Method

To build an AI which plays Gomoku we used a blend of some hard coded moves and heuristics from a neural network. Our version of GomokuAI begins its approach by checking if we can win in the current turn. If no open 4 in a row are found in our favor the AI checks if the opponent is winning in the next move. If no serious threats are found our AI chooses to build an open 3 position by choosing a move that minimizes the heuristic value provided by our neural network, thus creating a threat. If a serious threat is found, we return the action that blocks that threat. If a non-serious threat like an open 3 for the opponent is found, GomokuAI blocks that threat by choosing a move that minimizes the heuristic provided by our neural network. If no threats are found, winning moves or open 3s found on the board the AI will choose a move to perform using the heuristics from the neural network.

Neural Network

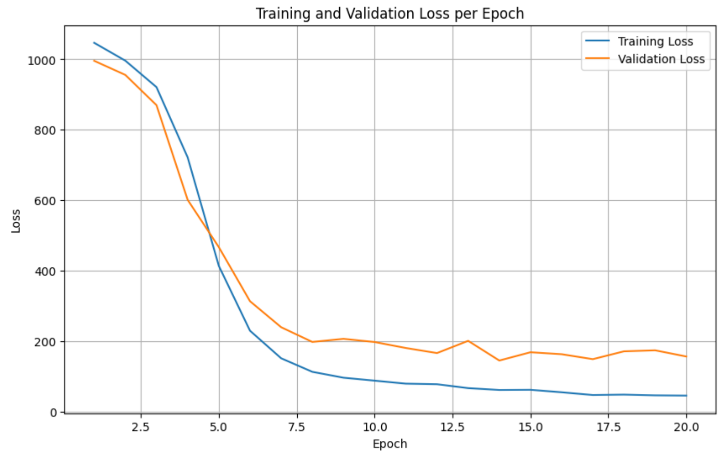
We wrote a script that ran 2000 games and saved the outputs in a text files. We parsed these files to create a dataset with columns for Game ID, move number, Player (Min or Max), the board for that move, which player won, final score and discounted score. The discounted score is calculated as

$$Discounted\ Score = Final\ Score\ \times \gamma^{\{Total\ Moves - Move\ Number - 1\}}$$

The final move has the minimum score (best for the min player) and low rewards are given to the initial moves. We experimented with a few values of the discount factor too see how it was affecting the loss and settled on a discount factor of 0.9. Thus, creating a value output for the value network [1].

Only picking the moves made by the min player we created 2 models using PyTorch [2]. For the first model we considered all games but since a losing game has positive rewards for the min player the neural network could not work out the heuristics of the initial moves as they would have high positive as well as high negative rewards. Thus, for our final model we considered only the games where Min player was the winner.

We played with a few architectures but settled for 3 convolution layers and 3 fully connected layers with a dropout of 0.3 between the fully connected layers 1 and 2. The dropout helped in preventing overfitting the data. We chose Mean Square Error as our loss function. The final loss curve shows that our model was overfitting a little bit still. We experimented with the learning rates and settled on a learning rate of 0.001.



Discussion

We ran the performance code multiple times to evaluate all different versions of our AI. With only the hard code (threat analysis and win moves) our AI was only able to draw a few games and was losing most games. With the heuristics from the CNN neural network the AI was able to win or draw between 60-70% of the games.

To improve the model, we can do some more hyperparameter tuning, looking at different learning rates, dropouts or adding more layers. We can also experiment further with the discount factor or choose a better formula to select the output heuristic. The most difficult challenge was to combine this approach with a Monte Carlo Tree Search Algorithm [1] which we are still working on.

References

[1] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., ... & Hassabis, D. (2017). Mastering the game of go without human knowledge. nature, 550(7676), 354-359.
[2] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., … Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32* (pp. 8024–8035). Curran Associates, Inc. Retrieved from http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
[3] Harris, C. R., Millman, K. J., Van Der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., ... & Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357-362. (Used in code)