Demystifying ANN with Mathematical and Graphical Insights: An Algorithmic Review for Beginners

Tayyaba Saeed, Syeda Zahra Kazmi, Amjad Ali*, Salma Zafar

Centre for Advanced Studies in Pure and Applied Mathematics (CASPAM)
Bahauddin Zakariya University (BZU), Multan, Pakistan.

*amjadali@bzu.edu.pk

Abstract

Developments in deep learning with ANNs (Artificial Neural Networks) are paving the way for revolutionizing all application areas, especially related to non-linear regression and classification problems of predictive modelling and forecasting. Although their explainability is more complicated and challenging, deep neural networks are preferred over conventional machine learning methods for high accuracy in non-linear and complex problems. However, machine learning and data science practitioners often use ANN like a black-box. The present article concisely overviews the mathematics and computations involved in simple feed-forward neural networks (FNNs) or multilayer perceptrons (MLPs). The purpose is to spot light on what deep neural networks' learning (or training) is and how it works. The article includes simplified derivations of the expressions for the main workhorse of neural networks (the backpropagation) and an example to explain how it works with graphical insights. An algorithm for a basic ANN application is presented in both component-form and matrix-form, together with a detailed note on the relevant data structures, to elaborate the scheme comprehensively. Python implementation of the basic algorithm is presented, and its performance results are compared with those produced using the TensorFlow library functions that implement the neural networks. The article discusses various techniques to improve the generalization capability of neural networks and how to address various training challenges. Finally, some well-established optimization approaches based on the Gradient Descent method are also discussed. The article may serve as a comprehensive premiere for a sound understanding of deep learning for undergraduate and graduate students before indulging in the relevant industry practices so that they can step into sustainable progress in the field.

Keywords: Artificial Neural Networks; ANN; Feed Forward Neural Networks; Deep Neural Networks; Deep Learning; Backpropagation; Gradient Descent; Pseudo Code of ANN; Hyperparameter Tuning; Regularization; Learning Curves; Batch Normalization; Adaptive Learning Rate

ACM Classification Codes (ccs98): **I.2.6**; **I.5.1**; **K.3.2**. MSC Codes (2020): **68T07**

1. Introduction

Artificial Neural Networks (ANNs) are machine learning (ML) algorithms modelled after the human brain's structure and function. The use of ANN has proliferated in recent years, fuelled by advancements in computational power, data availability, and algorithm development. As a result, ANNs have revolutionized various fields, including speech recognition, natural language processing, image recognition, computer vision, anomaly detection, recommender systems, and autonomous driving. The developments are ongoing, and the foreseeable future is expected to be even more promising. The success of ANNs in these applications is due to their ability to learn complex patterns in data and make accurate predictions or perform other deep learning tasks.

Cite as: Saeed, T., Kazmi, S.Z., Ali, A., & Zafar, S. (2023). Demystifying ANN with Mathematical and Graphical Insights: An Algorithmic Review for Beginners (Version 2). Zenodo. https://doi.org/10.5281/zenodo.8353418 [Sep. 17, 2023]

The history of ANNs can be traced back to the pioneering work of McCulloch and Pitts in the 1940s (McCulloch & Pitts, 1943), who proposed a simple mathematical model of a neuron that could perform logical operations. Rosenblatt (1958) developed a perceptron that was capable of learning from data and recognizing patterns in images. However, the limitations of the Perceptron were later discovered, leading to a decline in interest in ANNs in the 1970s. Nonetheless, researchers continued to work on improving the algorithms and architectures of ANNs, leading to a breakthrough in the development of the Backpropagation algorithm in the 1980s (Rumelhart et al., 1986). This renewed interest in ANNs in the 1990s and led to the development of deep neural networks in the 2000s and onward. Now, neural networks have achieved remarkable results in various fields and gained wider acceptance as a powerful tool for deep learning.

ANNs are composed of multiple interconnected processing nodes or units, called ***neurons***, that work together to learn from data and do ML tasks. For supervised learning, this is achieved by training the network using a set of labelled data and adjusting the connections between neurons to minimize the error between the predicted output and the target output. The neurons are arranged in multiple interconnected layers, which hierarchically represent the patterns in the data. These layers can be seen as a sequence of non-linear transformations that map the input data to a higher-level representation, which captures or extracts increasingly complex features and patterns from raw data without requiring explicit feature engineering. Other ML algorithms, such as decision trees or support vector machines, often require hand-crafted features that are specifically designed for the task at hand. In contrast, neural networks can learn these features directly from the data, allowing them to generalize better and perform well on a wide range of tasks. Neural networks can also handle large amounts of data and are capable of learning from unstructured and high-dimensional data such as images, audio, and text. This makes them well-suited for tasks such as image classification, speech recognition, and natural language processing. Thus, ANNs give rise to the field of deep learning. The distinguishing peculiarity of neural networks from conventional machine learning algorithms is their ability to learn directly from the raw data. Machine learning techniques consist of a series of steps: pre-processing, feature extraction, feature selection, and prediction (classification or whatever task is at hand). The model's accuracy depends on the quality of these handcrafted features, and expert knowledge plays a crucial role in generalization. On the other hand, deep learning using neural networks can automatically learn/extract complex and hierarchical features through multiple layers of computation. This eliminates the need for manual feature engineering, as the model learns representations directly from the raw data. This concept is depicted in Fig. 1.1.

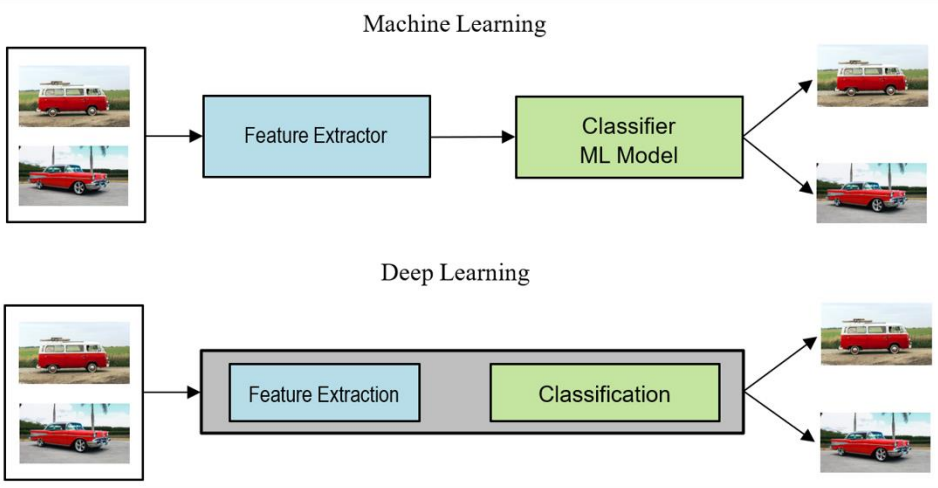


Fig. 1.1: Machine Learning versus Deep Learning: Binary classification of appropriately pre-processed images of a car and a van.

The input layer of a neural network receives the raw data, which is then passed through one or more hidden layers, where the neurons perform a weighted sum of their inputs, followed by the application of an activation function. The resulting output of each neuron is then passed to the next layer as input. By chaining multiple layers together, the network can learn to represent the input data at different levels of abstraction. The process of learning in a neural network involves adjusting the model parameters to minimize a loss function that measures the difference between the predicted output of the network and the target output. The learning is essentially an iterative optimization process involving the workhorse of the backpropagation method for finding the gradients of the loss with respect to the network's parameters (weights and biases). Neural networks can also incorporate regularization techniques to prevent overfitting or other learning issues and improve generalization.

One of the most influential ANN architectures for deep learning is the convolutional neural network (CNN) (Ali et al., 2023), which has been widely used in image and speech recognition tasks. CNNs are inspired by the organization of the visual cortex in the brain, with each layer of neurons processing increasingly complex features. Another important ANN architecture is the recurrent neural network (RNN), designed to handle sequential data (time series) such as text or speech. RNNs use feedback connections to enable the network to retain information over time, allowing them to model temporal dependencies in the data. In addition to the variants of these deterministic/supervised neural network architectures, the realm of generative neural network architectures is rapidly expanding, offering vast opportunities and applications in the field of generative AI.

To become an expert in deep learning with neural networks, firstly, a sound understanding of multivariate calculus, linear algebra, approximation theory, optimization, statistics, and probability is essential. Next, mathematical insights into the architectures and working of the neural networks are vital for innovative algorithm development. However, it is not uncommon for practitioners to use neural network models as a black box using pre-built tools, such as Keras (Chollet et al., 2015) and TensorFlow (Abadi et al., 2016). There are various newer and older well-reputed texts and articles available in the literature to explain the NN, including Bishop (2006), Hastie et al. (2016), Hagan et al. (2014), LeCun et al. (2015), Nielsen (2015), Schmidhuber (2015), Goodfellow et al. (2016), Vidal (2017), Aggarwal (2018), Marcus (2018), Higham & Higham (2019), Strang (2019), Dawani (2020), Sarker (2021), Kutyniok (2022), and Alexender (2023). However, an elaborative style of mathematical derivation of backpropagation with some visual representation was somewhat lacking. Moreover, a complete elaboration of the basic ANN algorithm (for both element-wise and vectorized implementation) in one place was lacking. The present article attempts to fill the gap and serve as a comprehensive yet compact premier for ML professionals and students to clearly understand the internal working of neural networks to achieve sustainable progress in the field.

This article provides a comprehensive overview of the mathematical and computational working of ANN, in particular for Feed-Forward Neural Networks (FNNs) without convolution or recurrent layers. Discussions about the neuron model, network architecture, forward pass, backpropagation, training process, regularization techniques to prevent overfitting, and optimization algorithms to improve the training process are presented. The article explains the underlying principles and mechanisms that drive the training and optimization of ANNs, from the basics of the neuron model to deep learning architectures. To illustrate the concepts, the article also presents pseudo codes of scalar and vectorized implementation for an ANN application for classification. The pseudo codes demonstrate the step-by-step process for implementing the method in a ready-to-code fashion. By providing intrinsic explanations and pseudo codes, we expect to help readers better understand the practical considerations crucial for success with ANNs.

The following sections explain the structure of the neural network, the forward pass, backpropagation, optimization algorithms used for computing the output, regularization techniques to prevent overfitting, and adaptive approaches in the optimization method. A detailed listing of the upcoming sections is as follows.

2. Schematic Structure of Simple Artificial Neural Networks (ANNs)

2.1. A Perceptron (A Single Neuron)

In Artificial Neural Networks (ANNs), a perceptron is a fundamental unit of computation that takes input values, multiplies them by corresponding weights, and applies a threshold function to produce an output. A perceptron is often referred to as an artificial neuron because it is modelled after the structure and function of a biological neuron, as depicted in Fig. 2.1.

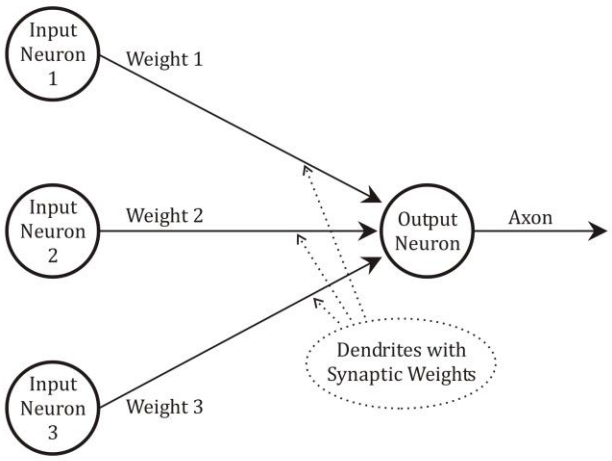


Fig. 2.1: A depiction of a perceptron architecture.

In a computational sense, a schematic diagram of a perceptron is shown in Fig. 2.2. The three input neurons (x_1, x_2, and x_3) together with the weights (w_1, w_2, and w_3) are used to create an intermediate value, z, of the output neuron as follows:

$$z = w_1x_1 + w_2x_2 + w_3x_3$$

The final value, $\hat{y} = \phi(z)$, at the output neuron is obtained by applying an appropriate non-linear activation function on z.

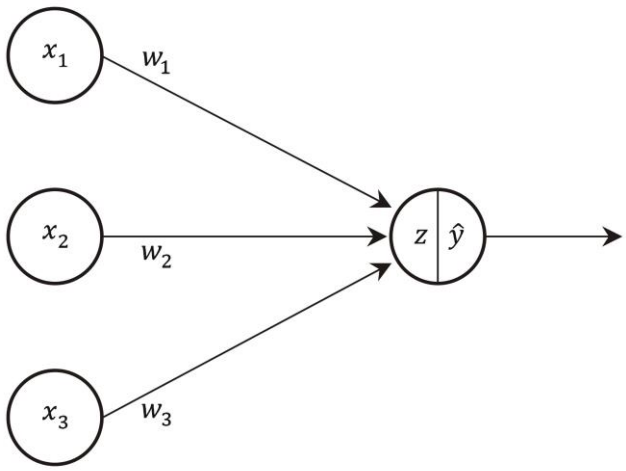


Fig. 2.2: A schematics diagram of a perceptron.

2.2. A Multi-Layer Perceptron (MLP) or a Feed-Forward Neural Network (FNN)

Multi-Layer Perceptron (MLP), specifically a simple Feed-Forward Neural Network (FNN), are the simplest of the ANNs. These are composed of multiple nodes, which imitate the biological neurons of the human brain. The neurons are arranged in multiple layers such that the neurons of consecutive layers are connected by links for passing on the values. A neuron receives values from the neurons in the previous layer and performs some operations to produce a new value, which is passed on to the neurons in the next layer. The input or output of each neuron is a numerical value, and the computation performed by the neuron is determined by a set of adjustable model parameters called ***weights*** and ***biases***.

