Mastering the game of Go from scratch

Michael Painter [*1] Luke Johnston [*1]

Abstract

In this report we pursue a transfer-learning inspired approach to learning to play the game of Go through pure self-play reinforcement learning. We train a policy network on a 5×5 Go board, and evaluate a mechanism for transferring this knowledge to a larger board size. Although our model did learn a few interesting strategies on the 5×5 board, it never achieved human level, and the transfer learning to a larger board size yielded neither faster convergence nor better play.

1. Introduction

In the recent paper "Mastering the game of Go with deep neural networks and tree search" [1], superhuman performance on the game of Go was achieved with a combination of supervised learning (from professional Go games) and reinforcement learning (from self-play). Traditional pure reinforcement learning approaches have not yielded satisfactory results on the game of Go on a 19×19 board because its state space is so large and its optimal value function so complex that learning from self-play is infeasible. However, neither of these limitations apply to smaller board sizes (for example, 5×5).

In this paper, we attempt to investigate this problem. Specifically, we evaluate an entirely reinforcement-learning based approach to 'mastering the game of Go', that first learns how to play on smaller board sizes, and then uses a form of transfer learning to learn to play on successively larger board sizes (without ever consulting data from expert play).

The inspiration for the transfer learning component comes from the observation that humans almost always learn large tasks by breaking them up into simpler components first. A human learning the game of Go would be taught techniques and strategies at a much smaller scale than the full 19×19 board, and only after mastering these concepts would they have any chance of mastering the larger board. So our ultimate goal is to set up a general framework for this sort of transfer learning, so that complex tasks such as Go can be learned without reference from expert data.

In a more general setting, it's an interesting question to consider how can we, in the absence of human professional datasets, attempt to apply machine learning techniques (specifically reinforcement learning techniques) to problems with large state spaces. We pursue this unsupervised learning approach due to one main motivation: for tasks which lack expert data, or for tasks which no expert exists, supervised training is impossible. Hence, this area is vital to the development of general artificial intelligence (GAI), since no GAI will be able to rely on expert data for all of its tasks.

Full of hope and optimism, we had hoped to coin the term "BetaGo" for our agent (excuse the pun). However, as will become clear in the remainder of the report, that we settled for a more apt "ZetaGo", leaving Beta, Gamma, and well, the rest of the Greek alphabet available for more capable AI.

2. Related Work

2.1. TD-gammon

TD-gammon [6] was the first reinforcement learning agent to achieve human master level play on a board game (backgammon) using only self-play. A neural network is used to estimate the value function of the game state, and the TD update is applied after every step:

$$\Delta w = \alpha[V(s_{t+1}) - V(s_t)] \sum_{k=1}^{t} \gamma^{t-k} \nabla V(s_k)$$

where α is the learning rate, w the weights of the network, V the value function, and γ the future reward discount. Starting from no knowledge, and only playing against itself, this network is able to learn a value function that is competitive with human masters with the simple TD learning update above. However, the game of backgammon has many advantages for this approach that other games, like Go, lack. First, backgammon state transitions are indeterministic, so a form of random exploration is inherently built into the policy. Second, simple evaluations of board state are relatively straightforward, as the board is one-

[*]Equal contribution [1]Stanford University, Palo Alto, USA.

dimensional, and the inherent objective is to simple move stones along the board in one direction (although the strategy is complex). It is also worthwhile noting that the TD-gammon agent uses human-constructed features for the input in addition to the basic board state. Finally, a 19×19 Go board has an much, much larger state space than backgammon - $\approx 10^{170}$ versus $\approx 10^{20}$.

2.2. Policy Networks

A policy network approximates a stochastic policy with a neural network. The policy gradient theorem [7] states that for a policy function approximator π with parameters θ, we can compute the derivative of the expected total discounted rewards p with respect to the parameters θ as follows:

$$\frac{\partial p}{\partial \theta} = \sum_s d^\pi(s) \sum_a \frac{\partial \pi(s,a)}{\partial \theta} Q^\pi(s,a)$$

where the s are all states, the a are all actions, d^π is the stationary distribution of states according to the policy, and Q is an estimate of the expected discounted reward from taking action a at state s. In this paper we use the actual returns, $Q(s_t, a_t) = R_t = \sum_{k=1}^{\infty} \gamma^{k-1} r_{t+k}$, although Q can also be approximated with another function approximator. In either case, the policy will converge to a locally optimal policy [7]. Policy networks have been used to great success for recent reinforcement learning tasks, such as AlphaGo below [1], and the A3C algorithm on diverse tasks such as navigating 3D environments and playing Atari games [8].

2.3. AlphaGo

AlphaGo [1] is an AI capable of defeating human masters at the game of Go. This feat was considered a milestone of AI and was accomplished recently, in the spring of 2016. To achieve this, they first train a 13-layer convolutional policy network to predict expert play. Then, they initialize a second policy network with these results, and train it further with reinforcement learning from self play. During self play, network parameters are saved every 500 iterations, and the opponent is selected randomly from one of those saves (to reduce overfitting to the current policy). Thirdly, they train a value network to estimate the value function for each state. It has the same architecture as the convolutional policy network, except the final layer, which outputs a single value instead of policy logits. This network was trained 30 million game positions, each a random position of a distinct self-play game. Positions are taken from unique games to reduce correlation between the training data and prevent overfitting. Finally, a form of Monte-Carlo tree search parameterized by the policy and value networks is utilized for action selection during both training and testing.

