Summary:

Mastering the game of Go with deep neural Networks and tree search

- A BRIEF SUMMARY OF THE PAPER'S GOALS OR TECHNIQUES INTRODUCED

The game of Go has long been viewed as the most challenging of classic games for artificial intelligence Owing to its enormous search space and the difficulty of evaluating board positions and moves. This paper discusses a new approach to solve the game of Go using 'value networks' to evaluate board positions And 'policy networks' to select moves. These deep neural networks were trained by a novel combination of Supervised learning from human expert games, and reinforcement learning from games of self-play. Without any look ahead search, the neural networks play Go at the level of state of-the-art Monte Carlo tree search programs that simulate thousands of random games of self-play. In addition, a new search algorithm that combines Monte Carlo simulation with value and policy networks is been introduced. Using this search algorithm, the program AlphaGo achieved a 99.8% winning rate against other Go programs, and defeated the human European Go champion by five games to zero. This is the first time that a computer program has defeated a human professional player in the full-sized game of Go, a feat previously thought to be at least a decade away.

AlphaGo is made up of a number of relatively standard techniques: behavior cloning (supervised learning on human demonstration data), reinforcement learning , value functions, and Monte Carlo Tree Search (MCTS). However, the way these components are combined is novel and not exactly standard. In particular, AlphaGo uses a SL (supervised learning) policy to initialize the learning of an RL (reinforcement learning) policy that gets perfected with self-play, which they then estimate a value function from, which then plugs into MCTS that uses the SL policy to sample rollouts. In addition, the policy/value nets are deep neural networks, so getting everything to work properly presents its own unique challenges (e.g. value function is trained in a tricky way to prevent overfitting).

Training Pipeline:

1. **Supervised learning of policy networks.**
   For the first stage of the training pipeline, the expert moves were predicted using supervised learning. The SL policy network $p\sigma(a \mid s)$ alternates between convolutional layers with weights σ, and rectifier nonlinearities. A final softmax layer outputs a probability distribution over all legal moves a. The input s to the policy network is a simple representation of the board state,The policy network is trained on randomly sampled state-action pairs (s, a), using stochastic gradient ascent to maximize the likelihood of the human move a selected in state s. a 13-layer policy network is trained from 30 million positions. The network predicted expert moves with an accuracy of 57% using all input features, and 55% using only raw board position and move history as inputs. Small improvements in accuracy led to large improvements in playing strength,larger networks achieve better accuracy but slow latency in evaluation.

2. **Reinforcement learning of policy networks**.
   The second stage of the training pipeline aims at improving the policy network by policy gradient reinforcement learning (RL). The RL policy network $p\rho$ is identical in structure to the SL policy network, and its weights ρ are initialized to the same values, $\rho = \sigma$. We play games between the current policy network $p\rho$ and a randomly selected previous iteration of the policy network. Randomizing from a pool of opponents in this way stabilizes training by preventing overfitting to the current policy. We use a reward function r(s) that is zero for all non-terminal time steps $t < T$. The outcome $zt = \pm r(sT)$ is the terminal reward at the end of the game from the perspective of the current player at time step t: +1 for winning and −1 for losing. Weights are then updated at each time step t by stochastic gradient ascent in the direction that maximizes expected outcome

3. **Reinforcement learning of value networks**.
   The final stage of the training pipeline focuses on position evaluation, estimating a value function $vp(s)$ that predicts the outcome from position s of games played by using policy p for both players
   Ideally, we would like to know the optimal value function under perfect play $v^*(s)$;

in practice, we instead estimate the value function v pρ for our strongest policy,
using the RL policy network pρ. We approximate the value function using a value network vθ(s)
with weights θ. This neural network has a similar architecture to the policy network, but outputs
a single prediction instead of a probability distribution.
We train the weights of the value network by regression on state-outcome pairs (s, z), using stochastic
gradient descent to minimize the mean squared error (MSE) between the predicted value
vθ(s), and the corresponding outcome z The naive approach of predicting game outcomes from data
consisting of complete games leads to overfitting. The problem is that successive positions are strongly
correlated; differing by just one stone, but the regression target is shared for the entire game. When trained
On the data set in this way, the value network memorized the game outcomes rather than generalizing to
new positions, achieving a minimum MSE of 0.37 on the test set, compared to 0.19 on the training
set. To mitigate this problem, we generated a new self-play data set consisting of 30 million distinct positions,
each sampled from a separate game. Each game was played between the RL policy network and
itself until the game terminated. Training on this data set led to MSEs of 0.226 and 0.234 on the training and
test set respectively, indicating minimal overfitting. A single evaluation of vθ(s) also approached the accuracy
of Monte Carlo rollouts using the RL policy network pρ, but using 15,000 times less computation.

4. Searching with policy and value networks

AlphaGo combines the policy and value networks in an MCTS algorithm
that selects actions by lookahead search. Each edge (s, a) of the search tree stores
an action value Q(s, a), visit count N(s, a), and prior probability P(s, a).
The tree is traversed by simulation (that is, descending the tree in complete games without backup),
starting from the root state. At each time step t of each simulation, an action at
is selected from state st so as to maximize action value plus a bonus that is proportional to the prior
probability but decays with repeated visits to encourage exploration.
When the traversal reaches a leaf node sL at step L, the leaf node may be expanded.
The leaf position sL is processed just once by the SL policy network pσ. The output probabilities are
stored as prior probabilities P for each legal action a,
P(s, a)=pσ(a|s) . The leaf node is evaluated in two very different ways:
first, by the value network vθ(sL); and second, by the outcome zL of a
random rollout played out until terminal step T using the fast rollout policy pπ; these evaluations are
combined, using a mixing parameter λ, into a leaf evaluation V(sL)
V(sL)=(1−λ)vθ(sL)+λzL
At the end of simulation, the action values and visit counts of all
traversed edges are updated. Each edge accumulates the visit count and
mean evaluation of all simulations passing through that edge
where sL is the leaf node from the ith simulation, and 1(s, a, i) indicates
whether an edge (s, a) was traversed during the ith simulation. Once
the search is complete, the algorithm chooses the most visited move
from the root position. It is worth noting that the SL policy network pσ performed better in
AlphaGo than the stronger RL policy network pρ, presumably because
humans select a diverse beam of promising moves, whereas RL optimizes
for the single best move. However, the value function
vθ(s)≈v pρ(s) derived from the stronger RL policy network performed

References: https://github.com/llSourcell/alphago_demo/blob/master/papers/alphago2016.pdf