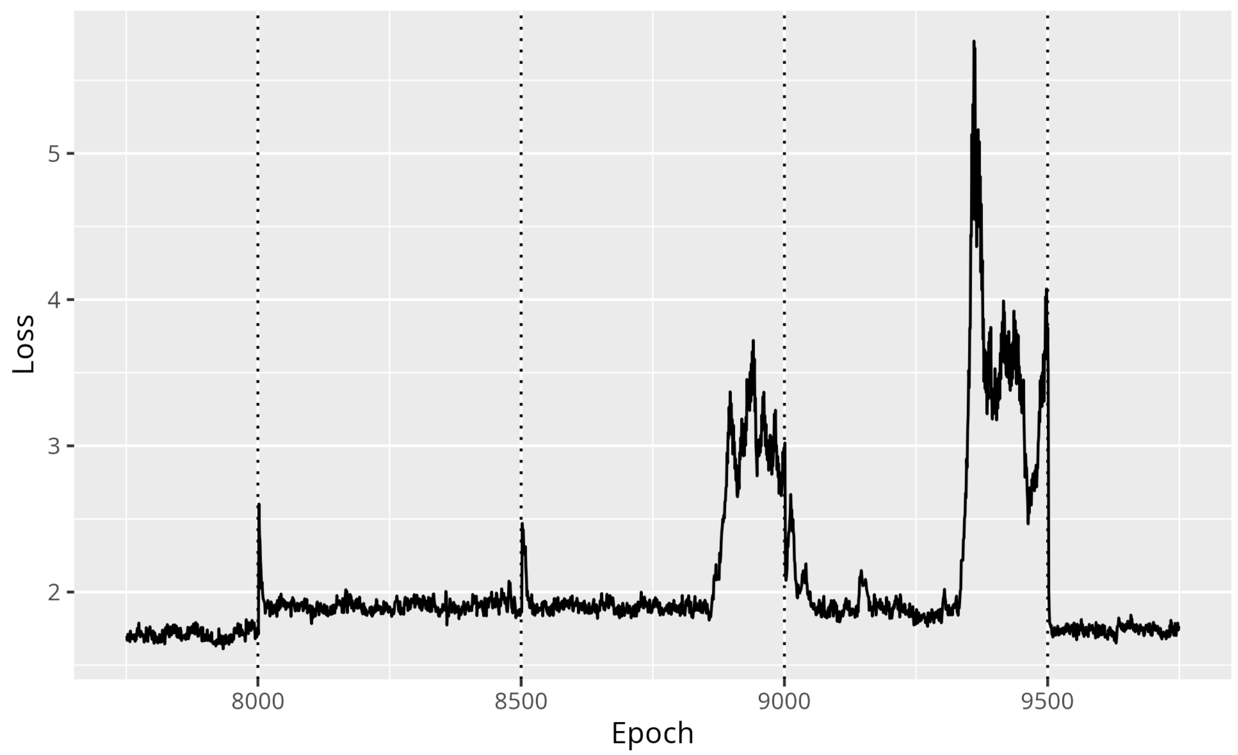

Figure 4: Epochs 7750 - 9750 for Model 2043

As seen, the model seemed to spike in loss right before some of the target updates. This type of behaviour is highly irregular, and stumped me for a great deal of time. Even now, I have no mathematical reasoning behind this phenomenon. I tried a few attempts and soon found out that it was an issue with the code.

3.3.3 Code Errors

The loss of the model for a training set uses the sum of squares method. The full equation used to calcualte the average loss is as follows:

$$\text{Loss} = \sum_i \left(\text{Q-Network}_{\text{Choice}_i}(\text{Initial State}_i) - \text{Reward}_i - \gamma \cdot \max\left(\text{Target}(\text{Next State}_i)\right)\right)^2$$

However, the implementation in python read as

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

loss = tf.math.square(loss_raw)
```

The error lied in the fact that I had seemingly swapped the first two variables. This issue meant that the network wasn't at all trying to reduce the loss between the Q-network and the Target network. This has a high possibility of being correlated to why the model had odd behaviours. In the following model, I corrected the code to be as follows:

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

loss = tf.math.square(loss_raw)
```

3.4 Models 2044-2046

Models 2044 and 2046 are similar models that followed Model 2043. After fixing the issues found with Model 2043, multiple models with slightly different configuration were ran in quick succession before making any serious changes.

