Date of Submission June 02, 2025.

# Multi Level Perceptron And Back Propagation

**AMRIT KANDEL[1], (THA078BEI004), PRASISH TIMALSINA[2], (THA078BEI026)**
[1]Department of Electronics and Computer Engineering, Thapathali Campus (e-mail: amrit.tha078bei004@tcioe.edu.np)
[2]Department of Electronics and Computer Engineering, Thapathali Campus (e-mail: prasish.tha078bei026@tcioe.edu.np)

**ABSTRACT** These instructions give you guidelines for preparing papers for IEEE Access. Use this document as a template if you are using LaTeX. Otherwise, use this document as an instruction set. The electronic file of your paper will be formatted further at IEEE. Paper titles should be written in uppercase and lowercase letters, not all uppercase. Avoid writing long formulas with subscripts in the title; short formulas that identify the elements are fine (e.g., "Nd–Fe–B"). Do not write "(Invited)" in the title. Full names of authors are preferred in the author field, but are not required. Put a space between authors' initials. The abstract must be a concise yet comprehensive reflection of what is in your article. In particular, the abstract must be self-contained, without abbreviations, footnotes, or references. It should be a microcosm of the full article. The abstract must be between 150–250 words. Be sure that you adhere to these limits; otherwise, you will need to edit your abstract accordingly. The abstract must be written as one paragraph, and should not contain displayed mathematical equations or tabular material. The abstract should include three or four different keywords or phrases, as this will help readers to find it. It is important to avoid over-repetition of such phrases as this can result in a page being rejected by search engines. Ensure that your abstract reads well and is grammatically correct.

**INDEX TERMS** Activation function, back propagation, encoding, gradient descent, optimizer, perceptron, sigmoid function.

## I. INTRODUCTION

### A. BACK PROPAGATION

Backpropagation is an algorithm used to train deep neural networks by minimizing the error between predicted and actual outputs. It enables efficient computation of gradients needed for weight updates across multiple layers, which is essential for learning complex patterns from large datasets. It works by calculating the gradient of a loss function with respect to each weight in the network through the application of the chain rule of calculus. These gradients indicate how much each weight contributes to the overall error, allowing the model to adjust its parameters and improve performance.

The Figure 1 illustrates the concept of backpropagation in a neural network, showing both the forward propagation of input data and the backward flow of error signals through the network layers.

### B. MULTI-LEVEL PERCEPTRON

The Single-Layer Perceptron (SLP) is the simplest form of a neural network, composed of only an input layer and an output layer with no hidden layers. It is capable of solving problems that are linearly separable, such as basic logic gates like AND, OR, and NOT.

However, it fails when dealing with problems that are not linearly separable, such as the XOR (Exclusive OR) gate. The XOR function produces an output of 1 only when the inputs differ. No straight line can separate the output classes in a 2D input space, making it impossible for an SLP to correctly classify XOR inputs.

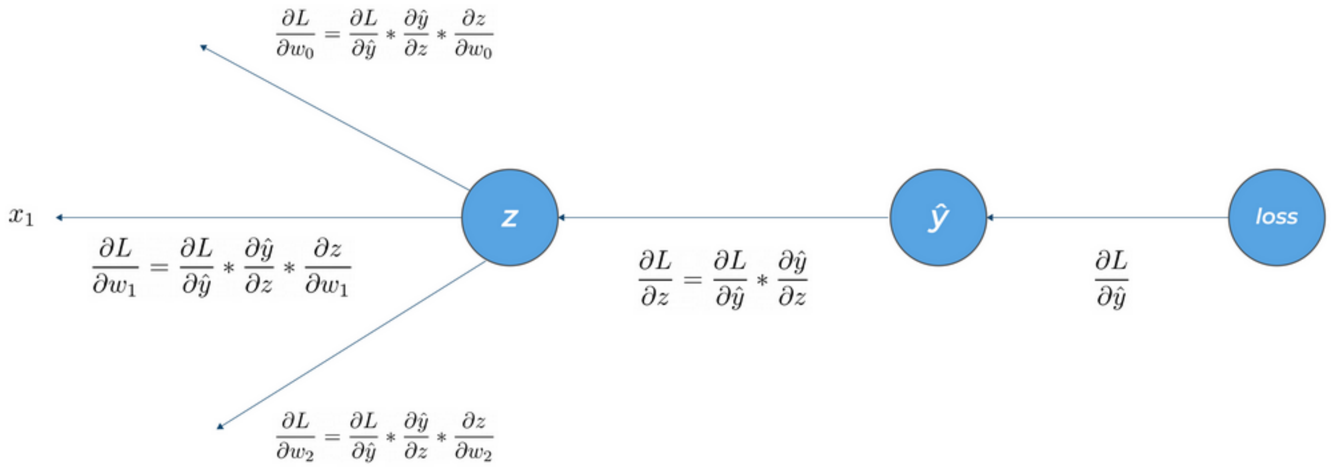
To overcome this limitation, the Multilayer Perceptron introduces one or more hidden layers between the input and output as shown in Figure 2. These hidden layers allow the network to learn non-linear decision boundaries using non-linear activation functions such as ReLU, sigmoid, or tanh.