Typically, a multilayer neural network has one ***input layer*** (numbered as, say $l = 0$) and L other layers, where L is a positive integer, such that the layers are numbered as $l = 1$ to $l = L$. Each of the layers numbered from $l = 1$ to $l = L - 1$ is called a ***hidden layer***, and the layer $l = L$ is called the ***output layer***. The input layer contains input neurons that send information to the hidden layer next to it. Each neuron in the input layer represents one feature or attribute of the input data. The hidden layer contains information from the input layer and performs some operations on the data. The hidden layers allow the ANN to learn and extract features from the input data that are not directly observable. In other words, hidden layers can identify complex patterns and relationships within the input data that can be used to make accurate predictions or classifications. The number of hidden layers and number of neurons in an ANN can vary, depending on the complexity of the problem being solved. Generally, the more complex the problem, the more hidden layers are needed to learn the necessary features and relationships in the

data. The complexity of the network increases with the number of layers and neurons in any layer. The output of the last hidden layer is then passed to an output layer for computing the final output of the network. The present article discusses mainly simple FNNs with no convolution or recurrent layers. We frequently call these as ANNs in the present document. Moreover, we confine the discussion in this document to supervised learning.

In a layer l, for $l = 1,2,3,\cdots,L$, the number of neurons can be denoted by n_l. In the subsequent discussion, the superscript $[l]$ is used to denote that the concerning value is for layer l. To form the values at the n_l neurons in a layer l, we define the notations as follows.

- The vector of input values at n_{l-1} neurons from the previous layer $(l-1)$ is denoted by

$$X^{[l-1]} = \begin{bmatrix} x_1^{[l-1]} \\ x_2^{[l-1]} \\ \vdots \\ x_{n_{l-1}}^{[l-1]} \end{bmatrix}_{n_{l-1}\times 1}$$

  Here, $X^{[l-1]} \in \mathbb{R}^{n_{l-1}\times 1}$.

- The vector of weights for generating the value at jth neuron (in the layer l), for $j = 1,2,\cdots,n_l$, are denoted by

$$W_j^{[l]} = \begin{bmatrix} w_{j,1}^{[l]} \\ w_{j,2}^{[l]} \\ \vdots \\ w_{j,n_{l-1}}^{[l]} \end{bmatrix}_{n_{l-1}\times 1}$$

  Here, $W_j^{[l]} \in \mathbb{R}^{n_{l-1}\times 1}$. A collective matrix of all the weights required to produce the n_l neurons is given by

$$\overline{\boldsymbol{W}}^{[l]} = \begin{bmatrix} w_{1,1}^{[l]} & w_{1,2}^{[l]} & \cdots & w_{1,n_{l-1}}^{[l]} \\ w_{2,1}^{[l]} & w_{2,2}^{[l]} & \cdots & w_{2,n_{l-1}}^{[l]} \\ & & \vdots & \\ w_{n_l,1}^{[l]} & w_{n_l,2}^{[l]} & \cdots & w_{n_l,n_{l-1}}^{[l]} \end{bmatrix} = \begin{bmatrix} -- & {W_1^{[l]}}^T & -- \\ -- & {W_2^{[l]}}^T & -- \\ & \vdots & \\ -- & {W_{n_l}^{[l]}}^T & -- \end{bmatrix}_{n_l\times(n_{l-1})}$$

  Here, $\overline{\boldsymbol{W}}^{[l]} \in \mathbb{R}^{n_l\times n_{l-1}}$.

- The vector of biases for generating the values at n_l neurons in the layer l, is denoted by

$$B^{[l]} = \begin{bmatrix} b_1^{[l]} \\ b_2^{[l]} \\ \vdots \\ b_{n_l}^{[l]} \end{bmatrix}_{n_l\times 1}$$

  Here, $B^{[l]} \in \mathbb{R}^{n_l\times 1}$.

- The vector $Z^{[l]} = \overline{\boldsymbol{W}}^{[l]} X^{[l-1]} + B^{[l]}$, or element-wise: $z_j^{[l]} = W_j^{[l]} \cdot X^{[l-1]} + b_j^{[l]}$, for $j = 1,2,\cdots,n_l$, for generating the values at n_l neurons in the layer l, is denoted by

$$Z^{[l]} = \begin{bmatrix} z_1^{[l]} \\ z_2^{[l]} \\ \vdots \\ z_{n_l}^{[l]} \end{bmatrix}_{n_l \times 1}$$

Here, $Z^{[l]} \in \mathbb{R}^{n_l \times 1}$.

- The vector $\hat{Y}^{[l]} = \boldsymbol{\phi}\left(Z^{[l]}\right)$, written component-wise: $\hat{y}_j^{[l]} = \phi\left(z_j^{[l]}\right)$, of output values generated at the n_l neurons in the layer l is denoted by

$$\hat{Y}^{[l]} = \begin{bmatrix} \hat{y}_1^{[l]} \\ \hat{y}_2^{[l]} \\ \vdots \\ \hat{y}_{n_l}^{[l]} \end{bmatrix}_{n_l \times 1}$$

Here, $\hat{Y}^{[l]} \in \mathbb{R}^{n_l \times 1}$. Note that the output from a layer $\mathrm{l}-1$ is the input to the layer l, i.e., $\hat{y}_i^{[l-1]} = x_i^{[l-1]}$.

- The vector of target/true values or labels at the n_L neurons in the output layer L is denoted by

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n_L} \end{bmatrix}_{n_L \times 1}$$

Here, $Y \in \mathbb{R}^{n_L \times 1}$.

A schematic diagram of an ANN architecture is given in Fig. 2.3. In general, any the value at jth neuron of the layer l is formed as:

$$\hat{y}_j^{[l]} = \phi\left(z_j^{[l]}\right) = \phi\left(W_j^{[l]} \cdot \hat{Y}^{[l-1]} + b_j^{[l]}\right)$$

2.3. Forward Pass: Finding Neuron Values in an Artificial Neural Network

Once a network architecture has been designed, by deciding the number of layers and number of neurons in each layer, also having initialized the network parameters (weights and biases), the computations across the network are performed. In the forward pass, the following three steps are performed.

Step 1 (Finding the Dot product of the Input Values and Weights):

Suppose that there are n_0 neurons in the input layer, where each neuron corresponds to an input feature of the dataset. The neuron values in the input layer are denoted by $x_i^{[0]}$, for $i = 1,2,\cdots,n_0$. These values form a vector, say $X^{[0]}$, which is used to compute n_1 neuron values of the first hidden layer $l = 1$. First,

for each neuron j, for $j = 1,2,\cdots,n_1$, of layer 1, the dot product of $X^{[0]}$ with the vector of the weights $W_j^{[1]}$, having components $w_{j,i}^{[1]}$ for $i = 1,2,\cdots,n_0$, is computed. In general, for each neuron j, for $j = 1,2,\cdots,n_l$, in a layer l, the dot product of $X^{[l-1]}$ with the vector of the weights $W_j^{[l]}$, having components $w_{j,i}^{[l]}$, for $i = 1,2,\cdots,n_{l-1}$, is computed. That said,

$$W_j^{[l]} \cdot X^{[l-1]} \quad = \quad X^{[l-1]} \cdot W_j^{[l]} \quad = \quad {W_j^{[l]}}^T X^{[l-1]} \quad = \quad \sum_{i=1}^{n_{l-1}} w_{j,i}^{[l]} \times x_i^{[l-1]} \qquad ---(F.1)$$

Here, l is the layer number: $l = 1,2,\cdots,L$. Note that the weight represents the strength of the connection between the neurons and decides how much influence the given input will have on the output.

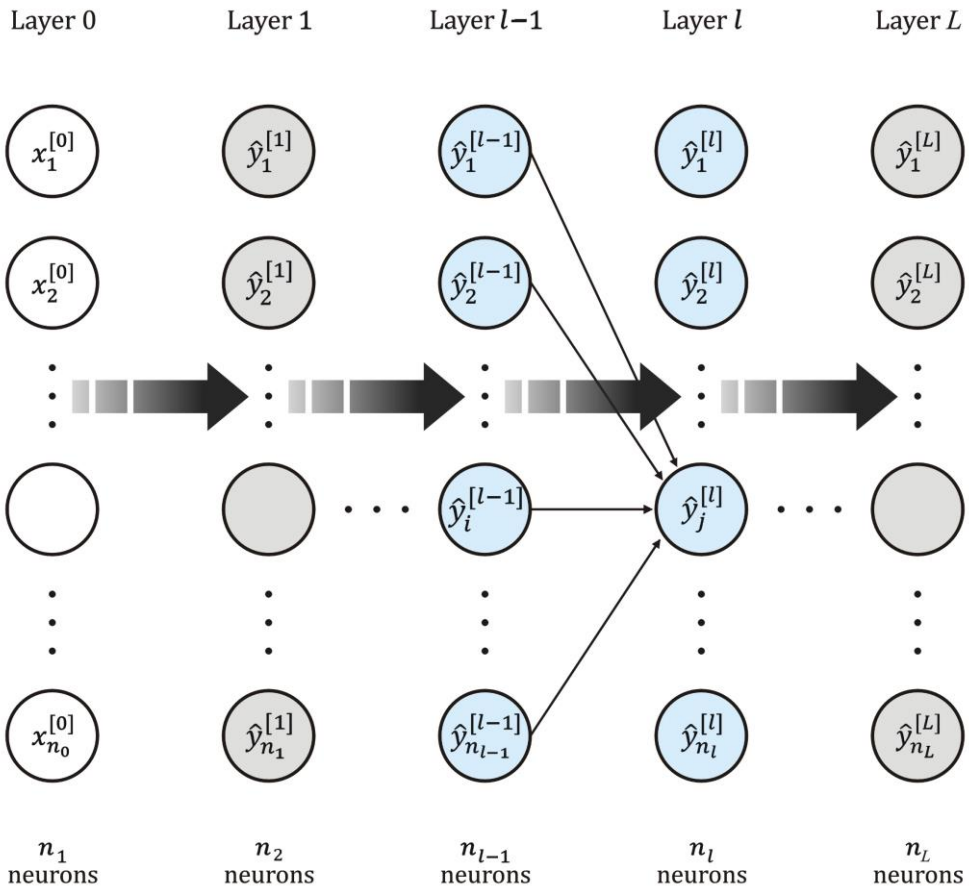


Fig. 2.3: A schematics diagram of a multilayer perceptron or FNN

Step 2 (Adding Biases):

Next, for each neuron j, for $j = 1,2,\cdots,n_l$, in the layer l, a bias $b_j^{[l]}$ is added to the dot product mentioned in $(F.1)$ to calculate the pre-activation/intermediate value, which represents a line. The result, denoted by $z_j^{[l]}$, can be expressed as:

$$z_j^{[l]} = \quad W_j^{[l]} \cdot X^{[l-1]} + \; b_j^{[l]} \quad = \quad \left(\sum_{i=1}^{n_{l-1}} w_{j,i}^{[l]} \times x_i^{[l-1]} \right) + b_j^{[l]}, \quad \text{for } j = 1,2,3, \cdots, n_l \quad (F.2)$$

The pre-activation value, $z_j^{[l]}$, the so-called linear neuron, is used to compute a non-linear output neuron. The bias is the offset, which is necessary in most cases, to move the pre-activation function or line to the left or right to generate the required output values. Using biases the neural network helps the model to make a better prediction by allowing the model to fit the training data more closely. It allows the model to have non-zero output when the input is zero. This is important in specific problems, such as image classification, where the input data may have zero mean. It can also be used to break the symmetry in the model, which can help with optimization during training. Without bias, multiple neurons could have the same weight and produce the same output, preventing the model from learning.