3.4.1 Overview of Configurations

The following table summarizes the differences between each of the models.

	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

The full configuration of each model can be found at the following links to the GitHub repository commits for each model.

- Model 2044: 7e2b15fd3c30c8333ba9d5192990ff7fc0fc6eea
- Model 2045: b2f7631d70db228bc0676e63e3d1790094f55684
- Model 2046: fd3300675f662864be00a3378dbba0d470933472

3.4.2 First 10,000 Epochs

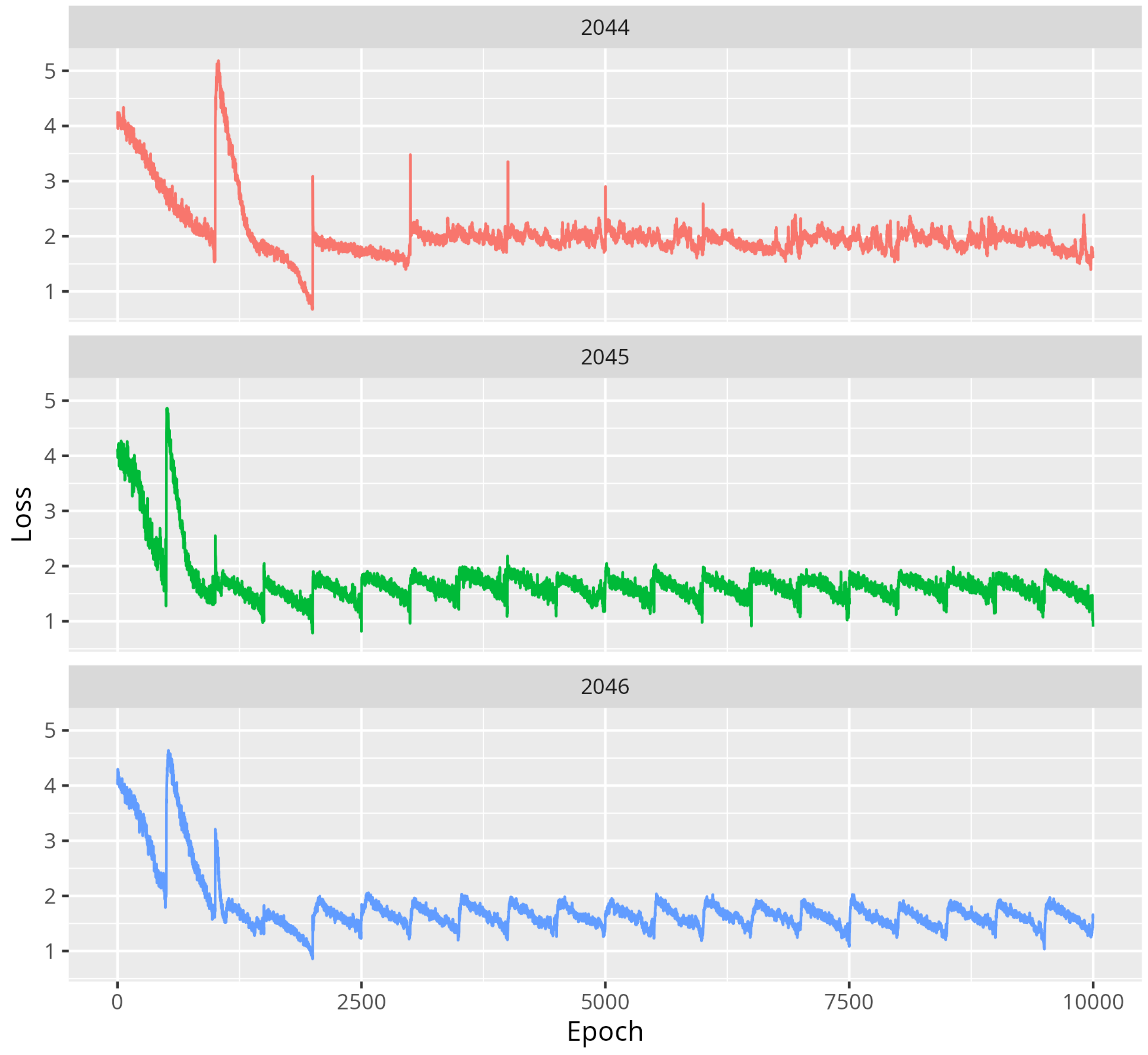

Figure 5: First 10,000 Epochs for models 2044, 2045, and 2046

It's important to note that model 2044 had a target update cycle of 1000, while models 2045 and 2046 had udpate cycles of only 500. For that reason, the vertical lines were not included in the graph above. To get a better sense of trends between the models, we can scale 2045 and 2046 by 2 so that the target updates line up at every 1000 epochs.