3. Approach

3.1. OpenAI Gym

The OpenAI Gym toolkit for reinforcement learning [2] provides an environment for learning the game of Go on both 9×9 and 19×19 sized boards, playing against an opponent controlled by the Pachi open-source Go program [3]. For our project, we added our own environments for each board width from 5 to 19, and modified the environment to allow self-play (we made it a two player environment instead of a single player environment where the opponent is always Pachi). In this environment, for a board width of W, the current player has $W^2 + 2$ actions: play at any location in the board, resign, or pass. If the agent attempts to make an impossible move, this is interpreted as resignation. Since this results in extremely frequent resignations in the early stages of training, we masked out the probabilities of obvious resignations, disallowing the model to attempt to play on top of an existing piece. The game ends when either player resigns, or when both players pass in succession (at which point the board is evaluated and the winner is determined). Rewards are 0 until the terminal state, at which point a reward of 1 indicates a win, -1 a loss, and 0.0 a draw. The state of the board is represented as a $(W \times W \times 3)$ array, where each entry is either 0 or 1, and one channel represents black moves, one white, and one empty spaces.

The OpenAI Gym toolkit also provides a Pachi agent [3] implementation, which uses a standard UCB1 search policy with monte-carlo tree search (described in section 3.6).

3.2. Policy network reinforcement learning

We train a policy network to estimate the probability of each action given a state at time t:

$$p_\theta(s_t) \in \mathbb{R}^{|A|}$$

where p_θ is the policy function with parameters θ, s_t is a state of the game, and $|A|$ is the number of actions. These values are normalized to probabilities with a softmax layer. To train the policy network we follow a similar approach as [1]. For a minibatch of size n, the policy network plays n games against a randomly selected previous iteration of the policy network. According to [1], this randomization prevents overfitting to the current policy. Let the outcome of the ith game be r_i at terminal turn T^i. Then the policy update we implement is

$$\Delta\theta = \frac{\alpha}{n} \sum_{i=1}^{n} \sum_{t=1}^{T^i} \frac{\partial \log p_\theta(a_t^i | s_t^i)}{\partial \theta} r_i$$

Note that this represents a modification from [1], which includes a baseline estimate of the value function for variance reduction.

$$\Delta\theta = \frac{\alpha}{n}\sum_{i=1}^{n}\sum_{t=1}^{T^i}\frac{\partial \log p_\theta(a_t^i|s_t^i)}{\partial\theta}(r_i - v_\rho(s_t^i))$$

where v_ρ is the value network learned in [1]. We chose to omit this optimization for computational speed, and as we discuss in section 4.1.3 appears to be crucial to a stable and fast convergence [7]. Every 500 steps, the current policy network is saved in the list of previous policy networks, for random sampling of the opponent policy.

3.3. Small model (5 x 5 board)

3.3.1. Board size motivation

In order for transfer learning to work effectively, we first need to train an effective agent on a small board size, however, we need for it to be able to learn about structures that appear on full size Go boards. We therefore choose to start with a 5×5 boards, because for any smaller board sizes, structures such as *eyes* cannot be formed.

Furthermore, for evaluation against the Pachi agent, due to the smaller board sizes, if Pachi is moving second, Pachi almost always resigns immediately.

On the 5×5 board, the Pachi agent will still resign against a central first move, but will play against any other move, but we still found this board size to be a challenge for training (it was difficult for the agent to learn that the middle is the optimal move).

3.3.2. Policy Network Architecture

The architecture for the 5×5 network was heavily inspired by [1]. The board input described in section 2.4 passed through C_S convolutional layers, each with k_S filters. The first layer has filter size set to 5, the rest are set to 3. Outputs of each layer are passed through a ReLU nonlinearity. The output of the final layer is the feature extractor of the small network, that will be used for the global scaling transfer learning below. In order to obtain the policy from these features, they are passed through one final convolutional layer with a single filter of size 1. The resulting values are passed through a two-dimensional softmax, to obtain probability estimates for playing in each location on the board. As mentioned in section 3.1, these probabilities are zeros for all actions that would place a stone on top of another existing stone. However, the model can still perform invalid moves if it attempts to *suicide* a *single* stone. This is considered forfeiting.

3.4. Board size transfer learning

Here we consider two ways that we could use the smaller network *locally* and *globally* when learning to play on a larger board. These are intended to mimic the way that humans would use knowledge gained from playing on a smaller board when moving up to a larger board size.

3.4.1. Local Convolution

Once we have an effective neural network trained to output the policy for a small (i.e. 5×5) board, we can "scale up" the network to create a larger network to estimate the values of a larger (i.e. 9×9) board as follows. The small network is applied to each window of the larger board, as a convolutional filter would be applied, with no padding and a stride of 1. For each window of the large board, the small board network generates a policy for play in that window. Each one of these policies is zero-padded back to the size of the larger board, as can be seen in figure 1.

In general terms, if transfer learning is applied from a board of size W_1 to a board of size W_2, this results in $F = (W_2 - W_1 + 1)^2$ convolution windows. The policies from each window are stacked, giving the first term of board features R_1 of shape (W_2, W_2, F). The weights of this part of the network are locked during training.

3.4.2. Global Downscaling

In addition to convolving the smaller network on the larger board, we learn a general down-scaling operation that "resizes" the larger board to the smaller size, so that it can be passed through the smaller network to extract features. The larger board state is passed through a new convolutional network, to represent it in the state space of the smaller board size. This convolutional network This will be done with N different convolutional reductions, each with k filters and filter size of 3. Each reduction is passed through the smaller network to extract $N \times k_s$ features, to give R_2 of shape (W_2, W_2, Nk_s), the second set of board features to the final output fully-connected layer of the network. The convolutional scaling operation weights are trained, but the smaller network weights are again locked.

