PROJECT REPORT VERSION 2.0

Jonathan Schory & Jonathan Schwartz

April 26th, 2020

Abstract

The MuZero algorithm which combines a tree-based search (MCTS) with a learned neural model has proved to yield superhuman results in a variety of domains such as Chess, Go, Atari. In addition, a fundamental feature of the MuZero algorithm is that it does not require prior knowledge of domain dynamics, game rules, or pre-determined strategy. The course exercise, which implements AlphaX over an NP-Complete problem can therefore be enhanced by applying the MuZero methodology. With Traveling Salesperson Problem (TSP) as an evaluation domain at a manageable scale, we seek to demonstrate state-of-the-art performance with MuZero.

The following report will demonstrate the progress and current outcomes of the TSP MuZero implementation (to be referred as MuZeroTSP). As such, the report will analyze the ongoing progress rooted at the initial exercise (AlphaTSP) and will then examine the methodology, resources, and approaches used to transition from the AlphaZero framework to the MuZero framework.

Introduction

It is a well known fact in the field of computer science that look-ahead prediction search can yield astonishing success and accuracy in the realm of artificial intelligence. Nonetheless, many if not most of these state-of-the-art planning frameworks require prior, elaborate familiarity with environment-specific details such as game rules, valid actions, potential strategies, etc. Making use of avant-garde reinforcement learning mechanisms, novel frameworks seek to first train a model on the environment and its dynamics for planning algorithms governed by the learned model. Specifically, the MuZero framework (Schrittwieser et. al) which extends the AlphaZero methodology (Silver et. al) uses a learned model to predict the key details for future planning.

The most astonishing variance between the AlphaZero approach to the MuZero framework, is that MuZero does not require any prior domain knowledge (i.e. the machine does not need to know the rules of the "game"). In this sense, MuZero develops a dynamic environment model which allows it to calculate probabilities and rewards that are needed for its plan-ahead mechanism. In addition to the MuZero original publication, the DeepMind team have released an elaborate appendix with the MuZero psuedocode which explains the intricate interactions between the algorithm components. The psuedocode, a main resource used in the implementation of our MuZeroTSP, will be examined in-depth throughout this report.

Problem, Goals, and Formulation

The original course exercise requires to implement AlphaX where X is any NP-complete problem over a graph. The NP-complete problem chosen for the exercise as well as the project is the Traveling Salesperson problem "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?"

The goals and formulation of the assignment, and our enhancements, are as follows:

- Formulate the problem as a single-player game and use MCTS to solve a small random instance of the problem with 10 nodes. Actions in this case are limited to node manipulation (adding or removing nodes to the graph). User may be prompted to directly determine how and when actions are taken (limited to computationally feasible actions). Node features will be xy coordinates and graph edges represent distances.
- The AlphaZeroTSP will then be on a small random instance of the problem with 10 nodes. Then to be enhanced to MuZero TSP and evaluated in a similar fashion on small scale dataset.
- Extend the dataset to a maximum of 20 nodes and thoroughly evaluate performance and limitations of MuZero implementation on an NP-Complete problem through plotting and probability analysis.
- Thoroughly compare and examine the performance difference between AlphaZeroTSP and MuZeroTSP considering a metrics, namely the amount of random instances solved within a close factor of the optimal Traveling Salesperson Problem solution (e.g. factor of 1.1 from the brute-force algorithm).

- Experiment with various neural networks as MuZero components to evaluate the advantages and disadvantages in each, mainly seeking to infer which framework may yield most beneficial, computationally efficient, and optimally calculated for this specific case of an NP-hard problem: TSP.

Related Work

The main resource for the project including understanding the MuZero framework, familiarity with its components, computational logic, and psuedocode implementation is rooted in the original MuZero Paper: "Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model." Furthermore, the exercise which requires implementation of AlphaZero made use of the Alpha Zero paper "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm" by Silver et al. Finally, the hands-on Python implementation of both the AlphaTSP and the MuZero TSP made use of open source Github repositories which will be referenced in the report's reference section[6][7][8]. As the project is founded on the related work provided in the reference paper, it is essential to outline the MuZero framework before discussing and elaborating implementation.