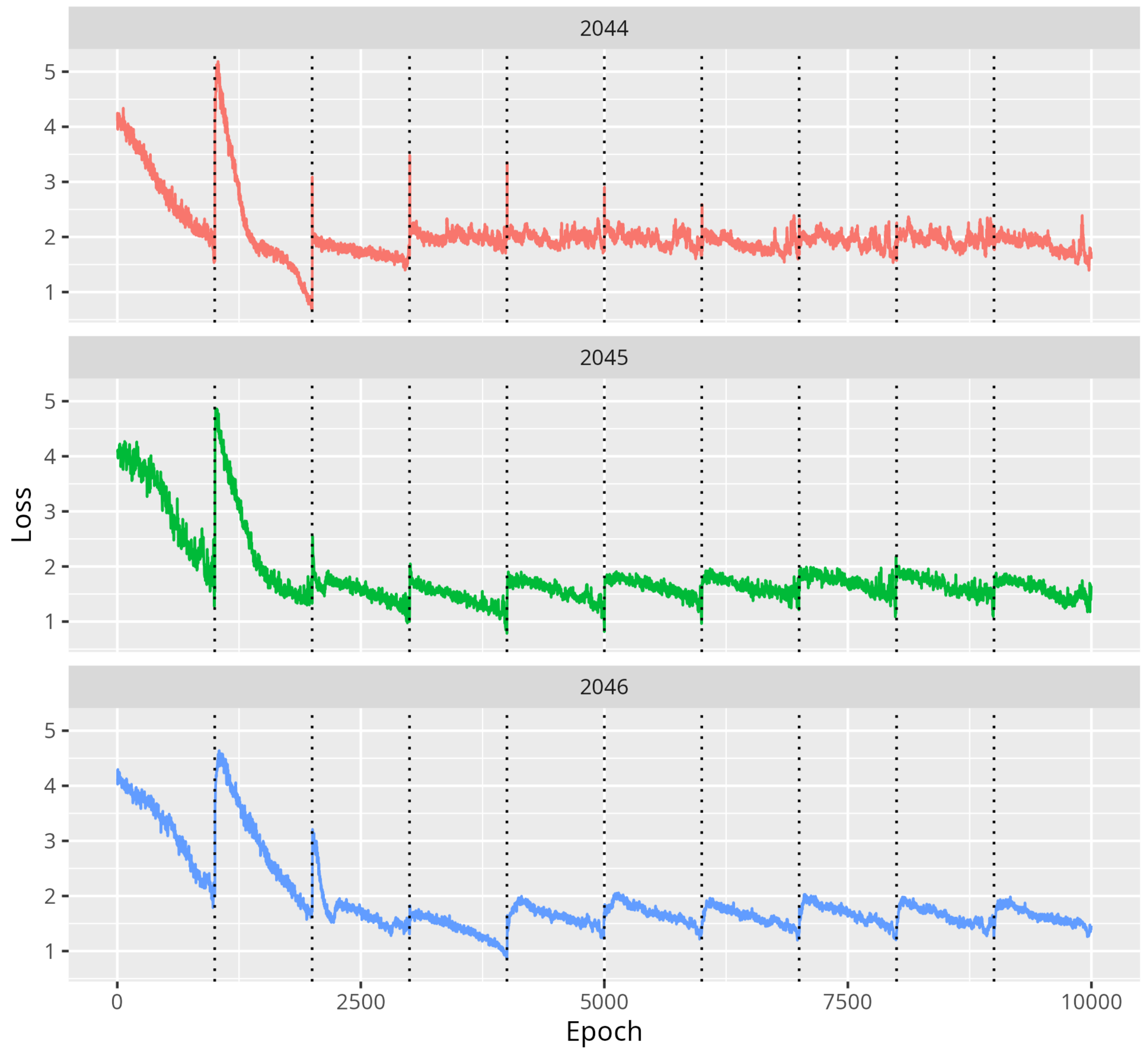

Figure 6: First 10,000 Epochs for models 2044, 2045, and 2046, scaled to syncronize target updates

With the target updates aligned, we can look at the differences in the patterns between each model. Notice how in 2044, the pattern seems to flatten out between each target update, almost to the point where a target udpate doesn't cause an increase in loss. However, in 2045 and 2046 we see the pattern where the loss jumps up momentarily after each target update, and then slowly descends back down to the lower value.

A potential explanation to this could be the fact that 2044 had a target update interval of 1000, while 2045 and 2046 had an interval of 500. With the shorter cycle period, the model might not have a chance to get close to the target network before the target network gets updated. This means that it's a game of cat and mouse between the Q-network and target network. It's not obvious if this is efficient or if this has little impact on the training.

