TTIC 31230, Fundamentals of Deep Learning

David McAllester, April 2017

AlphaZero

AlphaGo Fan (October 2015)

AlphaGo Defeats Fan Hui, European Go Champion.

AlphaGo Lee (March 2016)

AlphaGo Zero vs. Alphago Lee (April 2017)

AlphaGo Lee:

- Trained on both human games and self play.
- Trained for Months.
- Run on many machines with 48 TPUs for Lee Sedol match.

AlphaGo Zero:

- Trained on self play only.
- Trained for 3 days.
- Run on one machine with 4 TPUs.
- Defeated AlphaGo Lee under match conditions 100 to 0.

AlphaZero Defeats Stockfish in Chess (December 2017)

AlphaGo Zero was a fundamental algorithmic advance for general RL.

The general RL algorithm of AlphaZero is essentially the same as that of AlphaGo Zero.

AlphaGo

AlphaGo trains:

1. a fast rollout policy.
2. an imitation policy network.
3. a self-play policy network.
4. a value network trained to predict self-play rollout values.

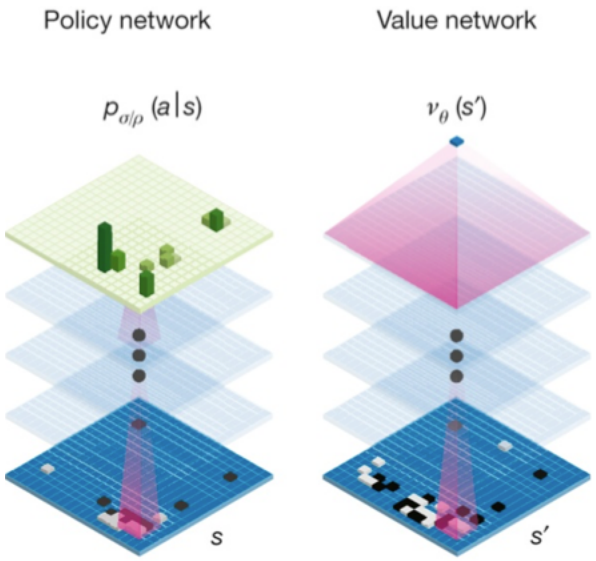
No tree search is used in training — in (3) and (4) "self-play" is just policy network rollouts.

1. Fast Rollout Policy

Softmax of linear combination of (hand designed) pattern features.

An accuracy of 24.2%, using just 2μs to select an action, rather than 3ms for the policy network.

2. Imitation Policy Network



[Silver et al.]

Networks have 13 Layers of 5×5 filters on 256 channels

The immatation policy network is trained from from 30 million positions from the KGS Go Server.

3. Self-Play Policy Network

Build a replay buffer by runing the self-play network against versions of itself to get (expensive) rollouts $a_1, b_1, a_2, b_2, \ldots, a_N, b_N$ with value z.

"Self-play" just draws moves from the policies — there is no tree search in training.

Draw rollouts from the replay buffer and update the network with REINFORCE.

$$\text{for } t \qquad \Theta_\pi \mathrel{+}= \eta \; z \; \nabla_{\Theta_\pi} \; \ln \pi(a_t | s_t; \Theta_\pi)$$

4. Value Network

Using self-play of the final self-play network policy we generate a database of 30 million pairs (s, z) where s is a board position and $z \in \{-1, 1\}$ is an outcome and each pair is from a different game.

Again, "self-play" just selects moves from the policies — there is no tree search in training.

We then train the value network on the pairs (s, z) in this database.

$$\Theta^* = \underset{\Theta}{\operatorname{argmin}}\, E_{(s,z)}\, (V(s, \Theta) - z)^2$$

Tournament Play uses Tree Search

Rollout position evaluation (Bruegmann, 1993)

Monte Carlo Tree Search (MCTS) (Bruegmann, 1993)

Upper Confidence Bound (UCB) Bandit Algorithm (Lai and Robbins 1985)

Upper Confidence Tree Search (UCT) (Kocsis and Szepesvari, 2006)

Rollouts and MCTS (1993)

To estimate the value of a position (who is ahead and by how much) run a cheap stochastic policy to generate a sequence of moves (a rollout) and see who wins.

Do a selective tree search using rollout averages for position evaluation.