### C. ACTIVATION FUNCTIONS

Activation functions are mathematical operations applied to the output of each neuron in a neural network layer. They determine whether a neuron should be activated or not by introducing non-linearity to the model. Without activation functions, neural networks would behave like simple linear models, regardless of their depth. Some popular activation functions are described below.

#### 1) Sigmoid function

The sigmoid function squashes input values into the range (0, 1), making it useful for binary classification problems. However, it suffers from vanishing gradient issues in deep networks.



$\frac{\partial L}{\partial w_0} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z} * \frac{\partial z}{\partial w_0}$
$x_1$
$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z} * \frac{\partial z}{\partial w_1}$
z
$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z}$
ŷ
loss
$\frac{\partial L}{\partial \hat{y}}$
$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z} * \frac{\partial z}{\partial w_2}$

**FIGURE 1.** **Back propagation using chain rule.**

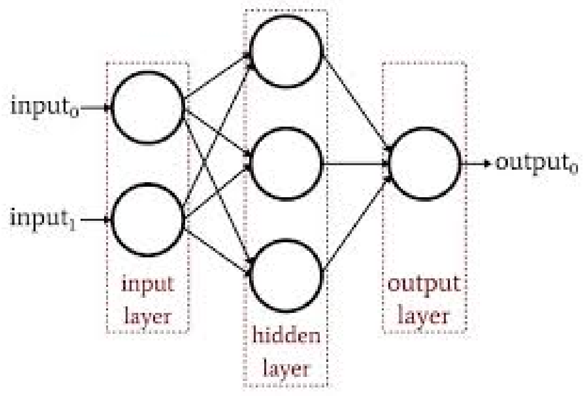


**FIGURE 2.** **Multi layer perceptron.**

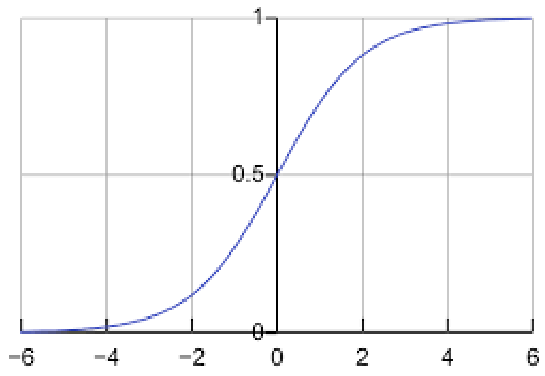


**FIGURE 3.** **Graph of a sigmoid function**

*a: Equation*

The equation for the sigmoid function is given as (1)

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

*b: Derivative*

The derivative of the sigmoid function is:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{2}$$

### 2) ReLU (Rectified Linear Unit)

ReLU is widely used in hidden layers of deep networks. It outputs the input if it is positive and zero otherwise. It is computationally efficient and helps avoid the vanishing gradient problem.

*a: Equation*

$$f(x) = \max(0, x) \tag{3}$$

*b: Derivative*

The derivative of ReLU is:

$$f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{4}$$

### 3) Leaky ReLU function

Leaky ReLU addresses the problem of dead neurons in standard ReLU by allowing a small, non-zero gradient when the input is negative.