Step 3 (Applying Activation Function):

The jth neuron's output at layer l, i.e., $\hat{y}_j^{[l]}$, is obtained by applying the activation function $\phi: \mathbb{R} \to \mathbb{R}$ to the pre-activation value $z_j^{[l]}$:

$$\hat{y}_j^{[l]} \quad = \quad \phi\left(z_j^{[l]}\right) \qquad ---(F.3)$$

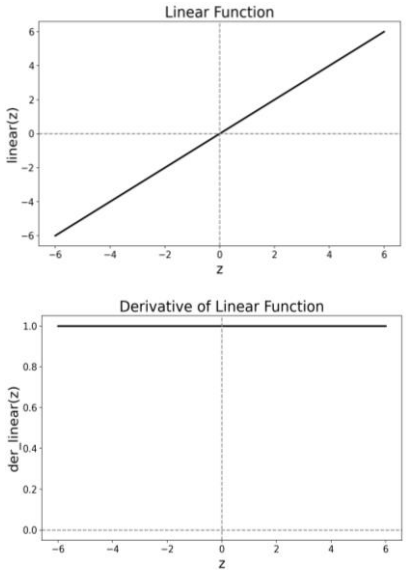
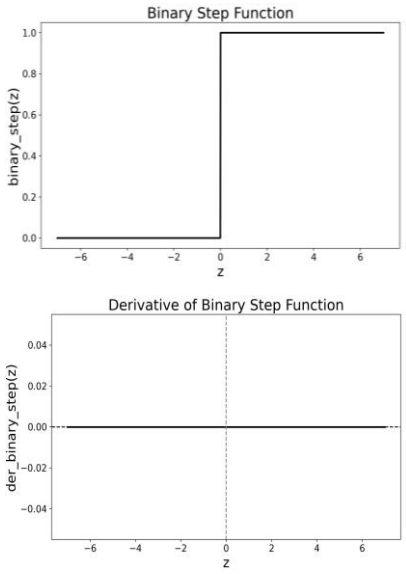
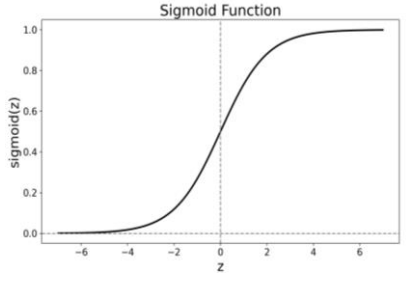
Conclusively, the jth artificial neuron in layer l with a vector of weights $W_j^{[l]} \in \mathbb{R}^{n_{l-1} \times 1}$, bias $b_j^{[l]} \in \mathbb{R}$ and an activation function $\phi: \mathbb{R} \to \mathbb{R}$ can be defined as a function $f: \mathbb{R}^{n_{l-1} \times 1} \to \mathbb{R}$

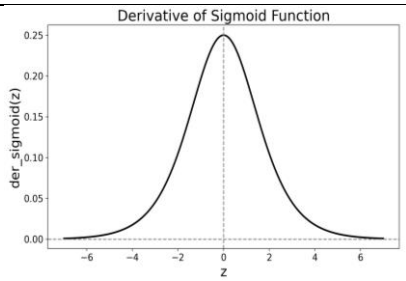
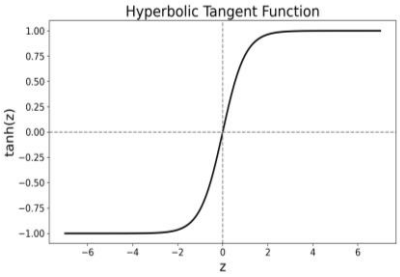
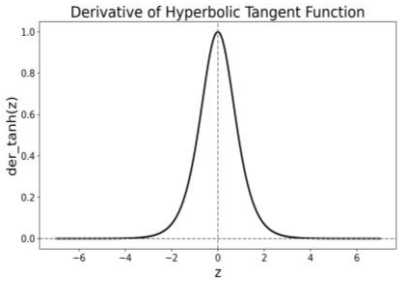
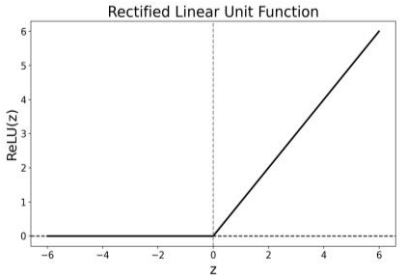
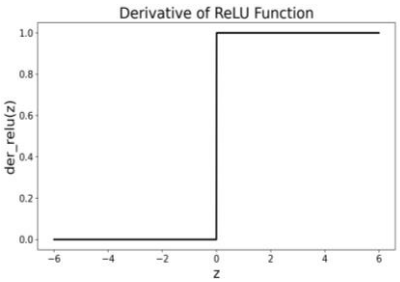
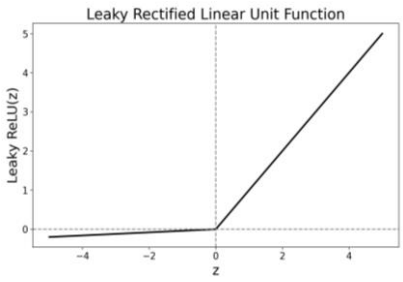
$$f\left(x_1^{[l-1]}, x_2^{[l-1]}, \cdots, x_{n_{l-1}}^{[l-1]}\right) \; = \; \phi\left(W_j^{[l]} \cdot X^{[l-1]} + \; b_j^{[l]}\right) \; = \; \phi\left(\left(\sum_{i=1}^{n_{l-1}} w_{j,i}^{[l]} \times x_i^{[l-1]}\right) + b_j^{[l]}\right)$$

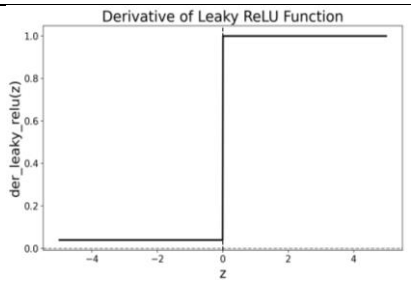
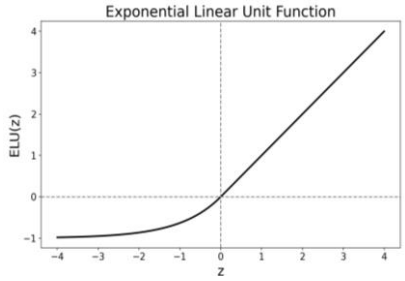
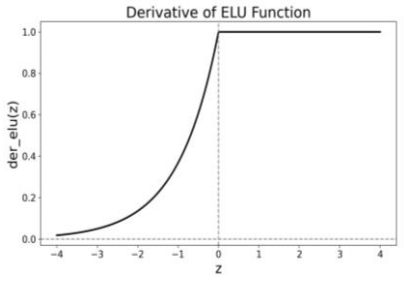
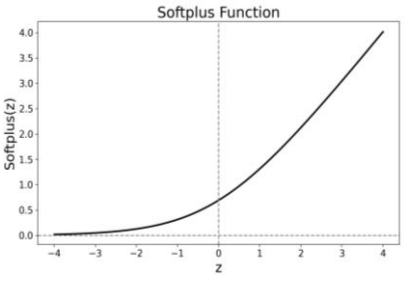
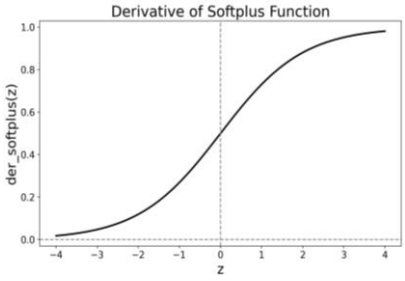
$$---(F.4)$$

An activation function, applied to the pre-activation linear neuron, introduces non-linearity into the neural network. This allows neural networks to learn complex and non-linear relationships between the input and output, which are inevitable in reality. Without non-linear activation functions, a neural network would act like a linear regression model that can only model linear relationships. Activation of a neuron determines whether the neuron should fire (i.e., output a signal) based on the input it receives from the previous layer. That said, the activation function decides whether the information the neuron receives is relevant or should be ignored.

Several definitions of the activation functions are available in the literature, each with its unique properties and advantages. Examples include the Threshold, Sigmoid, Tanh, ReLU, Leaky ReLU, Softmax functions, etc. Some of the well-known activation functions are listed in Table 2.1. The choice of activation function can have a significant impact on the performance of the model. The choice depends on the problem and data at hand and the characteristics of the data. Therefore, experimenting with different activation functions is often a good practice to see which works best. Research contributions proposed several other activation functions and demonstrated improved performance on some benchmarks. Agostinelli et al. (2015) proposed adaptive piecewise linear activation functions. Hendrycks and Gimpel (2020) proposed a high performance activation function, the Gaussian Error Linear Unit (GELU) as a variant to ReLU and ELU.

Function $\phi(z)$ and its derivative, $\phi'(z)$	**Description**	**Graph**
Linear/Identity Function: $\phi(z) = z$ $\phi'(z) = 1$ Ranges: $\phi(z) \in (-\infty, +\infty)$ $\phi'(z) \in \{1\}$	It produces an output that is directly proportional to its input. This means that for every unit change in the input, there is a corresponding unit change in the output. Due to its simplicity and linearity, it is rarely used in hidden layers of neural networks, as it lacks the expressive power to capture complex patterns. However, it finds utility in specific scenarios, such as regression tasks, where the output values are not bounded, and a direct, linear relationship between input and output is desired.	
Binary Step Function: $\phi(z) = \begin{cases} 0, & z < 0 \\ 1, & z \geq 0 \end{cases}$ $\phi'(z) = \begin{cases} 0, & z \neq 0 \\ \infty, & z = 0 \end{cases}$ Ranges: $\phi(z) \in \{0, 1\}$ $\phi'(z) \in \{0\}$	It maps its input to one of two discrete values: 0 or 1. Its gradient is nearly zero across the input space, which presents a challenge in parameter updates during training. Its binary nature makes it suitable for scenarios where a binary decision or activation is required, such as in binary classification problems. However, the near-zero gradient can impede gradient-based optimization methods, potentially leading to slow convergence or getting stuck in local minima. Therefore, it is often used sparingly in neural networks, with alternative activation functions preferred for tasks demanding smoother gradients and more efficient optimization.	
Sigmoid/ Logistic Function: $\phi(z) = \frac{1}{1 + e^{-z}}$ $\phi'(z) = \phi(z)\big(1 - \phi(z)\big)$ Ranges: $\phi(z) \in (0, 1)$ $\phi'(z) \in (0, 0.25]$	It transforms input values into a bounded range between 0 and 1, representing the likelihood or probability of belonging to a specific class in classification tasks. By compressing the output values into a probabilistic scale, it facilitates binary decisions, such as classifying objects as 'yes' or 'no.' Its smooth and differentiable nature allows for effective gradient-based optimization during neural network	

	training. However, its vanishing gradients can hinder learning in deep networks. Despite this limitation, the sigmoid function remains a fundamental choice for binary classification problems and as a component in more complex activation functions like the hyperbolic tangent. (Rumelhart et al., 1986)	
Hyperbolic Tangent (Tanh) Function: $\phi(z) = \frac{2}{1+e^{-2z}} - 1$ $\phi'(z) = 1 - \left(\phi(z)\right)^2$ Ranges: $\phi(z) \in (-1, 1)$ $\phi'(z) \in (0, 1]$	It is akin to the sigmoid function but extends its range from -1 to 1. This characteristic makes it suitable for a broad spectrum of tasks, including classification and regression. Like the sigmoid, tanh offers smooth and differentiable behavior, enabling efficient gradient-based optimization during neural network training. It is useful in scenarios where data varies both positively and negatively around a mean. Its centered nature, with a mean at 0, aids in mitigating vanishing gradient problems and contributes to its versatility in diverse network architectures. (Rumelhart et al., 1986)	 
Rectified Linear Unit (ReLU) Function: $\phi(z) = \max(0, z)$ $\phi'(z) = \begin{cases} 0, & z \leq 0 \\ 1, & z > 0 \end{cases}$ Ranges: $\phi(z) \in [0, \infty)$ $\phi'(z) \in \{0, 1\}$	It is known for its computational efficiency, making it a popular choice in neural networks. It introduces non-linearity by returning the input for positive values and zero for negative values (Hahnloser, 2000). ReLU may face the "dying ReLU" problem, where some neurons may become permanently inactive during training if they consistently output zero. This issue can be mitigated with variants like Leaky ReLU and Parametric ReLU, which introduce small slopes for negative inputs, ensuring gradients flow and maintaining the ability of neurons to learn effectively.	 
Leaky ReLU Function and Parametric ReLU Function: $\phi(z) = \max(\alpha z, z)$ $\phi'(z) = \begin{cases} \alpha, & z \leq 0 \\ 1, & z > 0 \end{cases}$ Here, α is a small positive constant,	These are designed to mitigate the "dying ReLU" issue. They allow small negative values to be passed through the activation function instead of being zero, preventing neuron inactivity. This enables parameter updates during backpropagation, even for negative inputs. This promotes more effective learning in deep neural	

For the Leaky-ReLU function, α is typically set to a small value like 0.01 or 0.2. For the Parametric-ReLU function, α is a learnable parameter that can be adjusted during training. Ranges: $\phi(z) \in (-\infty, +\infty)$ $\phi'(z) \in \{\alpha, 1\}$	networks. (Xu et al., 2015; He et al., 2015)	
Exponential Linear Unit (ELU) Function: $\phi(z) = \begin{cases} \alpha(e^{-z} - 1), z < 0 \\ z, \quad z \geq 0 \end{cases}$ $\phi'(z) = \begin{cases} \phi(z) + \alpha \ z \leq 0 \\ 1, \quad z > 0 \end{cases}$ Ranges: $\phi(z) \in (-\alpha, +\infty)$ $\phi'(z) \in (0, 1]$	It is an extension of the parametric ReLU that combines the benefits of ReLU with smoothness (Clevert, 2016). Like ReLU, it is computationally efficient and helps mitigate the vanishing gradient problem. However, it introduces smoothness to the activation output, making it more stable during optimization.	 
SoftPlus Function: $\phi(z) = \log(1 + e^{z})$ $\phi'(z) = \frac{1}{1 + e^{-z}}$ Ranges: $\phi(z) \in (0, \infty)$ $\phi'(z) \in (0, 1)$	It is a smooth and differentiable choice, well-suited for neural networks. It transforms input values into a positive range, offering benefits in terms of smooth gradients for optimization (Dugas et al., 2001). It is favored in scenarios where continuous and differentiable activations are crucial for network stability and training. Its output range spans from 0 to positive infinity, making it particularly useful for models requiring positive and unbounded outputs while maintaining differentiability.	 
SoftMax Function:	It is integral in the output layer of neural networks, particularly in multi-class classification tasks. Its role is pivotal in learning intricate decision boundaries. By producing a smooth and continuous output, it	

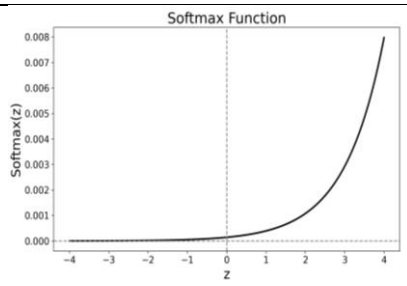
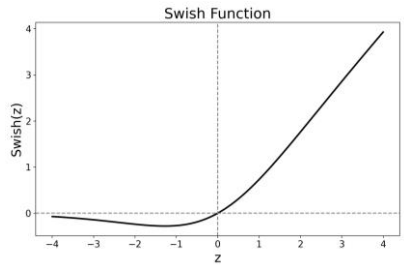
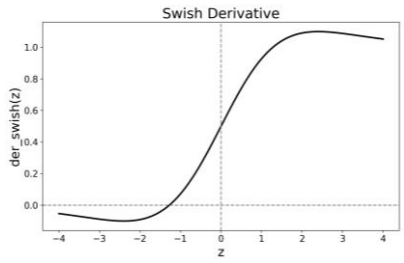
$\phi(z) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_i}} \in (0\,,1\,)$ $\phi'(z) = \phi(z_i)\left(\delta_{i,j} - \phi(z_j)\right)$ $\delta_{i,j} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$	fosters stable and robust training processes. It transforms raw scores or logits into class probabilities, allowing the network to assign a probability to each class. The class with the highest probability is selected as the final prediction (Bishop, 2006). Its output is well-suited for scenarios where data belongs to one of several mutually exclusive classes, making it a fundamental component of neural network architectures for classification.	
Swish Function: $\phi(z) = z \cdot \text{sigmoid}(z)$ $\phi'(z) = \phi(z)$ $+ \text{sigmoid}(z)\big(1 - \phi(z)\big)$ Range: $\phi(z) \in [-0.278, \infty\,)$ $\phi'(z) \in [-0.0998, 1.0998]$	it is a smooth and continuously differentiable activation function that exhibits a unique non-monotonic behavior, effectively addressing the vanishing gradient problem in deep neural networks. Bounded within the range of $(-1, \infty)$, it offers robust protection against the exploding gradient problem (Ramachandran, 2017). Swish has gained prominence for its potential to accelerate convergence and enhance generalization in neural network models.	 

