RNN and History State Representation for NP Problems with AlphaZero

Tal Zilkha, tiz2102[1]
Mat Hillman, mh3691[1]

Abstract

The AlphaZero algorithm combines a tree-based search (MCTS) and a learned neural network and can achieve superhuman performance in many challenging domains. Starting from random play, and given no domain knowledge except the game rules, AlphaZero achieved a superhuman level of play in the games of Chess, Shogi, and Go, and convincingly defeated a world-champion program in each case. The algorithm can definitely beat world-masters in various domains, but is not necessarily implemented to win efficiently with fewest number of steps.

In this paper, we seek to utilize the class exercise with the AlphaZero algorithm on the Traveling Salesperson Problem (TSP), an NP-complete problem. Instead of the classic AlphaZero algorithm which looks at the last state, we tweak it so it uses the history leading to said state. We do this by implementing an LSTM network which will account all board states leading to the current one. We compare this approach to GCN, CNN, and random-policy approaches to see whether an history based approach yields an efficient way of solving the TSP problem and whether it could be applied to various other domains in the NP Problem space.

1. Introduction

During the time of its release, AlphaZero was able to achieve results that were considered state of the art on the multi-player games Chess, Shogi, and Go. The addition of a deep neural network and self-play algorithm allowed it to surpass its predecessors which relied on actual game examples to train. The main stages of AlphaZero consist of:

- Self-Play. The model plays games against itself, storing game states s, search probabilities p, and rewards r. The state representation was a description of the current board.
- Training. A neural network is trained on the information stored during the self-play phase such that $f(s) = \hat{p}, \hat{v}$ where $\hat{p}$ and $\hat{v}$ are predicted values on optimal policies of a given state. These predictions are used to speed-up the Monte Carlo Tree Search.
- Evaluate. Test whether the new version of the network fairs better than the previous iterations'. If that is the case, the new version replaces the old version.

In the class exercise, we were tasked with implementing the AlphaZero algorithm to be able to work with single-player games. We had to choose between various NP-Complete problems and construct them as single-player games. Our problem of choice was the TSP and we formulated it as a single-player game. Next, we modified the MCTS algorithm from Alpha-Zero-General GitHub repository to work on single-player games. Finally, we were tasked to implement a Graph Neural Network to speed up the MCTS computations.

In many games such as Chess, the history of moves played before a particular state can give insights as to the future of the game. In our project, we choose to explore whether the model itself can benefit from a larger representation of history in the context of NP-Problems. That is, keep track of the history of board states and use them in predicting the policy and value using an LSTM network.

In our experiment, we aim to compare the AlphaZero algorithm on the TSP using a CNN, GNN, and an LSTM.

2. Related Work

The main reference we used is the original publication on AlphaZero to understand the algorithm (2). Our original idea involved modifying the MuZero algorithm and we had spent time reviewing its original publication (4). We also referenced (1) in order to understand how to implement an NP Problem into a single-player game. We referenced (3) for inspiration on GNN implementation for TSP. Lastly, we followed (5) to get an idea of a possible architecture to incorporate and how state history would feed into it as input.

3. TSP

The implementation of the Traveling Salesperson Problem into a single-player game is described as follows:

- Graph - The underlying positions of the nodes of the graph in 2D. That is, an $n \times 2$ matrix where each row corresponds to a node's position.
- Path - The current path is represented by an $n \times 2$ matrix in which the left column corresponds to nodes visited, that is a 1 if the node has been visited and a 0 otherwise. The right column of the matrix is a sparse vector of 0's with a 1 corresponding to the node currently at the end of the path.
- Edge Cost - a path cost is calculated through the euclidean distance between two points. This is used as the rewards.

Therefore we get that,

$$s = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, T(s,3) = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

and, $R(s,3) = Distance(2,3)$.

4. CNN

The CNN architecture was inspired by the architecture used in alphazero-general for the tic-tac-toe game (2).

