Giraffe

Using Deep Reinforcement Learning to Play Chess
(Summary)

M. Nassim Aliousalah, Mohammed D. Belgoumri, Ayoub Mokhtari, Oussama Djetaou

SIQ1 - SIQ2
ESI

Contents

1 Introduction

Chess has been an extensively-studied problem in artificial intelligence since the 1950s. In a chess problem, we basically try to find a move that maximizes our chance of winning. In high level chess, we usually model the opponent to be the same as ourselves.

An alternative formulation of the problem more often used is : given a chess position, we want to assign a score to it that corresponds to the chance of winning for the side to move, and for any given position, the score for one side is the negated score for the opposite side, because in chess scores are centered around 0.

2 Background and Related Work

2.1 Conventional Chess Engines

2.1.1 Minimax

The `minimax` algorithm is a simple recursive algorithm to score a position based on the assumption that the opponent thinks like we do, and also wants to win the game.

In chess, this algorithm works in theory but not in practice, the most common approach is a fixed-depth search, we go down in the tree and when we are at the end of the sub-tree we want to search, we call a static evaluation function that assigns a score to the position. The method generates a new problem, the horizon effect,the solution to it is called *"quiescent search"* (q-searche).

2.1.2 Alpha-beta pruning

This algorithm can be optimized by introducing a lowerbound/uperbound for each call to `minimax`(), we call this : the alpha-beta pruning. In alpha-beta pruning we only explore nodes that can potentially be "useful", we can stop searching a node as soon as we prove that the result will be outside the window.In the best case scenario, it reduces the effective branching factor to the square root of the original value, which would allow the search to go about twice as far in the same amount of time. In conventional chess engines, there are a few common heuristics to improve move ordering :

1. If we had previously searched the same position, the move that ended up being the best is probably still the best (even if the previous search was done to a shallower depth).

2. If a move proved to be good in a sibling node, there is a good chance it will be good for the node under consideration as well.

3. Capturing high-valued pieces with low-valued pieces is usually good, and capturing defended low-valued pieces with high-valued pieces is usually bad.

4. Queen promotions are usually good.

2.1.3 Evaluation

The job of the evaluation functions is to assign scores to positions statically. For instance, we can take a look at the evaluation function of Stockfish, which consists of 9 parts: Material, Piece-Square Tables, Pawn Structure, Piece-specific Evaluation, Mobility, King Safety, Threat, Space and Draw-ish-ness.

2.2 Machine Learning in Zero-Sum Games

2.2.1 Mate Solvers

There have been many attempts at using machine learning to solve chess endgames. We note that these situations are much simpler than average chess positions. Approaches using genetic programming do not scale to much more complicated positions seen in midgames and openings.

2.2.2 Learning Based on Hand-Selected Features

In a typical chess evaluation function, there are hundreds of parameters that must be tuned together. NeuroChess and The Falcon project are two attempts to tune evaluation parameters using generic programming with the help of a mentor, and a more recent attempt is the Meep project by Veness et al, whose evaluation function is a linear combination of hand-designed features, and where the weights are trained using reinforcement learning.

2.2.3 Temporal-Difference Learning

Temporal-difference reinforcement learning is an approach to evaluate the performance of models, where systems are tested to predict their own evaluation in the near future. This approach has several advantages over using game results, the most important one being that evaluations used to make moves in the same game can receive different error signals. In temporal-difference learning, only the blunder (which creates a temporal inconsistency) will get a high error signal.

This significantly improves the signal-to-noise ratio in error signals.The most famous study in using temporal-difference reinforcement learning to play zero-sum games is probably Tesauro's backgammon program TD-Gammon [19] from 1995.

2.2.4 Deep Learning

To go beyond weight tuning with hand-designed features we need a powerful and highly non-linear universal function approximator, where the input-output mapping is inherently hierarchical, the models would have to be made extremely large. In this project, we apply deep learning to chess. We use deep networks to evaluate positions, decide which branches to search, and order moves.