Table 2.1: Some commonly used activation functions

Now, we present the expressions using vector notations for generating values for the whole of the layer l. All the dot products of input values and the weights collectively for each neuron in layer l can be expressed as:

$$\overline{\boldsymbol{W}}^{[l]} X^{[l-1]}$$

Adding the biases into it, the pre-activation values for n_l neuron in the layer l are expressed in vector form as:

$$Z^{[l]} \;=\; \overline{\boldsymbol{W}}^{[l]} X^{[l-1]} + B^{[l]} \qquad ---(F.5)$$

Here, $\overline{\boldsymbol{W}}^{[l]} \in \mathbb{R}^{n_{l-1} \times n_l}$ is the weight matrix and $B^{[l]} \in \mathbb{R}^{n_{l-1} \times 1}$ is the vector of bias of layer l. The vector $Y^{[l]}$ of n_l neurons in the layer l is obtained through application of activation function:

$$\hat{Y}^{[l]} \;=\; \boldsymbol{\phi}\big(Z^{[l]}\big) \in \mathbb{R}^{n_l \times 1} \qquad ---(F.6)$$

Here, $\boldsymbol{\phi}$ is in the vector form. The computations across the whole network produce the predicted output, reaching the layer L. This constitutes a neural network model as a computational framework. Interestingly, the whole neural network architecture can be expressed as the function $\boldsymbol{\Phi}$: $\mathbb{R}^{n_0 \times 1} \rightarrow \mathbb{R}^{n_L \times 1}$ that maps the input data to the predicted output:

$$\boldsymbol{\Phi}\left(X^{[0]}\right) \;=\; \boldsymbol{\phi}\left(Z^{[L]}\boldsymbol{\phi}\left(Z^{[L-1]}\boldsymbol{\phi}\left(\cdots\boldsymbol{\phi}\left(Z^{[1]}\right)\right)\right)\right) \qquad ---(F.7)$$

Here, $X^{[0]} \in \mathbb{R}^{n_0 \times 1}$.

The function $\boldsymbol{\Phi}$ can serve as a universal function approximator means that, theoretically, a neural network with a sufficiently large number of hidden neurons or layers can approximate any continuous function, given enough training data and proper optimization. In other words, neural networks have the capacity to learn and represent a wide range of complex functions, making them versatile tools for various machine learning tasks. This concept is based on the Universal Approximation Theorem (Cybenko, 1989). The theorem essentially states that a feedforward neural network with a single hidden layer containing a finite number of neurons can approximate any continuous function on a closed and bounded input domain, provided that the network has the appropriate activation function and given sufficient data and proper training. Some more descriptions about the concept of the universal function approximator can be found in (Hornik, 1991; Mhaskar et al., 2016; Zhou, 2018; Rolnick and Tegmark, 2018; Strang, 2019).

2.4. Calculating the Cost Function (Loss Function)

A ***Loss function*** for the neural network is an important parameter determining how well a model performs for a given instance. The average of the loss function values for all the training instances under consideration (in an iteration of the learning process) is called the ***Cost function***. However, in machine learning literature, using the term loss function and cost function interchangeably is common. The loss function gives a scalar value as the difference/distance/norm between the vector of labels or target values $Y(p)$ and the vector of predicted output values $\hat{Y}^{[L]}(p)$ in the output layer L of the network for an instance p. The cost function is obtained as the average of the loss functions for all the training instances in an iteration of the learning process. The learning of the model occurs in the form of optimizing the network parameters (weights and biases). The neural network, together with optimal parameters, predicts the output for a given input more accurately. The parameters are modified carefully in a mathematically justified way to reduce the error or cost function. To find an update of the network parameters for optimization, the gradients of the cost function (i.e., the average of the gradients of the loss functions) with respect to each parameter are used. The iterative process of updating the parameters continues until a permissible prediction error is obtained. Section 3 discusses how to find the gradients of the cost function. Section 4 discusses using the gradients in an optimization method that updates the network parameters.

There are several loss/cost functions in literature, such as the Squared Error (or Quadratic Loss), Mean-Squared Error (MSE), Mean-Absolute Error (MAE), Binary Cross-Entropy, Multi-Class Cross-Entropy, Hinge Loss, Huber Loss, Focal Loss, etc. Table 2.2 lists some commonly used definitions of the cost functions for neural networks. The choice of the loss function depends on the problem at hand, the characteristics of the available dataset, and the desired properties of the solution.

Suppose that there is a sample of m instances under consideration (for an iteration of the model training process) from a population (dataset) of N instances. The loss function for the pth instance, where $p = 1, 2, 3, \cdots, m$, can be denoted by $c^{(p)}$ or c_p. If there are n_L neurons in the output layer L, then the Quadratic Loss function or MSE is expressed in element-wise form as,

$$c_p \quad = \quad \frac{1}{2}\sum_{j=1}^{n_L}\left(y_j - \hat{y}_j^{[L]}\right)^2 \qquad ---(B.1a)$$

In vector-notation,

$$c_p \quad = \quad \frac{1}{2}\left\|Y(p) - \hat{Y}^{[L]}(p)\right\|_2^2 \qquad ---(B.1b)$$

The Binary Cross-Entropy Loss function is given by

$$c_p \quad = \quad -\sum_{j=1}^{n_L}\left[y_j \ln \hat{y}_j^{[L]} + \left(1 - y_j\right)\ln\left(1 - \hat{y}_j^{[L]}\right)\right] \qquad ---(B.2a)$$

Note that each $\hat{y}_j^{[L]} \in (0,1)$, for $j = 1,2,\cdots,n_L$ is a probability (possibly estimated using the Sigmoid function) for instance p to belong to a particular class. The instance p will belong to the class corresponding to the Kth neuron (Kth component in the vector of the target classes) such that

$$\hat{y}_K^{[L]} \quad = \quad \max_{1\le j\le n_L}\left(\hat{y}_j^{[L]}\right)$$

In vector form,

$$c_p \quad = \quad -\left[Y(p)\cdot \ln\left(\hat{Y}^{[L]}(p)\right) + \left(I - Y(p)\right)\cdot \ln\left(I - \hat{Y}^{[L]}(p)\right)\right] \qquad ---(B.2b)$$

Here I is an $n_L \times 1$ vector with all values as 1.

The cost function (average of the loss function values for m training instances) can be calculated as:

$$C \quad = \quad \frac{1}{m}\sum_{p=1}^{m} c_p \qquad ---(B.3)$$

Cost Function, $E\left(Y^{(p)}, \hat{Y}^{[L](p)}\right)$	**Description**
Quadratic Cost or MSE (Mean Squared Error): $\frac{1}{2m}\sum_{p=1}^{m}\left\|Y^{(p)} - \hat{Y}^{[L](p)}\right\|_2^2$ $= \frac{1}{2m}\sum_{p=1}^{m}\sum_{j=1}^{n_L}\left(y_j^{(p)} - \hat{y}_j^{[L](p)}\right)^2$	The quadratic cost function, also known as mean squared error (MSE), is commonly used for regression problems. It calculates the average of the squared differences between the predicted and target values. The advantage of MSE is its differentiability, making it suitable for gradient-based optimization algorithms. It also emphasizes larger errors, encouraging accurate predictions. However, it is sensitive to outliers; the squared difference will be much larger than the squared differences of the other data point, which can significantly increase the cost. Moreover, the parabolic shape of MSE near the minimum can cause the derivative to approach zero, leading to slow convergence and requiring more iterations for the weights and biases to reach optimal values.
RMSE (Root Mean Squared Error):	RMSE stands for Root Mean Squared Error. It is a commonly used metric in regression analysis to measure the average deviation between predicted and actual values. The RMSE is calculated by taking the square root of MSE. By squaring the differences, RMSE

$$\sqrt{\frac{1}{2m}\sum_{p=1}^{m}\left\|Y^{(p)}-\hat{Y}^{[L](p)}\right\|_2^2}$$ $$=\sqrt{\frac{1}{2m}\sum_{p=1}^{m}\sum_{j=1}^{n_L}\left(y_j^{(p)}-\hat{y}_j^{[L](p)}\right)^2}$$	emphasizes larger errors and provides a measure of the overall accuracy of the predictions. The RMSE is particularly useful when the magnitude of errors is a concern and when comparing the performance of different models.
Mean Absolute Error (MAE): $$\frac{1}{m}\sum_{p=1}^{m}\left\|Y^{(p)}-\hat{Y}^{[L](p)}\right\|_1$$ $$=\frac{1}{m}\sum_{p=1}^{m}\sum_{j=1}^{n_L}\left\|y_j^{(p)}-\hat{y}_j^{[L](p)}\right\|$$	MAE measures the average absolute difference between the predicted and target values in regression problems. It measures the error, regardless of their direction (positive or negative). MAE is preferred over MSE when the dataset contains significant outliers, and minimizing the impact of outliers is crucial. In MSE, the squared difference for an outlier may be much larger than that for the other data points, which can significantly impact the overall error.
Binary Cross-Entropy: $$-\frac{1}{m}\sum_{p=1}^{m}\left[Y^{(p)}\ln\left(\hat{Y}^{[L](p)}\right)+\left(I-Y^{(p)}\right)\ln\left(I-\hat{Y}^{[L](p)}\right)\right]$$ $$=-\frac{1}{m}\sum_{p=1}^{m}\sum_{j=1}^{n_L}\left[y_j^{(p)}\ln\left(\hat{y}_j^{[L](p)}\right)+\left(1-y_j^{(p)}\right)\ln\left(1-\hat{y}_j^{[L](p)}\right)\right]$$	The Binary Cross-Entropy function, also known as Log Loss, is typically used for binary classification tasks. Its formula penalizes the model based on the divergence between the predicted probabilities and the true labels for the two classes. It encourages the model to assign high probabilities to the correct class and low probabilities to the other classes. It is preferred over quadratic loss function in classification problems because it directly optimizes the likelihood of the correct class prediction.
Categorical or Multi-class Cross-Entropy: $$-\frac{1}{m}\sum_{p=1}^{m}\left[Y^{(p)}\ln\left(\hat{Y}^{[L](p)}\right)\right]$$ $$=-\frac{1}{m}\sum_{p=1}^{m}\sum_{j=1}^{n_L}\left[y_j^{(p)}\ln\left(\hat{y}_j^{[L](p)}\right)\right]$$	The Categorical or Multi-class Cross-Entropy function is used for multi-class classification problems. It captures the contribution of each class to the overall cost and encourages the model to assign higher probabilities to the true class and lower probabilities to the other classes.
Hinge Loss: $$\frac{1}{m}\sum_{p=1}^{m}\max\left(0,1-\hat{Y}^{[L](p)}\cdot Y^{(p)}\right)$$ $$=\frac{1}{m}\sum_{p=1}^{m}\max_{1\le j\le n_L}\left(0,1-\hat{y}_j^{[L](p)}y_j^{(p)}\right)$$	The Hinge loss function is commonly used in support vector machines (SVMs) for binary classification tasks. When dealing with linearly separable datasets, it is advantageous and aims to find a decision boundary that maximizes the margin while correctly classifying the training samples. The Hinge loss penalizes misclassifications, assigning a non-zero loss only when a sample is on the wrong side of the decision boundary. The loss increases linearly with the magnitude of the incorrect prediction, encouraging the model to classify samples with a margin of at least one correctly. Unlike the other loss functions, such as the Binary Cross-Entropy, the Hinge loss does not directly optimize the predicted probabilities but focuses on maximizing the margin between the classes.

Table 2.2: Some commonly used loss/cost functions for neural networks.