(One Armed) Bandit Problems

Consider a set of choices (different slot machines).
Each choice gets a stochastic reward.

We can select a choice and get a reward as often as we like.

We would like to determine which choice is best and also to get reward as quickly as possible.

The Upper Confidence Bound (UCB) Algorithm

For each choice (bandit) a construct a confidence interval for its average reward.

$$\mu = \hat{\mu} \pm 2\sigma/\sqrt{n}$$

$$\mu(a) \leq \hat{\mu}(a) + U(N(a))$$

Always select

$$\operatorname*{argmax}_a \hat{\mu}(a) + U(N(a))$$

The Upper Confidence Tree (UCT) Algorithm

The UCT algorithm grows a tree by running "simulations".

Each simulation descends into the tree to a leaf node, expands that leaf, and returns a value.

In the UTC algorithm each move choice at each position is treated as a bandit problem.

We select the child (bandit) with the the lowest upper bound as computed from revious simulations selecting that child.

More Details

When a leaf is expanded each new leaf is assigned a value, originally by averaging random rollouts.

In AlphaGo each new leaf s is assigned the value

$$\lambda V_\Phi(s) + (1 - \lambda)V'(s)$$

where $V_\Phi(s)$ is the value of the value network and $V'(s)$ is the value of a fast rollout from s.

Each new leaf is considered to have be "visited" once by a simulation which assigned it the above value.

The simulation returns as a value to its parent nodes the largest value of the new leaves.

More Details

At a non-leaf node an AlphaGo simulation selects the move

$$\operatorname*{argmax}_a \ \hat{\mu}(s,a) + \lambda \pi_\Phi(a|s) \frac{\sqrt{N(s)}}{N(s,a)}$$

$N(s,a) \geq 1$ is the number of sims that have visited (s,a).

$N(s)$ is the number of sims that have visited s.

$\hat{\mu}(s,a)$ is the average value of the sims that have visited (s,a).

$\pi_\Phi(a|s)$ is the **imitation learned** action probability.

Eventually we select the root move $\operatorname{argmax}_a \ N(s,a)$.

AlphaGo Zero

- There is no fast rollout policy or imitation policy — just a self-play policy network and a self-play value network.

- No rollouts are ever used — just self-play UCT with leaf values computed by the value network alone.

- The networks are replaced with Resnet.

- A single dual-head network is used for both policy and value.

Learning from Tree Search

UTC tree search is used to generate a complete self-play game. Move selection during simulations is done using

$$\underset{a}{\text{argmax}}\ \hat{\mu}(s,a) + \lambda \frac{\pi_\Phi(a|s)}{N(s,a)}$$

Each self-play game has a final outcome z and generates data (s, π, z) for each position s in the game where

$$\pi(a) \propto N(s,a)^\beta$$

where β is a temperature hyperparameter and z is the final value of the game.

This data is collected in a replay buffer.

Learning from Tree Search

Learning is done from this replay buffer using the following objective on a single dual-head network.

$$\Phi^* \;=\; \underset{\Phi}{\operatorname{argmin}}\; E_{(s,\pi,z)\sim\text{Replay},\, a\sim\pi} \begin{pmatrix} (v_\Phi(s) - z)^2 \\ -\lambda_1 \log \pi_\Phi(a|s) \\ +\lambda_2 ||\Phi||^2 \end{pmatrix}$$

Selecting Root Moves

During self-play we have two forms of move selection.

- Selecting a child during a simulation as part of tree-growth.
- Selecting a root move to make progress in the actual play of the game.

In AlphaGo root moves were selected by $\text{argmax}_a N(s, a)$.

In AlphaZero self-play, exploration is maintained by selecting root moves in proportion to $N(s, a)$ for the first 30 moves and then selecting $\text{argmax}_a \ N(s, a)$ beyond 30 moves.

Throughout the game a random root move is used with probability $\epsilon << 1$.

