Playing draughts using reinforcement learning

An investigation into playing draughts using a reinforcement learning via a neural network library, with an implementation of automatic differention, built from scratch.

Edgar Maddocks

TO DO

1.1, 1.2.3, 1.3.1 (UML), 1.3.2, 1.3.3, 1.4

Graph Traversal	Autodiff backprop
Tree Traversal	MCTS Search
Advanced Matrix Operations	Dot Procuts, Convolution, Autdiff backprop, Neural Nets
Recursion	MCTS Tree BackProp, Autodiff backprop
User-defined algos	RL solution

Bedford School

03/05/2024

Contents

1 Analysis

1.1 Background

This project is an investigation into the use of reinforcement learning to play games (in this case draughts/checkers). The model will use self-play and monte carlo tree search algorithms, coupled with a multi-headed neural network to understand the game and estimate optimal actions.

The neural network will built using a library, that has an autograd engine implemented, all built from scratch. This investigation will however use some scientific computing libraries for faster simple matrix operations (such as numpy). Some more complex functions (such as cross-correlation and its derivative) will also be built from scratch.

1.2 Evidence of Analysis

1.2.1 Current systems

1.2.1.1 Draughts

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.

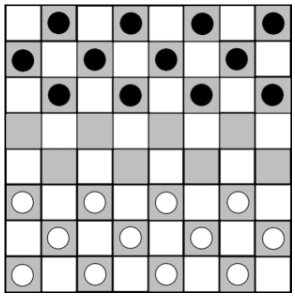

Figure 1: An image showing the starting position of a game of draughts

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 3.

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 is possible. In this case, the additional capture must be made. The capture sequence can only be made by one piece per move. i.e. You cannot make one 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 compulsory for the player to move in the direction, or with the piece, that will lead to the greatest number of captures in that move.

A 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.

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

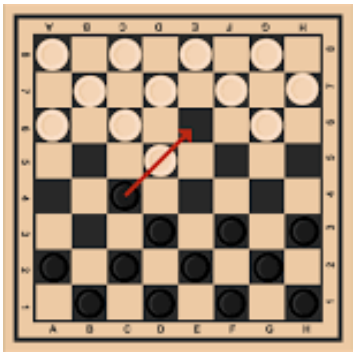

Figure 2: Example of a piece being taken in draughts

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

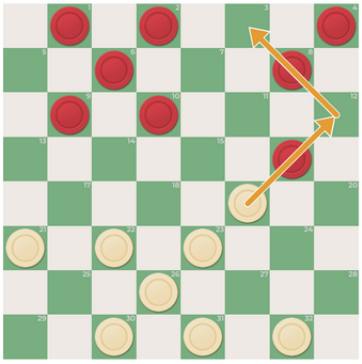

Figure 3: Visualization of multiple captures in one move

1.2.1.2 Neural Networks

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 below.

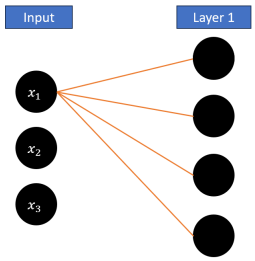

Figure 4: Example of a fully connected node and layer

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.

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:

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

Vectorization of processes

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.

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.

Forward Pass of a 1D Convolutional Layer

The main function of a convolutional layer is to extract key features and are often used in image recognition and classification solutions.

The input to a 1D convolutional layer is a 1 dimensional matrix which then has a cross-correlation operation completed with a filter. The forward pass is quite simple and consists of sliding the filter across the input and taking the dot product. An example of this can be seen below.

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \star \begin{bmatrix} 2 & -1 \end{bmatrix} = \begin{bmatrix} 2 \cdot 1 + -1 \cdot 2 & 2 \cdot 2 + -1 \cdot 3 \end{bmatrix} \tag{1}$$

$$= \begin{bmatrix} 0 & 1 \end{bmatrix} \tag{2}$$

The star here denotes the cross-correlation operation. And this specific method of cross-correlation is known as valid cross-correlation.

Gradient Descent

Now, to update the weights of our model, we can compute these values in closed form, however, 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.

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.

