ELL784: Introduction to Machine Learning

Shashank Shaurya Dubey (2022SMZ8158)

Assignment I

Question 1

Generate all possible boolean functions for n inputs where $n \in \{1, 2, 3, 4\}$.

Proof. A Boolean function with n inputs maps each possible combination of n binary values (0 or 1) to a single binary output (0 or 1). For n inputs, there are 2^n possible input combinations. Since each combination can be assigned to 0 or 1, there are $2^{(2^n)}$ possible Boolean functions for the input n.

Case 1: $n = 1$

- Number of input combinations: $2^1 = 2$
- Input combinations: (0) and (1)
- Number of Boolean functions: $2^{(2^1)} = 2^2 = 4$

The four Boolean functions are:

$$\begin{gathered} 1.\ (0 \to 0, 1 \to 0) \\ 2.\ (0 \to 0, 1 \to 1) \\ 3.\ (0 \to 1, 1 \to 0) \\ 4.\ (0 \to 1, 1 \to 1) \end{gathered}$$

Case 2: $n = 2$

- Number of input combinations: $2^2 = 4$
- Input combinations: $(00), (01), (10), (11)$
- Number of Boolean functions: $2^{(2^2)} = 2^4 = 16$

Some of the Boolean functions are:

$$\begin{gathered} 1.\ (00 \to 0, 01 \to 0, 10 \to 0, 11 \to 0) \\ 2.\ (00 \to 0, 01 \to 0, 10 \to 0, 11 \to 1) \\ 3.\ (00 \to 1, 01 \to 0, 10 \to 0, 11 \to 1) \\ \text{...total functions} = 16 \end{gathered}$$

Case 3: $n = 3$

- Number of input combinations: $2^3 = 8$
- Input combinations: $(000), (001), (010), (011), (100), (101), (110), (111)$
- Number of Boolean functions: $2^{(2^3)} = 2^8 = 256$

Some of the Boolean functions are:

$$\begin{aligned}
&1.\ (000 \to 0, 001 \to 0, 010 \to 0, \ldots, 111 \to 0)\\
&2.\ (000 \to 0, 001 \to 0, 010 \to 0, \ldots, 111 \to 1)\\
&3.\ (000 \to 1, 001 \to 0, 010 \to 0, \ldots, 111 \to 1)\\
&\quad \text{...total functions} = 256
\end{aligned}$$

Case 4: $n = 4$

- Number of input combinations: $2^4 = 16$
- Input combinations: $(0000), (0001), (0010), (0011), (0100), (0101), (0110), (0111), (1000), (1001), (1010), (1011), (1100), (1101), (1110), (1111)$
- Number of Boolean functions: $2^{(2^4)} = 2^{16} = 65536$

Due to a large number of functions (65,536), listing them is not possible here.

Summary

For n inputs, the number of possible Boolean functions is given by $2^{(2^n)}$. The results for different values of n are summarized below:

- For $n = 1$: 4 Boolean functions.
- For $n = 2$: 16 Boolean functions.
- For $n = 3$: 256 Boolean functions.
- For $n = 4$: 65,536 Boolean functions.

$\square$

Question 2

How many of these functions are learnable by an L-layer ANN for $L \in \{1, 2, 3, 4\}$?

Proof. The learnability of Boolean functions by a neural network depends on the network's capacity, which is influenced by the number of layers (L) and the number of neurons per layer.

Case 1: L = 1

For a single layer ANN, all those boolean functions are learnable which are linearly separable. Therefore, given a dataset of input vectors $\mathbf{x}_i \in \mathbb{R}^n$ and corresponding binary output labels $y_i \in \{0, 1\}$, we determine which functions are linearly separable for inputs $n \in \{1,2,3,4\}$.

We set parameters
* Set weights randomly: $\mathbf{w} \leftarrow \mathbf{0}$ * Set bias: $b \leftarrow 0$
* Set learning rate: η * Set maximum number of iterations: T

Model training
For $t = 1$ to T:
For each training example $(\mathbf{x}_i, y_i)$:
* Calculate the net input: $z_i = \mathbf{w}^T\mathbf{x}_i + b$
* Apply the activation function (step function): $\hat{y}_i = \text{step}(z_i) = \begin{cases} 1, & \text{if } z_i > 0 \\ 0, & \text{otherwise} \end{cases}$
* Calculate the error: $e_i = y_i - \hat{y}_i$
* Update weights and bias: a) $\mathbf{w} \leftarrow \mathbf{w} + \eta e_i \mathbf{x}_i$ b) $b \leftarrow b + \eta e_i$

Checking convergence If the number of errors in an epoch is zero, the data is linearly separable; otherwise, it is not.

