2CS492(I) Introduction to Deep Learning Final Project Report

Interactive RL Playground

Team 10

20180493 Jung In Rhee 20140005 Minsoo Kang 20180424 Tae Young Yoon

Github Link: https://github.com/jirheee/CS492-Team-Project

Contents

1. Introduction

Artificial Intelligence(AI) is a term that is being used more and more everyday. However, for most people it is hard to get familiar with AI since it contains relatively new technologies and jargon such as artificial neural networks or anything related to machine learning, which are difficult to understand without deep prior knowledge of mathematics and computer science. To bridge the gap between people and neural networks, google tensorflow provides Neural Network Playground[1], where people can easily construct neural networks to classify linearly inseparable data. Inspired by this playground, we create an Interactive Reinforcement Learning(RL) playground, where even users without any background in AI can train and evaluate their own RL agent. We chose the RL domain because it is arguably better known than any other AI tasks due to the AlphaGo event, but it still has limited beginner level resources compared to the classifying tasks of the aforementioned prior work.

2. Main Method

In our playground, users can build their own RL agents for playing gomoku. We chose gomoku for its familiarity and simplicity. For the agent to learn, an environment of the game should be built where the agent can get game states and rewards corresponding to the actions it takes. Fortunately, there is an implementation of the gomoku environment for RL already in [2], so we decided to use this environment.

Now we build an RL agent based on AlphaZero[3], the powerful RL algorithm for playing games such as Go, which has a similar state and action space with gomoku. Alphazero agents use a deep neural network to estimate the value of an action. Our playground helps users build this neural network with basic building blocks such as Convolution Layer, BatchNorm, and ReLU.

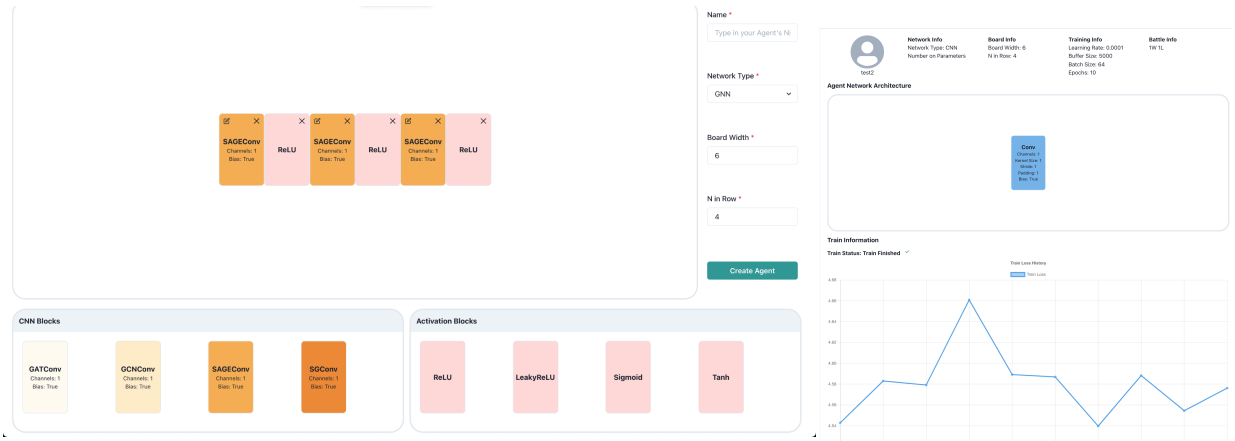
Although vanilla AlphaZero uses Convolution Neural Network(CNN) architecture to evaluate the state of the game board, we tried another method called Graph Neural Network(GNN). The GNN model transforms the graph into a lower dimension vector space whilst maximally preserving properties like graph structure and information by message passing between connected nodes. We can consider a game board as a graph intuitively, and expect that the GNN model can capture the relationship between nodes, which is important in playing a gomoku. Therefore, we provide users an additional neural network option GNN as well as CNN. Specifically, users can choose various GNN layers, for example, GCNConv, SGConv, and SAGEConv. Those layers can be easily implemented by the Pytorch Geometric Library[4].

There is one remaining question when exploiting GNN architecture, how to use node embedding acquired from the GNN model for estimating the value of an action. Unlike CNN, which has output of a single vector, GNN returns node embeddings, a list of vectors. We considered various options. FIrst option is simply averaging all node embeddings into a single vector. This approach is very simple but a lossy computation. Second option is concatenating all node embeddings. In this approach, we can save all node information with a cost of large dimension. Since the first approach shows a poor performance, we chose the second approach to implement GNN architecture.

Our end product has two parts: the client part and the server part. Client side application is where the users can build, train and monitor their agents. Server's IO layer takes in requests from the clients and creates corresponding (1)backend python processes and (2)manages agent information with agent DB tables and JSON files. The python processes, according to the request and JSON files, will create models, train them, and play games. The IO layer takes in the output files and logs made by these threads, and stores them in the database and pushes the generated data to client side.