3.4.3. Larger board features

The above two feature-extraction methods primarily use the information learned from the smaller model. However, we also need to be able to learn features specific to the larger board. So we will additionally train a convolutional feature extractor that takes as input the larger board representation (of shape $(W_2, W_2, 3)$). We use C_L convolutional layers, each with k_L filters, in the same structure as section 3.3.2. This results in a third set of board features R_3 of shape (W_2, W_2, k_L).

Parameter	Value	Parameter	Value
W_1	5	W_2	9
C_s	5	k_s	32
N	32		
C_L	5	k_L	32
C_f	3	k_f	32

Table 1. Parameters used in construction of 9x9 network.

3.4.4. Putting it all together

The three sets of board features are then concatenated together $R = (R_1; R_2; R_3)$. Thus, R is of shape $(W_2, W_2, F + Nk_s + k_L)$.

The board features are now used as input to another C_f convolutional layers, with k_f filters in each layer, each using a filter size of 3.

The output layer consists of a final convolutional layer, using a filter of size 1, and a softmax activation for output, to provide a probability distribution over actions, similar to the output in section 3.3.2. The final architecture is depicted in figure 1. And the parameter values were set to those in table 1

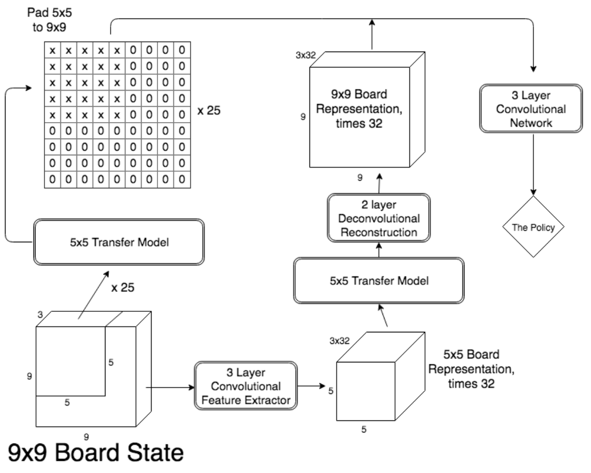

Pad 5x5
to 9x9
x 25
3x32
9x9 Board Representation, times 32
3 Layer Convolutional Network
2 layer Deconvolutional Reconstruction
The Policy
5x5 Transfer Model
x 25
5x5 Transfer Model
3x32
5x5 Board Representation, times 32
3 Layer Convolutional Feature Extractor
9x9 Board State

Figure 1. Transfer learning architecture

The three feature vectors R_1, R_2, and R_3 are concatenated and passed through a final fully-connected 2-layer output network to estimate Q-values for the larger board.

We train two 9×9 board agents: one with transfer learning, and one without. The one with transfer learning is trained the same number of iterations as the 5×5 board it uses for transfer learning, and the one with no transfer learning is trained with twice that number of iterations (so that in total, both models have been trained for the same number of iterations). This is so we can compare speed of training (since there is no useful loss or score curve to analyze).

Algorithm 1 Training Loop

```
Initialize ReplayBuffer, OpponentPool, PolicyNetwork
for $i = 1$ to NumEpisodes do
  if $i$ % 500 = 0 then
    Add PolicyNetwork to OpponentPool
  end if
  Initialize States, Actions, Environment
  Sample Opponent from OpponentPool
  while t < horizon do
    state, action = Sample from PolicyNetwork
    Add state to States
    Add action to Actions
    Update Environment from action
    if state not terminal then
      state, action = Sample from Opponent
      Update Environment from action
    end if
    Update PolicyNetwork (Gradient Descent)
  end while
  Compute Values using Environment.winner
  Add States, Actions, Values to ReplayBuffer
end for
```

3.5. Training

We include psuedo-code for our training loop in algorithm 1. One design choice we made was to store triples of (s, a, v) in a replay buffer, and use these triples to perform mini-batch stochastic gradient descent updates of the form described in section 3.2. Internally, within the replay buffer, for any (s, a, v) triple stored, we instead store eight triples (s_i, a_i, v), for $i = 1, ..., 8$, where s_i and a_i are obtained by flipping or rotating the board, which we found to lead to more even, symmetrical values being learned by the network.

3.6. Monte-Carlo Tree Search

3.6.1. Algorithm

Monte Carlo Tree Search (MCTS) is, as an tree search algorithm, frequently used within game playing agents. We have implemented our own version of MCTS.

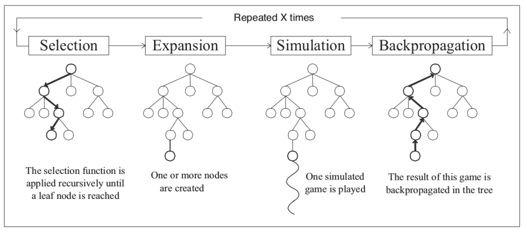

Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until a leaf node is reached
One or more nodes are created
One simulated game is played
The result of this game is backpropagated in the tree

Figure 2. Outline of MCTS. [5]

Each MCTS iteration consists of 4 steps, over which a partial search tree is built and that grows by a single node with each iteration. For each node in the search tree we store a few additional pieces of meta-data, including the number of times that we have visited the node, and the average value of the game state, as witnessed from this node (see backpropogation). As we progress through the iterations, we hope that our value for each node approximates the value of that state, and hence can be used to make informed decisions on what action to take.

