Applying Deep Double Q-Learning and Monte Carlo Tree Search to Playing Go

Booher, Jonathan jaustinb@stanford.edu
De Alba, Enrique edealba@stanford.edu
Kannan, Nithin nkannan@stanford.edu

I. Introduction

FOR our project we replicate many of the methods used in AlphaGo Zero to make an optimal Go player; however, we modified the learning paradigm to a version of Deep Q-Learning which we believe would result in better generalization of the network to novel positions.

The modification of Deep Q-Learning that we use is called Deep Double Q-Learning and will be described later. The evaluation metric for the success of our agent is the percentage of games that are won against our Oracle, a Go-playing bot available in the OpenAI Gym. Since we are implementing a version of reinforcement learning, there is no data that we will need other than the simulator. By training on the games that are generated from self-play, our player will output a policy that is learned at the end of training by Deep Double Q-Learning.

II. Related Work and our Modifications

We build off of the work from Google research and DeepMind outlined in two papers.

We use the Deep Double Q-Learning algorithm described in the first paper. In this algorithm, we have two sets of network parameters, using one set for choosing the optimal action and the other for position $valuation$. We present the exact changes with respect to traditional Q-Learning in the methods section.

The other paper that we drew on for this project is the AlphaGo Zero paper. We used the method of self play generation through MCTS that was described therein. However, we significantly altered the learning paradigm used to train the network from the one presented in the paper. In the paper they trained a classifier to predict game winners based on the current position. By sampling states from the self play games along with their respective rewards, the researchers were able to train a binary classifier to predict the outcome of a game with a certain confidence. Then based on the confidence measures, the optimal move was taken.

We depart from this method of training and use Deep Double Q-Learning instead. We use the same concept of sampling states and their rewards from the games of self play, but instead of feeding this information to a binary classifier, we feed the information to a modified Q-Learning formula, which we present shortly.

III. Challenges

The main challenge we faced was the computational complexity of the game of Go. To address these issues, we trained the agent on a 9×9 Go board instead of the full 19×19 board. This will limit the complexity of our MCTS searches as the branching factor and depth will be greatly reduced. Because we have limited computational resources, this was major bottleneck to how high of an ELO rating our agent was able to achieve.

Double Q-Learning also involves swapping the parameters that are trained periodically. Two major hyperparameters that needs to be addressed are the frequency of that swapping and the conditions under which swapping is allowed. This touches on another broader challenge in hyperparameter tuning and network initialization. We currently are using samples from a normal distribution to initialize the parameters of the network and are using the hyperparameters defined in the AlphaGo Zero paper. A lane of future work would be to see if the optimal hyperparameters on 19×19 Go are close to optimal

for the reduced 9×9 Go board. Since training our Go engine is quite computationally expensive, tuning these parameters given limited computing resources would take time on the order of several days.

IV. Our Work

All of the code has been written in python using TensorFlow for leveraging GPU acceleration. We are using a modified version of the Go environment in OpenAI Gym that allows us to easily and efficiently extract legal moves and store positions. The modifications made to the Go environment are as follow:

- Disable the OpenAI Go engine by default
- Provide our program with access to more of the internal data structures.

Note that these modifications are allowable under the license.

All of the algorithms have been written and tested independently. We began by testing the Deep Double-Q on multiple Open AI Gym environments including the Inverted Pendulum, the MCTS algorithm on Go, and the network on a basic set of inputs, drawn randomly from a normal distribution in order to ensure that dimensions matched correctly.

V. Deep Double Q-Learning and our Modifications

To begin to look at our algorithm for Double-Q learning, we will first introduce terminology for Q learning as well as the original Q-Learning algorithm. In standard Q-Learning, we write the value of action a at state s under a given policy π as

$$Q_\pi(s,a) = \mathbb{E}\big[r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots\big]$$

with discount factor $\gamma \in [0,1]$.

In traditional Q-Learning, there is a weight vector w that is updated using gradient descent according to the following update rule:

$$w \leftarrow w - \eta\Big[w \cdot \phi(s,a) - Y\Big]\nabla_w w \cdot \phi(s,a)$$

$$Y \equiv r + \gamma\Big(\arg \max_{a' \in Actions(s')} w \cdot \phi(s',a')\Big)$$

However, in a Deep Q-Learning setting, we replace $w \cdot \phi$ with the output of a network.

