A Simple Alpha(Go) Zero Tutorial

29 December 2017

This tutorial walks through a synchronous single-thread single-GPU (read malnourished) game-agnostic implementation of the recent AlphaGo Zero paper by DeepMind. It's a beautiful piece of work that trains an agent for the game of Go through pure self-play without *any* human knowledge except the rules of the game. The methods are fairly simple compared to previous papers by DeepMind, and AlphaGo Zero ends up beating AlphaGo (trained using data from expert games and beat the best human Go players) convincingly. Recently, DeepMind published a preprint of Alpha Zero on arXiv that extends AlphaGo Zero methods to Chess and Shogi.

The aim of this post is to distil out the key ideas from the AlphaGo Zero paper and understand them concretely through code. It assumes basic familiarity with machine learning and reinforcement learning concepts, and should be accessible if you understand neural network basics and Monte Carlo Tree Search. Before starting out (or after finishing this tutorial), I would recommend reading the original paper. It's well-written, very readable and has beautiful illustrations! AlphaGo Zero is trained by self-play reinforcement learning. It combines a neural network and Monte Carlo Tree Search in an elegant policy iteration framework to achieve stable learning. But that's just words- let's dive into the details straightaway.

The Neural Network

Unsurprisingly, there's a neural network at the core of things. The neural network f_θ is parameterised by θ and takes as input the state s of the board. It has two outputs: a continuous value of the board state $v_\theta(s) \in [-1, 1]$ from the perspective of the current player, and a policy $\vec{p}_\theta(s)$ that is a probability vector over all possible actions.

When training the network, at the end of each game of self-play, the neural network is provided training examples of the form $(s_t, \vec{\pi}_t, z_t)$. $\vec{\pi}_t$ is an estimate of the policy from state s_t (we'll get to how $\vec{\pi}_t$ is arrived at in the next section), and $z_t \in \{-1, 1\}$ is the final outcome of the game from the perspective of the player at s_t (+1 if the player wins, -1 if the player loses). The neural network is then trained to minimise the following loss function (excluding regularisation terms):

$$l = \sum_t (v_\theta(s_t) - z_t)^2 - \vec{\pi}_t \cdot \log(\vec{p}_\theta(s_t))$$

The underlying idea is that over time, the network will learn what states eventually lead to wins (or losses). In addition, learning the policy would give a good estimate of what the best action is from a given state. The neural network architecture in general would depend on the game. Most board games such as Go can use a multi-layer CNN architecture. In the paper by DeepMind, they use 20 residual blocks, each with 2 convolutional layers. I was able to get a 4-layer CNN network followed by a few feedforward layers to work for 6x6 Othello.

Monte Carlo Tree Search for Policy Improvement

Given a state s, the neural network provides an estimate of the policy $\vec{p}_\theta$. During the training phase, we wish to improve these estimates. This is accomplished using a Monte Carlo Tree Search (MCTS). In the search tree, each node represents a board configuration. A directed edge exists between two nodes $i \to j$ if a valid action can cause state transition from state i to j. Starting with an empty search tree, we expand the search tree one node (state) at a time. When a new node is encountered, instead of performing a rollout, the value of the new node is obtained from the neural network itself. This value is propagated up the search path. Let's sketch this out in more detail.

For the tree search, we maintain the following:

- $Q(s,a)$: the expected reward for taking action a from state s, i.e. the Q values
- $N(s,a)$: the number of times we took action a from state s across simulations
- $P(s,\cdot) = \vec{p}_\theta(s)$: the initial estimate of taking an action from the state s according to the policy returned by the current neural network.

From these, we can calculate $U(s,a)$, the upper confidence bound on the Q-values as

$$U(s,a) = Q(s,a) + c_{puct} \cdot P(s,a) \cdot \frac{\sqrt{\Sigma_b N(s,b)}}{1+N(s,a)}$$

Here c_{puct} is a hyperparameter that controls the degree of exploration. To use MCTS to improve the initial policy returned by the current neural network, we initialise our empty search tree with s as the root. A single simulation proceeds as follows. We compute the action a that maximises the upper confidence bound $U(s,a)$. If the next state s' (obtained by playing action a on state s) exists in our tree, we recursively call the search on s'. If it does not exist, we add the new state to our tree and initialise $P(s',\cdot) = \vec{p}_\theta(s')$ and the value $v(s') = v_\theta(s')$ from the neural network, and initialise $Q(s',a)$ and $N(s',a)$ to 0 for all a. Instead of performing a rollout, we then propagate $v(s')$ up along the path seen in the current simulation and update all $Q(s,a)$ values. On the other hand, if we encounter a terminal state, we propagate the actual reward (+1 if player wins, else -1).