There are 3 main pages in the client side application: Agent > Create, Agent > Manage, and Battle. This section will focus more on the UI elements and available actions for the clients, and details of how the backend works according to these actions will be covered in the next section.

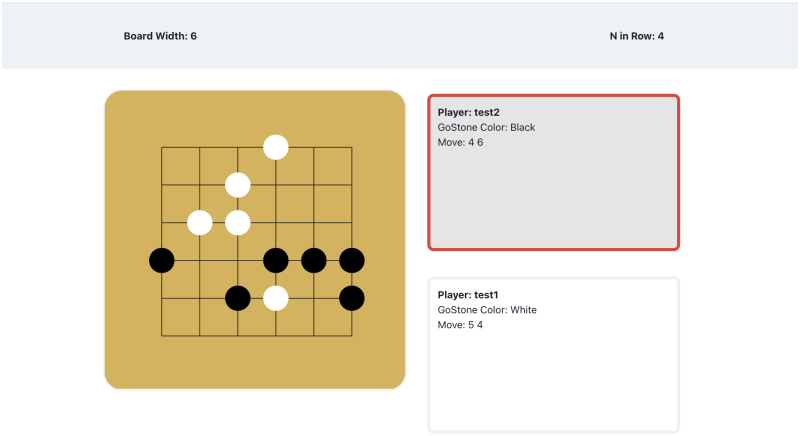
The <Create> page is where the users can get experimental and try building neural networks like building blocks. Since the target users of the product are mostly beginners to ML/RL, we made the agent graphically & interactively configurable. We thought this would help those who are not familiar with certain ML frameworks and let them focus on core ML/RL concepts appearing in the service. (1) They can choose to use CNN or GNN, (2) they can add/delete new layers to their model with clicking, (3) and they can configure each layer with the parameters according to each type of layers.



<Figure: Create Page and Manage Page>

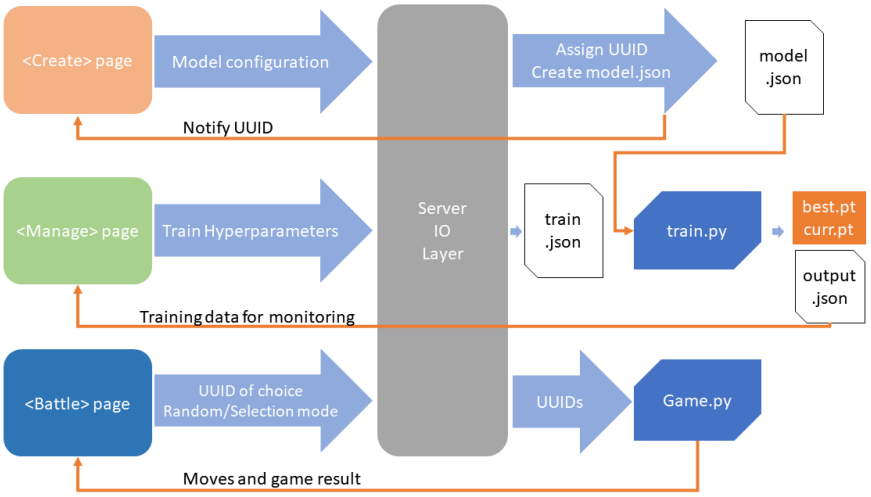
In the <Manage> page, users can see relevant information and stats of the models such as type of the network, training hyperparameters and its training progressions. Users can also start the training of the models in this page. After setting learning rate, epochs etc. the training progression will be updated in real time. Since training can take a long time depending on hyperparameters or model architecture, users can just start the training and come back after a few hours and come back to the website to see the train results.

The <Battle> is where users can test the performance of their agents against the others. For the opponents, users can choose to fight against a random model or a model of their choice from the database.