- The MuZero model is constituted from three interconnected components: representation, prediction, and dynamics. Using a prior hidden state h and a potential valid action k the dynamics function g yield a reward r alongside a new hidden state. The policy p,k as well as the value function v,k are both computed using the hidden state h through the prediction function f. The initial hidden state h is obtained by passing the past observation stack (in the TSP case, representations of a node graph with distances) into the representation function r.
- Afterwards, a Monte Carlo Tree Search is performed at every step t were an action is sampled each time from the search policy pi, proportional to the visit count from each node in the tree. The environment then receives a valid action to generate a new observation o and a reward u. At the end of each "training episode" the data is stored into the replay buffer.
- Training is performed by sampling trajectories from the replay buffer, where in each initial step the representation function r received an input of past observations o from the sampled trajectory. Then the model is "unrolled" recurrently for a predetermined number of k steps. For each step k, the dynamics function g obtains the input of a hidden state h from prior step and a real action. The parameters of these components are trained jointly using backpropogation to yield three essential quantities: policy, value, and reward.

Methods for AlphaTSP

The data and algorithms used in our first approach were sourced from the provided libraries recommended in the exercise description, namely the The Alpha Zero General repository provided in the course website [6]. The AlphaZero General framework is flexible and dynamic reinforcement learning environment for self-play based on the AlphaGo Zero paper (Silver et al). The AlphaZero repository provides a framework for reinforcement games that can be manipulated for the user's needs, in our case: the formulation of TSP as a single player game and implementation of MCTS algorithm for the tour optimization process. A key advantage in the AlphaZero framework is the ability to directly manipulate pre-existing .py files which execute the needed functions for TSP game incorporating the Monte Carlo Tree Search. Namely, the Game.py and MCTS.py files are user-friendly and easily maliable Python code segments which requires relatively little alteration to run properly on our TSP single player game. Both Game.py and MCTS.py outline and provide the necessary functions and algorithms required to carry out the game, which in turn allowed us to remove and manipulate particular aspects of the code and execute the game successfully.

Methods for MuZeroTSP

Our methodology for implementing the MuZero approach was constructed of two crucial parts. The first, was to thoroughly understand and examine the MuZero paper (as well as other online articles) to better grasp the convoluted logic behind the MuZero framework and its interconnected neural architecture (which will be elaborated later in the Architecture subsection). Afterwards, our Python implementation for the MuZeroTSP program was sourced from the MuZero General repository. Specifically, as with the AlphaTSP approach, we made direct use of the pre-existing TicTacToe MuZero implementation which allowed us to gain firsthand understanding of a gameified version of the MuZero algorithm. In doing so, we successfully visualized and experimented with a fully-functioning MuZero game. Moreover, the MuZero General repository provided us with the critical foundations for our MuZeroTSP which includes, but not limited to: models.py file, sharedStorage.py file, an implementation of Monte Carlo Tree Search, a ReplayBuffer class, and other elements for running the MuZeroTSP framework. In order to ensure full understanding and modify the existing code properly, we worked in parallel using the Github repository as well as the reference paper provided psuedocode. We then iteratively implemented aspects of the MuZeroTSP (from formulating it as a single player CSP game to the eventual representation of the search problem as a MuZero-solvable).

As will be explained in further detail in subsequent sections, the team experimented with a variety of models by iteratively modifying their hyperparameters. Mainly, we sough to increment the amount of training steps (a MuZero configuration parameter) from relatively low values such as 50 to large values over 10,000. In doing so, we were able to visualize and quantify the progress of the MuZero network, seeking to evaluate the threshold at which the algorithm functions within a reasonable margin of error. Since our Python notebook implementation allows for a user to load a pre-trained model, such experiments became a staple part of the research process.