For n = 1

- Number of Boolean functions: $2^{(2^1)} = 2^2 = 4$
- Number of linearly separable boolean functions = 4

x_1	f_1	f_2	f_3	f_4
0	0	0	1	1
1	0	1	0	1

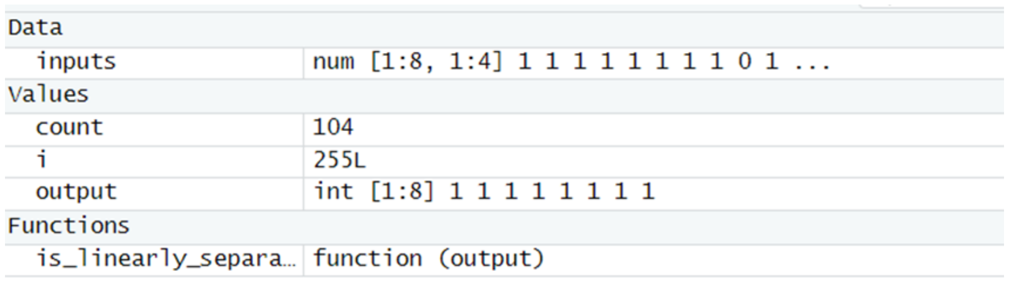
For n = 2

- Number of Boolean functions: $2^{(2^2)} = 2^4 = 16$
- Out of these sixteen functions two functions are not linearly separable - XOR and its complement. Therefore, the number of linearly separable boolean functions = 14

x_1	x_2	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	f_{16}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

For n = 3

- Number of Boolean functions: $2^{(2^3)} = 2^8 = 256$
- Out of the 256 possible boolean functions, the number of linearly separable boolean functions = 104

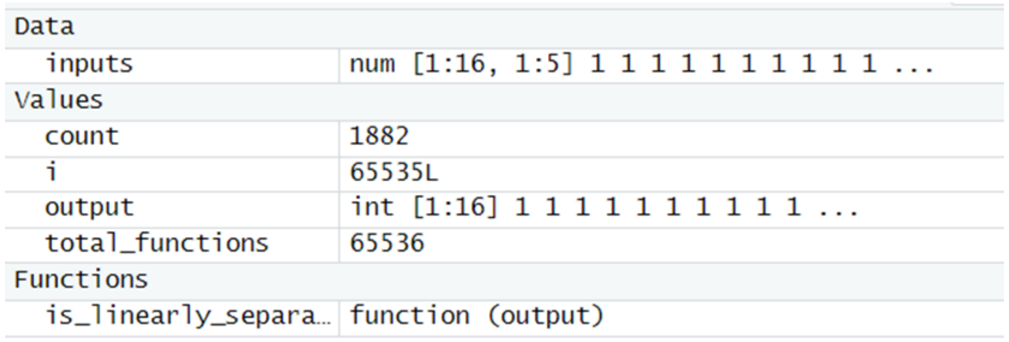


```
Data
  inputs                num [1:8, 1:4] 1 1 1 1 1 1 1 1 0 1 ...
Values
  count                 104
  i                     255L
  output                int [1:8] 1 1 1 1 1 1 1 1
Functions
  is_linearly_separa…   function (output)
```

Fig 1. Linearly separable functions for $n = 3$; Source: R code

For n = 4

- Number of Boolean functions: $2^{(2^4)} = 2^{16} = 65536$
- Out of the 65536 possible boolean functions, the number of linearly separable boolean functions = 1882



```
Data
  inputs                num [1:16, 1:5] 1 1 1 1 1 1 1 1 1 1 ...
Values
  count                 1882
  i                     65535L
  output                int [1:16] 1 1 1 1 1 1 1 1 1 1 ...
  total_functions       65536
Functions
  is_linearly_separa…   function (output)
```

Fig 2. Linearly separable functions for $n = 4$; Source: R code

Case 2: L = 2

The Universal Approximation Theorem (UAT) states that in a feedforward neural network with a single hidden layer containing a finite number of neurons can approximate any continuous function on a compact subset of [R^n].

