Backpropagation and Gradient-based optimisation

Mohammed Luqmaan Akhtar

December 2025

Introduction

First let us define the structure of the neural network which I have used for the Wine Quality Dataset. Based on this structure all the relevant concepts will be explained.
I have taken a two layer neural network. The input has 11 features and output 1 but we will use variables to generalize it.

- The input is matrix X of the shape (n_x, m) where n_x is the number of features and m is the number of training examples
- The hidden layer has n_h number of nodes.
- The weight and bias matrix associated with the hidden layer i.e W_1 and b_1 have shapes (n_h, n_x) and and $(n_h, 1)$ respectively.
- The output has n_y number of nodes.
- The weight and bias matrix associated with the output layer i.e W_2 and b_2 have shapes (n_y, n_h) and and $(n_y, 1)$ respectively.
- Z_1 and A_1 is are matrices of shape (n_h, m) and contain the value of nodes in the hidden layer. Similarly, Z_2 and A_2 contain that for the output layer.

Processes in a Neural Network

1. **Forward pass/Forward propagation**: In this step we calculate the values of the nodes of the hidden layers and the output layer. The equations are as follows:
   $Z_1 = W_1X + b1$
   $A_1 = g_1(Z_1)$ where g_1 is an activation function (ReLU chosen)
   $Z_2 = W_2X + b2$
   $A_2 = g_2(Z_2)$ where g is an activation function (Identity chosen)

2. **Computing cost**: This is not necessarily a step but conceptually it matters because we calculate the gradients of different parameters with respect to this to get their optimal value. The optimal value is obatined by minimizing this function. The loss function is chosen with respect to the use case, I chose Mean Squared Error.

$$MSE = \frac{1}{m}\sum_{i=1}^{N}(A_2 - Y)^2$$

where m is the number of training samples and Y is their true values.

3. **Back propagation**: This is the most mathematically complex part of the entire process and we calculate the gradients with respect to differnet parameters to use them during gradient descent. Chain rule is a very important component of this process.

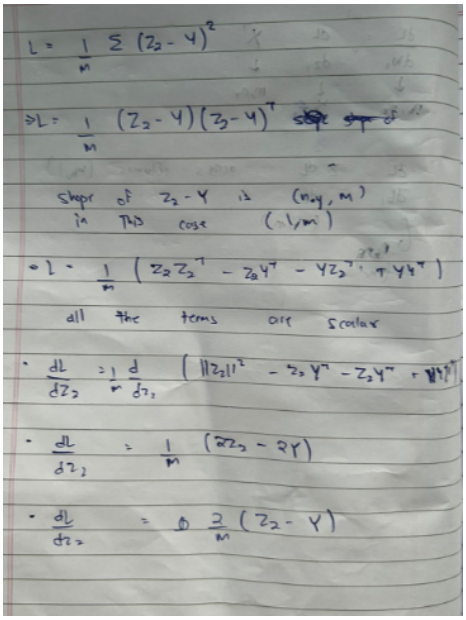

Figure 1:

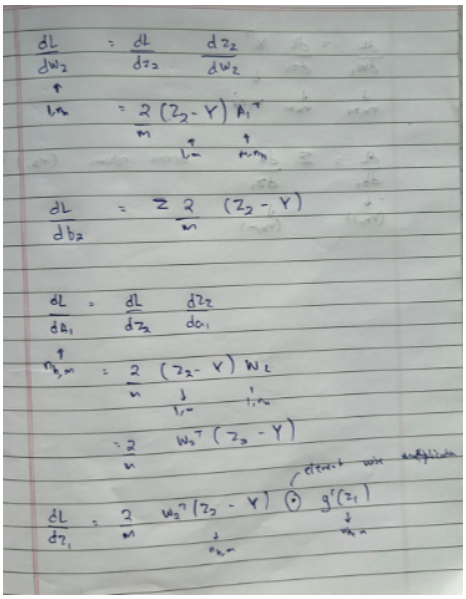

Figure 2:

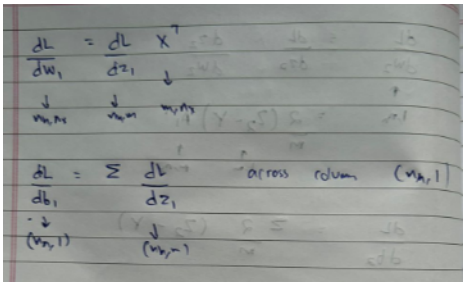

Figure 3:

Note: There is a fair bit of intuition involved in these derivations using a single sample or using the shapes of the matrices.

4. **Update parameters:** Now that we have the gradients of the parameters all we need to do is update it. Gradient descent is covered in the following section. These steps comprise one iteration of gradient descent we repeat this multiple times to get the optimal parameter values.

Gradient Descent

The loss function is minimized using the gradient descent method. Gradient Descent is an iterative optimization algorithm used to minimize a function by adjusting model parameters in the direction of the steepest descent of the function's gradient. In simple terms, it finds the optimal values of weights and biases by gradually reducing the error between predicted and actual outputs. The equation for gradient descent are:

$$w = w - \gamma \frac{\partial L}{\partial w}$$

where w is any weight and γ is the learning rate.
Choosing the correct value of learning rate is important. If we choose too large a value the loss function is never minimized due to overshooting. On the other hand too small a value also leads to the loss function never minimizing because we never reach the minimum in the given number of iterations.
Basically too large leads to divergence and too small leads to slow convergence.