3. Derivation of the Expressions for the Gradients using the Backpropagation

The learning phase of a neural network model consists of two phases: backpropagation and optimization. The backpropagation is a well-accepted and crucial algorithm for finding the gradients of the cost function with respect to the model parameters (weights and biases). The gradients determine how each of the parameters affects the cost function and hence the overall behaviour of the model. The gradients are needed for the optimization algorithm that modifies the weights and biases. The parameter modification attempts to minimize the cost function, thus giving rise to the learning of the model.

We derive expressions for finding the gradients of the cost function $c^{(p)}$or c_p for a training instance p. Unless otherwise needed, we omit (p) from the subscript of the notations, as it is considered understood that we are discussing the quantities relevant to an instance p.

In backpropagation, first, the gradients of c_p with respect to each of the weights $w_{j,i}^{[L]}$, $i = 1,2,3,\cdots,n_{L-1}$, and biases $b_j^{[L]}$ corresponding to each neuron j, $j = 1,2,3,\cdots,n_L$, in the layer L (the output layer), are obtained. Note that c_p does not depend directly on the weights and biases; rather, it depends on the corresponding neuron value in the layer L. Therefore, a gradient of c_p involves the use of the chain rule of differentiation.

Next, the gradient of c_p with respect to each of the weights $w_{k,i}^{[L-1]}$, $i = 1,2,3,\cdots,n_{L-2}$, and biases $b_k^{[L-1]}$ corresponding to each neuron k, $k = 1,2,3,\cdots,n_{L-1}$, in the layer $(L-1)$, are obtained. Again, note that c_p depends on neither the said weights and biases, nor the neuron values in the layer $L-1$, directly. Therefore, for the gradient of c_p, multiple applications of the chain rule are involved recursively. A similar approach is used for the gradients of c_p with respect to the weights and biases corresponding to the neurons in the previous layers. In general, the gradients of c_p with respect to each of the weights $w_{q,i}^{[r]}$, $i = 1,2,3,\cdots,n_{r-1}$, and biases $b_q^{[l]}$ corresponding to each neuron q, $q = 1,2,3,\cdots,n_r$ in each hidden layer r, for $r = L-1, L-2,\cdots,1$, are obtained with multiple applications of the chain rule recursively, as these parameters affect the neuron values in the current layer r and the subsequent layers, $r+1, r+2,\cdots,L$.

The expressions for the gradient of the cost function with respect to the weights and biases are derived below.

The gradient of c_p with respect to the weights and biases used for the neurons of the layer L:

Let's derive the expressions the gradients of c_p with respect to the weights $w_{j,i}^{[L]}$ and biases $b_j^{[L]}$. Since c_p is not directly dependent on $w_{j,i}^{[L]}$ and $b_j^{[L]}$, let's use the chain rule for the gradients of c_p:

$$\frac{\partial c_p}{\partial w_{j,i}^{[L]}} = \frac{\partial c_p}{\partial \hat{y}_j^{[L]}} \times \frac{\partial \hat{y}_j^{[L]}}{\partial z_j^{[L]}} \times \frac{\partial z_j^{[L]}}{\partial w_{j,i}^{[L]}}, \quad j = 1,2,3,\cdots,n_L \quad i = 1,2,3,\cdots,n_{L-1} \qquad ---(B.4)$$

For Eq. $(B.4)$, we need to find the following three gradients,

$$\frac{\partial c_p}{\partial \hat{y}_j^{[L]}} = ? \qquad \frac{\partial \hat{y}_j^{[L]}}{\partial z_j^{[L]}} = ? \qquad \frac{\partial z_j^{[L]}}{\partial w_{j,i}^{[L]}} = ?$$

The gradient of the predicted values $\hat{y}_j^{[L]}$ with respect to $z_j^{[L]}$ can be written as

$$\frac{\partial \hat{y}_j^{[L]}}{\partial z_j^{[L]}} = \frac{\partial}{\partial z_j^{[L]}} \phi\left(z_j^{[L]}\right) = \phi'\left(z_j^{[L]}\right) \quad ---(B.5)$$

The gradient of $z_j^{[L]}$ with respect to $w_{j,i}^{[L]}$ is given by

$$\frac{\partial z_j^{[L]}}{\partial w_{j,i}^{[L]}} = \frac{\partial}{\partial w_{j,i}^{[L]}}\left(z_j^{[L]}\right) = \frac{\partial}{\partial w_{j,i}^{[L]}}\left[\sum_{i=1}^{n_{L-1}}\left(x_i^{[L-1]} \cdot w_{j,i}^{[L]}\right) + b_j^{[L]}\right] = x_i^{[L-1]} = \hat{y}_i^{[L-1]} \quad ---(B.6)$$

Therefore, using Eqs. $(B.5 - B.6)$ in Eq. $(B.4)$, the gradient of c_p with respect to the weights $w_{j,i}^{[L]}$ can be expressed as (for each $j = 1,2,3,\cdots,n_L$ vary $i = 1,2,3,\cdots,n_{L-1}$):

$$\frac{\partial c_p}{\partial w_{j,i}^{[L]}} = \frac{\partial c_p}{\partial \hat{y}_j^{[L]}} \times \phi'\left(z_j^{[L]}\right) \times x_i^{[L-1]} \quad ---(B.7)$$

Here, we consider the quantity,

$$\delta_j^{[l]} = \frac{\partial c_p}{\partial z_j^{[l]}} \quad ---(B.8)$$

$\delta_j^{[l]}$ is error in the jth neuron at layer l (Higham and Higham, 2019). It tells how the change in the weighted input to the neuron effects the cost function. We will use it to define recursive expressions for simplicity in the derivation. For the output layer $l = L$, we have

$$\delta_j^{[L]} = \frac{\partial c_p}{\partial z_j^{[L]}} = \frac{\partial c_p}{\partial \hat{y}_j^{[L]}} \times \frac{\partial \hat{y}_j^{[L]}}{\partial z_j^{[L]}}$$

$$\delta_j^{[L]} = \frac{\partial c_p}{\partial \hat{y}_j^{[L]}} \times \phi'\left(z_j^{[L]}\right) \quad ---(B.9)$$

Then, Eq. $(B.7)$, becomes,

$$\frac{\partial c_p}{\partial w_{j,i}^{[L]}} = \delta_j^{[L]} \times x_i^{[L-1]} \quad ---(B.10)$$

Similarly, the gradient of c_p with respect to $b_j^{[L]}$ is computed as:

$$\frac{\partial c_p}{\partial b_j^{[L]}} = \frac{\partial c_p}{\partial \hat{y}_j^{[L]}} \times \frac{\partial \hat{y}_j^{[L]}}{\partial z_j^{[L]}} \times \frac{\partial z_j^{[L]}}{\partial b_j^{[L]}} \qquad j = 1,2,3,\cdots,n_L \quad ---(B.11)$$

The gradient of $z_j^{[L]}$ with respect to the $b_j^{[L]}$ is given by,

$$\frac{\partial z_j^{[L]}}{\partial b_j^{[L]}} = \frac{\partial}{\partial b_j^{[L]}}\left(z_j^{[L]}\right) = \frac{\partial}{\partial b_j^{[L]}}\left[\sum_{i=1}^{n_{L-1}}\left(x_i^{[L-1]} \cdot w_{j,i}^{[L]}\right) + b_j^{[L]}\right] = 1 \quad ---(B.12)$$

Using Eqs. $(B.9)$ and $(B.12)$ in Eq. $(B.11)$, we get (for $j = 1,2,3,\cdots,n_L$):

$$\frac{\partial c_p}{\partial b_j^{[L]}} = \delta_j^{[L]} \times 1 \qquad ---(B.13)$$

If we consider the Quadratic cost function, the gradient of c_p with respect to $\hat{y}_j^{[L]}$ is obtained as follows:

$$\frac{\partial c_p}{\partial \hat{y}_j^{[L]}} = \frac{\partial}{\partial \hat{y}_j^{[L]}} \frac{1}{2} \sum_{j=1}^{n_L} \left(y_j - \hat{y}_j^{[L]}\right)^2 \qquad \because \text{using Eq. } (B.1b)$$

$$\frac{\partial c_p}{\partial \hat{y}_j^{[L]}} = \frac{1}{2} \cdot 2 \left(y_j - \hat{y}_j^{[L]}\right)(-1)$$

$$\frac{\partial c_p}{\partial \hat{y}_j^{[L]}} = \hat{y}_j^{[L]} - y_j \qquad ---(B.14)$$

If we consider the Binary Cross-Entropy function, the gradient of c_p with respect to $\hat{y}_j^{[L]}$ is obtained as follows:

$$\frac{\partial c_p}{\partial \hat{y}_j^{[L]}} = -\frac{\partial}{\partial \hat{y}_j^{[l]}} \left[y_j \ln \hat{y}_j^{[L]} + (1 - y_j) \ln \left(1 - \hat{y}_j^{[L]}\right)\right] \qquad \because \text{using Eq. } (B.2b)$$

$$= -\left[y_j \frac{\partial}{\partial \hat{y}_j^{[L]}} \left(\ln \hat{y}_j^{[L]}\right) + (1 - y_j) \frac{\partial}{\partial \hat{y}_j^{[L]}} \ln \left(1 - \hat{y}_j^{[L]}\right)\right]$$

$$= -\left[\frac{y_j}{\hat{y}_j^{[L]}} + \frac{(1 - y_j)}{\left(1 - \hat{y}_j^{[L]}\right)}(-1)\right] = -\left[\frac{y_j}{\hat{y}_j^{[L]}} - \frac{(1 - y_j)}{\left(1 - \hat{y}_j^{[L]}\right)}\right] = \frac{(1 - y_j)}{\left(1 - \hat{y}_j^{[L]}\right)} - \frac{y_j}{\hat{y}_j^{[L]}}$$

$$\frac{\partial c_p}{\partial \hat{y}_j^{[L]}} = \frac{\hat{y}_j^{[L]} - y_j}{\left(1 - \hat{y}_j^{[L]}\right)\hat{y}_j^{[L]}} \qquad ---(B.15)$$

Therefore, for the Quadratic cost function, Eq. $(B.9)$ becomes,

$$\delta_j^{[L]} = \left(\hat{y}_j^{[L]} - y_j\right) \times \phi'\left(z_j^{[L]}\right) \qquad ---(B.16)$$

For the Binary Cross-Entropy function, Eq. $(B.9)$ becomes,

$$\delta_j^{[L]} = \frac{\hat{y}_j^{[L]} - y_j}{\left(1 - \hat{y}_j^{[L]}\right)\hat{y}_j^{[L]}} \times \phi'\left(z_j^{[L]}\right) \qquad ---(B.17)$$

If the Sigmoid function is used for the activation, then

$$\phi'\left(z_j^{[L]}\right) = \frac{\partial}{\partial z_j^{[L]}} \phi\left(z_j^{[L]}\right) = \frac{\partial}{\partial z_j^{[L]}} \left(\frac{1}{1 + e^{-z_j^{[L]}}}\right)$$

$$= \frac{e^{-z_k^{[L]}}}{\left(1 + e^{-z_j^{[L]}}\right)^2} = \frac{1}{1 + e^{-z_j^{[L]}}} \times \frac{e^{-z_k^{[L]}}}{1 + e^{-z_j^{[L]}}} = \frac{1}{1 + e^{-z_j^{[L]}}} \times \left(1 - \frac{1}{1 + e^{-z_j^{[L]}}}\right)$$

$$= \phi\left(z_j^{[L]}\right)\left(1 - \phi\left(z_j^{[L]}\right)\right)$$

$$\phi'\left(z_j^{[L]}\right) \;=\; \hat{y}_j^{[L]}\left(1-\hat{y}_j^{[L]}\right) \qquad ---(B.18)$$

Similarly, if the ReLU function is used for the activation, then

$$\phi'\left(z_j^{[L]}\right) \;=\; \frac{\partial}{\partial z_j^{[L]}}\phi\left(z_j^{[L]}\right) \;=\; \frac{\partial}{\partial z_j^{[L]}}\left(\max\left(0, z_j^{[l]}\right)\right) \;=\; \begin{cases} 0, & z_j^{[l]} \le 0 \\ 1, & z_j^{[l]} > 0 \end{cases} \qquad (B.19)$$

An appropriate expression for $\phi'\left(z_j^{[L]}\right)$, such as in Eq. $(B.18)$ or Eq. $(B.19)$, in an appropriate definition of $\delta_j^{[L]}$, such as in Eq. $(B.16)$ or Eq. $(B.17)$, can be used. The result, in turn, can be used in Eq. $(B.10)$ and Eq. $(B.13)$ to obtain specific expressions for the gradients of c_p with respect to $w_{j,i}^{[L]}$ and $b_j^{[L]}$.

