Neural Networks

Mustafa Khan

2021

1 Introduction

Acknowledgement: Aditya, a friend of mine, was kind enough to walk me through his writeup on the mathematics behind NNs and explain key concepts such as backpropagation. This paper is based on the conversation I had with him as well as his writeup.

Idea: Neural networks are a means of doing machine learning, in which a computer learns to perform a task by analyzing training examples. Through this process, a neural network is able to find patterns in data that consistently correlates with particular labels.

- Let's consider an arbitrary neural network and attempt to build up the components of it mathematically! We define inputs to our model as an $(n_{in}, 1)$ matrix, where n_{in} is the number of input nodes in our neural network.

$$\vec{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_{n_{in}} \end{bmatrix} \tag{1}$$

  We define outputs to our model as an $(n_{out}, 1)$ matrix, where n_{out} is the number of nodes in the final layer of our neural network.

$$\vec{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_{n_{out}} \end{bmatrix} \tag{2}$$

- Notation Note: Going forward, j refers to an arbitrary neuron in the l^{th} layer, whilst k refers to an arbitrary neuron in the $l-1^{th}$ layer. Additionally, we denote m as the final neuron in the l^{th} layer and n as the final neuron in the $l-1^{th}$ layer.

- We define the weight between two nodes in layer l and layer $l-1$ using the following notation:

$$w^l_{jk} \tag{3}$$

  This is the weight from the k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron on the l^{th} layer. Using this, we end up getting a weight matrix for every layer that looks like the following:

$$\boldsymbol{w^l} = \begin{bmatrix} w_{1,1} & \dots & w_{1,n} \\ \vdots & \ddots & \vdots \\ w_{m,1} & \dots & w_{m,n} \end{bmatrix} \tag{4}$$

- We define the bias for a neuron j in layer l as:

$$b^l_j \tag{5}$$

  Hence, we define our bias vector for layer l as:

$$\boldsymbol{b^l} = \begin{bmatrix} b^l_1 \\ \vdots \\ b^l_m \end{bmatrix} \tag{6}$$

- We define the activation of the j^{th} neuron in the l^{th} layer as:

$$a_j^l = \sigma(\sum_k w_{jk}^l a_k^{l-1} + b_j^l) \tag{7}$$

Hence, in the same manner as the bias, we can define the activations matrix for layer l as:

$$\boldsymbol{a^l} = \begin{bmatrix} a_1^l \\ \vdots \\ a_m^l \end{bmatrix} \tag{8}$$

- Next, in order to implement back-propagation, we need to define an intermediate quantity z_j^l, as the weighted input to neuron j in layer l:

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l \tag{9}$$

We can use this to define an matrix z_l, which consists of the weighted inputs to the neurons in layer l as:

$$\boldsymbol{z^l} = \begin{bmatrix} z_1^l \\ \vdots \\ z_j^l \end{bmatrix} \tag{10}$$

- Another way of expressing this is:

$$\boldsymbol{z^l} = \boldsymbol{w^l a^{l-1}} + \boldsymbol{b^l} \tag{11}$$

- Lastly, we define our cost function as a regular Mean Squared Error (MSE) cost:

$$C = \frac{1}{2N} \sum_x \left\| \boldsymbol{a^L}(\boldsymbol{x}) - \boldsymbol{y}(\boldsymbol{x}) \right\|^2 \tag{12}$$

Where $\boldsymbol{y}(\boldsymbol{x})$ is the label for a training example $\boldsymbol{x}$ and $\boldsymbol{a^L}(\boldsymbol{x})$ is the activations of the final layer when $\boldsymbol{x}$ is fed into the neural network (this is denoted by L, which represents the final layer). We include the $\frac{1}{N}$ because we are averaging the cost over N training examples.