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	(None, 7, 2)	0
input_3 (InputLayer)	(None, 7, 2)	0
reshape (Reshape)	(None, 7, 2, 1)	0
reshape_1 (Reshape)	(None, 7, 2, 1)	0
conv2d (Conv2D)	(None, 7, 2, 256)	1280
conv2d_2 (Conv2D)	(None, 7, 2, 256)	1280
batch_normalization (BatchNorma	(None, 7, 2, 256)	1024
batch_normalization_2 (BatchNor	(None, 7, 2, 256)	1024
activation (Activation)	(None, 7, 2, 256)	0
activation_2 (Activation)	(None, 7, 2, 256)	0
conv2d_1 (Conv2D)	(None, 7, 2, 256)	262400
conv2d_3 (Conv2D)	(None, 7, 2, 256)	262400
batch_normalization_1 (BatchNor	(None, 7, 2, 256)	1024
batch_normalization_3 (BatchNor	(None, 7, 2, 256)	1024
activation_1 (Activation)	(None, 7, 2, 256)	0
activation_3 (Activation)	(None, 7, 2, 256)	0
concatenate (Concatenate)	(None, 7, 2, 512)	0
flatten (Flatten)	(None, 7168)	0
dense_1 (Dense)	(None, 1024)	7341056
batch_normalization_4 (BatchNor	(None, 1024)	4096
activation_4 (Activation)	(None, 1024)	0
dropout (Dropout)	(None, 1024)	0
pi (Dense)	(None, 7)	7175
v (Dense)	(None, 1)	1025

Total params: 7,884,808
Trainable params: 7,880,712
Non-trainable params: 4,096

For each input to the model (the graph and the current state) there are two convolutional layers with a kernel size of 2, batch normalization and Relu activations. The two outputs from the convolutions are then concatenated. We included dropout after the concatenation and fully connected dense layers before the outputs.

The CNN was implemented using the Keras library.

5. GNN

The GNN Architecture we used was inspired by the one used in AlphaTSP (3).

```
gnn(
  (conv1): GCNConv(2, 32)
  (conv2): GCNConv(32, 16)
  (conv3): GCNConv(16, 1)
  (fc): Linear(in_features=16, out_features=1, bias=True)
)
```

It consists of three graph convolution operators.

Different from (3), we wanted the graph representations fed into the network to not include information about the history of the game in order to get a better comparison of whether history improves results.

Graph data was constructed using the original graph of the TSP, that is the node locations. Additionally, only the available edges to be chosen in the next iteration were added to the graph. For example, $s = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$ would yield the following edges in its graph representation: $[(2,3),(2,4),(2,5)]$.

The GNN was implemented using the Pytorch Geometric library.

6. RNN

The RNN architecture we used simply consisted of a single LSTM layer into a fully connected dense layer.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 7, 28)	0
lstm (LSTM)	(None, 128)	80384
dense (Dense)	(None, 128)	16512
pi (Dense)	(None, 7)	903
v (Dense)	(None, 1)	129

Total params: 97,928
Trainable params: 97,928
Non-trainable params: 0

The input for the RNN consisted of multiple time-steps of board states. More specifically each time-step consisted of the concatenation of the graph and board state. Each time-step was then flattened into an array and stacked on top of one another such that the total input to the RNN was a matrix of size $time\_steps \times 4n$.

$$\begin{bmatrix} graph, s_0 \\ graph, s_1 \\ ... \\ graph, s_t \end{bmatrix}$$

For padding, time-steps preceding the starting state were composed of $[graph, 0]$, that is the state representation was all 0s.

The RNN was implemented using the Keras library.

7. Random Policy

For comparison, one of the Monte Carlo instances we evaluated used a random policy to pick the next action rather than relying on a neural network to predict policy and value.