Training Time

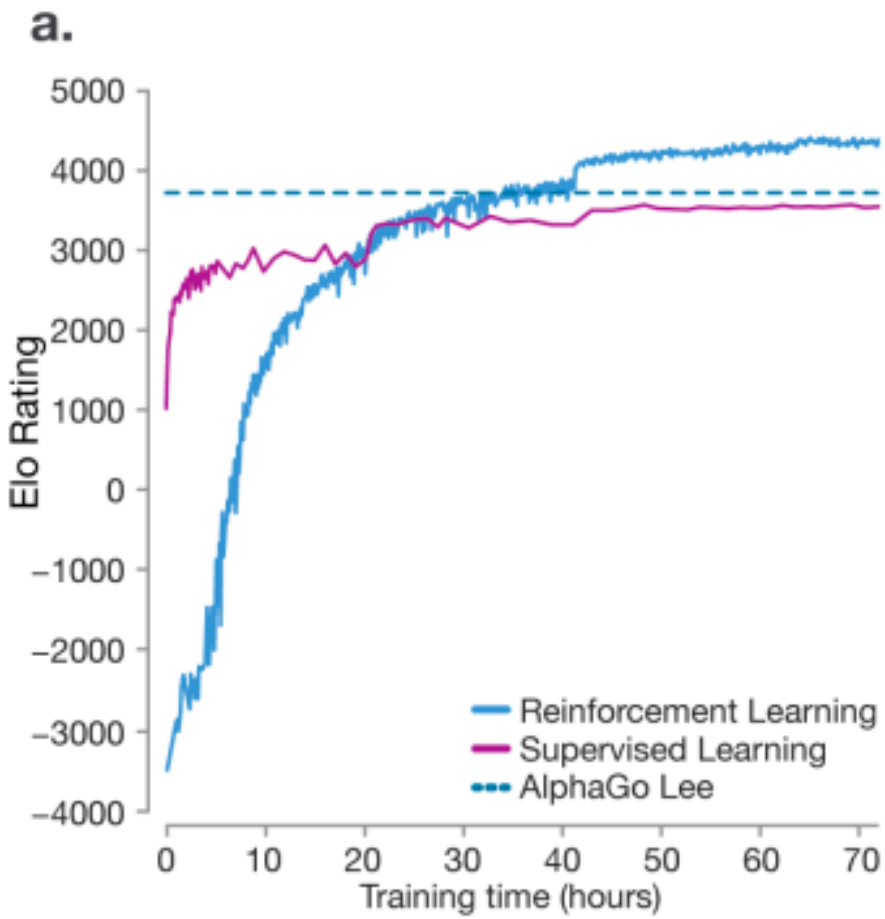
4.9 million games of self-play

0.4s thinking time per move

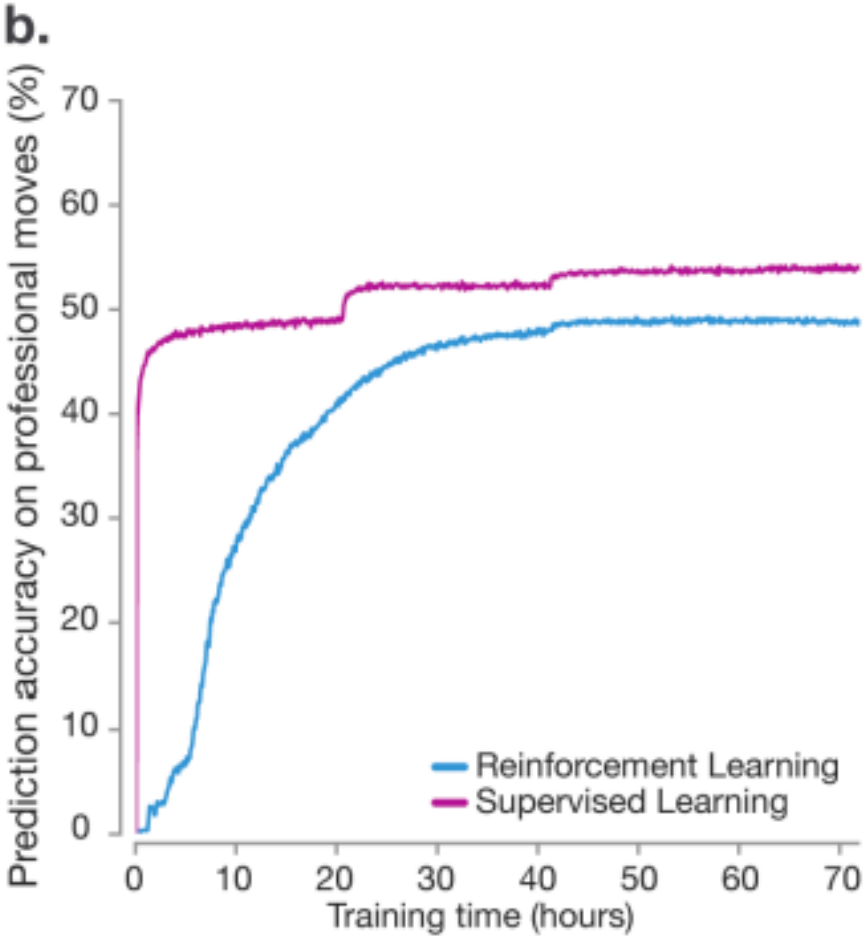
About 8 years of thinking time in training.