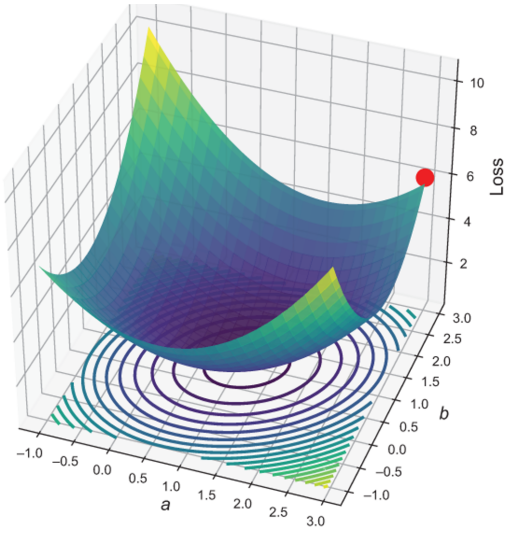

Figure 5: Plot of MSE in a single regression unit

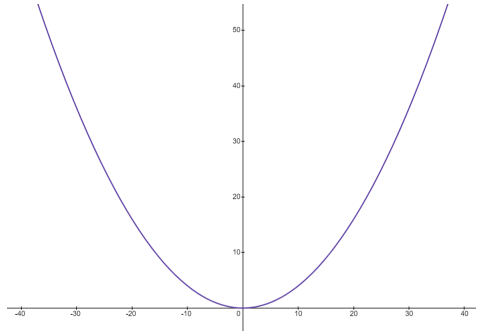

Figure 6: MSE with respect to one parameter

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

$$w_{t+1} = w_t - \alpha \frac{\partial E}{\partial w} \tag{3}$$

$$b_{t+1} = b_t - \alpha \frac{\partial E}{\partial b} \tag{4}$$

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 calculating all of the derivatives, with respect to each set of parameters, as this requires passing the output from the error function backwards through all of the layers and processes. To combat this, automatic differentiation can be used, which tracks computation dynamically at run time to compute derivatives.

1.2.1.3 Automatic Differentiation

As mentioned, automatic differentiation tracks computation dynamically and then computes derivatives using a computational graph. This allows models to have much more complex forward passes, including decision branches and loops where the length of the loop is decided at runtime.

Automatic differentiation operates by differentiating a complex function (that we don't know the derivative of), by treating it as a composition of more elementary functions (of which we do know the derivatives). Additionally, it treats each of these elementary functions as though their output is an intermediate variable

when computing the complex function. This becomes very useful when there are multiple inputs to the function, and we only want the derivative to one of these variables. Finally, this process makes use of the chain rule to compute the derivative with respect to the inputs, as will be shown later on.

Firstly, I will walkthrough an example of the process of automatic differentiation.
Say we have a function:

$$f(x) = ln\left(\frac{sin(x_1)}{cos(x_2)}\right)$$

This function has two arbitrary inputs (perhaps in the case of a neural network these could be our parameters or outputs for example). At first, this is quite a complex function to differentiate but we can simplify it using intermediate variables. For example

$$v_1 = sin(x_1) \tag{5}$$
$$v_2 = cos(x_2) \tag{6}$$
$$v_3 = \frac{v_1}{v_2} \tag{7}$$
$$v_4 = ln(v_3) \tag{8}$$
$$y = v_4 \tag{9}$$

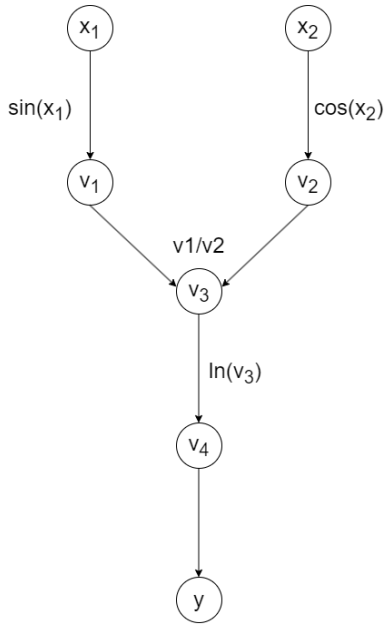

Figure 7: Example Computational Graph

Now that we have declared our intermediate variables, we can see how automatic differentiation splits up the original function into elementary ones. This then allows use to easily take derivatives. Firstly, we can draw out the computational graph for this function, using our intermediate variables, and this can be seen in Figure 7. This graph visualizes the forward pass of the function, and how the intermediate varibales compose to form our original function.

Now, when we want to take the derivative with respect to one of these inputs, we can simply reverse through the graph, taking the gradient of each intermediate variable, and using the chain rule to give us our final derivative.