In our version of the algorithm, we need two stochastic policies to be provided, the *selection policy* and *simulation policy*. Here, a stochastic policy is a function mapping actions, from some given state, to a weight, that intuitively represents how 'good' that action is.

Selection: In the selection phase, we stochastically walk down the partial search tree, according to the aforementioned selection policy, until we reach a state that is not in the tree.

Expansion: The expansion phase extends the partial search tree by one node. Our agent expands by simply adding the state that we reached in the selection phase to the tree.

Simulation: From the 'expansion node' (the node we added in the expansion phase), we finish playing a game, according to the simulation policy and see which player of the game won. Each action is chosen stochastically according to the simulation policy, and there is no backtracking.

Backpropogation: When we backpropogate, we update data stored in each of the nodes on the path we traversed in the selection phase, as can be seen in figure 2. The values updated are the number of times each node has been visited, and the average value of the node.

Intuitively we see that this agent can act in the same way as our Pure MC agent, if handed uniform policies, and we sample only according to that policy. However, to encourage exploration, when we are following our selection policy π_{sel} from some state s, we sample actions a from

$$p(a|s) \propto \frac{\pi_{sel}(a;s)}{n_{s'}+1}$$

where $s' = Succ(s,a)$, the successor state of s from action a, and $n_{s'}$ is the number of times that s' has been visited, as tracked by the partial search tree.

Finally, due to the model's flexibility we can try many policies for searching, such as the ones learned by our neural networks.

3.6.2. MCTS AGENT

For competitive play, we can use the Monte Carlo Tree Search algorithm with the policy output by policy network described in sections 3.2 to 3.4 as the selection and simulation policy. By the nature of using a search and using more evaluations of the network, the resulting agent tends to be stronger than a 'reflex agent' (an agent that doesn't use any form of searching), as it can avoid, with almost certainty, actions that lead to loosing in the near future.

4. Results

4.1. 5x5 board

4.1.1. 2-3 HOUR TRAINING RUNS

When moving first, our 5×5 board agent is able to defeat Pachi every single time. However, this is only because Pachi surrenders very quickly to good opening moves. When playing second, the agent never defeats Pachi, and when we (novice Go players) play our agent, we always win. However, this does not mean that our agent does not learn anything. In order to understand what is happening, we analyzed games played by different training runs of our agent against Pachi, and against us, with a couple representative positions depicted in figure 7. First, it is worthwhile to note that the agent consistently learns that the best starting move is the middle square. It learns a couple interesting strategies - what we will call the "grid" strategy and "rush eye" strategy. In the "grid" strategy, it places a series of stones on diagonals in order to cover as much territory as quickly as possible. This maximizes the fraction of moves that will cause the opponent to resign - so when playing against a random or unintelligent opponent, this will result in the quickest winning strategy. However, this strategy is very weak, and can be beaten by an agent capable of planning ahead to capture each diagonal stone, one at a time (as the agent never solidifies the stones into uncapturable eyes). The "rush eye" strategy involves quickly building a particular type of "eye", which is a stone formation that is uncapturable in go, and among the first techniques students of the game are taught. However, by focusing on this particular eye, the agent sacrifices the majority of the territory of the board, which allows intelligent player (like Pachi) to easily win in the long run. Note that the agent never learns to play both these strategies at once - these were taken from different training runs. In both cases, when the model learns one of these strategies, it consistently applies it in every game. So, we were unable to achieve the playing power we hoped for on a 5×5 board, but the agent did learn some techniques, and although transfer learning to a 9×9 board will probably not result in stellar results when working with a weak 5×5 player, we decided to proceed anyway to see if the techniques it did learn for the 5×5

board were beneficial for training speed or learning of the 9×9 board.

4.1.2. 10 HOUR TRAINING RUN

The above results occurred from two separate ≈ 2 hour training runs (of 100,000 iterations). We tried one much longer training run of ≈ 10 hours (500,000 iterations), and found the final results were not much different than the shorter runs. However, during these 10 hours of training, at one point the model played a game against itself of 1,311 moves - much longer than the average game length of ≈ 20 (Figure 7). This crashed the training run (we had a bug in the replay buffer), but we were able to resume training from this checkpoint. Later, we went back and investigated how the model was playing during this game, and were surprised with the results. Its performance at this point looked the most human-like to our qualitative analysis - It would create a wall of stones, blocking off territory on one side of the board, and then when it could not obtain any more territory, it would play the only valid moves available to it (which were to play in its own territory, decreasing its score). This strategy would have resulted in victory on the 5×5 board if it had only learned that the center square is the optimal location for the first move (as far as we can tell, it is impossible to defeat Pachi with any other opening move). Hence, these were our most promising results on the 5×5 board. Unfortunately we were unable to further explore the effect of longer training times before the deadline. Nevertheless this is a valuable lesson for the future, that even if short-term results are unsatisfactory, it is important to at least try long training runs to see if they result in significant improvements.

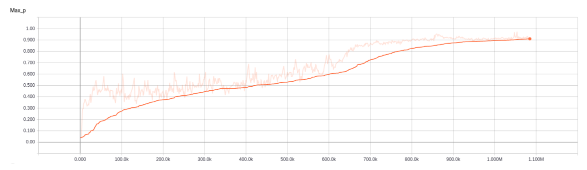


Figure 3. The maximum probability value output from the network throughout the 10 hour run, with respect to time.