*a: Equation*

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases} \tag{5}$$

*b: Derivative*

The derivative is:

$$f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ \alpha & \text{if } x \leq 0 \end{cases} \tag{6}$$

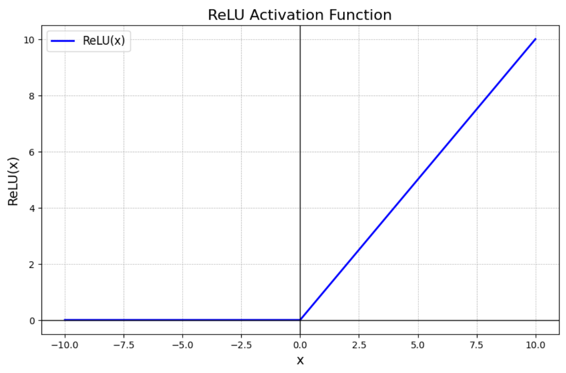


**FIGURE 4. Graph of a ReLU function**

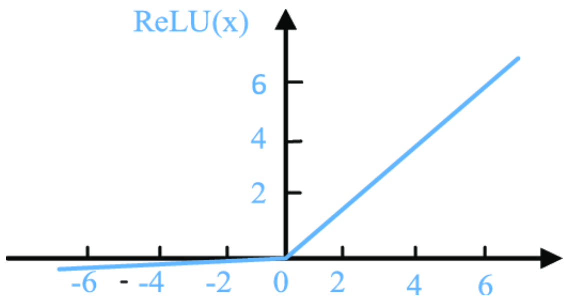


**FIGURE 5. Graph of a Leaky ReLU function**

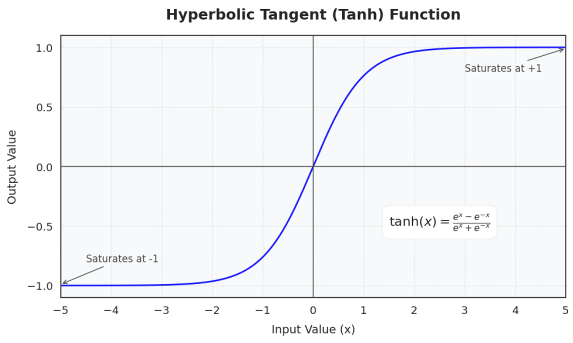


**FIGURE 6. Tanh Activation Function**

### 4) Tanh (Hyperbolic Tangent)

The Tanh function is similar to the sigmoid function but maps inputs to a range between -1 and 1. It is zero-centered, making it generally better than sigmoid for training.

#### *a: Equation*

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{7}$$

#### *b: Derivative*

The derivative of Tanh is:

$$\tanh'(x) = 1 - \tanh^2(x) \tag{8}$$

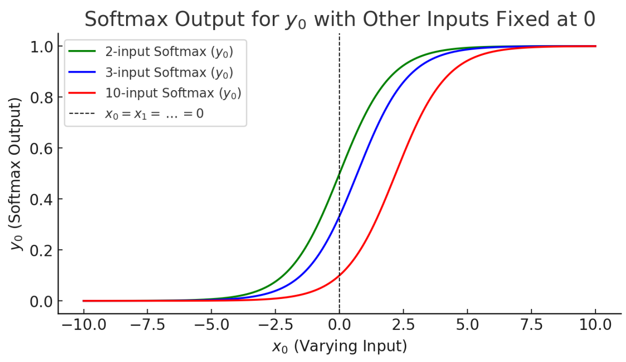


**FIGURE 7. Graph of a SoftMax function**

### 5) Softmax Function

The Softmax function is used in the output layer of neural networks for multi-class classification. It converts logits into a probability distribution over multiple classes.

#### *a: Equation*

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}, \quad \text{for } i = 1, 2, \dots, n \tag{9}$$

#### *b: Derivative*

The derivative of softmax is a Jacobian matrix given by:

$$\frac{\partial y_i}{\partial x_j} = \begin{cases} y_i(1 - y_i) & \text{if } i = j \\ -y_i y_j & \text{if } i \neq j \end{cases} \tag{10}$$

## *D. LOSS FUNCTIONS*

Loss functions and evaluation metrics play a vital role in training and assessing machine learning models. They quantify how well the model's predictions match the true outputs. In this subsection, we describe five commonly used functions: Mean Squared Error (MSE), Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), R-squared Score (R$^2$), and Adjusted R-squared Score.

### 1) Mean Squared Error (MSE)

MSE measures the average squared difference between the actual and predicted values. It penalizes larger errors more than smaller ones.

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{11}$$

Where:

- $y_i$ is the actual value
- $\hat{y}_i$ is the predicted value
- $n$ is the number of samples

### 2) Mean Absolute Error (MAE)

MAE calculates the average of absolute differences between predicted and actual values. It treats all errors equally.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{12}$$

### 3) Root Mean Squared Error (RMSE)

RMSE is the square root of MSE. It provides an error measure in the same unit as the output variable, making it more interpretable.