For the neurons in the hidden layer $L-1$:

Let's find the expressions for the gradient of the cost function c_p with respect to the weights $w_{k,i}^{[L-1]}$ and biases $b_k^{[L-1]}$ used for the neuron in the hidden layer $L-1$. Since c_p is not directly dependent on $w_{k,i}^{[L-1]}$, let's use the chain rule:

$$\frac{\partial c_p}{\partial w_{k,i}^{[L-1]}} \;=\; \frac{\partial c_p}{\partial \hat{y}_k^{[L-1]}} \times \frac{\partial \hat{y}_k^{[L-1]}}{\partial z_k^{[L-1]}} \times \frac{\partial z_k^{[L-1]}}{\partial w_{k,i}^{[L-1]}}, \quad k = 1,2,3,\cdots,n_{L-1} \quad i = 1,2,3,\cdots,n_{L-2}$$

$$---(B.20)$$

Similar to Eq. $(B.8)$, we have

$$\delta_k^{[L-1]} \;=\; \frac{\partial c_p}{\partial z_k^{[L-1]}} \;=\; \frac{\partial c_p}{\partial \hat{y}_k^{[L-1]}} \times \frac{\partial \hat{y}_k^{[L-1]}}{\partial z_k^{[L-1]}} \qquad ---(B.21)$$

Next

$$\frac{\partial z_k^{[L-1]}}{\partial w_{k,i}^{[L-1]}} \;=\; \frac{\partial}{\partial w_{k,i}^{[L]}}\left(z_k^{[L-1]}\right) \;=\; \frac{\partial}{\partial w_{k,i}^{[L]}}\left[\sum_{i=1}^{n_{L-2}}\left(x_i^{[L-2]} \cdot w_{k,i}^{[L-1]}\right) + b_k^{[L-1]}\right] \;=\; x_i^{[L-2]}$$

$$---(B.22)$$

Using Eqs. $(B.21)$ and$(B.22)$, in Eq. $(B.20)$ gives

$$\frac{\partial c_p}{\partial w_{k,i}^{[L-1]}} \;=\; \delta_k^{[L-1]} \times x_i^{[L-2]}, \quad k = 1,2,3,\cdots,n_{L-1} \quad i = 1,2,3,\cdots,n_{L-2} \qquad ---(B.23)$$

Recall that the output from the layer $L-1$ is the input to the layer L, i.e., $\hat{y}_k^{[L-1]} = x_k^{[L-1]}$. For Eq. $(B.21)$, we find $\frac{\partial c_p}{\partial \hat{y}_k^{[L-1]}}$ and $\frac{\partial \hat{y}_k^{[L-1]}}{\partial z_k^{[L-1]}}$.

Let's find the gradient of c_p with respect to $\hat{y}_k^{[L-1]}$ using the chain rule:

$$\frac{\partial c_p}{\partial \hat{y}_k^{[L-1]}} = \sum_{j=1}^{n_L} \frac{\partial c_p}{\partial \hat{y}_j^{[L]}} \times \frac{\partial \hat{y}_j^{[L]}}{\partial y_k^{[L-1]}}$$

$$\frac{\partial c_p}{\partial \hat{y}_k^{[L-1]}} = \sum_{j=1}^{n_L} \frac{\partial c_p}{\partial \hat{y}_j^{[L]}} \times \frac{\partial \hat{y}_j^{[L]}}{\partial z_j^{[L]}} \times \frac{\partial z_j^{[L]}}{\partial \hat{y}_k^{[L-1]}}$$

$$\frac{\partial c_p}{\partial \hat{y}_k^{[L-1]}} = \sum_{j=1}^{n_L} \delta_j^{[L]} \times \frac{\partial z_j^{[L]}}{\partial \hat{y}_k^{[L-1]}} \qquad \because \text{using Eq. } (B.9) \qquad ---(B.24)$$

Since

$$\frac{\partial z_j^{[L]}}{\partial \hat{y}_k^{[L-1]}} = \frac{\partial}{\partial \hat{y}_k^{[L-1]}}\left(z_j^{[L]}\right) = \frac{\partial}{\partial \hat{y}_k^{[L-1]}}\left[\sum_{k=1}^{n_{L-1}} \left(x_k^{[L-1]} \cdot w_{j,k}^{[L]}\right) + b_j^{[L]}\right] = w_{j,k}^{[L]} \qquad (B.25)$$

Therefore,

$$\frac{\partial c_p}{\partial \hat{y}_k^{[L-1]}} = \sum_{j=1}^{n_L} \delta_j^{[L]} \times w_{j,k}^{[L]} \qquad ---(B.26)$$

Next,

$$\frac{\partial \hat{y}_k^{[L-1]}}{\partial z_k^{[L-1]}} = \frac{\partial}{\partial z_k^{[L-1]}} \phi\left(z_k^{[L-1]}\right) = \phi'\left(z_k^{[L-1]}\right) \qquad ---(B.27)$$

Using Eqs. $(B.26) - (B.27)$ in Eq. $(B.21)$,

$$\delta_k^{[L-1]} = \left(\sum_{j=1}^{n_L} \delta_j^{[L]} \times w_{j,k}^{[L]}\right) \times \phi'\left(z_k^{[L-1]}\right) \qquad ---(B.28)$$

In a similar fashion, the gradient of c_p with respect to the biases $b_k^{[L-1]}$ can be obtained as:

$$\frac{\partial c_p}{\partial b_k^{[L-1]}} = \frac{\partial c_p}{\partial \hat{y}_k^{[L-1]}} \times \frac{\partial \hat{y}_k^{[L-1]}}{\partial z_k^{[L-1]}} \times \frac{\partial z_k^{[L-1]}}{\partial b_k^{[L-1]}}, \quad k = 1,2,3,\cdots,n_{L-1} \qquad ---(B.29)$$

Note that

$$\frac{\partial z_k^{[L-1]}}{\partial b_k^{[L-1]}} = \frac{\partial}{\partial b_k^{[L-1]}}\left(z_k^{[L-1]}\right) = \frac{\partial}{\partial b_k^{[L-1]}}\left[\sum_{i=1}^{n_{L-2}} \left(x_i^{[L-2]} \cdot w_{k,i}^{[L-1]}\right) + b_k^{[L-1]}\right] = 1$$

$$---(B.30)$$

Using Eq. $(B.28)$ and Eq. $(B.30)$,in Eq. $(B.29)$, gives (for each $k = 1,2,3,\cdots,n_{L-1}$),

$$\frac{\partial c_p}{\partial b_k^{[L-1]}} = \delta_k^{[L-1]} \qquad ---(B.31)$$

Using Eq. $(B.16)$, for the Quadratic cost function, Eq. $(B.28)$ becomes,

$$\delta_k^{[L-1]} = \left(\sum_{j=1}^{n_L}\left(\hat{y}_j^{[L]} - y_j\right) \times \phi'\left(z_j^{[L]}\right) \times w_{j,k}^{[L]}\right) \times \phi'\left(z_k^{[L-1]}\right) \quad ---(B.32)$$

Using Eq. $(B.17)$, for the Binary Cross-Entropy function, Eq. $(B.28)$ becomes,

$$\delta_k^{[L-1]} = \left(\sum_{j=1}^{n_L}\frac{\hat{y}_j^{[L]} - y_j}{\left(1-\hat{y}_j^{[L]}\right)\hat{y}_j^{[L]}} \times \phi'\left(z_j^{[L]}\right) \times w_{j,k}^{[L]}\right) \times \phi'\left(z_k^{[L-1]}\right) \quad ---(B.33)$$

An appropriate expression for derivatives of ϕ in an expression of δ can be used to obtain specific expressions for the gradients of c_p with respect to $w_{k,i}^{[L-1]}$ and $b_k^{[L-1]}$.

For the neurons in the hidden layer $L-2$:

Let's find the expression for the gradients of the cost function c_p with respect to the weights $w_{s,i}^{[L-2]}$ and biases $b_s^{[L-2]}$ used for the neuron of the hidden layer $L-2$. Since c_p is not directly dependent on $w_{s,i}^{[L-2]}$, let's use the chain rule:

$$\frac{\partial c_p}{\partial w_{s,i}^{[L-2]}} = \frac{\partial c_p}{\partial \hat{y}_s^{[L-2]}} \times \frac{\partial \hat{y}_s^{[L-2]}}{\partial z_s^{[L-2]}} \times \frac{\partial z_s^{[L-2]}}{\partial w_{s,i}^{[L-2]}}, \quad s = 1,2,3,\cdots,n_{L-2} \quad i = 1,2,3,\cdots,n_{L-3}$$

$$---(B.34)$$

Similar to Eqs. $(B.9)$ and $(B.21)$, we have

$$\delta_s^{[L-2]} = \frac{\partial c_p}{\partial z_s^{[L-2]}} = \frac{\partial c_p}{\partial \hat{y}_s^{[L-2]}} \times \frac{\partial \hat{y}_s^{[L-2]}}{\partial z_s^{[L-2]}} \quad ---(B.35)$$

Next,

$$\frac{\partial z_s^{[L-2]}}{\partial w_{s,i}^{[L-2]}} = \frac{\partial}{\partial w_{s,i}^{[L-2]}}\left(z_s^{[L-2]}\right) = \frac{\partial}{\partial w_{s,i}^{[L-2]}}\left[\sum_{i=1}^{n_{L-3}}\left(x_i^{[L-3]} \cdot w_{s,i}^{[L-2]}\right) + b_s^{[L-2]}\right] = x_i^{[L-3]}$$

$$---(B.36)$$

Using Eqs. $(B.35)$ and$(B.36)$, in Eq. $(B.34)$ gives

$$\frac{\partial c_p}{\partial w_{s,i}^{[L-2]}} = \delta_s^{[L-2]} \times x_i^{[L-3]}, \qquad s = 1,2,3,\cdots,n_{L-2} \quad i = 1,2,3,\cdots,n_{L-3} \quad --(B.37)$$

Recall that the output from the layer $L-2$ is the input to the layer $L-1$, i.e., $y_s^{[L-2]} = x_s^{[L-2]}$. For Eq. $(B.35)$, we find $\frac{\partial c_p}{\partial \hat{y}_s^{[L-2]}}$ and $\frac{\partial \hat{y}_s^{[L-2]}}{\partial z_s^{[L-2]}}$.

Let's find the gradient of c_p with respect to $y_s^{[L-2]}$ using the chain rule:

$$\frac{\partial c_p}{\partial \hat{y}_s^{[L-2]}} = \sum_{k=1}^{n_{L-1}}\frac{\partial c_p}{\partial \hat{y}_k^{[L-1]}} \times \frac{\partial \hat{y}_k^{[L-1]}}{\partial \hat{y}_s^{[L-2]}}$$

$$\frac{\partial c_p}{\partial \hat{y}_s^{[L-2]}} = \sum_{k=1}^{n_{L-1}} \frac{\partial c_p}{\partial \hat{y}_k^{[L-1]}} \times \frac{\partial \hat{y}_k^{[L-1]}}{\partial z_k^{[L-1]}} \times \frac{\partial z_k^{[L-1]}}{\partial \hat{y}_s^{[L-2]}} \quad ---(B.38)$$

$$\frac{\partial c_p}{\partial \hat{y}_s^{[L-2]}} = \sum_{k=1}^{n_{L-1}} \delta_k^{[L-1]} \times \frac{\partial z_k^{[L-1]}}{\partial \hat{y}_s^{[L-2]}} \quad \because \text{using Eq. } (B.21) \quad ---(B.39)$$

Since,

$$\frac{\partial z_k^{[L-1]}}{\partial \hat{y}_s^{[L-2]}} = \frac{\partial}{\partial \hat{y}_s^{[L-2]}}\left(z_k^{[L-1]}\right) = \frac{\partial}{\partial \hat{y}_s^{[L-2]}}\left[\sum_{s=1}^{n_{L-2}} \left(x_s^{[L-2]} \cdot w_{k,s}^{[L-1]}\right) + b_k^{[L-1]}\right] = w_{k,s}^{[L-1]} \quad (B.40)$$

Therefore,

$$\frac{\partial c_p}{\partial \hat{y}_s^{[L-2]}} = \sum_{k=1}^{n_{L-1}} \delta_k^{[L-1]} \times w_{k,s}^{[L-1]} \quad ---(B.41)$$

Next,

$$\frac{\partial \hat{y}_s^{[L-2]}}{\partial z_s^{[L-2]}} = \frac{\partial}{\partial z_s^{[L-2]}} \phi\left(z_s^{[L-2]}\right) = \phi'\left(z_s^{[L-2]}\right) \quad ---(B.42)$$

Using Eqs. $(B.41)$ and $(B.42)$ in Eq. $(B.35)$, gives

$$\delta_s^{[L-2]} = \left(\sum_{k=1}^{n_{L-1}} \delta_k^{[L-1]} \times w_{k,s}^{[L-1]}\right) \times \phi'\left(z_s^{[L-2]}\right) \quad ---(B.43)$$

In a similar fashion, the gradient of c_p with respect to the biases $b_k^{[L-1]}$ can be obtained as:

$$\frac{\partial c_p}{\partial b_s^{[L-2]}} = \frac{\partial c_p}{\partial \hat{y}_s^{[L-2]}} \times \frac{\partial \hat{y}_s^{[L-2]}}{\partial z_s^{[L-2]}} \times \frac{\partial z_s^{[L-2]}}{\partial b_s^{[L-2]}}, \quad s = 1,2,3,\cdots,n_{L-2} \quad ---(B.44)$$

Note that

$$\frac{\partial z_s^{[L-2]}}{\partial b_s^{[L-2]}} = \frac{\partial}{\partial b_s^{[L-2]}}\left(z_s^{[L-2]}\right) = \frac{\partial}{\partial b_s^{[L-2]}}\left[\sum_{i=1}^{n_{L-3}} \left(x_i^{[L-2]} \cdot w_{s,i}^{[L-1]}\right) + b_s^{[L-2]}\right] = 1 \quad (B.45)$$

Using Eqs. $(B.35)$ and $(B.45)$, in Eq. $(B.44)$, gives (for each $k = 1,2,3,\cdots,n_{L-1}$),

$$\frac{\partial c_p}{\partial b_s^{[L-2]}} = \delta_s^{[L-2]} \quad ---(B.46)$$

Using Eq. $(B.16)$, for the Quadratic cost function, and Eq. $(B.28)$, Eq. $(B.43)$ becomes,

$$\delta_s^{[L-2]} = \left(\sum_{k=1}^{n_{L-1}} \left(\sum_{j=1}^{n_L} \left(\hat{y}_j^{[L]} - y_j\right) \times \phi'\left(z_j^{[L]}\right) \times w_{j,k}^{[L]}\right) \phi'\left(z_k^{[L-1]}\right) w_{k,s}^{[L-1]}\right) \times \phi'\left(z_s^{[L-2]}\right)$$

$$---(B.47)$$

Using Eq. $(B.17)$, for the Binary Cross-Entropy function, and Eq. $(B.28)$, Eq. $(B.43)$ becomes,

$$\delta_s^{[L-2]} = \left(\sum_{k=1}^{n_{L-1}} \left(\sum_{j=1}^{n_L} \frac{\hat{y}_j^{[L]} - y_j}{\left(1 - \hat{y}_j^{[L]}\right) \hat{y}_j^{[L]}} \times \phi'\left(z_j^{[L]}\right) \times w_{j,k}^{[L]} \right) \phi'\left(z_k^{[L-1]}\right) w_{k,s}^{[L-1]} \right) \times \phi'\left(z_s^{[L-2]}\right) \quad ---(B.48)$$

An appropriate expression for derivatives of ϕ in an expression of δ can be used to obtain specific expressions for the gradients of c_p with respect to $w_{s,i}^{[L-2]}$ and $b_s^{[L-2]}$.