After a few simulations, the $N(s,a)$ values at the root provide a better approximation for the policy. The improved stochastic policy $\vec{\pi}(s)$ is simply the normalised counts $N(s,\cdot)/\sum_b(N(s,b))$. During self-play, we perform MCTS and pick a move by sampling a move from the improved policy $\vec{\pi}(s)$. Below is a high-level implementation of one simulation of the search algorithm.

```
def search(s, game, nnet):
    if game.gameEnded(s): return -game.gameReward(s)

    if s not in visited:
        visited.add(s)
        P[s], v = nnet.predict(s)
        return -v

    max_u, best_a = -float("inf"), -1
    for a in range(game.getValidActions(s)):
        u = Q[s][a] + c_puct*P[s][a]*sqrt(sum(N[s]))/(1+N[s][a])
        if u>max_u:
            max_u = u
            best_a = a
    a = best_a

    sp = game.nextState(s, a)
    v = search(sp, game, nnet)

    Q[s][a] = (N[s][a]*Q[s][a] + v)/(N[s][a]+1)
    N[s][a] += 1
    return -v
```

mcts.py hosted with ♡ by GitHub view raw

Note that we return the negative value of the state. This is because alternate levels in the search tree are from the perspective of different players. Since $v \in [-1,1]$, $-v$ is the value of the current board from the perspective of the other player.

Policy Iteration through Self-Play

Believe it or not, we now have all elements required to train our unsupervised game playing agent! Learning through self-play is essentially a policy iteration algorithm- we play games and compute Q-values using our current policy (the neural network in this case), and then update our policy using the computed statistics.

Here is the complete training algorithm. We initialise our neural network with random weights, thus starting with a random policy and value network. In each iteration of our algorithm, we play a number of games of self-play. In each turn of a game, we perform a fixed number of MCTS simulations starting from the current state s_t. We pick a move by sampling from the improved policy $\vec{\pi}_t$. This gives us a training example $(s_t, \vec{\pi}_t, \_)$. The reward _ is filled in at the end of the game: +1 if the current player eventually wins the game, else -1. The search tree is preserved during a game.

At the end of the iteration, the neural network is trained with the obtained training examples. The old and the new networks are pit against each other. If the new network wins more than a set threshold fraction of games (55% in the DeepMind paper), the network is updated to the new network. Otherwise, we conduct another iteration to augment the training examples.

And that's it! Somewhat magically, the network improves almost every iteration and learns to play the game better. The high-level code for the complete training algorithm is provided below.

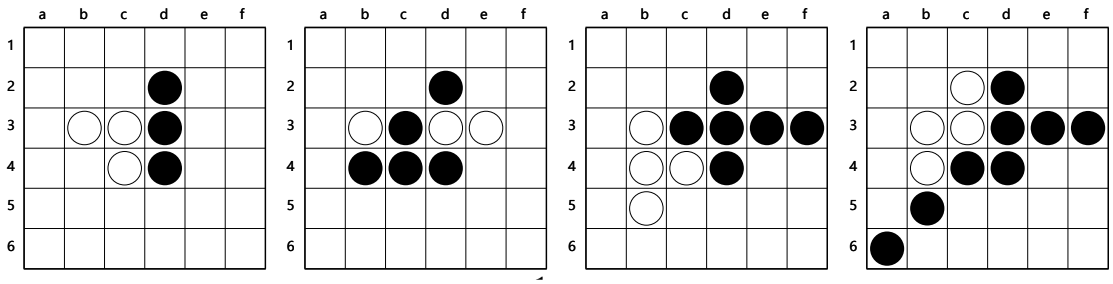
```
def policyIterSP(game):
    nnet = initNNet()                                   # initialise random neural network
    examples = []
    for i in range(numIters):
        for e in range(numEps):
            examples += executeEpisode(game, nnet)      # collect examples from this game
        new_nnet = trainNNet(examples)
        frac_win = pit(new_nnet, nnet)                  # compare new net with previous net
        if frac_win > threshold:
            nnet = new_nnet                             # replace with new net
    return nnet

def executeEpisode(game, nnet):
    examples = []
    s = game.startState()
    mcts = MCTS()                                       # initialise search tree

    while True:
        for _ in range(numMCTSSims):
            mcts.search(s, game, nnet)
        examples.append([s, mcts.pi(s), None])          # rewards can not be determined yet
        a = random.choice(len(mcts.pi(s)), p=mcts.pi(s))    # sample action from improved policy
        s = game.nextState(s,a)
        if game.gameEnded(s):
            examples = assignRewards(examples, game.gameReward(s))
            return examples
```