For n = 1

- Number of Boolean functions: $2^{(2^1)} = 2^2 = 4$
- Number of learnable functions = 4

For n = 2

- Number of Boolean functions: $2^{(2^2)} = 2^4 = 16$
- Number of learnable functions = 16

For n = 3

- Number of Boolean functions: $2^{(2^3)} = 2^8 = 256$
- Number of learnable functions = 256

For n = 4

- Number of Boolean functions: $2^{(2^4)} = 2^16 = 65336$
- Number of learnable functions = 65336

Case 3: L = 3

For n = 1

- Number of Boolean functions: $2^{(2^1)} = 2^2 = 4$
- Number of learnable functions = 4

For n = 2

- Number of Boolean functions: $2^{(2^2)} = 2^4 = 16$
- Number of learnable functions = 16

For n = 3

- Number of Boolean functions: $2^{(2^3)} = 2^8 = 256$
- Number of learnable functions = 256

For n = 4

- Number of Boolean functions: $2^{(2^4)} = 2^16 = 65336$
- Number of learnable functions = 65336

Case 4: L = 4

For n = 1

- Number of Boolean functions: $2^{(2^1)} = 2^2 = 4$
- Number of learnable functions = 4

For n = 2

- Number of Boolean functions: $2^{(2^2)} = 2^4 = 16$
- Number of learnable functions = 16

For n = 3

- Number of Boolean functions: $2^{(2^3)} = 2^8 = 256$
- Number of learnable functions = 256

For n = 4

- Number of Boolean functions: $2^{(2^4)} = 2^16 = 65336$
- Number of learnable functions = 65336

Hence, for L = 1 the boolean functions that are learnable for n $\in 1, 2, 3, 4$ would vary since it can only classify functions that are linearly separable. However, for L $\in 2, 3, 4$ all boolean functions for n $\in 1, 2, 3, 4$ are learnable since they can even classify functions that are not linearly separable such as XOR and $XNOR$. $\square$

Question 3

Implement the backpropagation algorithm for an L-layer ANN.

Proof. Backpropagation is an algorithm to train artificial neural networks. It's a gradient descent method that calculates the gradient of the error function to the network weights.

Steps to Implement Backpropagation for an L-Layer ANN

1. Network Architecture

Assume an L-layer network with n_0 input neurons, n_L output neurons, and hidden layers $l_1, l_2, \ldots, l_{L-1}$. Each layer l has weights $W^{[l]}$ and biases $b^{[l]}$.

2. Forward Pass

The forward pass in an artificial neural network (ANN) is the process of propagating input data through the network, layer by layer, to generate the output.

Input Data: Let the input data be represented by a vector $\mathbf{x} \in \mathbb{R}^{n_0}$, where n_0 is the number of input features. This is the input to the first layer (or the zeroth layer, denoted as $l = 0$) of the neural network.

Layers of ANN: The neural network consists of L layers, where each layer l contains n_l neurons. The input layer is considered as layer $l = 0$, and the output layer is considered as layer $l = L$.

Weights and Biases: Each layer l (except the input layer) has an associated weight matrix $\mathbf{W}^{[l]}$ and a bias vector $\mathbf{b}^{[l]}$:

- $\mathbf{W}^{[l]} \in \mathbb{R}^{n_l \times n_{l-1}}$: This weight matrix connects the neurons from layer $l-1$ to the neurons in layer l.
- $\mathbf{b}^{[l]} \in \mathbb{R}^{n_l}$: This bias vector is added to the weighted sum of inputs for each neuron in layer l.

Pre-Activation Calculation (Linear Transformation): For each layer l (starting from $l = 1$ to $l = L$), the pre-activation value (also called the "logit") is calculated as:

$$\mathbf{z}^{[l]} = \mathbf{W}^{[l]} \cdot \mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}$$

- $\mathbf{z}^{[l]} \in \mathbb{R}^{n_l}$ is the vector of pre-activation values for the lth layer.
- $\mathbf{a}^{[l-1]} \in \mathbb{R}^{n_{l-1}}$ is the activation from the previous layer.
- The operation $\mathbf{W}^{[l]} \cdot \mathbf{a}^{[l-1]}$ is a matrix-vector multiplication, producing a vector of size n_l.