For the neurons in the hidden layer r, (for $r = L-1, L-2, \cdots, 1$) :

In general, for any hidden layer r, (for $r = L-1, L-2, \cdots, 1$), the gradients of the cost function c_p with respect to the weights and biases can be expressed as:

$$\frac{\partial c_p}{\partial w_{q,i}^{[r]}} = \delta_q^{[r]} \times x_i^{[r-1]}, \qquad q = 1,2,3,\cdots,n_r, \quad i = 1,2,3,\cdots,n_{r-1} \quad ---(B.49)$$

$$\frac{\partial c_p}{\partial b_q^{[r]}} = \delta_q^{[r]}, \qquad q = 1,2,3,\cdots,n_r \quad ---(B.50)$$

where $\delta_q^{[r]}$ can be expressed recursively as:

$$\delta_q^{[r]} = \sum_{t=1}^{n_{r+1}} \delta_t^{[r+1]} \times w_{t,q}^{[r+1]} \times \phi'\left(z_q^{[r]}\right) \quad ---(B.51)$$

For $r = L-1$, in Eq. $(B.51)$, $\delta_t^{[r+1]}$ becomes $\delta_t^{[L]}$ that is defined in Eq. $(B.9)$:

$$\delta_t^{[L]} = \frac{\partial c_p}{\partial \hat{y}_t^{[L]}} \times \phi'\left(z_t^{[L]}\right), \qquad t = 1,2,3,\cdots,n_L \quad ---(B.52)$$

This concludes the derivation of expressions for derivatives of the cost function c_p with respect to the network parameters (weights and biases). For elaboration, consider the network as shown in Fig. 3.1.

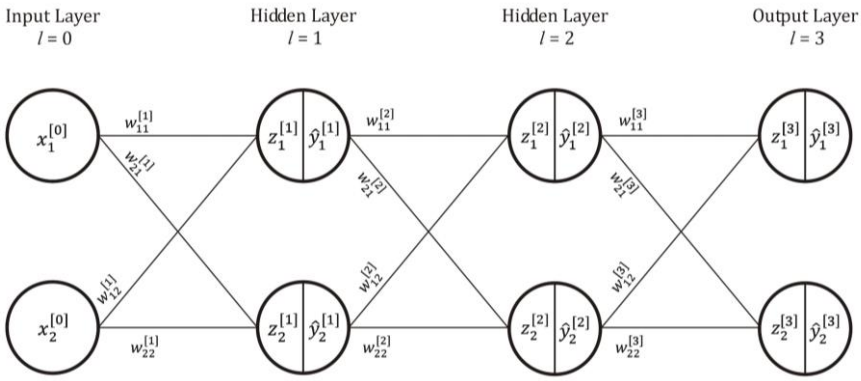


Fig. 3.1: A network with two hidden layers.

Using the formulas with the chain rule, the derivate of the cost function c_p for an instance p with respect to the weight $w_{11}^{[3]}$ is given by,

$$\frac{\partial c_p}{\partial w_{1,1}^{[3]}} = \frac{\partial c_p}{\partial \hat{y}_1^{[3]}} \times \frac{\partial \hat{y}_1^{[3]}}{\partial z_1^{[3]}} \times \frac{\partial z_1^{[3]}}{\partial w_{1,1}^{[3]}} \quad ---(B.53)$$

The quantities involved in Eq. $(B.53)$ for chaining of the derivatives due to the variable dependencies are highlighted in Fig. 3.2.

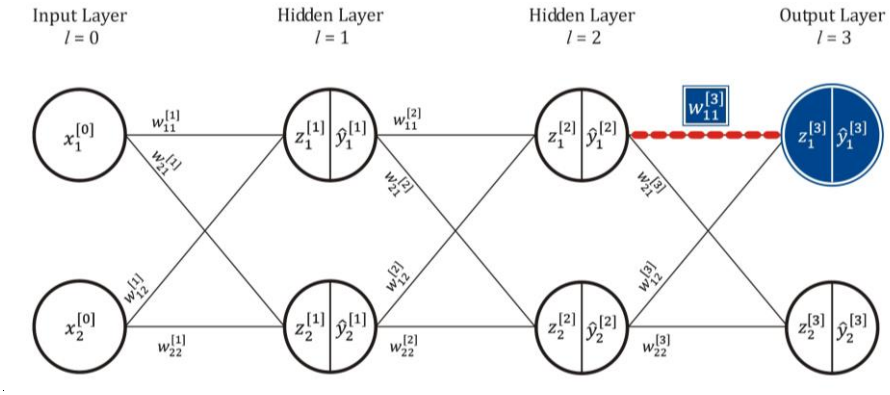


Fig. 3.2: Highlighting the quantities involve in the chain rule for gradient of c_p with respect to $w_{11}^{[3]}$.

The derivate of c_p with respect to the weight $w_{11}^{[2]}$ is given by,

$$\frac{\partial c_p}{\partial w_{1,1}^{[2]}} = \left(\sum_{j=1}^{n_3=2} \frac{\partial c_p}{\partial \hat{y}_j^{[3]}} \times \frac{\partial \hat{y}_j^{[3]}}{\partial z_j^{[3]}} \times \frac{\partial z_j^{[3]}}{\partial \hat{y}_1^{[2]}} \right) \times \frac{\partial \hat{y}_1^{[2]}}{\partial z_1^{[2]}} \times \frac{\partial z_1^{[2]}}{\partial w_{1,1}^{[2]}} \quad ---(B.54)$$

The quantities involved in Eq. $(B.54)$ for chaining of the derivatives due to the variable dependencies are highlighted in Fig. 3.3.

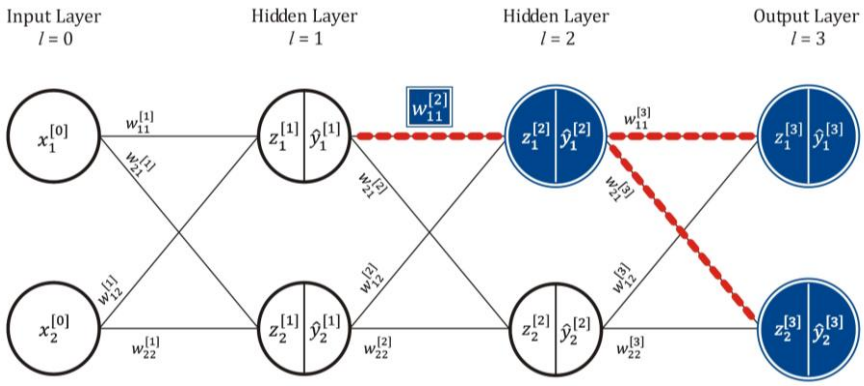


Fig. 3.3: Highlighting the quantities involve in the chain rule for gradient of c_p with respect to $w_{11}^{[2]}$.

The derivate of c_p for an instance p with respect to the weight $w_{11}^{[2]}$ is given by,

$$\frac{\partial c_p}{\partial w_{1,1}^{[1]}} = \left(\sum_{i=1}^{n_2=2} \left(\sum_{j=1}^{n_3=2} \frac{\partial c_p}{\partial \hat{y}_j^{[3]}} \times \frac{\partial \hat{y}_j^{[3]}}{\partial z_j^{[3]}} \times \frac{\partial z_j^{[3]}}{\partial \hat{y}_i^{[2]}} \right) \times \frac{\partial \hat{y}_i^{[2]}}{\partial z_i^{[2]}} \times \frac{\partial z_i^{[2]}}{\partial \hat{y}_1^{[1]}} \right) \times \frac{\partial \hat{y}_1^{[1]}}{\partial z_1^{[1]}} \times \frac{\partial z_1^{[1]}}{\partial w_{1,1}^{[1]}} \quad (B.55)$$

The quantities involved in Eq. $(B.55)$ for chaining of the derivatives due to the variable dependencies are highlighted in Fig. 3.4.

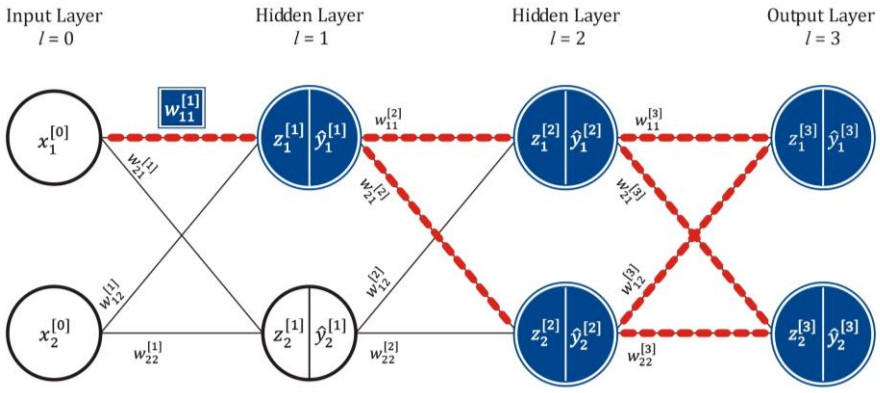


Fig. 3.4: Highlighting the quantities involve in the chain rule for gradient of c_p with respect to $w_{11}^{[1]}$.

4. Learning of the Neural Network Model Through Optimization

On completion of forward pass computations (together with initial values of weights and biases) of the neural network, an appropriate definition of the cost function (loss function) is used to compute the error between the predicted output and the target (label) output. This activity is performed for one or more instances depending on the choice of the optimization strategy. Next, through optimization, which is an integral component of a deep learning algorithm, the network weights and biases are updated to minimize the cost function for better generalization. The updated weights and biases are used for the next forward pass, and the cost function is computed again. If needed, the weights and biases are modified again through optimization. The iterative process of updating weights and biases (probably using a different set of instances) is continued until the desired accuracy is achieved. In fact, for a neural network model, the learning of the model occurs in the form of optimization of the weights and biases for the minimal cost function. Better weights and biases reflect better 'learning' of the model.

For deep learning, an optimizer is a function or algorithm that finalizes the gradients for updating the parameters and then modifies the parameters (weights and biases). In the literature, several optimization methods are available that are frequently used in deep learning. Examples include the Gradient Descent (GD) methods (Batch GD, Stochastic GD, Minibatch GD, etc.) with and without the Momentum term, the Nesterov Accelerated GD method, and the GD method with Adaptive learning rates (such as Adam).

Here, firstly, we discuss some basics of the Gradient descent method—secondly, we discuss various strategies for accumulating the gradients for the optimization scheme—finally, we discuss the formulas to update the weights and biases.

4.1. The Gradient Descent (GD) method for Multivariable Optimization

The GD method is an unconstrained first-order optimization algorithm. The GD method proceeds iteratively with the goal of converging to a vector that optimizes the objective function. The main idea of the GD method is to take steps proportional to the negative gradient of the function to minimize it.