Note that the cost for a single training example can be expressed as the following:

$$C = \frac{1}{2} \left\| \boldsymbol{a^L} - \boldsymbol{y} \right\|^2 = \frac{1}{2} \sum_j (a_j^L - y_j)^2 \tag{13}$$

2 The Backpropagation Algorithm

Idea: In machine learning, backpropagation is a widely used algorithm for training feedforward neural networks. When training a neural network by gradient descent, a loss function is calculated, which represents how far the network's predictions are from the true labels. Backpropagation allows us to calculate the gradient of the loss function with respect to each of the weights of the network. This enables every weight to be updated individually to gradually reduce the loss function over many training iterations.

- In order to do this, we define a vector δ^l containing elements of the form δ^l_j, which correspond to the error in the j^{th} neuron in layer l:

$$\delta^l = \begin{bmatrix} \delta^l_1 \\ \vdots \\ \delta^l_m \end{bmatrix} \tag{14}$$

- Furthermore, we define the error of single neuron j in layer l to be the partial derivative of the cost function with respect to the weighted input to the j^{th} neuron in layer l:

$$\delta^l = \begin{bmatrix} \frac{\partial C}{\partial z^l_1} \\ \vdots \\ \frac{\partial C}{\partial z^l_m} \end{bmatrix} \tag{15}$$

- Hence, using this concept, we can define four fundamental equations for backpropagation, which we will prove in order.

2.1 Equation 1: Error In The Final Layer L

The equation to compute the error of the final layer L is given by:

$$\delta^L = \nabla_{\boldsymbol{a^L}} C \odot \sigma'(\boldsymbol{z^L}) \tag{16}$$

We can show this by writing each term as a matrix of partial derivatives, and computing the final product. However, before we do this, we need to do some groundwork. Recall that n_{out} is the number of nodes in our final layer. Additionally, recall that the cost of a single training example can be expressed as:

$$C = \frac{1}{2}\sum_j (a^L_j - y_j)^2 \tag{17}$$

Hence, we do the following:

$$\frac{\partial C}{\partial a^L_j} = (a^L_j - y_j) \tag{18}$$

However, if we want to compute $\nabla_{\boldsymbol{a^L}} \boldsymbol{C}$, we just have to do the following:

$$\nabla_{\boldsymbol{a^L}} \boldsymbol{C} = \boldsymbol{a^L} - \boldsymbol{y} \tag{19}$$

Putting this all together, we can show the validity of this equation using the following:

$$\delta^L = \nabla_{\boldsymbol{a^L}} C \odot \sigma'(\boldsymbol{z^L}) \tag{20}$$

$$= \begin{bmatrix} \frac{\partial C}{\partial a_1^L} \\ \vdots \\ \frac{\partial C}{\partial a_{n_{out}}^L} \end{bmatrix} \odot \begin{bmatrix} \frac{\partial a_1^L}{\partial z_1^L} \\ \vdots \\ \frac{\partial a_{n_{out}}^L}{\partial z_{n_{out}}^L} \end{bmatrix} \tag{21}$$

As can be seen, once the element wise multiplication occurs, the equation shown in (15) arises for the final layer where $l = L$ and therefore (16) is a valid means of computing the error of the final layer.

2.2 Equation 2: Error In A Hidden Layer l

- The equation to compute the error in a hidden layer l is given by:

$$\delta^l = ((\boldsymbol{w}^{l+1})^T \delta^{l+1}) \odot \sigma'(\boldsymbol{z}^l) \tag{22}$$

- We can show this by writing each term as a matrix of partial derivatives and compute the final product:

$$\delta^l = ((\boldsymbol{w}^{l+1})^T \delta^{l+1}) \odot \sigma'(\boldsymbol{z}^l) \tag{23}$$

$$= \begin{bmatrix} w_{1,1}^{l+1} & \dots & w_{1,n}^{l+1} \\ \vdots & \ddots & \vdots \\ w_{m,1}^{l+1} & \dots & w_{m,n}^{l+1} \end{bmatrix}^T \begin{bmatrix} \frac{\partial C}{\partial z_1^l} \\ \vdots \\ \frac{\partial C}{\partial z_m^l} \end{bmatrix} \odot \begin{bmatrix} \frac{\partial a_1^l}{\partial z_1^l} \\ \vdots \\ \frac{\partial a_k^l}{\partial z_n^l} \end{bmatrix} \tag{24}$$