The original approach has been found by researchers at Google to provide overly optimistic value predictions and be brittle to novel experiences. These problems led to the creation of Double Q-Learning.

In addition, instead of having one set of parameters (call them Θ), we have two (Θ and Θ'), with one used for approximating the value and the other for making the predicted move. After every learning iteration (`iter`), the parameters are swapped so that they can both be trained. We only run gradient descent on the parameters, choosing the action at any given time step under the assumption that the correct parameters for picking moves will provide values that are consistent with that of the optimal move. We then have the following modification to the Q-Learning Algorithm:

$$\Theta \leftarrow \Theta + \eta(Y_t^Q - \Theta \cdot \phi(s,a))\nabla_\Theta \Theta \cdot \phi(s,a)$$

$$Y_t^Q = r + \gamma \Theta' \cdot \phi(s', (\arg\max_a \Theta \cdot \phi(s,a)))$$

For sake of simplicity we use $\Theta \cdot \phi(s,a)$ to represent the output of the network using parameters Θ.

See Algorithm 1 below for pseudo code.

Algorithm 1: Deep Double Q-Learning Algorithm

```
batchCount ← 0
while batchCount ≤ maxBatches do
    if learnStepCounter % iters then
        runParamSwap()
    end
    batch ← randomBatch( batchSize )
    qNext ← runQNext( batch[s'] )
    qNextEval ← runEval( batch[s'] )
    qEval ← runEval( batch[s] )
    maxActions ← argmax( qEvalNext )
    selectedQs ← qNext[ maxActions ]
    qTarget ← reward + gamma × selectedQs
    runGradientAscent( batch[s], qTarget )
    learnStepCounter += 1
    batchCount += 1
end
```

VI. Monte Carlo Tree Search

We implement Monte Carlo Tree Search in accordance with the implementation presented in AlphaGo Zero. We perform

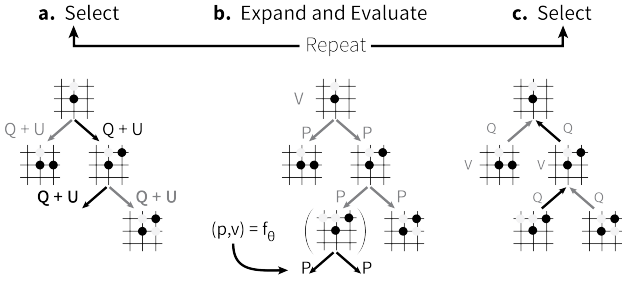

Fig. 1. MCTS Overview

1600 simulations to choose each move for the games of self play. There are three phases to this version of the tree search (for a graphical representation, see Fig 1)

Phase 1: Selection: At time step 0, *i.e.* the root node, we perform selection differently:

We add noise to the prior probabilities to encourage exploration at the root, giving us the following expression for exploration:

$$(1-\epsilon)p(s,a) + \epsilon\eta_a, \text{ where } \eta \sim Dir(0.03)$$

where we have $\epsilon = 0.25$ as it is in the AlphaGo paper. We use this value in place of $p(s,a)$ in the following formulas.

We do selection up to a time step L based off of the version of PUTC presented in the AlphaGo Zero paper. This will choose a random move in accordance with the probability distribution retrieved from the network if there are edges from that state that have not already been explored (*i.e.* an action that hasn't been explored). If all edges have been explored, we choose based on the following formula (PUTC) where c is a constant determining the amount of exploration:

$$action = \arg\max_a \big(VisitCount(s,a) + U(s,a)\big)$$
$$U(s,a) = c \cdot p(s,a)\frac{\sqrt{\sum_{b \in actions} VisitCount(s,b)}}{1 + VisitCount(s,a)}$$

Phase 2: Expansion and Evaluation: The key difference between MCTS in AlphaGo Zero and previous iterations of AlphaGo is the fact that all leaf nodes are expanded and we do not roll-out all the way to the end of the game. Instead, we feed the position through the Double-Q network and get the value and probability of each action. Initialize all edges from this leaf node with the respective probability from the network and then back the value up the tree (we implement this with recursive backtracking).