This process would looks as follows:

$$\frac{\partial y}{\partial x_1} = \frac{\partial y}{\partial v_4} \cdot \frac{\partial v_4}{\partial v_3} \cdot \frac{\partial v_3}{\partial v_1} \cdot \frac{\partial v_1}{\partial x_1} \tag{10}$$

If we computed each of these derivatives:

$$\frac{\partial y}{\partial v_4} = 1 \tag{11}$$
$$\frac{\partial v_4}{\partial v_3} = \frac{1}{v_3} \tag{12}$$
$$\frac{\partial v_3}{\partial v_1} = \frac{1}{v_2} \tag{13}$$
$$\frac{\partial v_1}{\partial x_1} = cos(x_1) \tag{14}$$

And finally, evaluating equation (8), as well as substituting our intermediate variables in terms of x_1, to

obtain $\frac{\partial y}{\partial x_1}$:

$$\frac{\partial y}{\partial x_1} = \frac{cos(x_1)}{v_3 \cdot v_2} \tag{15}$$

$$= \frac{cos(x_1)}{v_1} \tag{16}$$

$$= \frac{cos(x_1)}{sin(x_1)} \tag{17}$$

This process of traversing through the graph backwards is known as reverse accumulation auto differentiation, and in practice each of the gradients would be stored numerically at computation, rather than computing a symbolic version of the final derivative and plugging the values in.

1.2.1.4 Reinforcement Learning (RL)

Reinforcement learning aims to capitalize on learning through an environment (similar to how infants learn by playing), without a teacher, to improve computer's ability at a given task. Simply put, problems in reinforcement learning involve learning how to map certain situations to certain actions to maximize a reward signal. The models are not told what actions to take and instead must explore the environment and available actions to learn what yields the most rewards. In some cases, actions may not only affect immediate rewards, but also the possibility and/or size of future rewards.

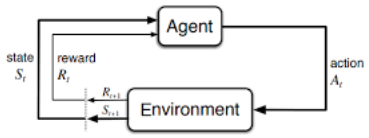

Figure 8: Diagram of Markov Decision Process

The solution to reinforcement learning problems can be modelled using a markov decision process (A diagram of which can be seen in Figure ˙). This diagram shows the process of one iteration of training. The agent will recieve a state (S_t) and then decide on an action to take (A_t). This action is sent to the environment, which process the action, evaluates the reward gained from that action (R_t), and then returns that reward and the next state (S_{t+1}).

One challenge that becomes in apparent in reinforcement learning, that is not present in other machine learning methods (supervised and unsupervised), is that of *exploration* vs *exploitation*. This problem comes from the fact that to maximize any given reward signal, an agent must take the action which gains the most reward i.e. it must *exploit* what it knows, and choose the best action. However, to have knowledge of which action is best at any given state, the agent must have *explored* many different actions - in many different states. This challenge will be addressed in more depth later on, during the design of the agent.

There are two main approaches to reinforcement learning problems, a model-free solution or model solution. Within model-free solutions there is another two categories of policy optimization and Q-learning. In policy optimization, the agent learns a policy which maps states to actions. There are two types of these policies - deterministic and stochastic. A deterministic policy maps without uncertainty i.e. the agent will take the same action given the same state. Stochastic policies on the other hand output a distribution which maps a state to the probability of each action.

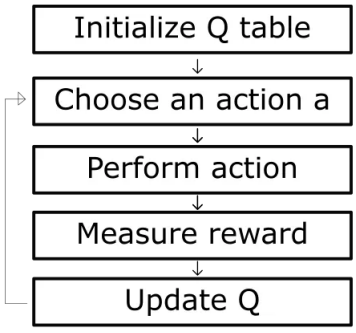

Figure 9: Representation of Q-learning process

Q-learning can be represented as a tabular learning method, and works by learning the value of a function - usually $Q(s, a)$ - which represents how successful an action was at a certain state. Each of these values is stored in the 'table' and when the model next observes a given state it consolidates the table and chooses the action which was most successful in its past experience.

For model-based solutions, the models 'know' the rules of the games and learn from planning and construct a functional representation of the environment. This is different to non-model based which aims to learn by trial-and-error and experience. AlphaZero for example would be defined as a model-

based agent. To better define this, if the agent is able to forecast the reward of an action given any state, allowing it to plan what actions to take, it is model-based.

Model-based solutions are best when trying to optimize the reward for a task - such as playing chess. Whereas model-free solutions are best for tasks such as self-driving cars (as a model-based approach may run over a pedestrian just to try and complete the journey in the shortest time).

1.2.2 Similar Systems

Chinook is a checkers playing computer program that was developed at the University of Alberta between 1989 and 2007. The program utilises an opening book, deep-search algorithms alongside a position evaluation function and a solved endgame database for positions with 8 pieces or fewer. All of Chinook's knowledge has been programmed in by humans, with no use of an artificial intelligence model. Despite this, Chinook managed to beat the draughts world champion at the time of 1995, and after this no longer competed, but instead was tasked with solving checkers. (A solved game is one where the outcome can be predicted from any position) - and this was achieved in 2007.