<Figure: Battle Page>

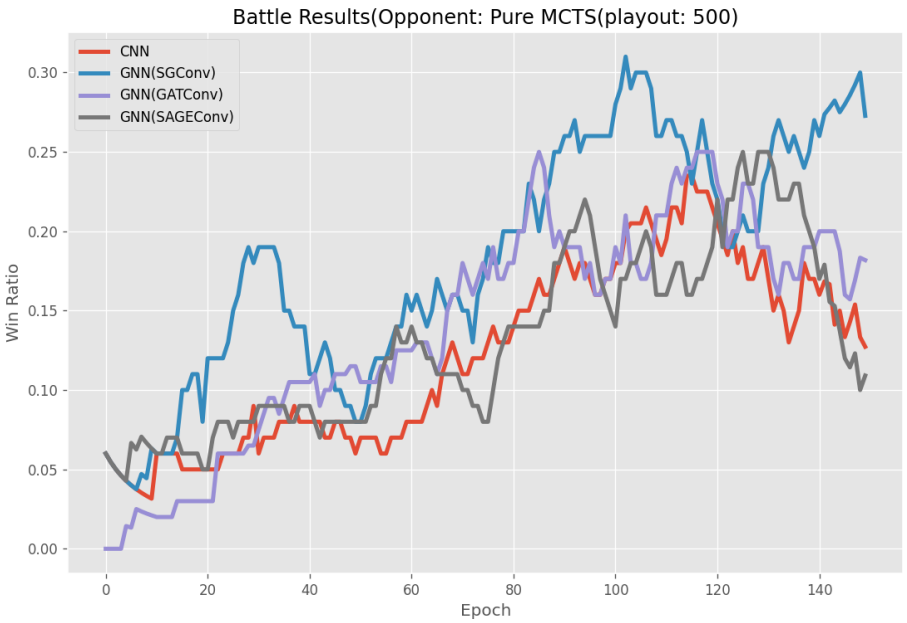
From the server side, when the user creates a model, the server stores the model information in DB and creates a model.json file with configured information. The user then takes the UUID to the <Manage> page. Then the server fetches any relevant information according to the UUID from the database and sends it back to the client. After the user clicks on ‘start train’, the server takes in the hyperparameters and hands the parameters in .json format to a python thread of ‘train.py’. The server simultaneously fetches output data and sends it back to the client to show. The agent-training python process generates checkpoints and training progression data. In the <Battle> page, when the user sends in a battle request along with the UUIDs, the server starts a python thread of ‘game.py’ and reports back the moves and result of the game.



<Figure: Software Architecture>

3. Results and Analysis

We wanted to check whether GNN is actually more powerful than CNN architecture in playing a gomoku. We trained GNN models with various GNN layer types and CNN model, which is already implemented for training AlphaZero with 400 MCTS playouts. Then compete with pure MCTS players with 500 playouts. As shown in the Figure, we can get comparable results with CNN models using GNN models. Furthermore, when we compare the GNN model with 4 SGConv layers with node dimension size 6 and CNN model, GNN model outperforms even with a relatively small number of parameters.



<Figure: Win ratio of CNN and GNN models>

	CNN	GNN
# of Parameters	105,317	30,277
Training Time(100 epochs)	1h 11m 38s	1h 14m 01s
Battle Results	0 Win 100 Lose	100 Win 0 Lose

<Table: Battle between CNN and GNN model>

4. Conclusion

We create an interactive RL playground where users can build their own agent with basic neural network building blocks, train their agent, monitor it, and battle with other opponents. We also introduced the GNN model to represent a state more appropriate for playing a gomoku. We hope that our playground is helpful for people who are not familiar with RL. Furthermore, we anticipate our playground can be a new sandbox with various ideas for more general and powerful RL applications.

5. Contribution

- **Jung In Rhee**
  - Developed frontend of the web service using React.js
  - Developed backend apis to create/train/battle agents.
- **Tae Young Yoon**
  - Implementation of Alphazero for playing a gomoku
  - Implementation of GNN architecture and conduct experiments for comparing performance between CNN and GNN architecture.
- **Minsoo Kang**
  - Connecting server IO layer with the python processes.
  - Developing APIs of the python modules for communicating with the backend api.

6. References

[1] Neural Network Playground - https://playground.tensorflow.org

[2] Alphazero_Gomoku - https://github.com/junxiaosong/AlphaZero_Gomoku

[3] AlphaZero: Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm - https://arxiv.org/abs/1712.01815

[4] Torch Geometric - https://pytorch-geometric.readthedocs.io/en/latest/

Appendix

Network Architecture

<CNN>

	Input → Output shape	Layer Information
Layer 1	(h, w, 4) → (h, w, 32)	Conv(32, K3, S1, P1), BatchNorm, ReLU
Layer 2	(h, w, 32) → (h, w, 64)	Conv(64, K3, S1, P1), BatchNorm, ReLU
Layer 3	(h, w, 64) → (h, w, 128)	Conv(128, K3, S1, P1), BatchNorm, ReLU
Layer 4	(h, w, 128) → (h, w, 1)	Conv(1, K1, S1, P1), BatchNorm, ReLU

<GNN(SGConv)>

	Input → Output shape	Layer Information
Layer 1	(h, w, 4) → (h, w, 6)	SGConv(6), ReLU
Layer 2	(h, w, 6) → (h, w, 6)	SGConv(6), ReLU
Layer 3	(h, w, 6) → (h, w, 6)	SGConv(6), ReLU
Layer 4	(h, w, 6) → (h, w, 6)	SGConv(6), ReLU

After CNN/GNN architecture, state representations are flattened/concatenated and the computation is separated into two different heads, for computing the policy p and the value v. p is computed using 2 fully-connected layers from input of size to output of size |A| (number of possible actions), followed by a log-softmax operation, yielding the probability vector. v is computed using 2 fully-connected layers from input of size to output of size 1, followed by a tanh nonlinearity function.