3.4.3 Evaluation Statistics

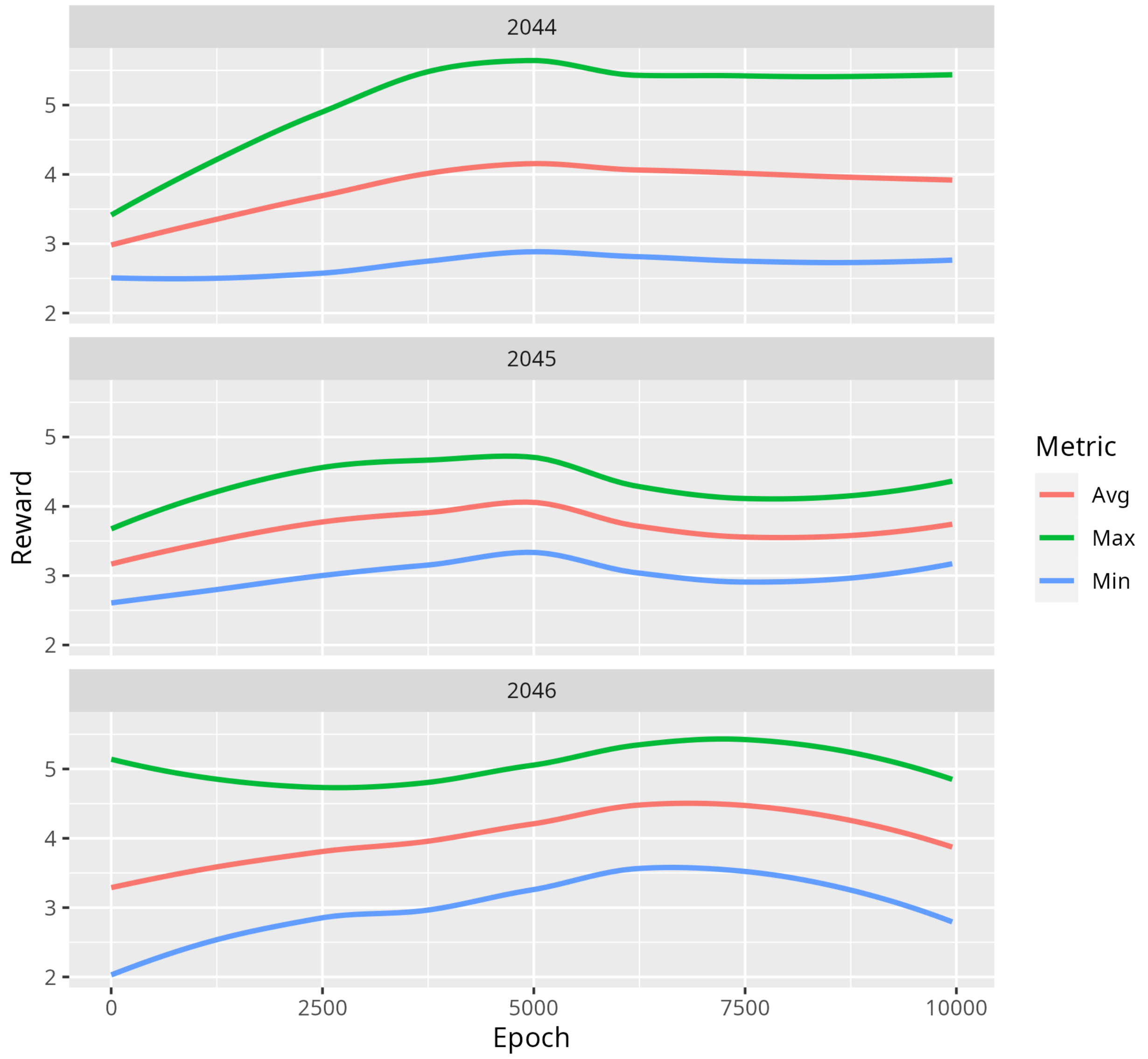

Figure 7: Evaluations during the first 10,000 epochs for models 2044, 2045, and 2046

The above graph summarizes each of the evaluations during the first 10,000 epochs. Evaluations were taken every 50 epochs, and the graph above shows the minimum, average, and maximum reward trend for each of the models[5].

[5]A point graph or line graph would not suffice as numbers are all over the place and messy otherwise. Thus, we show the general trends using ggplot2's geom_smooth function

4 Conclusions

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

```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(31) %>%
  filter(Epoch < 1000) %>%
  ggplot(aes(x = Epoch, y = Loss)) +
  geom_line() +
  geom_vline(xintercept = c(0, 500, 1000), linetype = "dotted")
ggsave(file_name("model_31.png"), width = 6.5, height = 4, units = "in")

get_epochs(39) %>%
  filter(Epoch < 1000) %>%
  ggplot(aes(x = Epoch, y = Loss)) +
  geom_line() +
  geom_vline(xintercept = c(0, 500, 1000), linetype = "dotted")
save_gg("model_39.png")

get_epochs(2043) %>%
  mutate(Loss = sqrt(Loss)) %>%
  filter(Epoch < 10000) %>%
  ggplot(aes(x = Epoch, y = Loss)) +
  geom_line() +
  geom_vline(xintercept = c(500, 1000, 1500, 2000, 2500, 3000, 3500, 4000,