Loss functions in NNs

Until now, I have used Neural Networks either for predicting a value or for binary classification and the loss functions for each case is:

1. **Mean Squared Error**: This is for value prediction

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (y_{pred_i} - y_i)^2$$

   where N is the total number of data samples.

2. **Binary Cross Entropy**: This is for binary classification. It works best when we use sigmoid as the activation function for the last layer/node as we have seen in Logistic Regression.

$$J = -\frac{1}{N} \sum_{i=1}^{N} [y_i log p_i + (1 - y_i) log(1 - p_i)]$$

   where N is the total number of data samples.

Activation Functions

I have come across three kinds of activation functions:

1. **Sigmoid**:

$$\sigma = \frac{1}{1 + e^{-z}}$$

This function maps any value between 0 and 1. This is not very suitable for NNs because after a certain value the slope becomes almost equal to zero.

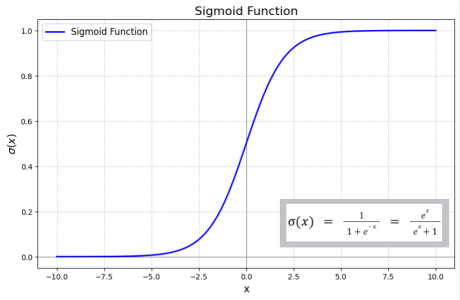

Figure 4:

2. **Hyperbolic Tan**:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

This function maps any value between -1 and 1. This is better than sigmoid because the data is more centered around zero

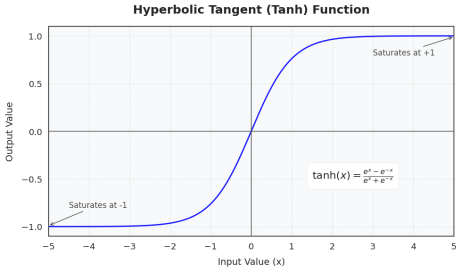

Figure 5:

3. **Rectified Linear Unit**:

$$ReLU = \max(0, x)$$

This function is best suited for activation because the slope is either 0 or 1. So no matter how large the value the slope remains 1 and does not approach 0. This is very suitable for learning.

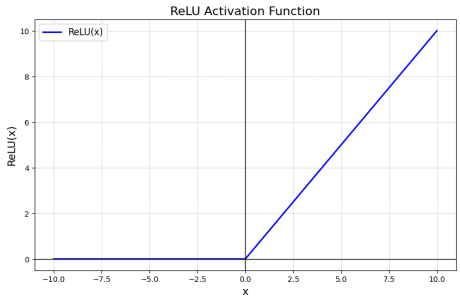

Figure 6:

RMSprop Optimizer

RMSprop (Root Mean Square Propagation) is an adaptive learning rate optimization algorithm designed to address the diminishing learning rate problem of standard gradient descent and AdaGrad. It maintains an exponentially decaying average of squared gradients to normalize parameter updates.

Given a parameter w and its gradient $g_t = \nabla_w L_t$, RMSprop computes:

$$s_t = \beta s_{t-1} + (1 - \beta) g_t^2$$

where $\beta \in [0, 1)$ is the decay rate, typically set to 0.9.

The parameter update rule is:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{s_t + \epsilon}} g_t$$

where η is the learning rate and ϵ is a small constant to prevent division by zero.

By scaling the learning rate inversely proportional to the magnitude of recent gradients, RMSprop enables stable and faster convergence, particularly in deep neural networks.

Adam Optimizer

Adam (Adaptive Moment Estimation) combines the advantages of RMSprop and momentum-based gradient descent. It maintains exponentially decaying averages of both the first moment (mean) and second moment (uncentered variance) of the gradients.

Let $g_t = \nabla_w L_t$. Adam computes:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

To correct bias introduced during initialization, bias-corrected estimates are used:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

The parameter update rule is:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

Adam is widely used due to its fast convergence, robustness to noisy gradients, and minimal need for manual tuning. Typical values are $\beta_1 = 0.9$, $\beta_2 = 0.999$.

Exploding and Vanishing Gradients

In deep neural networks, gradients are propagated backward through multiple layers using the chain rule. The gradient of the loss with respect to early-layer weights can be expressed as:

$$\frac{\partial L}{\partial W^{(1)}} = \left(\prod_{l=2}^{L} \frac{\partial z^{(l)}}{\partial z^{(l-1)}} \right) \frac{\partial L}{\partial z^{(L)}}$$

If these terms have values less than one, gradients shrink exponentially as they propagate backward, resulting in vanishing gradients. This leads to extremely slow learning in early layers.

Conversely, if these terms have values greater than one, gradients grow exponentially, resulting in exploding gradients. This can cause numerical instability and divergence during training.

Vanishing gradients are common with sigmoid and tanh activations, while exploding gradients often occur in deep or recurrent networks. Solutions include:

- Proper weight initialization (Xavier, He)
- Use of ReLU and its variants
- Gradient clipping
- Batch normalization