Training took just under 3 days — about 1000 fold parallelism.

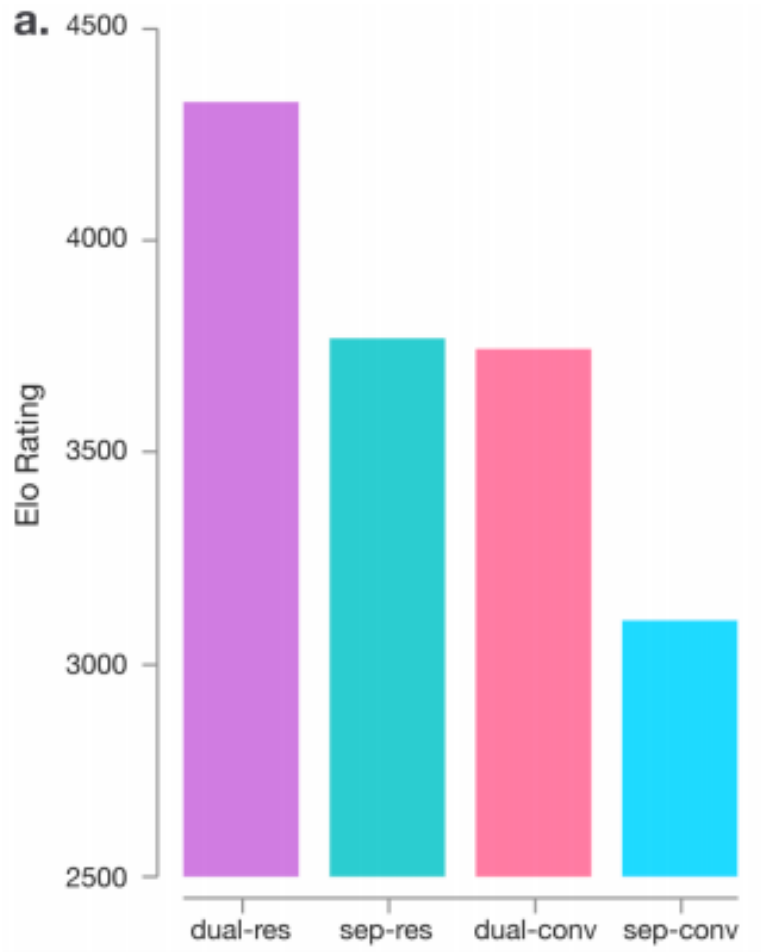
Elo Learning Curve



Learning Curve for Predicting Human Moves



Ablation Study for Resnet and Dual-Head



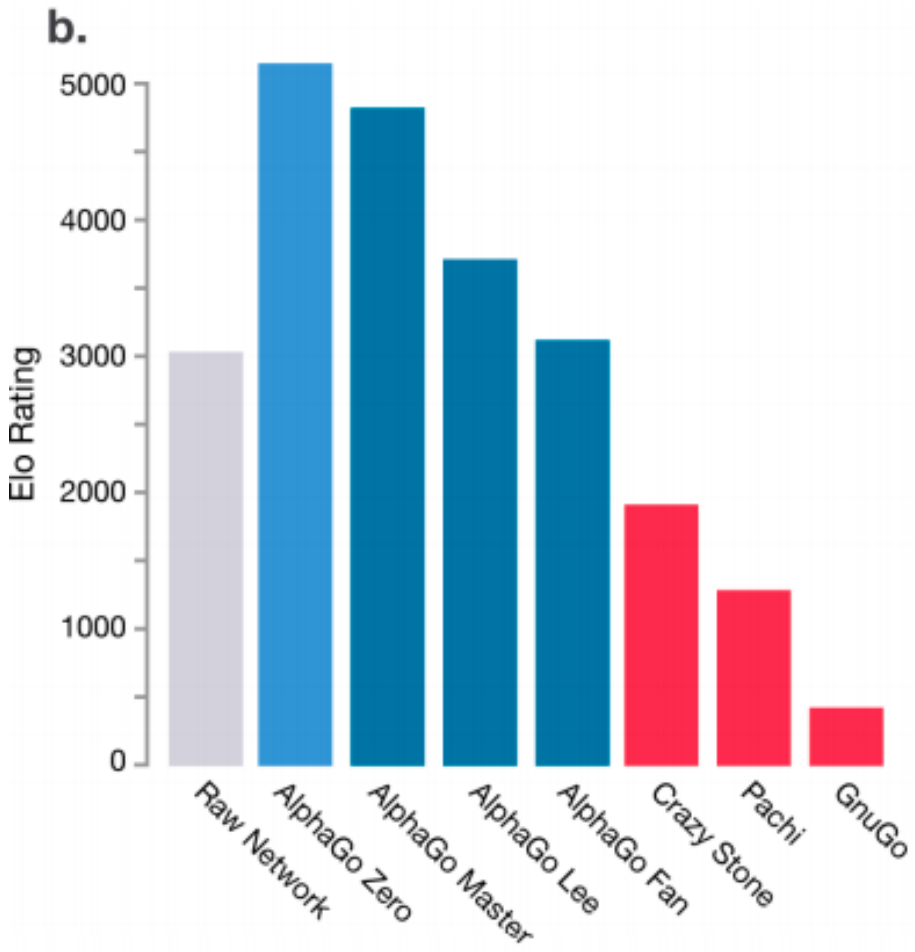
Increasing Blocks and Training

Increasing the number of Resnet blocks form 20 to 40.

Increasing the number of training days from 3 to 40.

Gives an Elo rating over 5000.

Final Elo Ratings



AlphaZero — Chess and Shogi

Essentially the same algorithm with the input image and output images modified to represent to game position and move options respective.

Minimal representations are used — no hand coded features.

Three days of training.

Tournaments played on a single machine with 4 TPUs.

Alpha vs. Stockfish

From white Alpha won 25/50 and lost none.

From black Alpha won 3/50 and lost none.

Alpha evaluates 70 thousand positions per second.

Stockfish evaluates 80 million positions per second.

Checkers is a Draw

In 2007 Jonathan Schaeffer at the University of Alberta showed that checkers is a draw.

Using alpha-beta and end-game dynamic programming, Schaeffer computed drawing strategies for each player.

This was listed by Science Magazine as one of the top 10 breakthroughs of 2007.

Is chess also a draw?

Grand Unification

AlphaZero unifies chess and go algorithms.