Phase 3: Backup: We update the statistics of the edges that led to the leaf node according to the value returned by the recursive call as follows:

$$VisitCount(s,a) \leftarrow VisitCount(s,a) + 1$$
$$TotalValue(s,a) \leftarrow TotalValue(s,a) + v$$
$$MeanValue(s,a) \leftarrow \frac{TotalValue(s,a)}{VisitCount(s,a)}$$

Move Selection For Play: We then chose the move that is selected for play according to the dist:

$$\frac{VisitCount(s_0,a)^{1/\tau}}{\sum_{b \in actions} VisitCount(s_0,b)^{1/\tau}}$$

Where $\tau = 1$ when the number of moves is less than a threshold and infinitesimally small when greater than or equal to that threshold.

Note that we keep the tree throughout the game, we just discard the parts of the tree that we no longer need (*i.e.* the parts that we can no longer reach).

We reproduce the algorithm in the following page in pseudo-code for completeness See Algorithm 2.

VII. Network Design

The network takes in a tensor with the following shape:

$$[batchSize, 17, boardSize, boardSize]$$

We have 17 for the number of previous board states the network needs to know for how the position arose. Thus, we pass in 17 planes, 8 of which carry the information on the locations of the black stones at time steps $t, \ldots, t-7$ and another 8 that carry the information on the locations of the white stones at the same time steps. The last plane is filled with either 1 or 0 based on whose turn it is to move. All this information is needed by the network because the rules of the game are dependent upon the previous states (*i.e.* no repetition); in addition, it is impossible to discern who's move it is based on the board position alone.

The network itself has three main parts:

Algorithm 2: Run A Single Monte Carlo Simulation

```
Recurse Function;
  Input : state, depth
if depth = 0 then
    val ← value( state )
    for all a in legalActions( state ) do
        p ← probability( state, action )
        tree[( state, a )] ← { 0, 0, 0, p }
    end
else
    if tree[( state, a )] for a in allActions( state ) then
        action ← arg max_a ( PUTC(state, a) )
        stateCp ← copy( state )
        stateCp.step( action )
        value ← recurse( stateCp, depth−1)
        updateStatistics( state )
        return state[MeanValue]
    else
        action ← getNetworkAction( state )
        prob ← getNetorkValue( state, action )
        stateCp ← copy( state )
        tree[( state, action )] ← { 0, 0, 0, prob }
        value ← recurse( stateCp, depth−1 )
        updateStatistics( state )
        return state[MeanValue]
    end
end
Single Monte Carlo Simulation;
  Input : state, depth
batchCount ← 0
while batchCount ≤ maxBatches do
    noise ← Dirichlet(0.03)
    for a ∈ allActions(state) do
        prob ← getNetworkProb( state, action )
        p ← (1 − ϵ) × prob + noise[a] × ϵ
        tree[( state, action )] ← { 0, 0, 0, p }
    end
    action ← arg max_a( tree[( state, a )] )
    stateCp ← copy( state )
    value ← recurse( state, depth−1)
    updateStatistics( state )
end
```

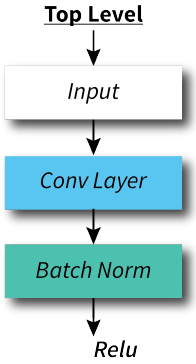

Fig. 2. The Top Layer Of The Network

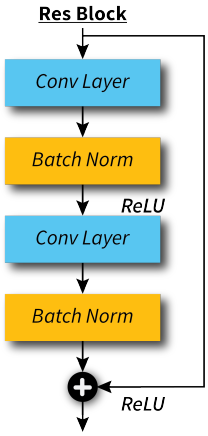

Fig. 3. A Residual Block

Part 1: The first part consists of (See Figure 2 for visual):

- Conv layer across all 17 input planes with kernel size $3 \times 3 \times 17$
- Batch-norm layer
- RELU activation

Part 2: This is a residual tower with variable height where each residual block contains (See Figure 3 for visual):