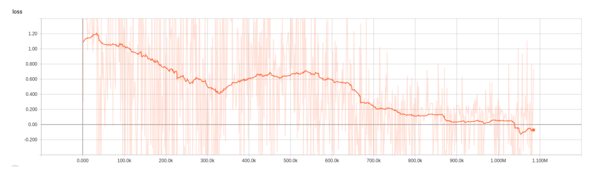


Figure 4. The value of the loss throughout the 10 hour run, with respect to time.

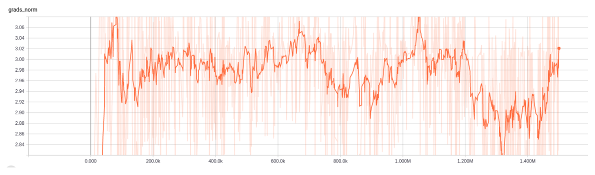


Figure 5. The gradient magnitude throughout the 10 hour run, with respect to time.

4.1.3. REFLECTIONS ON THE TRAINING RUNS

Whilst watching the training during the above runs, we saw a lot of oscillatory behavior. On some runs we witnessed the probability for picking the center move in the initial state reach highs of 0.99, and then plummet down to 0.03 later in the run. We also see in the non-smoothed curves in figures 4 and 5 that there is a high variation in values through the whole training run.

One technique for reducing this variance is subtracting a *reinforcement baseline* as described in [7] and [9], and used in [1]. When included as a term in the policy gradient update from section 2.2, it can be shown to reduce variance in the discounted reward R that we are aiming to maximize the expected value of, whilst not effecting it's expected value, provided the reinforcement baseline is chosen suitably. Thus this suggests that we should work to implement a value network as a reinforcement baseline, as used in [1] and described in section 3.2

4.2. 9x9 board

Unfortunately, all our attempts at training any type of agent (with or without transfer learning) on the 9×9 board failed miserably. Our agents never defeat Pachi, and when we observe their self-games, or play them ourselves, their moves appear no better than random. This does not necessary mean that our transfer learning method is useless - it is possible that it would perform better if we could train a better 5×5 agent to start with.

5. Conclusion

5.1. Future Work

We have decided that we would like to continue working on ZetaGo beyond this project, because we have identified a number of potential avenues of exploration that we simply did not have time to try in the course of this quarter. Most importantly, in this paper we omitted the value network for the baseline in the policy update, but according to [7], this baseline is essential for rapid learning. And we observe a lot of variance in our training results. Hopefully this addition will prevent oscillatory behavior in training, and reduce the likelihood of landing in local optima. We

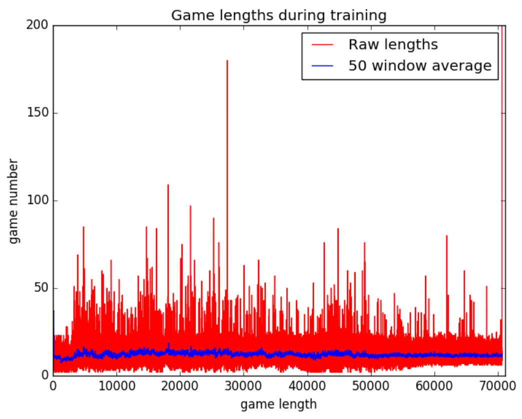
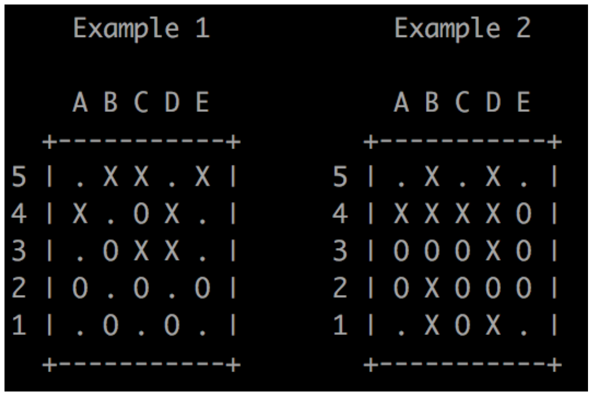


Figure 6. When our model crashed after encountering a game of length 1311 during training, we decided to plot the game lengths. However the plot shows a mostly constant average game length, although the variance does seem to change a bit during training.



```
   Example 1          Example 2

   A B C D E          A B C D E
 +-----------+      +-----------+
5 | . X X . X |    5 | . X . X . |
4 | X . O X . |    4 | X X X X O |
3 | . O X X . |    3 | O O O X O |
2 | O . O . O |    2 | O X O O O |
1 | . O . O . |    1 | . X O X . |
 +-----------+      +-----------+
```

Figure 7. Examples of a couple strategies that our 5×5 agent learns. The agent plays white ("O") in example 1 and uses the "grid" strategy. After a separate training run, the agent in example 2 plays black ("X") and uses the "rush eye" strategy. The eye is the structure in the upper left corner enclosing two empty (".") squares.

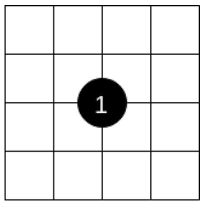


Figure 8. The exciting game where ZetaGo beats Pachi on a 5×5 board.

have another idea for improving the model, inspired by a very successful recent reinforcement learning paper, ”Reinforcement Learning with Unsupervised Auxiliary Tasks” [10]. In this paper, auxiliary tasks are added to the loss computation, so that the reinforcement learning agent must learn those tasks (as well as the policy for maximizing expected reward) during training. This can drastically speed up training, and improve results. For the Go agent, we could add auxiliary tasks such as predicting the next board state. This would force the model to learn explicit representations of when a move will capture opponents stones, and when a move will suicide a group of your own stones - something that we never saw our model learn (it seemed perfectly willing to sacrifice large groups of stones, and rarely captured opponent stones).

