Data Science 210 Final Project

Thomas Kwashnak

Fall 2021

.

Contents

1 Background

1.1 Goals

The main goal of this project is to create an algorithm that will predict whether or not a given player is considered winning in any given state of connect-4. To do this, I created a neural network that takes in a connect-4 board state (6 wide x 7 high), and outputs a single number between $[0, 1]$ that represents the predicted confidence that player 1 is winning. To train the network, I will use a stochastic gradient descent algorithm, as well as a back propagation algorithm, in order to update the weights.

1.2 Mathematics Overview

1.3 Dataset

In order to train the network to evaluate a state of a connect-4 game, I used a dataset of connect-4 games, which provided a list of connect-4 game states, and the resulting value of the game.

> John Tromp. (1995). Connect-4 `https://archive.ics.uci.edu/ml/datasets/Connect-4`. Irvine, CA: University of California, School of Information and Computer Science.

The dataset contains 67,557 entries, each one representing a state in a game of Connect-4. As decribed on the database page:

> This database contains all legal 8-ply positions in the game of connect-4 in which neither player has won yet, and in which the next move is not forced.
> x is the first player; o the second.
> The outcome class is the game theoretical value for the first player.

The data is stored in a .csv file. Each row of the dataset represents a legal position. Each row is composed of 43 comma separated variables. The first 42 variables represent the board state, and the last one represents the expected value of whether player 1 wins, loses or draws.
For the board position, there are three possible states. 'x' means that the first player has placed a piece in that square. 'o' indicates that the second player has placed their piece in that square. 'b' indicates that the square is vacant.
There are three possible "results" of the board state. "win" means that player 1 is expected to win from this game state. "lose" means that player 2 is expected to win from this game state. "draw" means that neither player is expected to win, or that the board will be completely filled before any player can get 4 in a row.

2 Design

During this section, a variety of variables and variable syntax will be used. The following is a list that explains each variable. Remember that for a given "layer", the output is considered the 0th layer, the hidden layer is considered the 1st layer, and the input is considered the 2nd layer.

y^n The output vector of the nodes in the n^{th} hidden layer.

$y^n_{i,0}$ The output value of node i in the n^{th} hidden layer.

y By default, if y is left alone, it represents the output value, known as $y^0_{0,0}$

$\hat{y}$ The expected final output as a scalar value

W^n The weight matrix for a given layer n, as it applies to the values Y^{n+1} (the outputs of the previous layer)

$w^n_{i,j}$ The weight value for the value passed from node j of layer $n+1$ as it is passed to node i of layer n

η The step size of the gradient descent as it nudges the weights.

I've sectioned off the design of the algorithms into the independant algorithms, ordering them to try and explain the progression of deriving the algorithms. This will take some experimentation, but the first idea is to

2.1 Sigmoid Activation Function

In neural networks, an activation function is typically used to to condense the output of a node down to a [0-1] scale. In this lab, I used a sigmoid function. Below is the sigmoid function and it's derivative.

$$\sigma(n) = \frac{1}{1+e^{-n}}$$

$$\sigma'(n) = \sigma(n) \cdot (1 - \sigma(n))$$

I used the following algorithm so that if I passed in a matrix, it would just apply the sigmoid function to each value in the matrix.

Algorithm 1: $\sigma(n)$ function for both constants and matrices

Input: A - which can either be a constant, or an array, with the first dimension being length n
Output: $\sigma(A)$ - The result of putting A through the $\sigma()$ function

if *A is a Matrix (or array)* **then**
    $B \leftarrow$new Matrix of 0s in the same shape as A
    for $i = 0, 1, 2...n$ **do**
        `// This works recursively until it reaches scalar values`
        $B[i] \leftarrow \sigma(A[i])$
    end
    return B
else
    return $1/(1+e^{-A})$
end