Figure 10: Image of Chinook playing a game of draughts - 1992

Figure 11: Chess champion Garry Kasparov loses to Chess engine - 1997

Other similar systems include chess engines - of which there is a multitude. Chess engines operate similarly to Chinook, creating trees of possible moves and using a minimax algorithm, as well as alpha-beta pruning to determine the best move. Additionally, there are some chess engines which use reinforcement learning, such as Google DeepMind's AlphaZero engine. This engine has also been trained on games such as Go and Shogi. AlphaZero is trained mainly using self-play, where the AI plays against itself and then learns from those games. In a 100 game match against StockFish (one of the most popular chess engines) AlphaZero won 28 games, lost 0 and drew the remaining 72. This shows the power of reinforcement learning as a tool for board games and its possibility to be used for draughts.

For automatic differentiation, some of the most popular systems include PyTorch's autograd, as well as autodiff by JAX. PyTorch has a very well developed library for deep-learning alongside it's autograd, as well as extensive documentation. Additionally, autograd is more similar to regular python than JAX and may be easier to use when first delving into the area. However, JAX builds on top of autograd and accelerates the linear algebra using an XLA backend. This backend optimizes operations for CPU's GPU's and TPU's as well as using a just-in-time compiler to decrease runtimes.

1.2.3 Interview of end-user(s)

1.2.3.1 Interview of 3rd-party

1.2.3.2 Interview with end-user

1.3 Modelling of the problem

1.3.1 Identification of Objects

1.3.1.1 Draughts

For draughts, we will need two classes, a checkers board/environment and a checkers game. The environment will be used for training the agent and It will require methods that execute an action in the environment, as well as functions to compute the available moves and check for a win. An example of UML for this environment can be seen below.

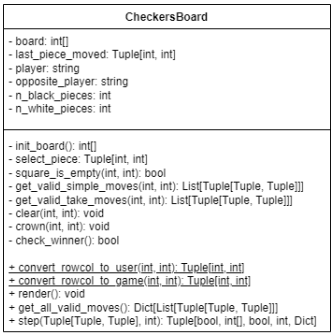

Figure 12: UML for draughts/checkers game

The checkers game varies only slightly and will inherit from the checkers board. It will however add some functionality such as a GUI and the ability to choose different models to play against (such as just the MCTS or the full agent). Additionally, it should allow two people to play each other as checkers usually would.

1.3.1.2 Automatic Differentiation Engine

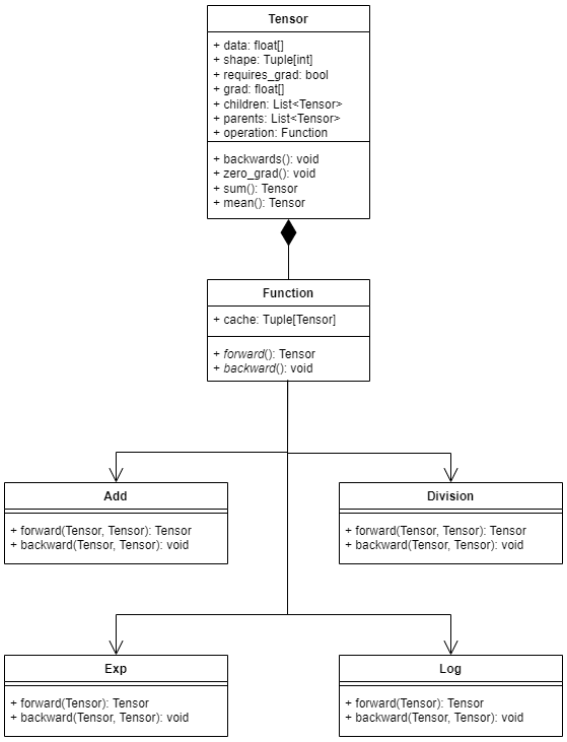

Figure 13: Example pre-design UML for Auto

The automatic differentiation engine will be based off of the 'Tensor' object. This object will be similar to a matrix but have some additonal attributes, such as its gradient and if its gradient requires computing. If the tensor does require a gradient then it will also store its parent tensors, which will be used for the backward DFS of the computational graph.

Additionally, all operators for the tensor object will have to be overridden to allow for gradient computation if the tensor requires it. The tensor class will also have functions to zero out the gradient as well as begin the backwards traversal through the computational graph. Below is an example of some possible UML for the automatic differentiation library.