5.2. Final Remarks

Throughout this project we had been hoping that ZetaGo would learn to play Go well, *from scratch*. We saw some promising initial results, and our agent learns a couple interesting strategies. Unfortunately the transfer learning components never conferred any advantage to the larger models, and we were unable to train a strong agent on even the small 5×5 board. From observing some major oscillatory behavior from the agent during training, we think the best immediate next steps will involve implementing the variance reduction techniques mentioned, and we remain hopeful for the eventual success of ZetaGo.

5.3. Team member contributions

We both discussed every technical decision before and / or after implementation. Michael implemented most of the training loop, replay buffer, network configuration, and monte carlo tree search. Luke implemented most of the policy network and transfer learning architecture, and saving and loading opponents. We both feel that our relative contributions were equivalent in scope.

References

[1] Silver, David, et al. “Mastering the game of Go with deep neural networks and tree search.” Nature 529.7587 (2016): 484-489.

[2] Brockman, Greg, et al. “OpenAI gym.” arXiv preprint arXiv:1606.01540 (2016).

[3] Baudi, Petr, and Jean-loup Gailly. “Pachi: State of the art open source Go program.” Advances in computer games. Springer Berlin Heidelberg, 2011.

[4] Storkey, Amos. “Training deep convolutional neural networks to play go.” Proceedings of the 32 nd International Conference on Machine Learning. 2015.

[5] Chaslot, Guillaume and Bakkes, Sander and Szita, Istvan and Spronck, Pieter. “Monte-Carlo Tree Search: A New Framework for Game AI.” AIIDE. 2008.

[6] Tesauro, Gerald. “Temporal difference learning and TD-Gammon.” Communications of the ACM 38.3 (1995): 58-68.

[7] Sutton, Richard S., et al. “Policy gradient methods for reinforcement learning with function approximation.” Advances in neural information processing systems. 2000.

[8] Mnih, Volodymyr, et al. “Asynchronous methods for deep reinforcement learning.” International Conference on Machine Learning. 2016.

[9] Williams, Ronald J. “Simple statistical gradient-following algorithms for connectionist reinforcement learning.” Machine learning 8.3-4 (1992): 229-256.

[10] Jaderberg, Max, et al. ”Reinforcement learning with unsupervised auxiliary tasks.” arXiv preprint arXiv:1611.05397 (2016).

Mastering the game of Go from scratch

Michael Painter, Luke Johnston
Stanford University

Introduction

In the landmark paper “Mastering the game of Go with deep neural networks and tree search” [1], superhuman performance on the game of Go was achieved with a combination of supervised learning (from professional Go games) and reinforcement learning (from self-play). However, traditional pure RL approaches haven’t yielded satisfactory results on the game of Go on a 19x19 board because its state space is so large and its optimal value function so complex that learning from self-play is infeasible. However, these limitations do not apply to smaller board sizes. In this project, we investigate an entirely unsupervised, reinforcement-learning based approach to playing Go, by learning how to play on a smaller board and using a form of transfer learning to learn to play on larger board sizes. If successful, this suggests an effective strategy using RL for approaching large state space problems, for which we, as humans, are not experts (and cannot generate any good dataset).

Reinforcement Learning Details

A policy $p_\rho(s)$ parameterised by ρ, is learned, by optimising over the expected reward:

$$U(p_\rho) = \sum_\tau \Pr(\tau|\rho) R(\tau)$$

where τ is a path of states. By the policy gradient theorem [2] and using an empirical estimate, we can arrive at an update rule for ρ of

$$\Delta\rho = \frac{\alpha}{n} \sum_{i=1}^{n} \sum_{t=1}^{T^i} \frac{\partial \log(p_\rho(s_t^i, a_t^i))}{\partial \rho} v_t^i$$

where for each episode i, s_t, a_t and v_t are the t’th state, action and value.

To train the network we play games against an old opponents. Where we play against policy ρ^-, chosen uniformly from the pool of old opponents. The current policy is copied every 500 episodes into the opponent pool.

To evaluate the policy learned we play against the Pachi AI [4], provided by OpenAI’s Go Environment [5]. We track two metrics, average game length (to see if the game was ‘competitive’) and the average value of the game from our agent’s perspective (the win ratio).

A number of different design choices were made compared to Alpha Go:

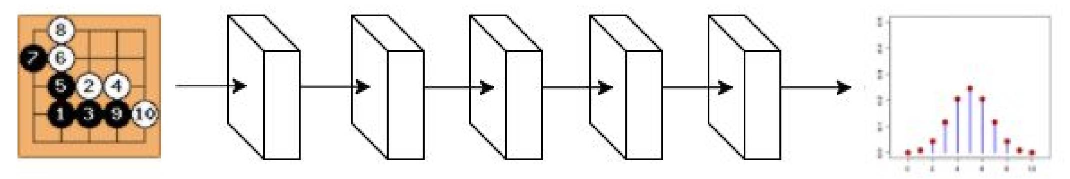
- Removal of variance reduction term from update rule, as it would significantly increase the computation time.
- No use of Monte Carlo Tree Search (MCTS) in training, allowing for the self play to run some orders of magnitudes quicker.
- We still learn a policy (softmax output) rather than a Q function, as many actions could have a large Q-values, leading to a large branching factor when run with MCTS (for competitive play).