Activation Function
After calculating the preactivation value $\mathbf{z}^{[l]}$, an activation function $f(\cdot)$ is applied to introduce nonlinearity:

$$\mathbf{a}^{[l]} = f(\mathbf{z}^{[l]})$$

- $\mathbf{a}^{[l]} \in \mathbb{R}^{n_l}$ is the vector of activations for the lth layer.
- The activation function $f(\cdot)$ is typically a non-linear function such as the sigmoid function, ReLU (Rectified Linear Unit), or tanh.

Output: The output of the last layer $\mathbf{a}^{[L]}$ is the final output of the neural network. This output could be a probability distribution (in the case of classification) or a continuous value (in the case of regression).

3. Compute Loss

Loss Function: We take the Mean Squared Loss Function (MSE)

$$\text{MSE} = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{z}^{(i)} - z^{(i)}\right)^2$$

Where:

- m is the number of examples in the dataset.
- $\hat{z}^{(i)}$ is the predicted value for the ith example.
- $z^{(i)}$ is the actual target value for the ith example.
- $\hat{z}^{(i)} - z^{(i)}$ is the error (difference) between the predicted value and the actual value for the ith example.
- The sum is taken over all examples in the dataset, and the average of these squared differences is computed by dividing by m.

4. Backward Pass

Compute Gradients for the Output Layer (Layer L): Start by calculating the error in the output layer L

$$\delta^{[L]} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{[L]}} \circ f'(\mathbf{z}^{[L]})$$

- $\delta^{[L]}$ is the gradient of the loss with respect to the pre-activation values $\mathbf{z}^{[L]}$.
- $\frac{\partial \mathcal{L}}{\partial \mathbf{a}^{[L]}}$ is the gradient of the loss with respect to the output $\mathbf{a}^{[L]}$.
- $f'(\mathbf{z}^{[L]})$ is the derivative of the activation function applied to $\mathbf{z}^{[L]}$.

- $\circ$ denotes element-wise multiplication (Hadamard product).

Backpropagate the Error to the Previous Layers (Layer l from $L-1$ to 1) For each layer l, the error is propagated backward:

$$\delta^{[l]} = \left(\mathbf{W}^{[l+1]}\right)^\top \delta^{[l+1]} \circ f'(\mathbf{z}^{[l]})$$

- $\delta^{[l]}$ is the gradient of the loss with respect to the pre-activation values $\mathbf{z}^{[l]}$.
- $\left(\mathbf{W}^{[l+1]}\right)^\top$ is the transpose of the weight matrix from layer $l+1$.

Compute Gradients for Weights and Biases: The gradient of the loss with respect to the weights $\mathbf{W}^{[l]}$ and biases $\mathbf{b}^{[l]}$ is computed as:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[l]}} = \delta^{[l]} \cdot \left(\mathbf{a}^{[l-1]}\right)^\top$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{[l]}} = \delta^{[l]}$$

- $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[l]}}$ is the gradient of the loss for the weights in layer l.
- $\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{[l]}}$ is the gradient of the loss for the biases in layer l.
- Here, $\mathbf{a}^{[l-1]}$ is the activation from the previous layer $l-1$.

5. Update Weights and biases

The weights and biases are updated during each iteration of the training process using the Gradient Descent optimization algorithm. The goal is to minimize the loss function L, which measures the difference between the predicted output and the actual target.
Updating Weights and Biases: The update rule for the weights and biases at layer l using gradient descent is given by:

$$\mathbf{W}^{[l]} := \mathbf{W}^{[l]} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[l]}}$$

$$\mathbf{b}^{[l]} := \mathbf{b}^{[l]} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{[l]}}$$

where:

- $\mathbf{W}^{[l]}$ are the weights at layer l.
- $\mathbf{b}^{[l]}$ are the biases at layer l.
- η is the learning rate.
- $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[l]}}$ is the gradient of the loss function with respect to the weights at layer l.

- $\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{[l]}}$ is the gradient of the loss function with respect to the biases at layer l.

But the gradients have already been computed as:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[l]}} = \delta^{[l]} \cdot \left(\mathbf{a}^{[l-1]}\right)^{\top}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{[l]}} = \delta^{[l]}$$

Thus, the update rules for weights and biases become:

$$\mathbf{W}^{[l]} := \mathbf{W}^{[l]} - \eta \left(\delta^{[l]} \cdot \left(\mathbf{a}^{[l-1]}\right)^{\top}\right)$$

$$\mathbf{b}^{[l]} := \mathbf{b}^{[l]} - \eta\delta^{[l]}$$

□

Question 4

Train your ANN for classification of handwritten images of the MNIST dataset (0-9). Choose the number of layers and neurons accordingly.