$$= \begin{bmatrix} \frac{z_1^{l+1}}{\partial a_1^l} & \dots & \frac{z_m^{l+1}}{\partial a_1^l} \\ \vdots & \ddots & \vdots \\ \frac{z_1^{l+1}}{\partial a_n^l} & \dots & \frac{z_m^{l+1}}{\partial a_n^l} \end{bmatrix} \begin{bmatrix} \frac{\partial C}{\partial z_1^l} \\ \vdots \\ \frac{\partial C}{\partial z_m^l} \end{bmatrix} \odot \begin{bmatrix} \frac{\partial a_1^l}{\partial z_1^l} \\ \vdots \\ \frac{\partial a_k^l}{\partial z_n^l} \end{bmatrix} \tag{25}$$

$$= \begin{bmatrix} \frac{\partial C}{\partial z_1^l} \frac{z_1^{l+1}}{\partial a_1^l} & \dots & \frac{\partial C}{\partial z_m^l} \frac{z_m^{l+1}}{\partial a_1^l} \\ \vdots & \ddots & \vdots \\ \frac{\partial C}{\partial z_1^l} \frac{z_1^{l+1}}{\partial a_n^l} & \dots & \frac{\partial C}{\partial z_m^l} \frac{z_m^{l+1}}{\partial a_n^l} \end{bmatrix} \odot \begin{bmatrix} \frac{\partial a_1^l}{\partial z_1^l} \\ \vdots \\ \frac{\partial a_k^l}{\partial z_n^l} \end{bmatrix} \tag{26}$$

As can be seen, once the element wise multiplication occurs, the equation shown in (15) arises for an arbitrary hidden layer l and therefore (22) is a valid means of computing the error of a hidden layer.

2.3 Equation 3: Derivative Of Cost Function W.R.T A Bias b_j^l For Neuron j In layer l

- The way to compute the $\frac{\partial C}{\partial b_j^l}$ is given by:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{27}$$

- We get this through the derivation shown below.

  Before proceeding, recall that we defined $\boldsymbol{z^l} = \boldsymbol{w^l a^{l-1}} + \boldsymbol{b^l}$ in (11) and we also defined $z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$ in (9).

  Hence, we can compute the following:

$$\frac{\partial z_j^l}{\partial b_j^l} = \frac{\partial}{\partial b_j^l}(\sum_k w_{jk}^l a_k^{l-1} + b_j^l) \tag{28}$$

$$= 0 + \frac{\partial}{\partial b_j^l}(b_j^l) \tag{29}$$

$$= 1 \tag{30}$$

Therefore, using the chain rule, we get that:

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l} \tag{31}$$

Using what was previously computed from (28) to (30):

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} = \delta_j^l \tag{32}$$

Therefore, we can say that:

$$\boldsymbol{\nabla_b^l} C = \delta^l \tag{33}$$

2.4 Equation 4: Derivative Of Cost Function W.R.T A Weight w_j^l For Neuron j In Layer l

- The way to compute the $\frac{\partial C}{\partial w_{jk}^l}$ is given by:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \tag{34}$$

- We prove that these two statements are equivalent below. Recall that $z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$ as was shown in (9).

To begin, we compute:

$$\frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1} \tag{35}$$

Therefore, we have:

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} \tag{36}$$

Substituting $\frac{\partial C}{\partial z_j^l}$ as δ_j^l and $\frac{\partial z_j^l}{\partial w_{jk}^l}$ as a_k^{l-1}, as was calculated previously. By doing so, we get the following:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \tag{37}$$

This gives us the result we were trying to prove in (34).

Now, when programming a neural network, we usually want to compute the a matrix of the same dimension as the weights matrix, with each element being the partial derivative of the weight at that position with respect to the cost.

