Playing draughts using PPO

A deep learning model that plays draughts via an implementation of PPO using a neural network library with auto differentiation built from scratch

Edgar Maddocks

Bedford School
03/05/2024

Contents

1 Analysis

1.1 Background

TO BE WRITTIEN

1.2 Evidence of Analysis

TO BE WRITTIEN

1.3 Current Systems

TO BE WRITTIEN

1.4 Identification of end-user

TO BE WRITTIEN

1.5 Modelling of the problem

TO BE WRITTIEN

1.6 Set of objectives

TO BE WRITTIEN

2 Research Log

2.1 Draughts Rules

https://www.mastersofgames.com/rules/draughts-rules.htm

Draughts is an English board game played on an 8x8 checkered board, identical to a chessboard. Each player begins a game with 12 pieces, usually flat round discs. The pieces and board are usually black and white, and will be referred to as such. The board is first placed between the two players such that the bottom right-hand corner is a white square, for both players.

A coin is tossed to decide who plays black, and that player has the first move. Each player places their pieces on the 12 black squares closest to themselves. The setup of the board can be seen in Figure 1.

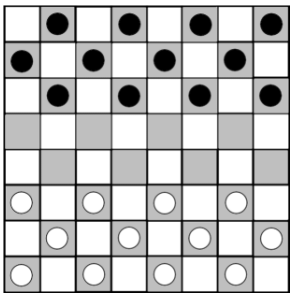

Figure 1: An image showing the starting position of a game of draughts

The pieces only move diagonally (so will always be on black squares) and the aim is to take all of the opposing players pieces, or to put the opposing player in a position with no possible moves. Players take turns moving their shade of pieces. If at any point of the game, a player's piece reaches the opposing players edge of the board, the piece becomes a 'King', and another piece should be placed on top of said piece to indicate so.

Unless a piece is crowned and a 'King' it may only move and take pieces diagonally forwards. Kings may move and take both forwards and backwards.

If an adjacent square has an opponents piece and the square immediately beyond the oppositions piece is empty, the opponents piece may be captured. If the player who's go it is has the opportunity to capture one or more pieces, then they must do so. A piece is taken by moving your own piece over the opposing player's, into the vacant square, and then removing the opposing piece from the board. An example of this process can be seen in Figure 2.

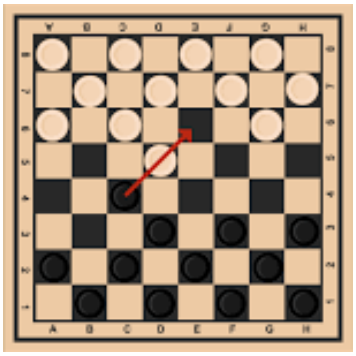

Figure 2: Example of a piece being taken in draughts

Unlike a regular move, a capturing move may make more than one 'hop'. This is if the capture places the piece in a position where another capture. In this case, the additional capture must be made. An example of this successive capturing behaviour can be seen in Figure 3.

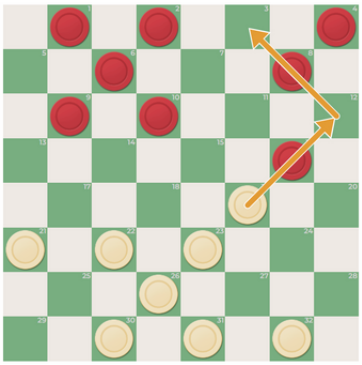

Figure 3: Visualization of a double hop capture

The move may only end when the position has no more captures available or an uncrowned piece reaches the opposing edge of the board and becomes a King. The capture sequence can only be made by one piece

per move. I.E. You cannot make on capture with one piece, and then another capture with another piece in the same move.

However, if more than one piece can capture, the player has free choice over which piece to move. Likewise, if one piece can capture in multiple directions then the player has the choice in which direction to move. **Note:** it is not compulsary for the player to move in the direction, or with the piece, that will lead to the greatest number of captures in that move.

If no capturing moves can be made, then any piece may be moved diagonally onto a vacant square.

The game ends when all of a players piece's have been captured, or a player has no available moves.

2.2 Neural Networks

https://www.ibm.com/topics/neural-networks

2.2.1 What is a neural network

A neural network is a machine learning model which aims to mimic the processes of the human brain. Each network contains inputs and outputs, as well as one or more layers of hidden nodes - which act as artificial neurons. In a fully connected network, each node is connected once to each node in the next layer - an example of how one node connects to the next layer can be seen in Figure 4.

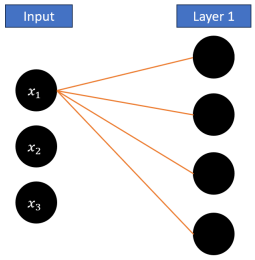

Figure 4: Example of a fully connected node and layer

Neural networks are a supervised learning model, meaning that they learn from labeled data (which has the objective correct answer in the data). They are sometimes referred to as artificial neural networks (ANNs) or simulated neural networks (SNNs).

Neural networks can be modelled as a collection linear regression units.