↪  4500, 5000, 5500, 6000, 6500, 7000, 7500, 8000, 8500, 9000, 9500),
↪  linetype = "dotted")

save_gg("model_2043_1000.png")

get_epochs(2043) %>%
  mutate(Loss = sqrt(Loss)) %>%
  filter(Epoch > 7750 & Epoch < 9750) %>%
  ggplot(aes(x = Epoch, y = Loss)) +
  geom_line() +
  geom_vline(xintercept = c(8000, 8500, 9000, 9500), linetype = "dotted")

save_gg("model_2043_1.png")


get_epochs(2044, 2045, 2046) %>%
  mutate(Model = factor(ModelId)) %>%
  filter(Epoch < 10000) %>%
  ggplot(aes(x = Epoch, y = Loss, color = Model)) +
  geom_line() +

  facet_wrap(~Model, ncol = 1) +
  theme(legend.position = "none")
save_gg("models_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)) +
  geom_line() +
  geom_vline(xintercept = c(1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000,
↪  9000), linetype = "dotted") +
  facet_wrap(~Model, ncol = 1) +
  theme(legend.position = "none")
save_gg("models_2044_2046_1.png", height = 6)

get_evaluation_data(2044, 2045, 2046) %>%
  filter(Epoch < 10000) %>%
  ggdata_summarize_evaluation_data() %>%
  ggplot(aes(x = Epoch, y = Value, color = Metric)) +
  facet_wrap(~ModelId, ncol = 1) +
  geom_smooth(se = FALSE) +
  labs(
    y = "Reward"
  )
save_gg("models_2044_2046_2.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`