2.2 Feed-Forward Algorithm

The Feed-Forward algorithm is the algorithm used to "run" a neural network. This is one of the fundamental steps in how a neural network works. This step is used whenever you want to find the output that a neural network will calculate from a given set of inputs. Since the neural network is siplit into 3 layers, the transition of the values from one layer to the next can be described as:

$$\vec{y^n} = \sigma(W_n \cdot \vec{y^{n+1}})$$

If we substitute in all of our layers, we can get the following equation as the result of the entire network.

$$y = y^0_{0,0} = \sigma(W_0 \cdot \sigma(W_1 \cdot \vec{y^2}))_{0,0}$$

2.3 Loss Function

In order to use gradient descent, we will need to create a function that measures the correctness of our network. This is known as the "Loss Function". The premise of the loss function is that the higher it is, the worse the network is. Therefore, in order to make our network more accurate, we need to minimize this value.

$$L = (y^0_0 - \hat{y})^2 = (y - \hat{y})^2$$

2.4 0th Layer Weight Derivative

In back propagation, we need to find the derivative of each of the weights in terms of the loss function. First, we need to find the derivative of the loss function with respect to the weights that connect the hidden layer with the output layer.
Given the following equations:

$$y^1_{i,0} = \sigma(\sum_h w^1_{i,h} \cdot y^2_{h,0})$$

$$y^0_{0,0} = \sigma(\sum_h w^0_{0,h} \cdot y^1_{h,0})$$

We can calculate the derivative as follows:

$$\frac{\delta L}{\delta w^0_{0,i}} = \frac{\delta L}{\delta y^0_{0,0}} \cdot \frac{\delta y^0_{0,0}}{\delta \sum_h w^0_{0,h} \cdot y^1_{h,0}} \cdot \frac{\delta \sum_h w^0_{0,h} \cdot y^1_{h,0}}{\delta w^0_{0,i}}$$

$$\frac{\delta L}{\delta w^0_{0,i}} = 2 \cdot (y^0_{0,0} - \hat{y}) \cdot \sigma'(\sum_h w^0_{0,h} \cdot y^1_{h,0}) \cdot y^1_{i,0}$$

$$\frac{\delta L}{\delta w^0_{0,i}} = 2 \cdot (y^0_{0,0} - \hat{y}) \cdot \sigma(\sum_h w^0_{0,h} \cdot y^1_{h,0}) \cdot (1 - \sigma(\sum_h w^0_{0,h} \cdot y^1_{h,0})) \cdot y^1_{i,0}$$

$$\frac{\delta L}{\delta w^0_{0,i}} = 2 \cdot (y^0_{0,0} - \hat{y}) \cdot y^0_{0,0} \cdot (1 - y^0_{0,0}) \cdot y^1_{i,0}$$

2.5 1st Layer Weight Derivative

Next, we need to find the derivative of each of the weights that connect the input layer to the hidden layer.
Given the following equations:

$$y^1_{i,0} = \sigma(\sum_h w^1_{i,h} \cdot y^2_{h,0})$$

$$y^0_{0,0} = \sigma(\sum_h w^0_{0,h} \cdot y^1_{h,0})$$

We can calculate the derivative as the following:

$$\frac{\delta L}{\delta w^1_{i,j}} = \frac{\delta L}{\delta y^0_{0,0}} \cdot \frac{\delta y^0_{0,0}}{\delta \sum_h w^0_{0,h} \cdot y^1_{h,0}} \cdot \frac{\delta \sum_h w^0_{0,h} \cdot y^1_{h,0}}{\delta y^1_{i,0}} \cdot \frac{\delta y^1_{i,0}}{\delta \sum_h w^1_{i,h} \cdot y^2_h} \cdot \frac{\delta \sum_h w^1_{i,h} \cdot y^2_{h,0}}{\delta w^1_{i,j}}$$

$$\frac{\delta L}{\delta w^1_{i,j}} = 2 \cdot (y^0_{0,0} - \hat{y}) \cdot \sigma'(\sum_h w^0_{0,h} \cdot y^1_{h,0}) \cdot w^0_{0,i} \cdot \sigma'(\sum_h w^1_{i,h} \cdot y^2_{h,0}) \cdot y^2_{j,0}$$