2.2.2 Linear Regression Unit

A single linear regression unit output has the formula:

$$\hat{y} = \sum_{i=0}^{n} w_i x_i + b$$

Where $\hat{y}$ is the predicted output, n is the number of inputs, x_i is the ith input, w_i is the weight of x_i, and b is a bias. If, for example, there were 3 inputs the full equation for $\hat{y}$ would be:

$$\hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$$

2.2.3 Vectorization of summations

https://courses.cs.washington.edu/courses/cse446/20wi/Lecture8/08_Regularization.pdf

This calculation can be vectorized to improve efficiency and would be notated:

$$\hat{y} = XW + b$$

Where we let

$$W = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} \qquad X = \begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix}$$

X here is a row vector as this is the most common format for data as an input to a network (e.g being read from a csv). This parallelized computation is much faster, and can be parallelized using the GPU to further improve speed and efficiency.

2.2.4 Forward Pass of Dense Layer

In the case of neural networks, lots of these linear regression units can be combined to form a vector of outputs. Each of these regression units will have the same inputs, therefore X can have the same definition. However, W will now be composed of multiple vectors of weights, instead of just one.

Here we let

$$W = \begin{bmatrix} w_{11} & w_{21} & \dots & w_{j1} \\ w_{12} & w_{22} & \dots & w_{j2} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1n} & w_{2n} & \dots & w_{jn} \end{bmatrix}$$

Where j is now the number of nodes in the layer. If we rewrite our forward pass equation to use this weight matrix, with each node as its own regression unit:

$$Y = XW + B$$

Where:

$$Y = \begin{bmatrix} \hat{y_1} & \hat{y_2} & \dots & \hat{y_n} \end{bmatrix} \qquad B = \begin{bmatrix} b_1 & b_2 & \dots & b_j \end{bmatrix}$$

With W and X having the same definition as most recently defined.

2.2.5 Gradient Descent

Although back propogation can be solved in closed form, it comes with a large time complexity (greater than $O(n^3)$), and therefore a process called gradient descent is usually employed.

Gradient descent works to minimize the error of a model by iteratively locating a minimum in the error function. For example, a common cost (error) function is mean squarred error.

$$MSE(Y, \hat{Y}) = \frac{1}{n} \sum_{i=0}^{n} (y_i - \hat{y}_i)^2$$

This function effectively calculates the absolute distance between the predicted value of one data point ($\hat{y_i}$) and the true value at that same point (y_i), for every value in the dataset (of size n) and takes the mean of these distances. Again, this calculation can be vectorized using the equation below.

$$MSE(Y, \hat{Y}) = \frac{1}{n}((Y - \hat{Y})^T \cdot (Y - \hat{Y}))$$

If we plot the MSE curve of different weights and biases - of a single regression unit, it looks as follows.

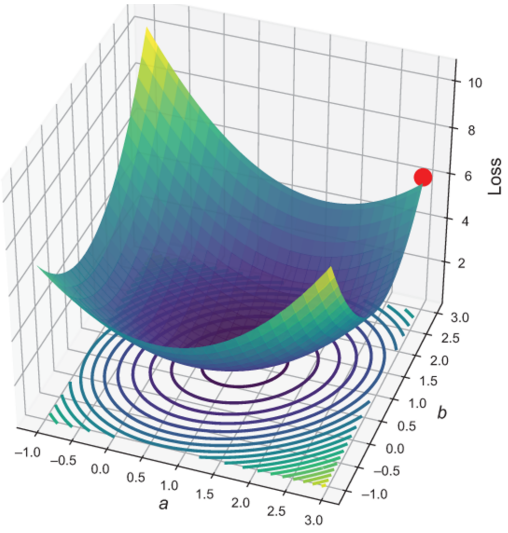

Figure 5: Plot of MSE in a single regression unit

Here the weight has been labelled a, but the bias has remained as b. As we can see, there is a clear local minimum of this error function, and the values that obtain this minimum, is what gradient descent aims to achieve.

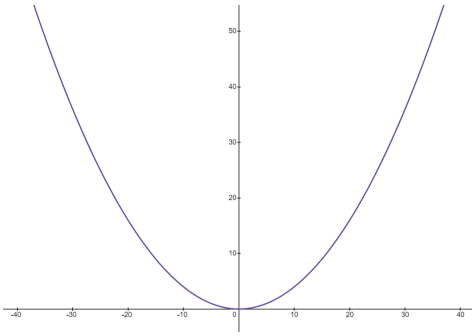

Figure 6: MSE with respect to one parameter

In figure 6 we can see a representation of the MSE plot with respect to only one of the parameters. This makes understanding the process of gradient descent much simpler. For example, the minimum of this function ($f(x) = \frac{x^2}{25}$) is at $x = 0$. So, if we were to have an initial x value of say -40 and we calculated the gradient to be 3.2, we could then subtract this from 40. Resulting in a new value of 36.8, which is closer to our optimal value of 0. This process is completed iteratively until the minimum, or near to it, is reached.

To denote this mathematically, we can say that

$$w_{t+1} = w_t - \alpha \frac{\partial E}{\partial w} \tag{1}$$

$$b_{t+1} = b_t - \alpha \frac{\partial E}{\partial b} \tag{2}$$

Where w_{t+1} and b_{t+1} are the parameters at the next timestep, w_t and b_t is the value of the parameters at the current timestep, $\frac{\partial E}{\partial w}$ and $\frac{\partial E}{\partial b}$ are the derivatives of the error with respect to each parameter, and α is the learning rate. This learning rate controls the size of our 'jumps' and prevents the process from beginning to spiral away from the optimal values.

That is a basic overview of gradient descent in a single regression unit, but in a dense network, the process is almost identical. An error function is evaluated, the derivative with respect to the inputs calculated, and the new values updated. The main expense computation-wise is

https://en.wikipedia.org/wiki/Automatic_differentiation Automatic differentiation