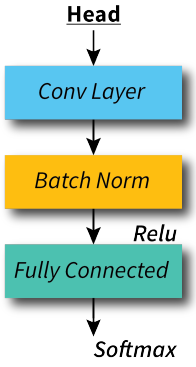

Fig. 4. The head of the network

- Conv layer (kernel size $3 \times 3 \times 256$)
- Batch-norm layer
- RELU Activation
- Conv Layer (kernel size $3 \times 3 \times 256$)
- Batch-norm layer
- RELU Activation
- Skip Connection

Part 3: This consists of (See Figure 4 for visual):

- Conv layer (kernel size $1 \times 1 \times 2$)
- Batch-norm layer
- RELU Activation
- A single fully connected layer to a $boardSize^2$ vector (indexed 0 to $boardSize^2$ where the $boardSize^2$ the action represents a pass) that represents the probabilities output by the network. Note that we have disabled the agent from resigning to ensure that games get played to their conclusion.

This network architecture is based off of the description of the network in the AlphaGo Zero paper. They used a network with a residual tower of height 20, but for us in testing on a 9×9 Go board, we will shorten the tower size to about 10 or less depending on the computational constraints.

VIII. Playing A Game And Learning

The algorithm learns through the the Q-Learning algorithm which requires a distribution of $(state, action, reward, nextState)$ quadruplets to learn. In order to get this data, we use self play as per the AlphaGo Zero paper. For each game of self play, we run 100 MCTS simulations with depth 5. The algorithm then selects the move to play in that position according to the exponentiated visit count of the moves at the root.

$$\frac{VisitCount(s_0, a)^{1/\tau}}{\sum_{b \in actions(s_0)} VisitCount(s_0, b)^{1/\tau}}$$

The parameter τ is set to 1 for the first 20 moves of the game and set close to 0 afterwards. This encourages exploration at the beginning of the game and low at the end. This makes the search tree more useful for exploration. The same search tree is used at subsequent time steps and is simply expanded upon.

At the end of running 100 games through the MCTS procedure, we train using random samples of $batchSize$ from the $(state, action, reward, nextState)$ quadruplets created by the last 200 games. This will allow us to refresh the data with new data produced by the stronger network.

IX. Experimental Results

We trained the network for 3 days where the network played over 1000 games of self play.

After every 10 games of self play, we perform one training epoch which consists of 20 minibatches of size 32 randomly sampled from the (s, a, r, s') transitions from the previous 2 training epochs.

After every 10 training epochs, we run 10 games of regular play against the oracle with 2400 ELO so that we can observe how well our agent is playing. We do not use any of this data for training purposes. Rather, we can use the win rate w to calculate the ELO difference that exists between our model and the engine that we are testing against according to the following formula:

$$\text{ELO difference} \equiv \log\left(\frac{1-w}{w}\right) \cdot 85$$

where 85 is a constant that is derived from the ELO of the engine we test against.

After this training, the model appeared to be converging albeit slowly (See Figure 5 where the $x-$axis is epochs and the $y-$axis is the value of the loss). Further you can see the progress that our agent was making in terms of ELO difference (see Figure 6). In that Figure, the $x-$axis is the epoch and the $y-$axis is the ELO difference. The points represent the ELO difference at a specific epoch. We fit a trend line to the data for clarity of the trend of gradual improvement. It is clear that the ELO difference does not monotonically decrease; however, as time progressed, the agent got progressively better.

The best performance that was reached by the agent was getting a score of $3/10$. Most of the points that our agent received were from *draws* (ties) in which the agent was assigned a score of $+0.5$: however, our agent did win several games during the course of training and the live demo presented at the poster session.