$$\frac{\delta L}{\delta w^1_{i,j}} = 2 \cdot (y^0_{0,0} - \hat{y}) \cdot \sigma(\sum_h w^0_{0,h} \cdot y^1_{h,0}) \cdot (1 - \sigma(\sum_h w^0_{0,h} \cdot y^1_{h,0})) \cdot w^0_{0,i} \cdot \sigma(\sum_h w^1_{i,h} \cdot y^2_{h,0}) \cdot (1 - \sigma(\sum_h w^1_{i,h} \cdot y^2_{h,0})) \cdot y^2_{j,0}$$

$$\frac{\delta L}{\delta w^1_{i,j}} = 2 \cdot (y^0_{0,0} - \hat{y}) \cdot y^0_{0,0} \cdot (1 - y^0_{0,0}) \cdot w^0_{0,i} \cdot y^1_{i,0} \cdot (1 - y^1_{i,0}) \cdot y^2_{j,0}$$

We can then reorder it as..

$$\frac{\delta L}{\delta w^1_{i,j}} = 2 \cdot (y^0_{0,0} - \hat{y}) \cdot y^0_{0,0} \cdot (1 - y^0_{0,0}) \cdot y^1_{i,0} \cdot w^0_{0,i} \cdot (1 - y^1_{i,0}) \cdot y^2_{j,0}$$

And substitute in the 0^{th} layer derivative!

$$\frac{\delta L}{\delta w^1_{i,j}} = \frac{\delta L}{\delta w^0_{0,i}} \cdot w^0_{0,i} \cdot (1 - y^1_{i,0}) \cdot y^2_{j,0}$$

2.6 Backwards Propagation

Backwards propagation is calculating the "gradient" of weights for a given input. I say "gradient" because in reality, the outputs of this algorithm will be two matricies, which each contain the derivative for each variable. We will be able to use these nudges in order to reduce the overall loss function. I also included an error calculations, such that we can also keep track of how accurate the network is at each batch of inputs.

Algorithm 2: Back Propagation for a 2-layer neural network

Input: Y^2 - The input vector (matrix), dimensions m x 1.
W^1 - The weight matrix, dimensions h x m, that represents the weights used as variables go from $Y^2 \rightarrow Y^0$
W^0 - The weight matrix, dimensions 1 x h, that represents the weights used as variables go from $Y^1 \rightarrow Y^0$
$\hat{y}$ - The expected result from Forward-Feeding

Output: δW^1 - A matrix of the derivatives of the weights in W^1, as a n x m matrix.
δW^0 - A matrix of the derivatives of the weights in W^0, as a 1 x h matrix.
Err - The absolute error of the network for this given input

$\delta W^1 \leftarrow$ new matrix of dimensions n x m
$\delta W^0 \leftarrow$ new matrix of dimensions 1 x h
$Y^1 \leftarrow \sigma(W^1 \cdot Y^2)$
$Y^0 \leftarrow \sigma(W^0 \cdot Y^1)$
Err $\leftarrow |Y^0_{0,0} - \hat{y}|$
for $i \leftarrow (0, 1, \ldots, h-1)$ **do**
    $\delta W^0_{0,i} \leftarrow 2 \cdot (Y^0_{0,0} - \hat{y}) \cdot Y^0_{0,0} \cdot (1 - Y^0_{0,0}) \cdot Y^1_{i,0}$
    for $j \leftarrow (0, 1, \ldots, m-1)$ **do**
        $\delta W^1_{i,j} \cdot W^0_{0,i} \cdot (1 - y^1_{i,0}) \cdot y^2_{j,0}$
    end
end
return $\delta W^1, \delta W^0$, *Err*

2.7 Stochastic Gradient Descent

For a good portion while working on this project, this was the part I was the most confused on. Stochastic steepest descent, at least my interpretation of it, is taking the accumulation of all of the weight changes taken from the back propagation algorithm, and averaging them before applying them to the network. The general formula is described as follows:

$$W^i = W^i - \frac{\eta}{n} \cdot \sum_{i}^{n} \text{Back-Propagation}(\ldots)_{W^i}$$