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{13}$$

### 4) R-squared Score ($R^2$)

$R^2$ score, or coefficient of determination, measures the proportion of the variance in the dependent variable that is predictable from the independent variables.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{14}$$

Where $\bar{y}$ is the mean of the actual values.

### 5) Adjusted R-squared Score

Adjusted $R^2$ modifies the $R^2$ score to account for the number of predictors in the model, penalizing unnecessary complexity.

$$\text{Adjusted } R^2 = 1 - \left(\frac{(1-R^2)(n-1)}{n-k-1}\right) \tag{15}$$

Where:

- $n$ = number of data points
- $k$ = number of independent variables

## E. OPTIMIZERS

Optimizers are algorithms used to minimize the loss function during training by updating the weights of the neural network. They determine how the model learns from the data by adjusting parameters based on gradients. Below are five commonly used optimizers in deep learning:

### 1) Stochastic Gradient Descent (SGD)

SGD updates the model's parameters using the gradient of the loss function with respect to each parameter, based on a random subset (mini-batch) of the training data.

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta_t) \tag{16}$$

Where:

- $\theta$ are the model parameters
- $\eta$ is the learning rate
- $J(\theta)$ is the loss function

### 2) Momentum

Momentum builds on SGD by adding a fraction of the previous update to the current one, smoothing the update trajectory and accelerating convergence.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta_t) \tag{17}$$

$$\theta_{t+1} = \theta_t - v_t \tag{18}$$

Where $\gamma$ is the momentum coefficient (typically around 0.9).

### 3) AdaGrad (Adaptive Gradient Algorithm)

AdaGrad adapts the learning rate for each parameter individually based on the accumulated squared gradients, making it suitable for sparse data.

$$r_t = r_{t-1} + \nabla_\theta J(\theta_t)^2 \tag{19}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{r_t + \epsilon}} \nabla_\theta J(\theta_t) \tag{20}$$

Where $\epsilon$ is a small constant to avoid division by zero.

### 4) RMSProp

RMSProp addresses the diminishing learning rate problem in AdaGrad by using an exponentially decaying average of squared gradients.

$$r_t = \rho r_{t-1} + (1-\rho)\nabla_\theta J(\theta_t)^2 \tag{21}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{r_t + \epsilon}} \nabla_\theta J(\theta_t) \tag{22}$$

Where $\rho$ is the decay rate (e.g., 0.9).

### 5) Adam (Adaptive Moment Estimation)

Adam combines the benefits of Momentum and RMSProp by maintaining both first and second moment estimates of the gradients.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta J(\theta_t) \tag{23}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\left(\nabla_\theta J(\theta_t)\right)^2 \tag{24}$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{25}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t \tag{26}$$

Where $\beta_1$ and $\beta_2$ are decay rates for the first and second moments, typically 0.9 and 0.999 respectively.

## F. ENCODING TECHNIQUES

In machine learning, most algorithms can only handle numerical input. Therefore, categorical data must be converted into numerical format. Two common encoding techniques used for this purpose are **Label Encoding** and **One-Hot Encoding**.

**TABLE 1. Comparison Between Label Encoding and One-Hot Encoding**

| Feature | Label Encoding | One-Hot Encoding |
|---|---|---|
| **Definition** | Assigns a unique integer to each category | Creates binary columns for each category |
| **Example for 'Color = [Red, Green, Blue]'** | Red → 0, Green → 1, Blue → 2 | Red → [1,0,0], Green → [0,1,0], Blue → [0,0,1] |
| **Output Format** | Single column with integer values | Multiple columns with binary values |
| **Ordering Information** | Introduces ordinal relationship (even if not intended) | No ordinal relationship |
| **Best Use Case** | Ordinal categorical variables | Nominal (unordered) categorical variables |
| **Disadvantage** | May mislead models into thinking one value is greater than another | Can cause high dimensionality with many categories |

### 1) Label Encoding

Label Encoding is a method where each unique category value is assigned an integer. This method is simple and efficient but can be misleading for machine learning models, as it may imply a natural ordering among categories even if none exists. It is more suitable for ordinal categorical variables, where such ordering is meaningful.

**Example:**

- Red → 0
- Green → 1
- Blue → 2

### 2) One-Hot Encoding

One-Hot Encoding transforms each category into a new binary column, where the presence of a category is marked as 1 and the rest are 0. This avoids any ordinal implications between categories and is suitable for nominal (unordered) variables. However, it increases the dimensionality, especially if there are many unique categories.