Tensor operations will all have a backward and forward method - inherited from the function class. These functions will represent elementary functions (Addition, Division, Log, etc.). The operations such as sum and mean, that operate on the tensor itself will also be represented by a function object.

1.3.1.3 Neural Network Library

Their will be a large collection of objects inside of the neural network library, all based around the functionality provided by the tensor object. Firstly, the parameter class will allow for tensors of a given shape to be created and act as parameters for a network - this will also help with backpropogation. The module object will have a parameters property which yields every parameter related to that module. This module class is what all layers will inherit from. It will also have methods to zero out the gradients of all parameters.

Secondly, their will be classes for each different type of layer (dense and activations). The dense layer will have attributes which store the parameters required for computation as well as a method to complete the forward pass through the layer. Activation functions will compute the forward pass, as well as store the gradient function for any tensors which require it. The activation layers will not need any attributes as the have no parameters.

The optimizer class is the base class for any optimizers such as stochastic gradient descent or momentum-based gradient descent. This class will have attributes which store the learning rate as well as any parameters the optimizer is related to. It will also have virtual methods - which will be overriden by the specific optimizer classes - of a step (which updates all parameters), a function to zero the gradient of all parameters and finally a function to add parameters.

The final class in the library will be the Sequential class. This class will represent a model (collection of layers) and abstract some complexity of working with individual layers. The class will have a list of layers which make up the model, a loss function to use and an optimizer. It will abstract away training of the model via a train function, with default values of some hyperparameters.

1.3.1.4 Reinforcement Learning Solution

The final solution will also require a large set of classes including: a model class - to output the model policy and predicted value; a monte carlo tree searach (MCTS) class to compute a search of available actions to train the network towards - composed of a node class to make up this tree - and finally an agent class which abstracts the processes of the previous classes.

1.3.2 Identification of algorithms

1.3.2.1 Automatic Differentiation Engine

The auto-diff engine will use a recursive DFS of the computational graph to calculate gradients. It uses a DFS as any of the computational graphs that built are directed acyclical graphs, and therefore DFS provides a way to search the whole graph in O(n) time. Some pseudocode for this search can be seen below.

Algorithm 1 Backward Function of Tensor

```
function BACKWARD(Tensor self, Array grad, Tensor out)
Ensure: t.requiresGrad is True                    ▷ t is the tensor being called to start DFS
    if t.Grad is None then
        grad ← [1  1  1  ...] in shape of t.data
    t.Grad += grad
    if t.Operation is not None then
        t.Operation.backward(t.Grad, t)
```

Algorithm 2 General Backward Function of a Tensor Operation

function BACKWARD(Operation self, Array grad, Tensor t)
  savedTensors ← self.cache.0
  for tensor in savedTensors **do**
    if tensor.requiresGrad is True **then**
      newGrad ← $\frac{\partial output}{\partial tensor}$ ▷ Computed by chain rule, specific for each operation
      Remove broadcasted dims from newGrad ▷ newGrad.shape should equal tensor.shape
      tensor.backward(newGrad, t)

1.3.2.2 Neural Network Library

1.3.2.3 Reinforcement Learning Solution

1.3.3 Mathematical Formula

1.3.3.1 Automatic Differentiation

1.3.3.2 Neural Network Library

1.3.3.3 Reinforcement Learning Solution

1.4 Set of objectives

2 Research Log

2.1 Draughts Rules

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

2.2 Neural Networks

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

2.2.1 Vectorization of summations

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

2.3 Automatic Differentiation

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

2.4 Implementing Custom Classes using Numba

https://numba.pydata.org/numba-doc/latest/extending/interval-example.html Numba does not allow user-declared classes and therefore they must be explicity defined using Numba's types.

2.5 AlphaZero

https://ai.stackexchange.com/questions/13156/does-alphazero-use-q-learning https://suragnair.github.io/posts/alphazero.html Explanations of AlphaZero's loss function
https://liacs.leidenuniv.nl/~plaata1/papers/CoG2019.pdf AlphaZero policy explanation

2.6 Existing Auto differentiation systems

https://medium.com/@utsavstha/jax-vs-pytorch-a-comprehensive-comparison-for-deep-learning-10a84f934e17
PyTorch vs JAX

2.7 Reinforcement Learning

https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf What is reinforcement learning?
https://smartlabai.medium.com/reinforcement-learning-algorithms-an-intuitive-overview-904e2dff5bbc Intro to reinforcement learning
https://medium.com/the-official-integrate-ai-blog/understanding-reinforcement-learning-93d4e34e5698 Different types of reinforcement learning
https://neptune.ai/blog/model-based-and-model-free-reinforcement-learning-pytennis-case-study Model-based vs Model-free learning