To update each of the weights, we take the average changes in the back propagation, and multiply it by a scalar coefficient η, which represents the step size of the learning. The smaller η is, the less the network will learn. For a single cycle in gradient descent, we will use the following algorithm to update the weights accordingly:

Algorithm 3: Stochastic Gradient Descent Iteration

Input: input - Length-n list of input vectors, as matrices. Each input vector is of size m x 1
output - Length-n list of all expected values, as scalars. Each output corresponds to the input at that same index
W^1 - The weight matrix, dimensions h x m, that represents the weights used as variables go from $Y^2 \to Y^0$
W^0 - The weight matrix, dimensions 1 x h, that represents the weights used as variables go from $Y^1 \to Y^0$
η - The step coefficient to nudge the weights by.

Output: W^1 - The updated weight matrix, dimensions h x m, where values have been modified by their average derivative across all inputs/outputs, scaled by η
W^0 - The updated weight matrix, dimensions 1 x h, where values have been modified by their average derivative across all inputs/outputs, scaled by η
Err - The average absolute error of the network on each input.

```
Err Total ← 0
ΔW¹ ← matrix of zeros with dimensions h x m
ΔW⁰ ← matrix of zeros with dimensions 1 x h
for i ← 0, 1, ..., n − 1 do
    δW¹, δW⁰, δErr ← Back Propagation on input[i] and output[i]
    ΔW¹ ← ΔW¹ + δW¹
    ΔW⁰ ← ΔW⁰ + δW⁰
    Err Total ← Err Total + Err
end
ΔW¹ ← η/n · ΔW¹
ΔW⁰ ← η/n · ΔW⁰
Err Total ← Err Total/n
return ΔW¹, ΔW⁰, Err Total
```

2.8 Training the Network

Now, in order to train the network, I need to send in batches of data for it to train on, and each batch of data I need to reduce the step function so the network can fine-tune its training. I plan to do this by using a random sample approach. I'll have a set of modifyable parameters, namely the batch size and the iteration count. These will allow me to change how many data points are put in to the network for each iteration, and how many iterations I will have. For step size, I will pass in a function, such that I can experiment with different step size functions. During implementation, I may also allow the ability to allow for keyboard interruption instead, so I can leave it running for however long I want, and come back to a trained network.

The algorithm to perform this training is described as follows:

Algorithm 4: Neural Network Training Algorithm

Input: input - Length-n list of input vectors, as matrices. Each input vector is of size m x 1
output - Length-n list of all expected values, as scalars. Each output corresponds to the input at that same index
W^1 - The weight matrix, dimensions h x m, that represents the weights used as variables go from $Y^2 \rightarrow Y^0$
W^0 - The weight matrix, dimensions h x m, that represents the weights used as variables go from $Y^2 \rightarrow Y^0$
$f(i, I)$ - Step size function, which when passed in a current iteration i, and total iteration count I, returns the step size for that iteration
I - The total number of iterations to run
c - The total number of inputs to feed into each iteration

Output: W^1 - The updated weight matrix, dimensions h x m, trained on the given inputs and outputs W^0 - The updated weight matrix, dimensions 1 x h, trained on the given inputs and outputs

for $i \leftarrow 0, 1, \ldots, n-1$ **do**
    $\eta \leftarrow f(i, I)$
    indexes $\leftarrow$ list of uniform-random indexes within the range $[0, n-1]$
    input-iteration $\leftarrow$ list of inputs at each corresponding index from indexes
    output-iteration $\leftarrow$ list of outputs at each corresponding index from indexes
    $W^1, W^0, Err \leftarrow$ Stochastic Gradient Descent of weights and step size for input-iteration and output-iteration
    Log the Err of the training
end
return W^1, W^0

3 Implementation

3.1 Data Importing

Since the data was given in a string csv format, an additional python script was created to import the data into two matricies, one contianing the inputs and one contianing the expected outputs.

4 Validation

5 Reflection