Assignment 2: Alpha-Go Zero

Overall Design

We used a similar approach as mentioned in the original paper of Alpha-Go Zero by doing self-play reinforcement learning along with Monte-Carlo Tree Search(MCTS) to get the target policy and value for a board state. The following diagram from the same paper is shown for representational purposes. It captures our algorithm perfectly, our bot plays a game s_1, s_2, ... s_T against itself. In one iteration we train our Neural Network over some number of mcts simulations over some number of episodes and finally at the end of the iteration update our network if it wins over the older network but not if it loses. While simulating the mcts tree we use the U(s, a) function dependant on Q(s, a) & N(s, a) values stored dynamically and P(s) and V(s) estimated using the neural network, described in detail in the paper which balances exploration and exploitation.

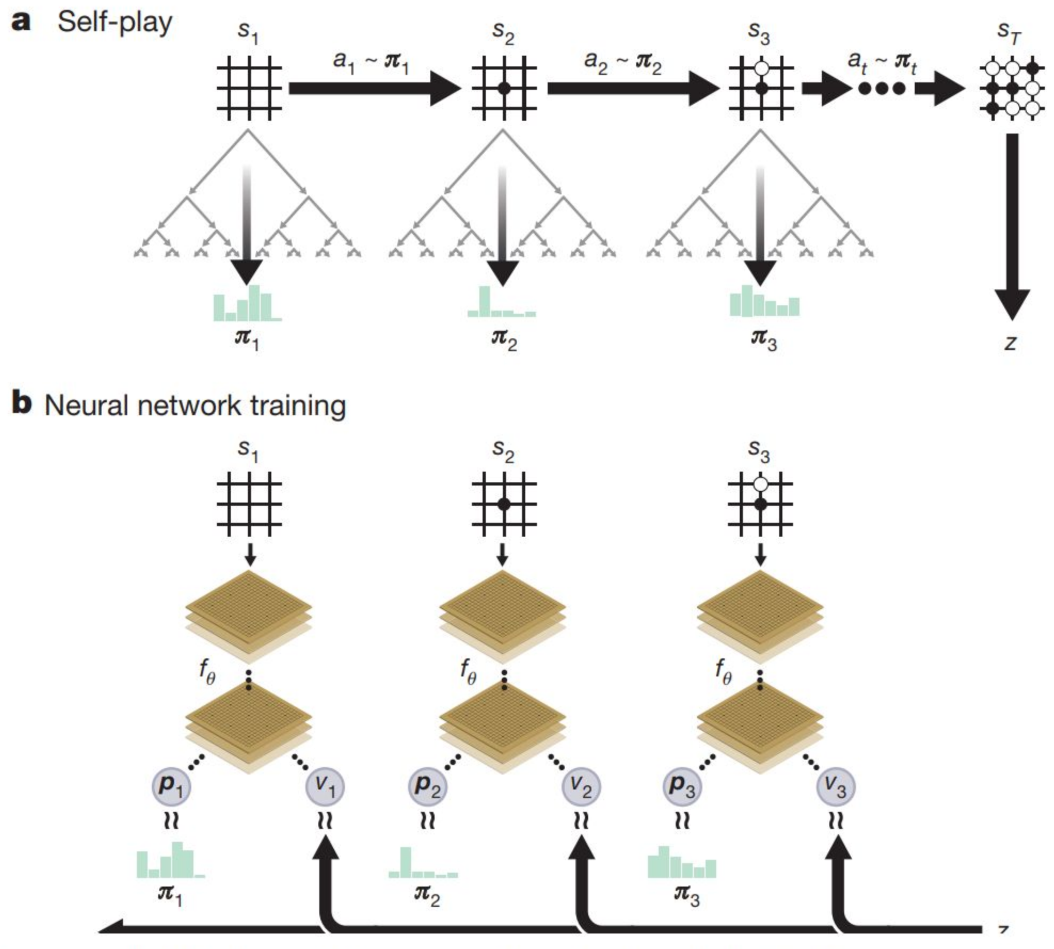

Figure 1 | Self-play reinforcement learning in AlphaGo Zero. a,

Hyperparameter Tuning:

1. **Number of Residual Blocks:** We experimented over a different number of residual blocks in our neural network starting from 0 to 19 and noticed the learning rate (by seeing the change in loss) and decided to use 0 blocks as it was faster and not significantly worse.
2. **Adam Optimizer**: We tried different learning rates and parameters $Beta_1$ & $Beta_2$ and decided on using the default values of Beta that are 0.9, 0.999; and $1e^{-4}$ learning rate.
3. We tuned over the **number of iterations**, the **number of episodes per iteration** and **number of MCTs simulation** per episode and played matches against different parameter models to select the best model with Num. of Iteration = 20, Num. of Episodes = 10 & Num. of MCTS Simulation = 75.
4. **Exploration Parameter:** We kept an exploration parameter that returns the best action using a greedy approach after we go below a particular depth in MCTS Simulation and stops exploring.
5. **Neural Net update threshold:** After each iteration, we play six matches between the old and the new network and update the old network only when the ratio of wins is greater than the Neural Net update threshold.

Experimental Findings & Tricks used to solve them:

We initiated by implementing the actual architecture and the loss function as in the paper however realized that with given compute and the time constraints a successful model won't be possible so the following methods were adopted:

1. **Reducing the number of model parameters:** Reducing the number of residual layers helped fasten the learning significantly
2. Initial Random Adversary: We initially trained our model against a random bot which randomly picked a valid action as earlier incomplete self-play both the bots learned to pass and the game didn't proceed further and to learn something effectively.
3. **Training with past generation:** Training was found to be susceptible to exploding gradients where the model ends up following in local valleys where the dissent towards a bad policy decision was further enhanced by repetitive self-play over the same policy. As a fix, we chose an adversary which is two generations older in self-play.

4. **Policy Loss Function:** We found that very smoothing for logarithm was dominating the eventual action probabilities which led to an equiprobability distribution which was detrimental for learning so we used Hadamard product of the target and the output values which led to stable training and still principally in the same optimization objective.
5. **Normalizing Value and Policy Loss:** For training of a successful model we found that a comparable value and policy loss is desirable as overemphasize on either one of them led to instability in the gradient descent.

Special Mention: Implementing the entire codebase in C++ using libtorch

A considerable amount of effort was spent in this work to implement the entire Alpha-Go Trainer in C++ - from **MCTS** to the **neural network** (using libtorch) to even writing a **valid simulator** that tells if the board is valid and executes the corresponding move.
Another ordeal was to get the libtorch code to be compatible with the HPC server.

This was considerably fast as compared to the python version of the code. In our experiments we found it to be **23x faster**. Especially because the majority train time is spent in MCTS.

We could not use this as the rules of our simulator were based on the Chinese Rules. However, in the actual pachi-py implementation there were many other nuances that were difficult to understand through their codebase and finally, we had to take the difficult decision of rejecting this work.