3 Neural Network-Based Evaluation

The first challenge we have to deal with is the evaluation function, because it is required in other systems. This function is called at the leaf nodes, takes as input a position and estimate the probability of winning

3.1 Feature Representation

In order to make the neural networks work efficiently, we need to make the feature representation of the input space relatively smooth. Representing positions in bitmaps is unlikely to work well because in some cases, different positions have the same distance from each other in feature space. Thus we will be using slot system. Feature representation consists of the following parts :

1. Side to Move

2. Castling Rights

3. Material Configuration

4. Piece Lists
5. Sliding Pieces Mobility
6. Attack and Defend Maps

3.2 Network Architecture

The evaluator network is a 3-layer network, where all hidden nodes use Rectified Linear activation (ReLU). The output node uses hyperbolic tangent activation to constrain the output between -1 and 1. there are three modalities :

1. piece-centric
2. square-centric
3. position-centric

3.3 Training Set Generation

Generating a good set of positions needs to satisfy a few potentially conflicting objectives :

1. Correct Distribution
2. Variety
3. High Volume

3.4 Network Initialization

Even TD-Leaf can train a system from random initialization, in a complex problem like chess, random initialization would lead to extremely long training times. Therefore a simple evaluation function introduced to bootstrap the training process to puts the neural network at a point in the parameter space that is closer to a good minimum.

3.5 TD-Leaf

One way to train the network is through temporal-difference reinforcement learning (a modified version of TD-Leaf(λ) - for this project) making the evaluation function a better predictor of its own evaluation later in time.
The update rule for TD-Leaf(λ) :

$$w = w + \alpha \sum_{t=1}^{N-1} \Delta J(x_t, w)[\sum_{j=t}^{N-1} \lambda^{j-t} d_t]$$

4 Probabilistic search

4.1 Alternative formulation of the problem

Instead of trying to construct the theoritical proncipal variation –which necessitates a complete traversal of the game tree–, the authors propose the goal of "*finding as much of it as possible*".

4.2 Probability-limited search

In contrast to the more conventional *depth-limited search* where one explores all nodes up to a certain depth (in this case all possible games with a certain number of moves), probablity-limited search looks for all nodes with a certain probability of being part of the theoreitical PV of the root position.

At each node, we compute the probability of its descendents being in the PV given that the node itself is. We start at the root –which has a 100% propability of being in the PV–, and we propagate the probabilities recursively downwards.

Instead of the halting condition beeing the reaching of a maximum depth, we stop when the probability drops below a certain threshold. We then evaluate the current node and return the result from the `minimax` call.

5 Neural Network-Based Probability Estimation

5.1 Neural Network-Based Probability Estimation

Can be resumed into two parts:

1. Describing the parent position
2. Describing the move under consideration

The part that describes the parent position is exactly the same as the feature representation we use for position evaluation, as described in the feature representation of Neural Network-Based Evaluation.With additional features as follows:

1. Piece Type
2. From Square
3. To Square
4. Promotion Type
5. Rank

5.2 Network Architecture

Similar to the architecture for the evaluation network described in network architecture of Neural Network-Based Evaluation .The only difference is that the output node has logistic activation instead of hyperbolic tangent activation. It also has rectified linear activation for hidden nodes, and separately-connected layers for features from the three different modalities.

5.3 Training Positions Generation

We need a corpus of unlabeled positions. But not the same corpus used in Evaluation network training. For training the evaluation network using TD-Leaf(λ), we need the training positions to be representative of root positions that will appear in actual games. To generate this training set :

1. Performing time-limited searches on the root positions from the training set for evaluation network training
2. Performing a random sample from the positions encountered as internal nodes in those searches

5.4 Network Training

We have a trained evaluator which can find the best move for each training position, as a consequence, we don't need an iterative temporal-difference learning.

1. **First step:** label each training position with the best move by performing a time-limited search on each one of them.

2. **Second step:** the generation of the training set for gradient descent by combining each position with all legal moves from that position, labeled by a binary training target specifying whether the move is the best move or not.

3. **Last step:** performing a stochastic gradient descent with AdaDelta update rules using the cross-entropy loss function.