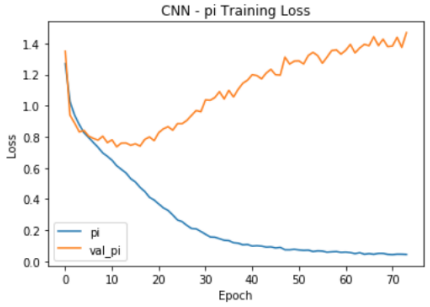
8. Optimal Training Experiment

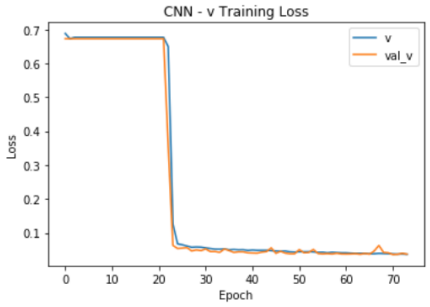
Before testing the neural networks using self-play as in the AlphaZero algorithm, we decided to test the networks using an optimal data-set. That is generate samples where the policy is based on the optimal solution of the TSP. We hoped this would give a first indicator of the potential of each of the models.

In our experiment we used TSP games with 7 nodes. We trained each model on the same data-set which consisted of a total of 5,000 games. 5,000 translates into 30,000 total samples as there are 6 actions per game.

9. Optimal Training Experiment - Training

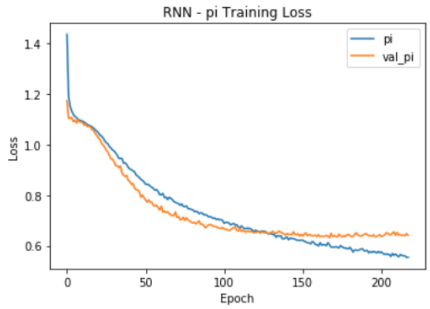
For training, we used early stopping with a patience of 50 epochs and 80/20 train/test split.
The following are the training curves for the CNN model:

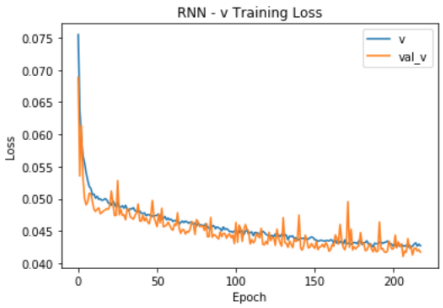




The following are the training curves for the GNN model:

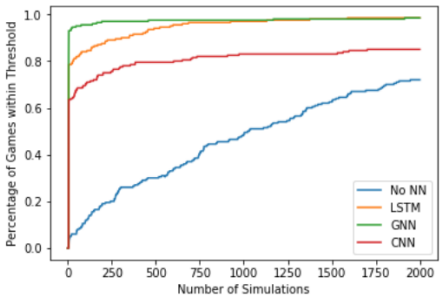
The following are the training curves for the RNN model:





10. Optimal Training Experiment - Results

For Evaluation, we solved 100 TSP games using an MCTS for each model, including the random-policy model, and recorded after how many iterations the solution was within a 1.1 factor of the optimal solution.



At a first glance it seemed that the GNN and LSTM networks gave similar results, out-performing the CNN.

11. Self-Play Experiment

Our next experiment consisted of training the different models using self-play, as in alphazero-general. We used an arena style training method to decide on which samples to include in the training set.

- Generate x TSP instances.
- For each x,
  - Naively pick the best action based on prediction from neural network to get a naive path.
  - Use MCTS and predictions from the neural network to get an MCTS path.
  - Compare the total reward from the two paths.
  - If the path using MCTS was better than the naive path, we turn it into a training set and use it to train on the next iteration
- Train the neural net on the training example and repeat.

The use-fullness in self-play for the case of TSP is that creating optimal training sets as in the last experiment for TSPs with large number of nodes can be intensely time-costly. By self-play, we are not relying on brute force to solve the TSPs.