5x5 Board Architecture

The architecture used for the 5x5 board, the ‘base case’, is a 5 layer convolutional neural network. Padding is used so that each of the intermediate layers maintains the same shape of the input.

- The first layer uses a 5x5 convolution, with _ filters and __ activation.
- The second to fourth layers use a 3x3 convolution, with _ filters and a ReLU activation.
- The final, fifth layer uses a 1x1 convolution, with 1 filter, and a softmax activation to provide a probability distribution as output.
- Each layer includes bias terms.

To encode the input, we use 3 channels, one for the white pieces, one for the black pieces and one for free spaces.



Example of Transfer Learning to 9x9 Board

Black box:

- A network trained to play on a 5x5 board is used as a black box (weights frozen),
- The black box is used like an oracle to query about ‘global’ information and ‘local’ information.

Transfer method 1 - convolution, or ‘local’ use:

- The 5x5 network is applied to every possible window of the 9x9 board (as if the 5x5 network were a convolutional filter).
- Each application yields a 5x5 output of ‘policy values’ for actions at each board location:
  - Which is padded with zeros to match original 9x9 board;
  - All outputs concatenated to form 9x9x25 tensor.

Transfer method 2 - scaling, or ‘global’ use:

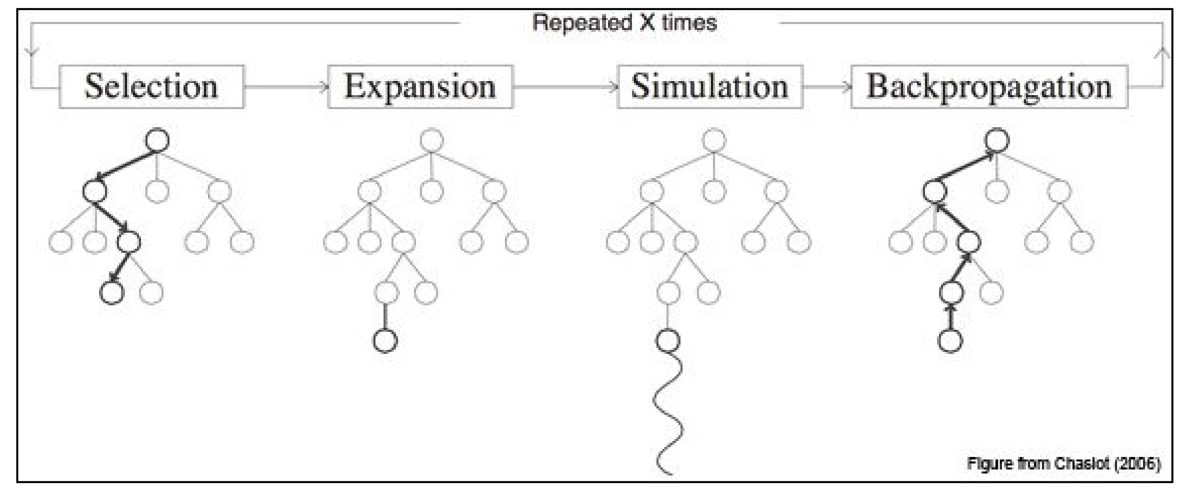
- A convolution is applied to the 9x9 board, such that the output is 5x5, by using a 5x5 convolution, and no padding
- The convolutions output is run through the black box
- The transpose of the *same* filter is applied to the 5x5 output from the black box, to mimic an ‘inverse’, and yielding a 9x9 output.

MCTS

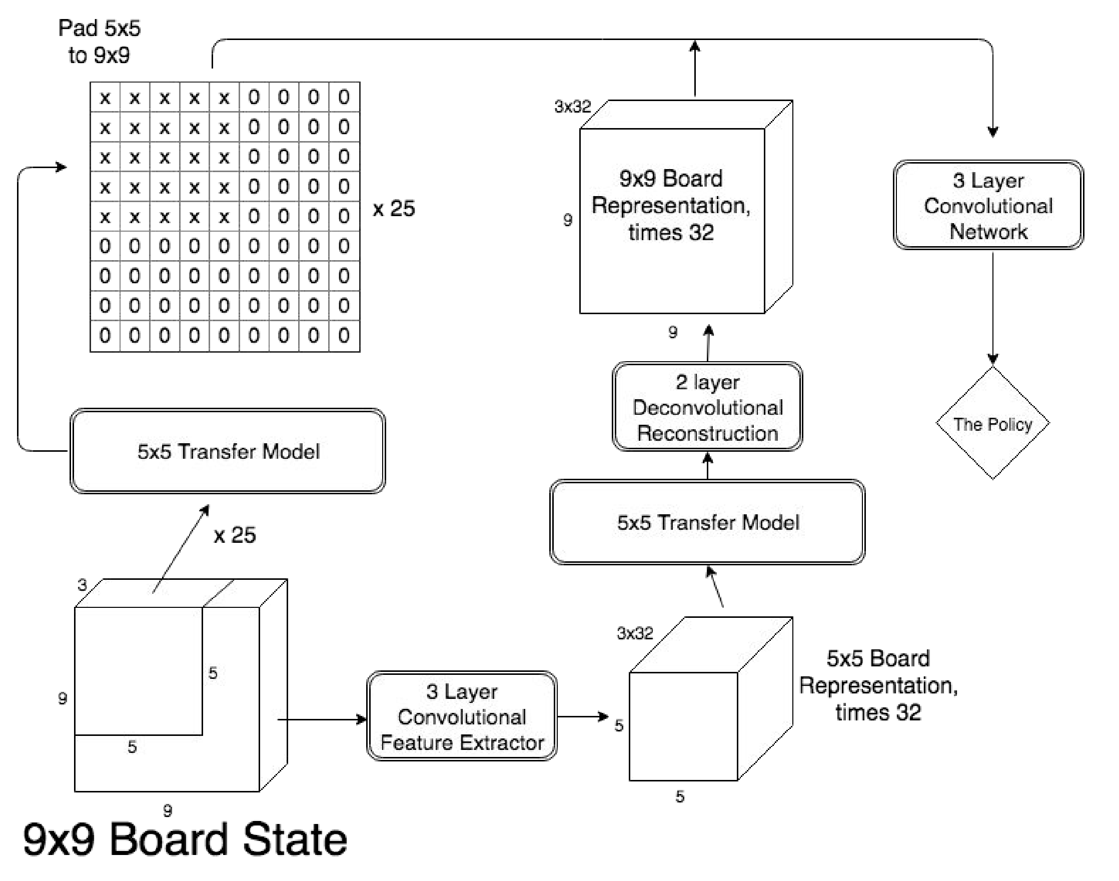
For competitive play we use the Monte Carlo Tree Search (MCTS) algorithm, which uses a stochastic policy to guide a tree search for a zero sum game. The algorithm maintains a partial search tree, adding one node to it on every iteration. Internally, every node maintains a ‘value’.