This matrix looks like the following:

$$\begin{bmatrix} \frac{\partial C}{\partial z_1^l}\frac{z_1^l}{\partial w_{1,1}^l} & \cdots & \frac{\partial C}{\partial z_1^l}\frac{z_1^l}{\partial w_{1,n}^l} \\ \vdots & \ddots & \vdots \\ \frac{\partial C}{\partial z_m^l}\frac{z_m^l}{\partial w_{m,1}^l} & \cdots & \frac{\partial C}{\partial z_m^l}\frac{z_m^l}{\partial w_{m,n}^l} \end{bmatrix} \tag{38}$$

We can compute this using the following formula:

$$\delta^l (a^{l-1})^T \tag{39}$$

This formula can be shown to work as follows. We begin by writing each term as a matrix, to get:

$$\delta^l (a^{l-1})^T = \begin{bmatrix} \frac{\partial C}{\partial z_1^l} \\ \vdots \\ \frac{\partial C}{\partial z_m^l} \end{bmatrix} \begin{bmatrix} a_1^l & \dots & a_n^{l-1} \end{bmatrix} \tag{40}$$

$$= \begin{bmatrix} \frac{\partial C}{\partial z_1^l} a_1^{l-1} & \cdots & \frac{\partial C}{\partial z_1^l} a_n^{l-1} \\ \vdots & \ddots & \vdots \\ \frac{\partial C}{\partial z_m^l} a_1^{l-1} & \cdots & \frac{\partial C}{\partial z_m^l} a_n^{l-1} \end{bmatrix} \tag{41}$$

We previously found that $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$ in (35). This statement means that the $\frac{\partial C}{\partial w_{jk}^l}$ does not rely on any terms indexed with neurons in layer $l-1$. Hence, we have that for all $j \in \{1,\dots,m\}$, the $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$.

Therefore, we get the following:

$$\begin{bmatrix} \frac{\partial C}{\partial z_1^l}\frac{z_1^l}{\partial w_{1,1}^l} & \cdots & \frac{\partial C}{\partial z_1^l}\frac{z_1^l}{\partial w_{1,n}^l} \\ \vdots & \ddots & \vdots \\ \frac{\partial C}{\partial z_m^l}\frac{z_m^l}{\partial w_{m,1}^l} & \cdots & \frac{\partial C}{\partial z_m^l}\frac{z_m^l}{\partial w_{m,n}^l} \end{bmatrix} \tag{42}$$

This simplifies to the derivative of a cost function with respect to a weight w_j^l for a neuron j in layer l as was shown initially in (34).

3 Programming A NN From Scratch

- It's time to take the mathematical insights and test whether this can be programmed.

```
#Imports
import math
import numpy as np

#Creating the NN Class
class NeuralNet:
    def __init__(self, layer_list):

      #Initialize class variables
      self.layer_list = layer_list
      self.weights = []
      self.bias = []

      #Initialize random weights and biases
      for i in range(1, len(layer_list)):
        rows = layer_list[i]
        cols = layer_list[i-1]
        self.weights.append(np.random.rand(rows,cols)*0.01)
        self.bias.append(np.random.rand(rows,1)*0.01)

      #Checking what the weights and biases look like
      print("Weights")
      print([element.shape for element in self.weights])

      print("\n" + "Biases")
      print([element.shape for element in self.bias])

    #Sigmoid activation function
    def sigmoid(self, x):
      sigmoid = 1/(1+(np.e)**-x)
      return sigmoid

    #Derivative of the sigmoid activation function
    def derivative_sigmoid(self, x):
      derivative = self.sigmoid(x)*(1-self.sigmoid(x))
      return derivative

    #Cost function (using MSE)
    def cost(self, a, y):
      cost = 0.5*((a-y)**2)
      cost = np.sum(cost)
      return cost

    #Derivative of cost function
    def derivative_cost(self, a, y):
      derivative = a - y
      return derivative
```

```
    #Feed forward steps of the NN
    def feed_forward(self, x, y):
      self.y = y
      self.x = x
      self.z = []
      self.activation = [x] #Note: Activation of first layer is just the inputs x

      for i in range(len(self.weights)):
        z_value = np.dot(self.weights[i], np.array(self.activation[i])) + np.array(self.bias[i])
        self.activation.append(self.sigmoid(z_value))
        self.z.append(z_value)

      cost = self.cost(self.activation[-1], y)

      return cost

    #Calculating the gradients and biases
    def gradients_and_biases(self):
      grad_weights = [np.zeros(element.shape) for element in self.weights]
      grad_bias = [np.zeros(element.shape) for element in self.bias]
      delta = self.derivative_cost(self.activation[-1], self.y) * self.derivative_sigmoid(self.
      derivative_cost_wrt_weight = np.dot(delta, (self.activation[-2]).T)
      derivative_cost_wrt_bias = delta
      grad_weights[-1] = derivative_cost_wrt_weight
      grad_bias[-1] = derivative_cost_wrt_bias

      for i in range(2, len(self.layer_list)):
        delta = np.dot((self.weights[-i+1].T), delta) * self.derivative_sigmoid(self.z[-i])
        derivative_cost_wrt_weight = np.dot(delta, (self.activation[-i-1]).T)
        derivative_cost_wrt_bias = delta
        grad_weights[-i] = derivative_cost_wrt_weight
        grad_bias[-i] = derivative_cost_wrt_bias

      return grad_weights, grad_bias

    #Updating the weights and biases
    def update_weights_and_biases(self, alpha, grad_weights, grad_bias):
      for i in range(len(self.weights)):
        self.weights[i] = self.weights[i] - alpha*grad_weights[i]
        self.bias[i] = self.bias[i] - alpha*grad_bias[i]
```