To combat having examples which are not close to optimal from early generations, we cap the total amount of training examples to a constant and replace the old samples by the new in a queue like fashion.

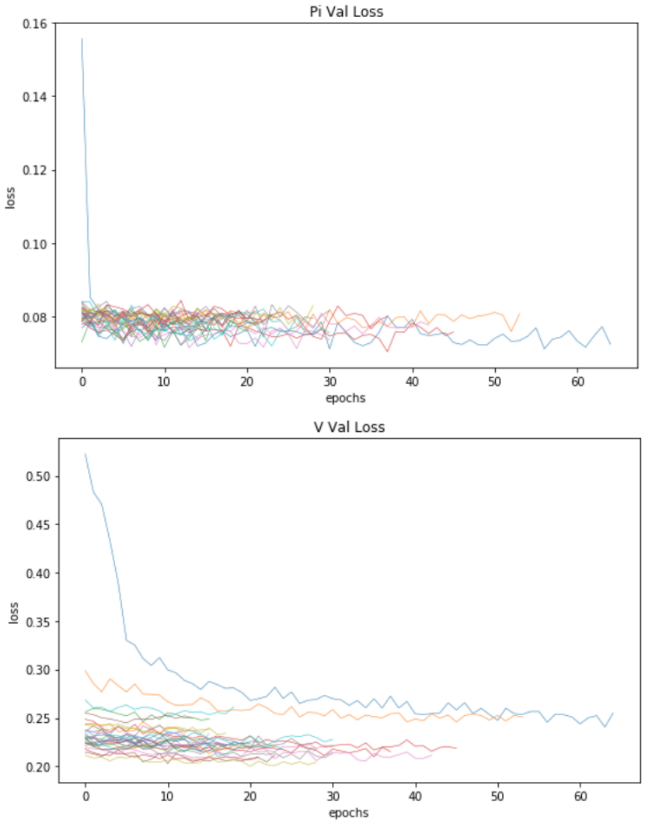
In our first self-play experiment we used TSPs of 7 nodes. The arena self-played 300 games, out of which those that saw improvement with mcts were kept for training.

12. Self-Play Experiment - Training

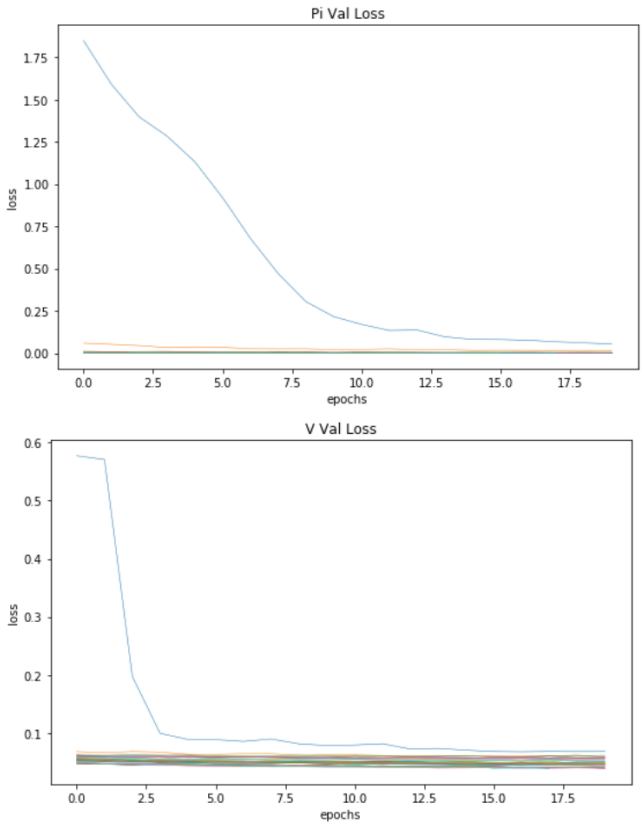
The three networks were each trained for 30 generations. Each generation self-played 300 games and the training-sample queue was capped at 3000 examples, which translates to 500 full TSP games for training.