- Selection: traverse the partial tree (probabilistically, according to the search policy), until a node not in the tree is reached.
- Expansion: add the node to the tree.
- Simulation: use a policy to (greedily) pick actions, until a terminal state. The value of this run of the game is then known.
- Backpropagation: update the value of all nodes traversed in this iteration using the value obtained from the simulation.

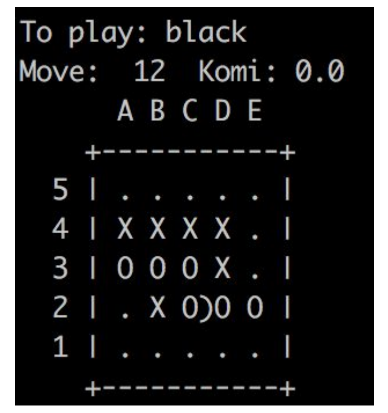
After some fixed number of iterations or time, the MCTS agent selects the action the leads to the child with maximum value from the root of the tree.



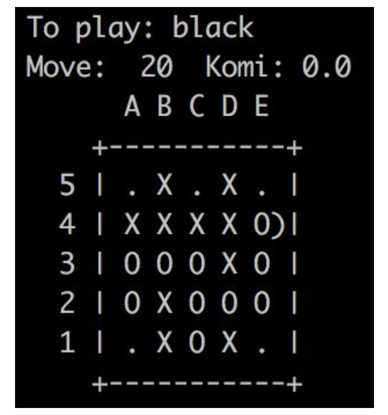
Composite Network Architecture


Pad 5x5 to 9x9
x 25
5x5 Transfer Model
x 25
3
9
5
5
9
9x9 Board State
3 Layer Convolutional Feature Extractor
3x32
5
5
5x5 Board Representation, times 32
5x5 Transfer Model
2 layer Deconvolutional Reconstruction
9
3x32
9
9x9 Board Representation, times 32
3 Layer Convolutional Network
The Policy

Gameplay

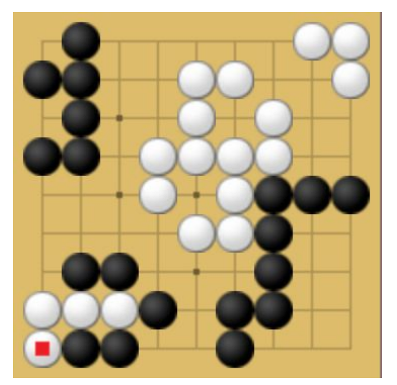


```
To play: black
Move:  12  Komi: 0.0
     A B C D E
   +-----------+
 5 | . . . . . |
 4 | X X X X . |
 3 | O O O X . |
 2 | . X O)O O |
 1 | . . . . . |
   +-----------+
```



```
To play: black
Move:  20  Komi: 0.0
     A B C D E
   +-----------+
 5 | . X . X . |
 4 | X X X X O)|
 3 | O O O X O |
 2 | O X O O O |
 1 | . X O X . |
   +-----------+
```

- 5x5 board: learns to contest middle 3x3 squares first
  - Does not learn optimal first move (center square)
- Forms an “eye” structure on top border of board
- Makes a couple obvious mistakes - 2B in first image, 1D in second image



- Above diagram illustrates a couple Go concepts

References

[1] Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." Nature 529.7587 (2016): 484-489.
[2] Sutton, Richard S., et al. "Policy gradient methods for reinforcement learning with function approximation." NIPS. Vol. 99. 1999.
[3] Greensmith, Evan, Peter L. Bartlett, and Jonathan Baxter. "Variance reduction techniques for gradient estimates in reinforcement learning." Journal of Machine Learning Research 5.Nov (2004): 1471-1530.
[4] Baudiš, Petr, and Jean-loup Gailly. "Pachi: State of the art open source Go program." Advances in computer games. Springer Berlin Heidelberg, 2011.
[5] Brockman, Greg, et al. "OpenAI gym." arXiv preprint arXiv:1606.01540 (2016).