- Next, the NN class that was programmed will be tested on MNIST. The following is an implementation:

```
#Imports
import tensorflow

#Preparing MNIST dataset
from tensorflow.keras.datasets import mnist
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()

X_train = X_train.reshape(X_train.shape[0], X_train.shape[1] * X_train.shape[2]) / 255

X_test = X_test.reshape(X_test.shape[0], X_test.shape[1] * X_test.shape[2])
```

```
#Preparing batches of data
def prepare_mini_batches(minibatch_size):
    random_indexes = []
    indexes = np.random.choice(X_train.shape[0], minibatch_size, replace=False)
    random_indexes.append(indexes)
    return random_indexes

#Function to one hot encode classes
def one_hot_encode(length, index):
    output = [0 for i in range(length)]
    output[index] = 1
    return np.asarray(output).reshape(length, 1)

#Function to compute the running loss
def compute_test_loss(test_set, net):
    total_loss = []
    for i in range(len(X_test)):
        loss = net.feed_forward(X_test[i].reshape(X_test[i].shape[0], 1), one_hot_encode(10 ,Y_
        total_loss.append(loss)
    total = 0
    for i in total_loss:
      total += i
    return total/len(total_loss)

#Instantiate the NN
nn = NeuralNet([784, 392, 196, 10])
epochs = 200
batch_size = 32

for i in range(epochs):
  random_indexes = prepare_mini_batches(batch_size)
  for mini_batch in random_indexes:
    grad_weights = [np.zeros(element.shape) for element in nn.weights]
    grad_bias = [np.zeros(element.shape) for element in nn.bias]
    for j in mini_batch:
       nn.feed_forward(X_train[j].reshape(X_train[j].shape[0], 1), one_hot_encode(10 ,Y_train[j
       grad_w, grad_b = nn.gradients_and_biases()
       for k in range(len(grad_w)):
         grad_weights[k] = grad_weights[k] + grad_w[k]
         grad_bias[k] = grad_bias[k] + grad_b[k]
    grad_weights = [grad_weights[i]/batch_size for i in range(len(grad_weights))]    #The act o
    grad_bias = [grad_bias[i]/batch_size for i in range(len(grad_bias))]
    nn.update_weights_and_biases(0.003, grad_weights, grad_bias)

  print("Updating Weights")
  print("Test Loss = " + str(compute_test_loss(X_test, nn)))
```

- To see an implementation of this on Google Colab, check out this page. This links to the ML Playbook repository where you can also see other implementations of a range of network architectures including but not limited to: Multi-Layer Perceptrons, Convolutional Neural Networks, RNNs and more!