Training in each iteration was done using early stopping when no improvement was seen for 15 epochs.

The following plot describes training iterations for the GNN.



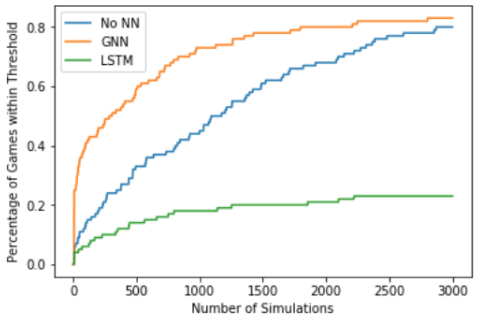
The following plot describes training iterations for the RNN.



During our experiment we had problems training the CNN using self-play. It seemed that loss for v would not go down. As seen through the optimal training of the CNN, there is a local minima that took about 30 epochs to pass. Perhaps our training sample cap for self-play was not enough to successfully train a CNN with a complex architecture such as ours.

13. Self-Play Experiment - Results

The self-play trained models were evaluated on 100 TSP instances and solutions were checked to be within a 1.1 factor of the optimal solution.



The results were surprising in that the LSTM performed worse than the random policy even though training suggested that loss was minimized adequately.

On separate occasions, it was not the case that the LSTM performed worse than random policy.

To conclude, we believe that the idea that history is beneficial to NP problems is still open for experiments. Scaling this toy example to an experiment with larger training sets and more nodes will likely produce results which can be better understood.

Furthermore, we would like to note that there was not much thought behind each of the model's complexity, that is number of hidden units, layers etc. We suggest that in next iterations of this experiment, multiple architectures from each type of neural network should be trained and compared.

14. Implementation

Our project Repo can be found here.
The separation of classes and methods into modules allows the code to be easily understood and also for Jupyter experiments to be simple. The code itself is not the most generic in that adding new components to test is possible but not too easy to do.

15. Conclusions and Plans to Update

Our assessment is that our results are not able to give conclusion to our original mission and that more experiments need to be made. Perhaps changes to the implementation of the self-play algorithm will yield a better performance that was expected.

We are currently working on seeing whether there is an underlying flaw in the self-play process as well as running the self-play experiment on a larger scale, that is more data generated, more exploration, more games to assess on.

We plan on updating this document once more note-worthy results are achieved.

Note - Conclusions still in Progress

16. Notes and Future Work

The following are possible follow ups to this project's experiments.

- Scale up experiment.
- Try different network architectures.
- Try different state representations inputs to GNN, and RNN.
- Experiment with a larger number of nodes.
- Implement experiments with different NP-problems other than TSP.
- Experiment with the length of history.

References

[1] Anonymous authors. A graph neural network assisted monte carlo tree search approach to traveling salesman problem, 2020.

[2] Julian Schrittwieser Ioannis Antonoglou Matthew Lai Arthur Guez Marc Lanctot Laurent Sifre Dharshan Kumaran Thore Graepel Timothy Lillicrap Karen Simonyan Demis Hassabis David Silver, Thomas Hubert. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, Dec 2017.

[3] Darius Irani Felix Parker. Alphattsp: Learning a tsp heuristic using the alphazero methodology, 2019.

[4] Thomas Hubert Karen Simonyan Laurent Sifre Simon Schmitt Arthur Guez Edward Lockhart Demis Hassabis Thore Graepel Timothy Lillicrap David Silver Julian Schrittwieser, Ioannis Antonoglou. Mastering atari, go, chess and shogi by planning with a learned model, Feb 2020.

[5] Ralf C. Staudemeyer and Eric Rothstein Morris. Understanding lstm – a tutorial into long short-term memory recurrent neural networks, Sep 2019.