Proof. We use the following logic to train our ANN to classify the handwritten images of the MNIST dataset.

1. Import Libraries

- *numpy*(*np*): Provides support for large, multi-dimensional arrays and matrices, along with high-level mathematical functions to operate on these arrays
- *keras.datasets.mnist* : Contains the MNIST dataset
- *keras.utils.to_categorical* : Converts class vectors (integers) to binary class matrices, often used as input to neural networks for classification tasks.

2. Load the MNIST dataset

We load the MNIST dataset into four NumPy arrays:

- X_train: Training images
- y_train: Corresponding training labels
- X_test: Test images
- y_test: Corresponding test labels

3. Normalize pixel values

Normalize pixel values from 0-255 to 0-1 for better numerical stability. Pixel values in the range 0-255 can lead to numerical instability during gradient descent updates in neural networks due to large differences in magnitude. Normalizing to 0-1 brings all pixel values to a similar scale, improving convergence speed and preventing exploding/vanishing gradients.

4. Reshape data

Flattens the 28x28 image matrices into 784-dimensional vectors for input to the neural network. Flattening reshapes the image matrix into a single long vector, preserving pixel information while making it compatible with the network's architecture.

5. Convert class vectors to categorical matrices

Flattens the 28x28 image matrices into 784-dimensional vectors for input to the neural network. One-hot encoding transforms each digit into a 10-dimensional vector with a single '1' at the corresponding index and '0' elsewhere, representing each class uniquely.

6. Testing model parameters and hyper-parameters

Case 1: Hyperparameter testing by altering hidden layers ($L = 1, 2, 3$) and layer sizes ($N = 64, 256, 512$)

1.1 Constant Layer ($L = 1$)**, Different Layer Sizes** ($N = 64, 256, 512$)
On decreasing the layer size or the number of neurons in a layer the test accuracy consistently dropped from 0.9800 for $N = 512$ to 0.9793 for $N = 256$ to 0.9745 for $N = 64$. The result supports the hypothesis of a correlation between increased layer size and enhanced test accuracy.

1.2 Different Layers ($L = 1, 2, 3$)**, Constant Layer Size** ($N = 64$)
On decreasing the number of layers while keeping the layer size constant at $N = 64$ in all layers, the test accuracy marginally increases from 0.9745 for $L = 1$ to 0.9761 for $L = 2$ but drops substantially to 0.9708 for $L = 3$.

Possible explanation for 1.2
a) Vanishing gradients: Deeper layer networks are more prone to this phenomenon which hinders the learning process.
b) Over-fitting: While a single layer might be under-fitting, leading to lower accuracy, a deeper network $L = 3$ might be over-fitting, capturing noise in the training data and leading to a drop in test accuracy.
c) Model complexity: More complex models need more data to train effectively and since dataset size is constant this is leading to a drop in test accuracy for $L = 3$ layer model.

Case 2: Parameter testing by altering epochs $(10, 20, 40)$ and learning rate ($\eta = 0.001, 0.01, 1$)
2.1 Constant learning rate ($\eta = 0.001$)**, Different epochs** $(10, 20, 40)$
On increasing the epoch, or the number of iterations, the test accuracy first dropped from 0.9761 for $epoch = 10$ to 0.9745 and then increased for $epoch = 20$ to 0.9747 for $epoch = 40$.
Possible explanation for 2.1
a) Initial drop: The optimization process might be stuck in local minima, and increasing the number of epochs can help the model escape this sub-optimal solution.
b) Subsequent increase: With more epochs, the model has more opportunities to converge to a better solution. This can lead to improved test accuracy.

2.2 Different learning rate ($\eta = 0.001, 0.01, 1$), **Constant epoch** (10)
On increasing the learning rate η, the step size at each iteration while moving toward a minimum of a loss function, the test accuracy dropped from 0.9761 for $\eta = 0.001$ to 0.9653 for $\eta = 0.01$. Test accuracy nosedived to 0.4620 for $\eta = 0.1$ and further to 0.0891 for $\eta = 1.0$.
Possible explanation for 2.2
a) Unstable optimization: Large steps taken during gradient descent can cause the model to overshoot the optimal solution, resulting in poor performance.
b) Affects convergence: High learning rate can prevent the model from converging.

$\square$

**