This unification of intuition (go) and calculation (chess) is surprising.

This unification grew out of go algorithms.

But are the algorithmic insights of chess algorithms really irrelevant?

Chess Background

The first min-max computer chess program was described by Claude Shannon in 1950.

Alpha-beta pruning was invented by various people independently, including John McCarthy, about 1956-1960.

Alpha-beta has been the cornerstone of all chess algorithms until AlphaZero.

Alpha-Beta Pruning

```
def MaxValue(s,alpha,beta):
    value = alpha
    for s2 in s.children():
      value = max(value, MinValue(s2,value,beta))
      if value >= beta: break()
    return value

def MinValue(s,alpha,beta):
    value = beta
    for s2 in s.children():
      value = min(value, MaxValue(s2,alpha,value))
      if value <= alpha: break()
    return value
```

Conspiracy Numbers

Conspiracy Numbers for Min-Max search, McAllester, 1988

Each node s has a min-max value $V(s)$ determined by the leaf values.

For any positive integer N and potential value V we define $L(N, V)$ to be the set of leaf nodes s_1 such that there exist $N - 1$ other leaf nodes $s_2, \ldots, s_N$ such that by changing the values of $s_1, \ldots, s_N$ the root node can be changed to V.

Algorithm:

Repeatedly select some N and V such that $L(N, V)$ is non empty and expand some leaf in $L(N, V)$.

END