Evaluation Criteria

All evaluation will be compared to the brute-force TSP solution which consist of a tour (Hamiltonian circuit) that visits each node in the graph *exactly* once and seeks to *minimize* the total weight along the path. The evaluation criteria for the TSP was split into the two steps of the project formulation: AlphaZero and MuZero, with both cases being compared with the brute force optimal solution algorithm. In this particular case, the brute force algorithm makes uses of examining all possible permutations of nodes in the "tour" yielding an optimal solution at exponential time complexity. The evaluation for the AlphaZeroTSP single player game is mainly based on the exercise evaluation which calls for plotting the percentage of random instances which MCTS solves within a 1.1 factor of the optimal (brute force) solution of the tour. The evaluation criteria for the MuZero implementation of TSP is based on a similar analysis of plots and comparison to brute force solution as mentioned in the AlphaTSP framework. Specifically, the MuZero evaluation included a plot analysis of the MCTS random instances in comparison to the 'hard-coded' optimal solution applying brute force calculation. For the MuZeroTSP, iterative experimentation with various hyperparameters was an intrinsic part of the evaluation process, allowing us to better decipher the fine-tuning which may increase of decrease performance.

MuZeroTSP Process

The framework's point of entry is the function muzero which passes an object of type MuZeroConfiguration. This provides some parameter information about the running of the algorithm. In our case, this included the action space size (space size is equivalent to number of nodes in the TSP route) and number of actors (in our case, a single player game). Furthermore, our code implements two essential objects for running the MuZero framework: Shared Storage and the Replay Buffer. Specifically, the Shared Storage object allows the MuZero algorithm to store and load a trained neural network object needed for the planning-ahead algorithm, and the Replay Buffer stores data and important information from previous games (e.g. past attempts to solve the TSP problem on a randomized node graph). Following the creation of these objects, our MuZero framework launches parallel game environments, each running the self-play mechanism which makes use of the most recent network and shares the outcome game data (optimal strategy for short path) to the buffer.

Network Architecture

While the AlphaTSP makes use of a single neural network, which in our exercise implementation was a CNN, the MuZero framework gains its rule-agnostic advantage from its three separate neural networks (prediction, dynamics, representation).

The figure below illustrates our choice for environment representation (used both in the AlphaTSP and MuZeroTSP portions):

- Prediction Network: The task of the AlphaZero prediction network f is to predict policy and value (p,v) of a graph state. As such, the network generates a probability distribution over possible moves yielding a predicted reward for each possible legal action. A move prediction is made when the MCTS algorithm reaches a leaf node in the search tree, assigning a value to a state and its possible following action. After propagating these values back to the root node, the network has a solid understanding of future values and fruitful strategies to explore the action space. On the other hand, our MuZero also has a prediction network yet since our MuZero does not know the rules of "TSP world" the observations are encoded into hidden states which are fed into the Dynamics Network which models the game environment.

- Dynamics Network: The input for the Dynamics Network is the hidden state h yielded by the representation function in concatenation with a transition action representation (legal action). Actions that are valid are all spatially encoded in the hidden state resolution.

- Representation Network: The Representation network maps the current observed state of the TSP game to the initial representation.

Graph Representation

```
Graph:
[[0.51027367 0.06753336]
 [0.71149285 0.69771643]
 [0.9195707  0.34988506]
 [0.97805973 0.70346914]
 [0.09729456 0.81933297]]
```

State Representation

```
State:
[[1. 0.]
 [0. 0.]
 [1. 1.]
 [0. 0.]
 [0. 0.]]
```

1. The TSP graph is represented by using a distance matrix of xy coordinates on a randomized plane of num_nodes
2. The TSP state is represented using an array representation where the left column indicates visited nodes and right column indicates current node. In example above, we observe a tour than began from node 1 (location 0) and visited node 3 (location 2)