politer-selfplay.py hosted with ♡ by GitHub view raw

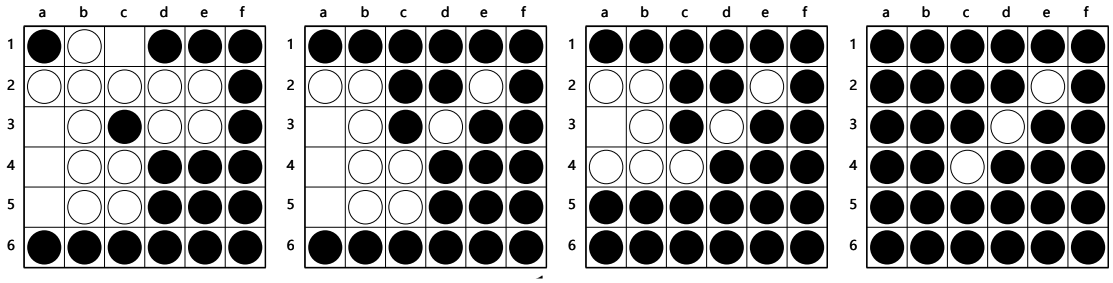
Experiments with Othello

We trained an agent for the game of Othello for a 6x6 board on a single GPU. Each iteration consisted of 100 episodes of self-play and each MCTS used 25 simulations. Note that this is orders of magnitude smaller than the computation used in the AlphaGo paper (25000 episodes per iteration, 1600 simulations per turn). The model took around 3 days (80 iterations) for training to saturate on an NVIDIA Tesla K80 GPU. We evaluated the model

against random and greedy baselines, as well as a minimax agent and humans. It performed pretty well and even picked up some common strategies used by humans.



The agent (Black) learns to capture walls and corners in the early game



The agent (Black) learns to force passes in the late game

A Note on Details

This post provides an overview of the key ideas in the AlphaGo Zero paper and excludes finer details for the sake of clarity. The AlphaGo paper describes some additional details in their implementation. Some of them are:

- **History of State**: Since Go is not completely observable from the current state of the board, the neural network also takes as input the boards from the last 7 time steps. This is a feature of the game itself, and other games such as Chess and Othello would only require the current board as input.
- **Temperature**: The stochastic policy obtained after performing the MCTS uses exponentiated counts, i.e. $\vec{\pi}(s) = N(s,\cdot)^{1/\tau} / \sum_b (N(s,b)^{1/\tau})$, where τ is the temperature and controls the degree of exploration. AlphaGo Zero uses $\tau = 1$ (simply the normalised counts) for the first 30 moves of each game, and then sets it to an infinitesimal value (picking the move with the maximum counts).
- **Symmetry**: The Go board is invariant to rotation and reflection. When MCTS reaches a leaf node, the current neural network is called with a reflected or rotated version of the board to exploit this symmetry. In general, this can be extended to other games using symmetries that hold for the game.
- **Asynchronous MCTS**: AlphaGo Zero uses an asynchronous variant of MCTS that performs the simulations in parallel. The neural network queries are batched and each search thread is locked until evaluation completes. In addition, the 3 main processes: self-play, neural network training and comparison between old and new networks are all done in parallel.
- **Compute Power**: Each neural network was trained using 64 GPUs and 19 CPUs. The compute power used for executing the self-play in unclear from the paper.
- **Neural Network Design**: The authors tried a variety of architectures including networks with and without residual networks, and with and without parameter sharing for the value and policy networks. Their best architecture used residual networks and shared the parameters for the value and policy networks.

This code presented in this tutorial provides a high-level overview of the algorithms involved. A complete game and framework independent implementation can be found in this GitHub repo. It contains an example implementation for the game of Othello in PyTorch, Keras and TensorFlow.

Feel free to leave questions/comments/suggestions below :).