**AlphaGo Review:**
The goals of the alpha go paper were to use deep learning to 1) Narrow the search tree for a search algorithm. 2) Learn the value function for a given position. These main 'functions' are used to augment monte carlo simulation. In the final algorithm, these were combined with various weights for generating and evaluating a search. This allows the algorithm to learn without hand-coded evaluation functions (like those in the assignment), opening books, or variants on those using linear combinations of features like most game playing programs.

The baseline architecture was convolutional neural network which take representations of the board as 48 channels of board features, including player's positions, open spaces, several common 'tactical' features (such as whether a move is already enclosed by one's pieces, and is therefore useless), and a few normalizing channels (all ones, all zeros) to indicate the board extents, for a final representation of 19 x 19 x 48.

The networks were trained by a deep learning training pipeline consisting of

| | **Input** | **Output** | **Goal** |
|---|---|---|---|
| 1) Supervised Training of policy networks | position | Best move(s) | decrease ST |
| 2) Reinforcement Training of policy | position | Best move(s) | improve decreased ST |
| 3) Supervised Training Value network | position | ~likelyhood of win | Value Function |
| 3) Reinforcement Training Value net | position | ~likelyhood of win | improve value function |

The training approach is a combination of bootstrapping and tournament. First, supervised training of policy was done by giving the network positions from professional Go games and having it predict the next move. After each batch of games, loss is back-propagated through the network with Stochastic gradient descent. The idea being that if a prediction is not correct, the values contributing to that incorrect prediction are adjusted over in proportion to their contribution to the incorrect prediction.
The Policy network is then improved by playing previous versions of itself in a tournament. AlphaGo used a reinforcement learning strategy here to give positive reward all actions in games where the player won, and negative reward when the player lost. ***This is where I got stuck in my implementation – I tried several ways of this and each time got exploding gradients, therefore I do not fully understand this.*
There is a similar process in training versions of a network that gives a single evaluation of the position. In the isolation context, this takes the place of the handcrafted evaluation function.

The final algorithm uses the best of each of these networks as the way to chose moves to iteratively sample and evaluate resulting positions in at tree search. Top moves are chosen for search with the Policy network. At successive steps of traversal, the edge is updated with the mean of its child evaluations. This is more appropriate in a game with a lot of uncertainty such as Go, than in more narrow game. In the narrow Isolation case, there one can assume that a 'best position' can be found by an opponent who is calculating similar trees. In wider search space where evaluation functions are more prone to some style or other, a running average will tend to less risky play. It is also a nice feature that if a terminal state is found in the game, that moving average will still go to the terminal value.

The results of the AlphaGo expirement are well publicized. The final engine defeated top ranked human opponent 8/10 games (5/5 formal games). The alpha go engine also defeated all computer opponents, which relied mostly on monte carlo simulation of games. In computer vs computer play, programs were allowed 5seconds per move, and alphago wins 90% or more of games. The nice result is that while evaluation by tree search of simulations is too expensive at a certain depth, reducing the choices with a fast function gives a huge performance boost. Adding the step of 'seeing what looks good' before calculating further is at least how strong chess players operate, as opposed to pure tree search, when a favorite move is chosen after you are done (or run out of time)

**My (working) Implementation of AlphaGo on Isolation**

| **My Network architecture (Policy baseline):** | **AlphaGo Network architecture** |
|---|---|
| k = 64 (also tried 16, but unstable) | k=128 (also with 256, 384) |
| 1: Convolution with k filters, size 3, stride 1, pad 2, relu | 1: Convolution w/ k filters, stride 5, pad 2, relu |
| 2-5: Convolution with k filters, size 3, stride 1, relu | 2-12: Convolution w/ k filters, size 3, stride 1, relu |
| 6: Convolution with 1 filter, size 1, stride 1, bias, relu | 13: Convolution 1 filter, 1 3, stride 1, bias, softmax |
| **Value Networks Architecture (goal is to get one answer of position value):** | |
| 7: (not implemnted) FC layer with 128 units | 14: Additional fully connected layer, 256 units |
| 8: (not implemnted) FC layer with 1 tanh | 15: Fully Connected Layer with 1 tanh. |

** my layer 6 is a relu which was an oversight on my part, which I realized too late. The relu function cuts off most inputs, leaving only a few activations. The softmax just normalizes everything, which would allow network to maintain a representation of board state going into the value network. If the value network sees just a few activations, it has no way of knowing how good they are (except by magnitude, which is wrong by intuition)

**Features Preprocessing Comparison:**

My Final network input was 7x7x7. channels were:

| | |
|---|---|
| 1: Player's position | (same as Alphago) |
| 2: Opponent's position | (same as Alphago) |
| 3: Legal Moves | (same as Alphago, generated for each move by game object) |
| 4: Ones | (same as alphago – used to control for padding) |
| 5: Open moves | |
| 6: Closed moves (not including self and opponent) | |
| 7: Zeros (same as alphago – not sure why, but I kept this) | |
| 8: Opponents moves (I ended up not using this, probably a mistake) | |

**Implementation:**

Networks were created in Pytorch. This was mostly because I wanted to learn pytorch, and since the alphago training pipeline has a decent amount of adding layers and initializing networks from other networks, I thought the dynamic model pytorch has would be a good way to do this. Training for policy network was done for 3 epochs on 1.5 million positions on a cuda 980 gpu. Successive self play experiments (defined in controller.py) were done on a shared K80 gpu instance on floydhub.

**Results (or current state thereof):**

Policy

The base line policy network training ended up at a stable ~50% prediction rate. The smaller version was still fluctuating over 4 epochs. Predictions are legal moves at ~99% rate, so at least that approximation exists.

Both large and small networks had roughly 70% win rates vs random initially. They performed better against the central bias player 80% , (see training log for backup) but horribly against Open Move scores and better heuristics. This is probably because the network has learned some form of central space maximization, since some of the samples are are from Central Move algorithm, and there is probably a central bias no matter what. If network has not learned to do an approximate calculation, it would make sense it defaults to this.

My main problem is reinforcement learning (which may not be necessary, but I am falling to sunk cost fallacy here). AlphaGo has the loss of each action in a minibatch of games reinforce by whether or not the game was won or lost. This gives me exploding gradients if I use the standard pytorch REINFORCE method. This is probably because when I sample from some distribution of possible last moves, often I will get not the best move, as opposed to when I just use the maximum activation of outputs. This produces a lot of illegal moves, and the network quickly goes worse than random.

I thought a workaround was to just use a standard loss on moves in games that were won by the network (positive reinforcement??) but this did pretty much nothing. After a day of training on floydhub, the network had not improved significantly against any opponent other than its past iterations, which is interesting in itself.

code:
.controller.py – main entry point for running experiments and tournaments (play_game_rl function is a mess, this is where I am trying to figure out how to apply REINFORCE per alphago)
.aindnn/slnn.py – pytorch modules for networks.

.nn_players.py – IsolationPlayer subclass for using the neural net in a game
.loader.py – dataloader from generated games dataset.
.dataloaders.py – some loading and saving utilities
.outputx/ - folder for network checkpoints
.misc/ - folder with logfiles, and floydhub outputs