Figure 1: TSP State and Graph Representation

Visual Representation of Networks

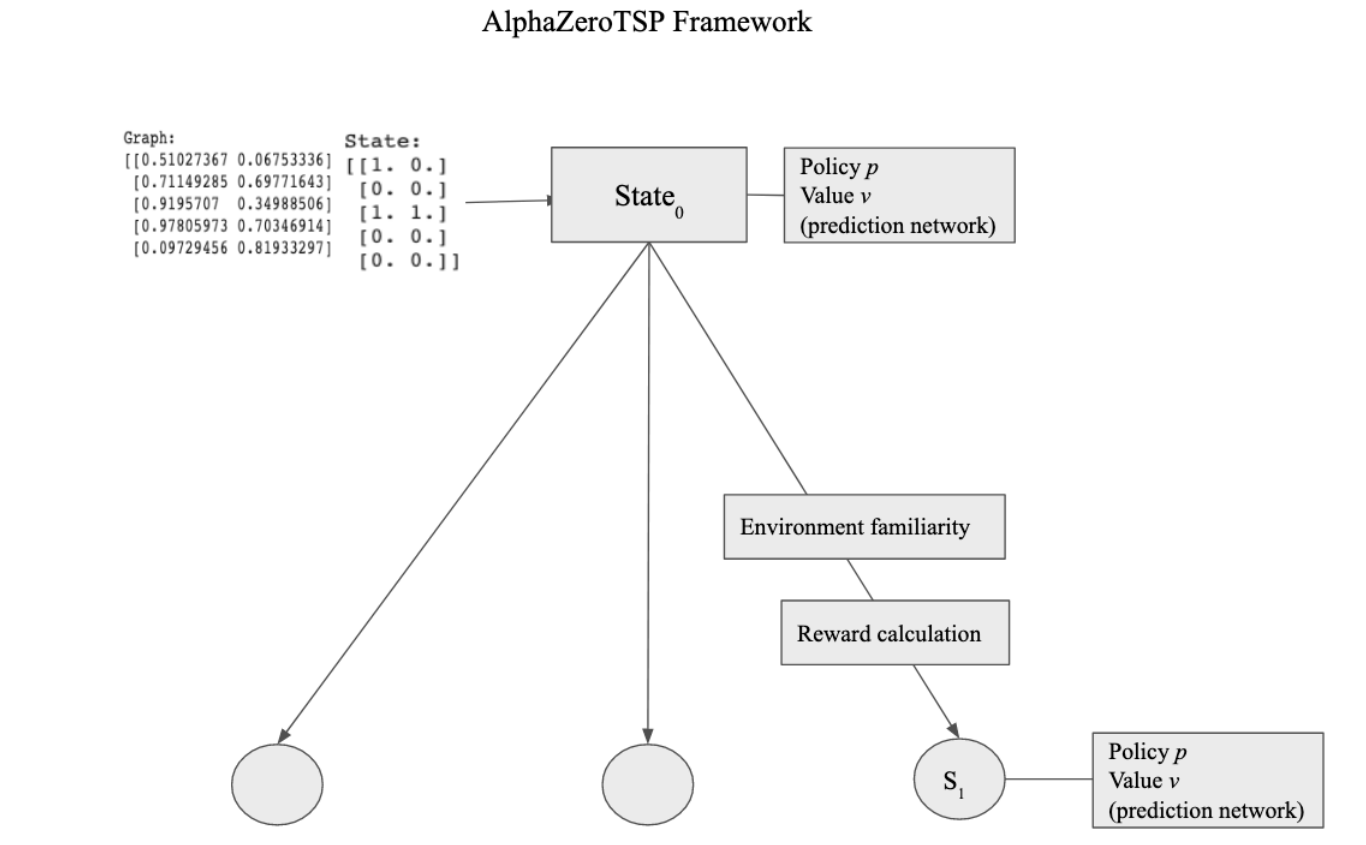

Figure 2: AlphaTSP Network Architecture

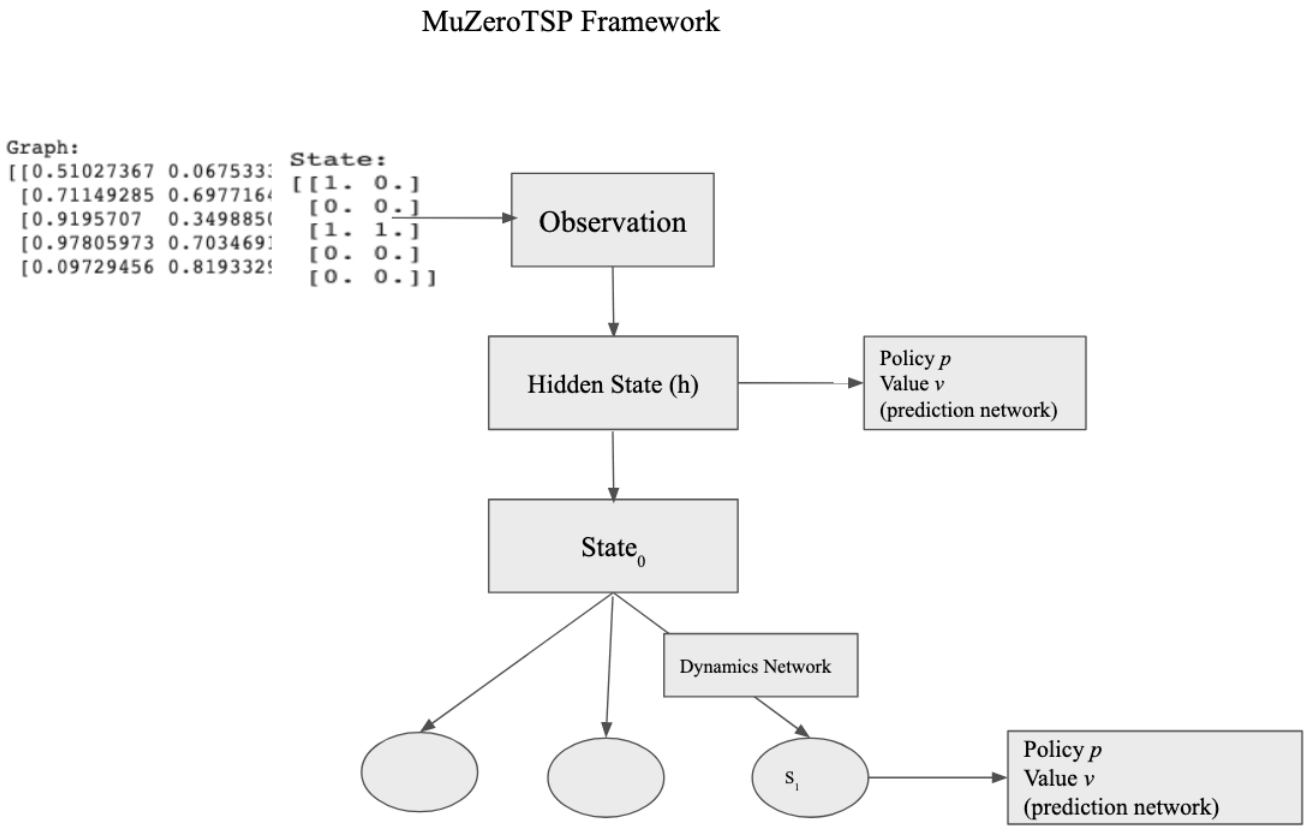

Figure 3: MuZeroTSP Network Architecture

MCTS Results for AlphaTSP (Course Exercise)

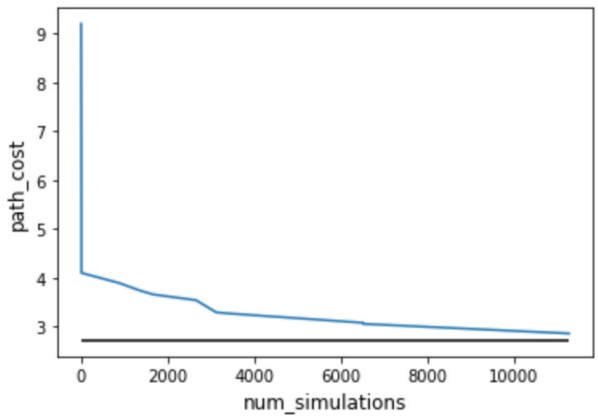

Figure 4: MCTS Progression

As noticeable from the graph, we observe that over time (e.g. as the number of simulations increases) the tour length approaches the optimal tour length. Specifically, we note that the blue curve (representing the returned reward from the Monte Carlo Tree Search algorithm) converges towards the shortest tour length (computed using brute force calculation on possible permutations). Currently, our data shows that many of our sample MCTS runs yield an ultimate result that is within a 1.1 factor of the optimal solution.