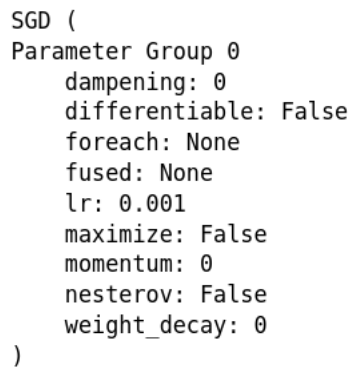
**Example:**

- Red → [1, 0, 0]
- Green → [0, 1, 0]
- Blue → [0, 0, 1]

The comparison between one hot encoding and label encoding is shown in Table 1.

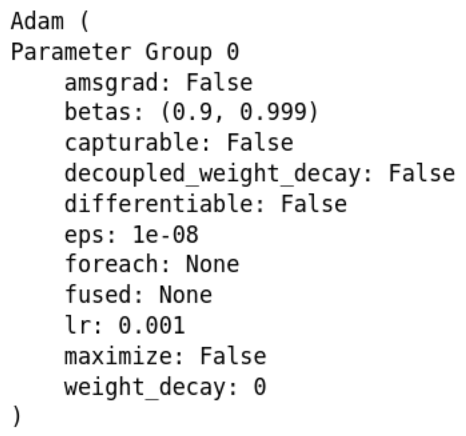
## II. EXPERIMENTATION MODEL

In this experiment, we built a Multilayer Perceptron (MLP) model using PyTorch to understand how backpropagation works and how different optimizers affect training. The model consisted of three fully connected layers: the first layer had 16 neurons with a LeakyReLU activation function, the second had 8 neurons with a Sigmoid activation, and the final layer had 2 output neurons with a Softmax activation for classification. We used the CrossEntropyLoss function to measure the error between predicted and actual outputs. To compare optimization methods, we trained the model using both Stochastic Gradient Descent (SGD) and Adam optimizers. The input was a fixed random tensor to ensure consistent results across



```
SGD (
Parameter Group 0
    dampening: 0
    differentiable: False
    foreach: None
    fused: None
    lr: 0.001
    maximize: False
    momentum: 0
    nesterov: False
    weight_decay: 0
)
```

**FIGURE 8.** SGD Optimizer



```
Adam (
Parameter Group 0
    amsgrad: False
    betas: (0.9, 0.999)
    capturable: False
    decoupled_weight_decay: False
    differentiable: False
    eps: 1e-08
    foreach: None
    fused: None
    lr: 0.001
    maximize: False
    weight_decay: 0
)
```

**FIGURE 9.** Adam Optimizer

runs. This setup allowed us to observe the impact of optimizer choice on model accuracy and learning efficiency.

## III. RESULTS

In lab, we used two optimizers; SGD and Adam, and compared their efficiency. It was found that, Adam performed with greater accuracy in our data. We kept the batch size 16 for experiment.

### A. USING SGD

Using SGD Optimizer, the accuracy of 50.10% was obtained.

### B. USING ADAM

Using Adam Optimizer, the accuracy of 90% was obtained.

## IV. DISCUSSION

**Generation of random number by torch.randn() function**

The torch.randn() function in PyTorch generates random numbers from a standard normal distribution, which has a mean of 0 and a standard deviation of 1. This means the values are centered around 0 and follow a bell-shaped curve, where most numbers are close to 0 and fewer are far from it. The function creates a tensor of any given shape, and each element is a random number drawn independently from this distribution. It is commonly used in deep learning to initialize weights in a way that helps models learn better. Unlike torch.rand(), which gives values between 0 and 1 from a uniform distribution, torch.randn() gives both positive and negative values from a normal distribution.

**Performance comparison: SGD VS Adam**
Our results demonstrate a significant performance difference: the SGD-trained model achieved a precision of 50. 1%, while the Adam-trained model reached a precision of 90%. This highlights the influence of optimizer choice on convergence speed and final model performance.

The relatively low performance of SGD can be attributed to its sensitivity to learning rate and its inability to adapt the learning process dynamically. In contrast, Adam combines the benefits of both AdaGrad and RMSProp, using adaptive learning rates and momentum, which accelerates convergence and leads to better generalization in this context.

## V. CONCLUSION

In this lab, we understood how backpropagation and neural networks work by letting us build and train a model ourselves. We saw that the choice of optimizer makes a big difference. Adam gave much better results than SGD, showing that it's a stronger choice for training models like this. This shows how important it is to try different settings and optimizers when training deep learning models.