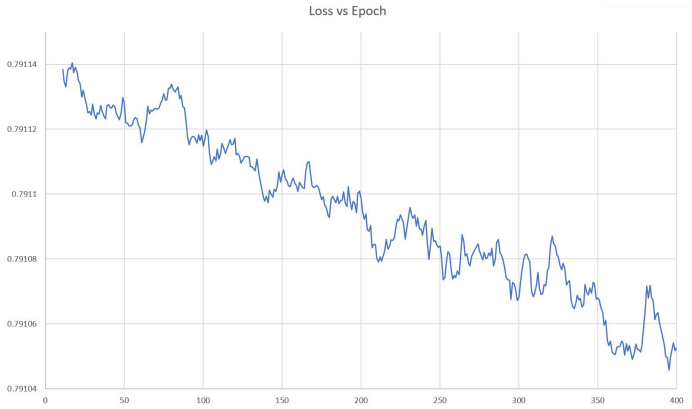

Fig. 5. The Loss Function over Time

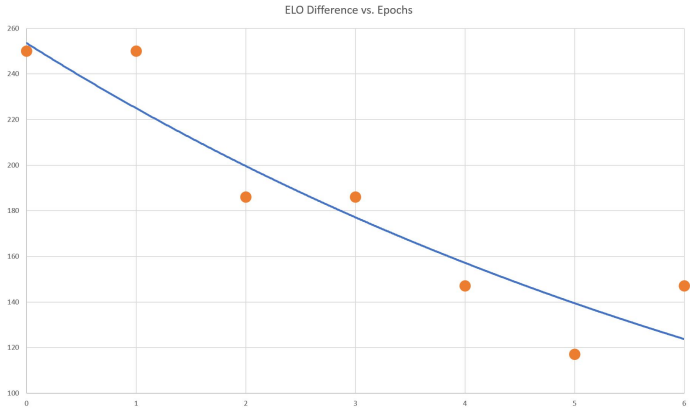

Fig. 6. ELO Difference Between our Agent and the OpenAI Go Engine

It is interesting to note that the games that our agent won were relatively short games where our agent gained an immediate advantage and both agents realized this and passed (two passes in a row is an agreement to end the game). Draws tended to be longer games.

X. Error Analysis

The weak performance of our agent as compared to AlphaGo Zero is attributable to two factors:

- Training time
- Quality of MCTS searches

Google was able to train continuously for 40 days while we were only able to train for 3. This is a big difference as in reinforcement learning, training time directly correlates to an increase in experience and better policies.

Secondly, Google was able to run 1600 MCTS simulations per move. We were able to run at maximum 160 simulations to a decent depth of 5. Any more and the computation time per move would have been too high for the amount of training time that we had available. The number of simulations is correlated to the *quality* of the data that the Double Q-Learning algorithm learns from. This is important as the agent can only get better when learning from good data. Currently there are not enough collisions in the MCTS search that we run for the expected value of the visit counts to be consistently close to the actual value.

It is simple to see that at the beginning of the game, with our implementation, each move at the root will be visited approximately 2 times while Google's implementation would see each node approximately 20 times (assuming the Dirichlet parameter is large). This effect of a decreased number of visits only gets exacerbated the further down the search tree you go, since the game tree grows exponentially. This is important as MCTS searches to get quality data is dependent upon exploration and the fact that better positions will be visited frequently. However, when positions are visited extremely rarely like they are in our implementation, the algorithm cannot make very informed choices on what nodes to visit thus impacting the quality of the data and the strength of the agent we can create.

XI. Further Work

Currently we are bound by the speed at which we can create the self play games as MCTS currently takes on the order of 0.6 seconds per move and running the training operation requires only a fraction of that amount of time (0.001 seconds), since we are leveraging GPU acceleration through TensorFlow across two Titan Xp GPUS.

Multi threading the MCTS would allow use to speed up the algorithm, which would in turn allow us to run more simulations to a lower depth and increase the quality of the data that we train the network on. This would require us to write the code using distributed TensorFlow as we would be making asynchronous calls to the network evaluation functions.

One more potential improvement that could be made was implemented in AlphaGo Zero and consisted of saving the current best player's parameters (call that player a_0) and using

that network to create the data to train from. That player would play many games of self play and then train to create player a_1. After several training epochs, a_1 would play against a_0 and would replace a_0 if it wins over 55% of the time. In this way we can ensure that the strength of our engine monotonically increases. This is different from what happened during our training as the strength of our engine alternated between strengthening and weakening periods.

REFERENCES

[1] Silver, David, et al. "Mastering the game of go without human knowledge." Nature 550.7676 (2017): 354-359.

[2] Van Hasselt, Hado, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-Learning." AAAI. 2016.