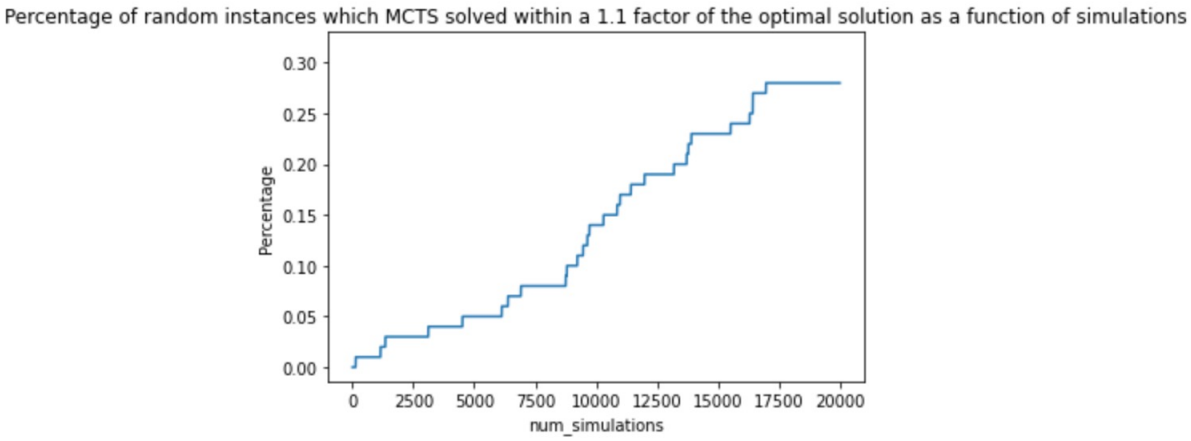

Figure 5: Percentage of random instances which MCTS solved within a 1.1 factor of the optimal solution as a function of samples

Metrics for MuZeroTSP

In order to keep track of real time training metrics and progress over episodes, the team made use of the Tensorboard extension of the TenserFlow package. By analyzing the Tensorboard dashboard, we were able to better grasp and visualize the levels of accuracy, efficiency, and overall performance of the various models used in the MuZeroTSP environment. Some of the metrics included in our analysis were weighted_loss, reward_loss, and policy_loss.

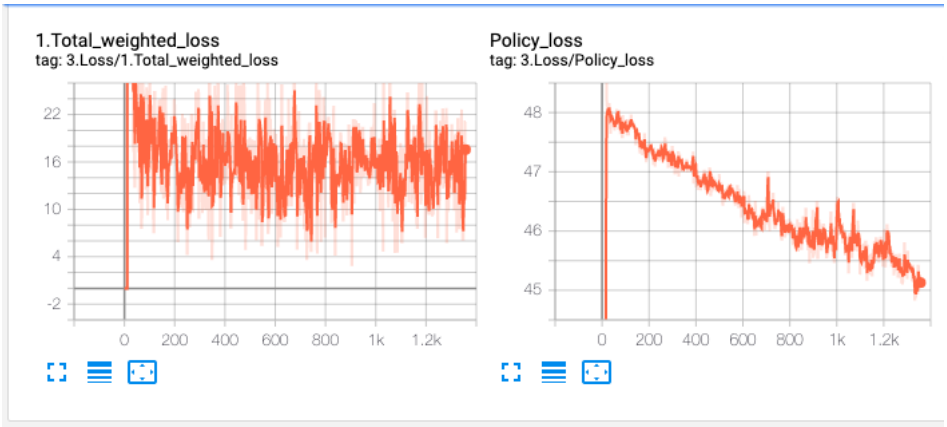

Figure 6: TensorBoard Metrics Example 1

Figure 7: TensorBoard Metrics Example 2

Parameters and Experimentation for MuZeroTSP

In order to better evaluate the performance of our MuZero model, while also deciphering the parameters which impact the algorithm's result, the team engaged in a variety of experiments in which parameters were modified. Specifically, we made use of our codes implementation which allows to load a pre-trained model and use it to render MuZero self-play games on a predetermined amount of randomized TSP graphs. As such, we formulated an array of models with varying parameters. These alterations mainly included:

- Modifying the number of simulations (e.g. the number of future moves simulated by the MuZero model
- Modifying the number of training steps in the model's training process, which correlates directly with the amount of "games played" (e.g. the number of graphs that were solved using the MuZero framework
- Modifying the Replay Buffer "Unroll Steps" variable which controls the number of game moves to keep for every batch element

Results

As mentioned earlier in the report, results of the MuZeroTSP will be evaluated in comparison to the optimal path calculated by the brute-force TSP algorithm. Moreover, we sought to demonstrate results of various models in terms of number of completed routes where the solution was within a various factors of the optimal solution (e.g. factors of 1.1,1.2,1.3,1.5). In the following graphs we illustrate the ratio of games solved within these factors with regards to the amount of games played (e.g. completed TSP graphs with MuZero algorithm):

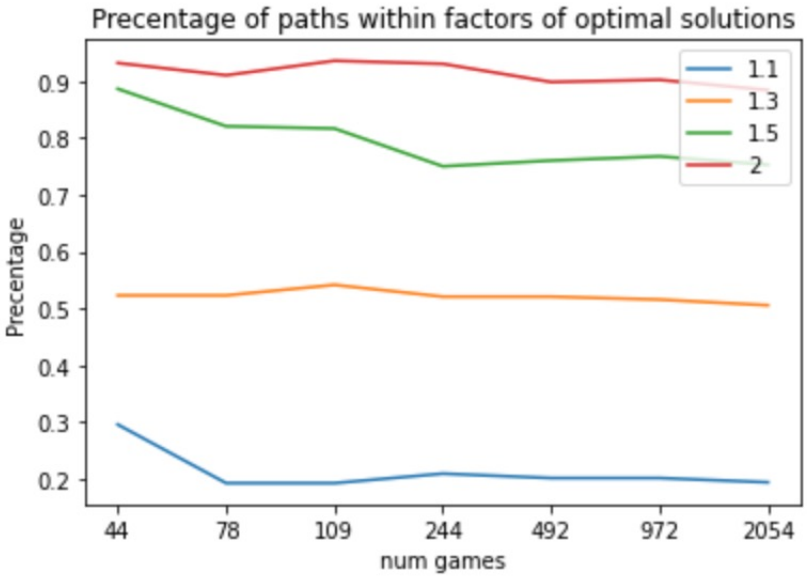

Figure 8: Percentage of games solved within factors of optimal solution

Conclusions and Future Steps

As of right now, the conclusions are that the MuZero network has not yet been able to properly learn the dynamics and rewards of the TSP environment well enough to succeed in increasing its percentage of games solved within 1.1 factor. While we can certainly observe that a vast majority of games (over 90%) have been solved within a factor of 2 from the brute force solution, our algorithm requires enhancement to increase precision.

As noticeable from the results provided above, the learning rate of the framework is somewhat stable, illustrating that there is not significant improvement despite and increase of number of games played and an increase in training episodes. With that being said, the following steps are to be taken in order to improve and increase the performance of the MuZeroTSP framework:

- Run the network on an incredibly large value of games on VM (such as 100k episodes which yields countless completed games
- Investigate and re-parameratize the neural networks used in the MuZero code to seek out enhancements in learning rate and performance
- Ensemble various neural networks and analyze performance enhancements

References and Previous Work

1. https://github.com/jonSc8/deep_learning_final_project
2. https://github.com/suragnair/alpha-zero-general/blob/master/MCTS.py
3. TA provided tutorials on CourseWorks
4. https://deepmind.com/blog/article/alphago-zero-starting-scratch
5. https://github.com/werner-duvaud/muzero-general
6. Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model: arxiv.org/abs/1911.08265
7. Silver, David, et al. "Mastering chess and shogi by self-play with a general reinforcement learning algorithm." arXiv preprint arXiv:1712.01815 (2017).