Let $C: \mathbb{R}^n \to \mathbb{R}$ be an n -dimensional, differentiable, objective function of variables: $v = (v_1, v_2, \cdots, v_n)^T$. The gradient of C is the vector of partial derivatives of C with respect to the components of v. It is represented by $\nabla C(v)$:

$$\nabla C(v) \quad = \quad \left[\frac{\partial C(v)}{\partial v_1} \quad \frac{\partial C(v)}{\partial v_2} \quad \cdots \quad \frac{\partial C(v)}{\partial v_n}\right]^T \quad \in \mathbb{R}^n \qquad ---(D.1)$$

If $\Delta v = (\Delta v_1, \Delta v_2, \cdots, \Delta v_n)^T \in \mathbb{R}^n$ is the vector of change in v, then by Taylor expansion, we have

$$C(v + \Delta v) \quad \cong \quad C + \frac{\partial C(v)}{\partial v_1}\Delta v_1 + \frac{\partial C(v)}{\partial v_2}\Delta v_2 + \cdots + \frac{\partial C(v)}{\partial v_n}\Delta v_n \qquad ---(D.2)$$

According to the Calculus, a change in C can be expressed as follows,

$$\Delta C \quad \cong \quad \frac{\partial C(v)}{\partial v_1}\Delta v_1 + \frac{\partial C(v)}{\partial v_2}\Delta v_2 + \cdots + \frac{\partial C(v)}{\partial v_n}\Delta v_n \quad = \quad \nabla C(v) \cdot \Delta v \qquad ---(D.3)$$

Using Eq. $(D.3)$ in Eq. $(D.2)$,

$$C(v + \Delta v) \quad \cong \quad C + \Delta C \quad = \quad C + \nabla C(v) \cdot \Delta v \qquad ---(D.4)$$

The objective is to minimize the objective function C. Using the Cauchy–Schwarz inequality, the upper bound of the magnitude of $\nabla C(v) \cdot \Delta v$ is $\|\nabla C(v)\|_2 \|\Delta v\|_2$. That is,

$$|\nabla C(v) \cdot \Delta v| \quad \leq \quad \|\nabla C(v)\|_2 \|\Delta v\|_2 \qquad ---(D.5)$$

For minimization, the upper bound should be negative. $\|\nabla C(v)\|_2 \|\Delta v\|_2$ is negative if $\Delta v = -\alpha \nabla C(v)$. Thus, Eq. $(D.3)$. becomes

$$\Delta C \quad \cong \quad \nabla C(v) \cdot \left(-\alpha \nabla C(v)\right) \quad \cong \quad -\alpha \|\nabla C(v)\|^2 \qquad ---(D.6)$$

$\|\nabla C(v)\|^2 \geq 0$, this implies that $\Delta C \leq 0$. This ensures that the value of C is minimized. The update is made as follows:

$$v \quad \longleftarrow \quad v - \alpha \nabla C(v)$$

4.2. Computation of the Gradients for Optimization: Variants of the GD Method

For performing an iteration of the Gradient Descent method, a subset/mini-batch of m instances is selected from the training set of N instances. The computations of the forward pass (together with a set of initial weights and biases) of the neural network are performed for each of m selected instances. This way, the predicted output for each of m instances is obtained. Next, the loss function is computed as the error between the predicted output and the target (label) output for each of the m instances separately. That is, the loss function $c^{(p)}$ is computed for $p = 1,2, \cdots, m$. Next, the gradient of each of

the loss functions $c^{(p)}$ with respect to a specific parameter is computed using the backpropagation as discussed in Section 3. Then, the average of these gradients is considered as the gradient of the cost function C. An averaged gradient of C serves as a more accurate estimate of the gradient (Goodfellow, 2016; Aggarwal, 2018). Similarly, the averaged gradients of C are formed with respect to each of the parameters separately. That said, we compute:

$$\frac{\partial C}{\partial w_{j,i}^{[l]}} = \frac{1}{m}\sum_{p=1}^{m}\frac{\partial c_p}{\partial w_{j,i}^{[l]}} \quad ---(G.1)$$

$$\frac{\partial C}{\partial b_{j}^{[l]}} = \frac{1}{m}\sum_{p=1}^{m}\frac{\partial c_p}{\partial b_{j}^{[l]}} \quad ---(G.2)$$

The notations have the same meaning as used in Section 3. Depending on the value of m for an iteration, there are three common variants of using the Gradient Descent optimization method:

4.2.1. The Batch-Gradient Descent Method

If $m = N$ (i.e., using all the instances in each iteration), the GD method is called the ***Batch-Gradient Descent*** method. In this case, $\gamma = 1$, where γ is the number of mini-batches of the set of N training instances. This approach shows a smooth convergence of the cost function to its minimum. However, practically, it slows down the training process and becomes exorbitantly expensive computationally. Hence, it is seldom used.

4.2.2. The Stochastic Gradient Descent (SGD) Method

If $m = 1$ (i.e., using only one instance in each iteration) and the instance is selected randomly, then the GD method is called the ***Stochastic Gradient Descent*** (***SGD***) method or ***Online Gradient Descent*** method. For SGD, $\gamma = N$. The term *stochastic* corresponds to a random selection of the instance for an iteration. The random selection may be made *with replacement* or *without replacement*. Sampling with replacement is sometimes referred to as *bootstrapping* or *bagging*.

The SGD method converges faster when the dataset is large, as it causes updates to the parameters more frequently. The SGD method is generally noisier than the Batch GD method, and it takes more iterations to reach the minimum because of its randomness. Still, it is far less computationally expensive comparatively. Hence, SGD is often preferred over the Batch GD method. In fact, the stochastic approach with the GD method has turned out to be the workhorse of neural network computations. The success is obtained due to the remarkable fact that randomization (stochastic sampling) results in reliable generalization when there are so many features/variables.

4.2.3. The Mini-Batch Gradient Descent Method

If $1 < m < N$, then the GD method is called the ***Mini-Batch Gradient Descent*** method, or precisely, the ***Mini-Batch Stochastic Gradient Descent*** method, as the selection of the instances for a minibatch is made stochastically. In this case $\gamma = N/m$. That is, N is an integer multiple of m such that $\gamma m = N$.

In the Mini-Batch Gradient Decent method, the parameters are updated after every subset of the data, so the progress of updating the cost function is smoother compared to that of the SGD. Also, since the entire dataset is not used at a time, the computation cost is less than the batch gradient descent. Therefore, this approach is preferred mostly.

Remark: A comparison of the error drop pattern for the three variants of the GD methods is shown in Fig. 4.1.

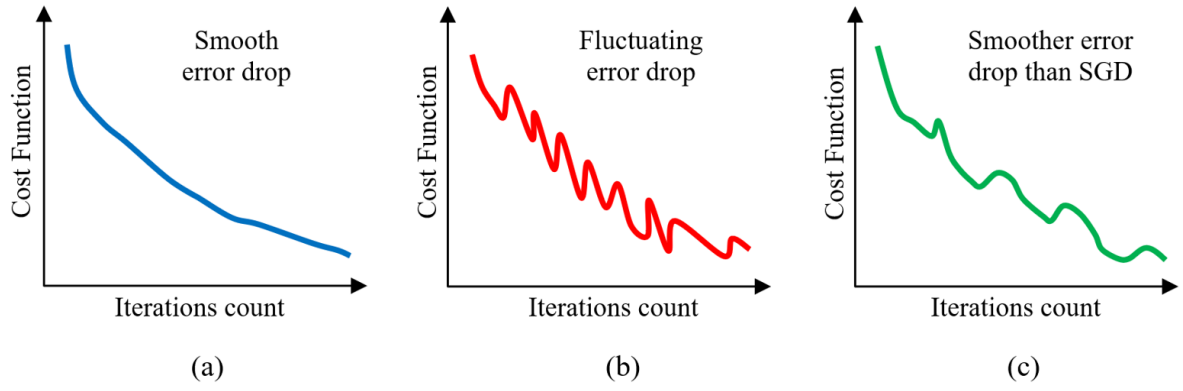


Fig. 4.1: A comparison of error drop pattern among the three variants of the Gradient Descent method: (a) the Batch GD method, (b) the SGD method, and (c) the Mini-Batch GD method.

4.3. Updating the Parameters (Weights and Biases): The Learning of the Model

Once the gradients of the parameters (weights and biases) are ready, as in Eqs. $(G.1, G.2)$, then according to the Gradient Descent strategy, the parameters are updated as follows. The averaged gradient of C with respect to a specific parameter serves as the respective component of the vector of advancement step sizes in the Descent direction. This step is used to update the relevant parameter.

$$w_{j,i}^{[l]} = w_{j,i}^{[l]} - \alpha\left(\frac{\partial C}{\partial w_{j,i}^{[l]}}\right) \quad j = 1,2,3,\cdots,n_l \quad i = 1,2,3,\cdots,n_{l-1} \qquad ---(G.3)$$

$$b_{j}^{[l]} = b_{j}^{[l]} - \alpha\left(\frac{\partial C}{\partial b_{j}^{[l]}}\right) \quad j = 1,2,3,\cdots,n_l \quad i = 1,2,3,\cdots,n_{l-1} \qquad ---(G.4)$$

Here, α is the learning rate. α is a hyperparameter. It serves as the step size to control how much the weights and biases are to be changed. In this way, all the parameters are updated using the respective averaged gradients.

The iterative process of updating the parameters is repeated until C is minimized. For the first iteration, the initial approximation of $w_{j,i}^{[l]}$ and $b_j^{[l]}$ are set (may be randomly) to start the iterative process. Once the optimal values of $w_{j,i}^{[l]}$ and $b_j^{[l]}$ are obtained that minimizes the cost function C, Eq. $(F.7)$, together with the network, serves as the trained model to predict the output value for any input.

Remarks:

- It is important to note that each time the parameters (weights and biases) are updated, it is known as an **iteration** or **step**. The GD method involves incremental training in which the network parameters are updated in each iteration. The updated parameters will be used in the next iteration (probably with a different set of m instances from the training set of N instances). Then, the cost functions and the averaged gradients are computed again. If needed, the weights and biases are modified again through optimization. The iterative process of updating weights and biases is continued until the convergence to the global minimum of the cost function.

- After a certain number of iterations, when all instances have been used in at least one iteration to update the weights and biases, an ***epoch*** is said to be completed. One epoch means going through the entire training set once.

- In the Batch Gradient Descent method, there is only one iteration per epoch, as $m = N$. Thus, one iteration serves as an epoch. This allows the cost function to move smoothly to the global minimum by updating the parameters once per epoch.

- In the SDG, the random selection of an instance is made ***with replacement*** or ***without replacement***. If the instance selection is made ***without replacement***, then N iterations would be required to complete one epoch, as there are N instances in the training set and $m = 1$. This allows the cost function to move rapidly to the global minimum by updating the parameters N times per epoch.

- In the Mini-Batch Gradient Decent method, there exists an integer γ such that $\gamma m = N$. γ specifies the number of mini-batches in which the N instances are partitioned. Thus, γ iterations/steps are required to complete one epoch if the sampling is made without replacement. This also allows the cost function to move rapidly to the global minimum by updating the parameters γ times per epoch.

Remarks: For gradient descent to reach the global minimum of the cost function, we must set the learning rate to an appropriate value that is neither too small nor too large. High learning rates cause larger steps towards the minimum (fewer iterations) but risk overshooting. On the other hand, low learning rates cause smaller step sizes and stability, requiring more iterations and computation time to reach the minimum. A comparison of the error drop pattern for different values of the learning rate is shown in Fig. 4.2.

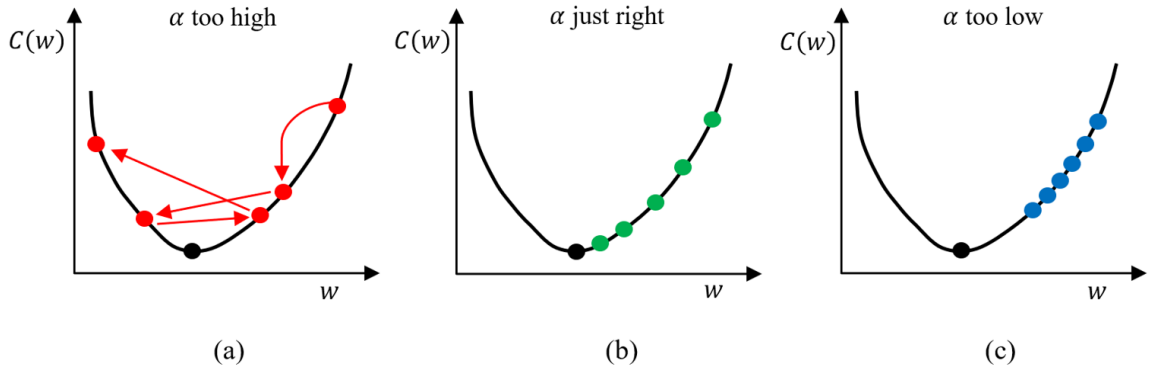


Fig. 4.2: A comparison of different values of the learning rate α. Here, w is a parameter with respect to which the error or loss $C(w)$ is spotted to be changing after every iteration. Clearly, having the learning rate neither too high nor too low is the appropriate way for the best pattern of error drop.

Fig. 4.3 gives a comparison of error drop for different levels of values of the learning rate α as the epochs of the iterative learning process progress.

The learning rate (α) and the size of a mini-batches (m) are two critical hyperparameters that control the optimization process. However, the right choices for these hyperparameters are not always obvious. Interestingly, strategies for an adaptive learning rate, such as Adam (discussed later), help to learn so efficiently that the effect of hyperparameters is abridged. This way, an extensive grid search for the best values of hyperparameters can be avoided.

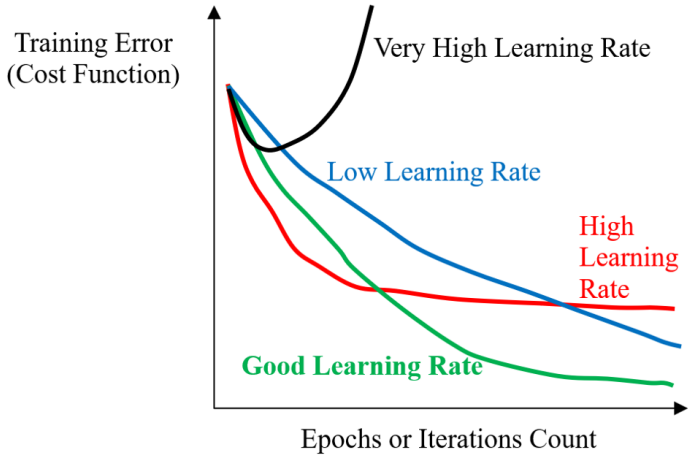


Fig. 4.3: A comparison of different values of the learning rate

Remark: There are certain pitfalls and shortcomings associated with the optimization methods. Researchers and practitioners have been proposing methods and variants for efficient optimization. Section 9 discusses some advanced Gradient Descent strategies for efficient training. Advanced literature can be consulted for more versatile optimization techniques.

5. Illustrating the Neural Networks Model Training: A Flow Chart

Fig. 5.1 depicts a work flow of the generic process of training a neural network model.

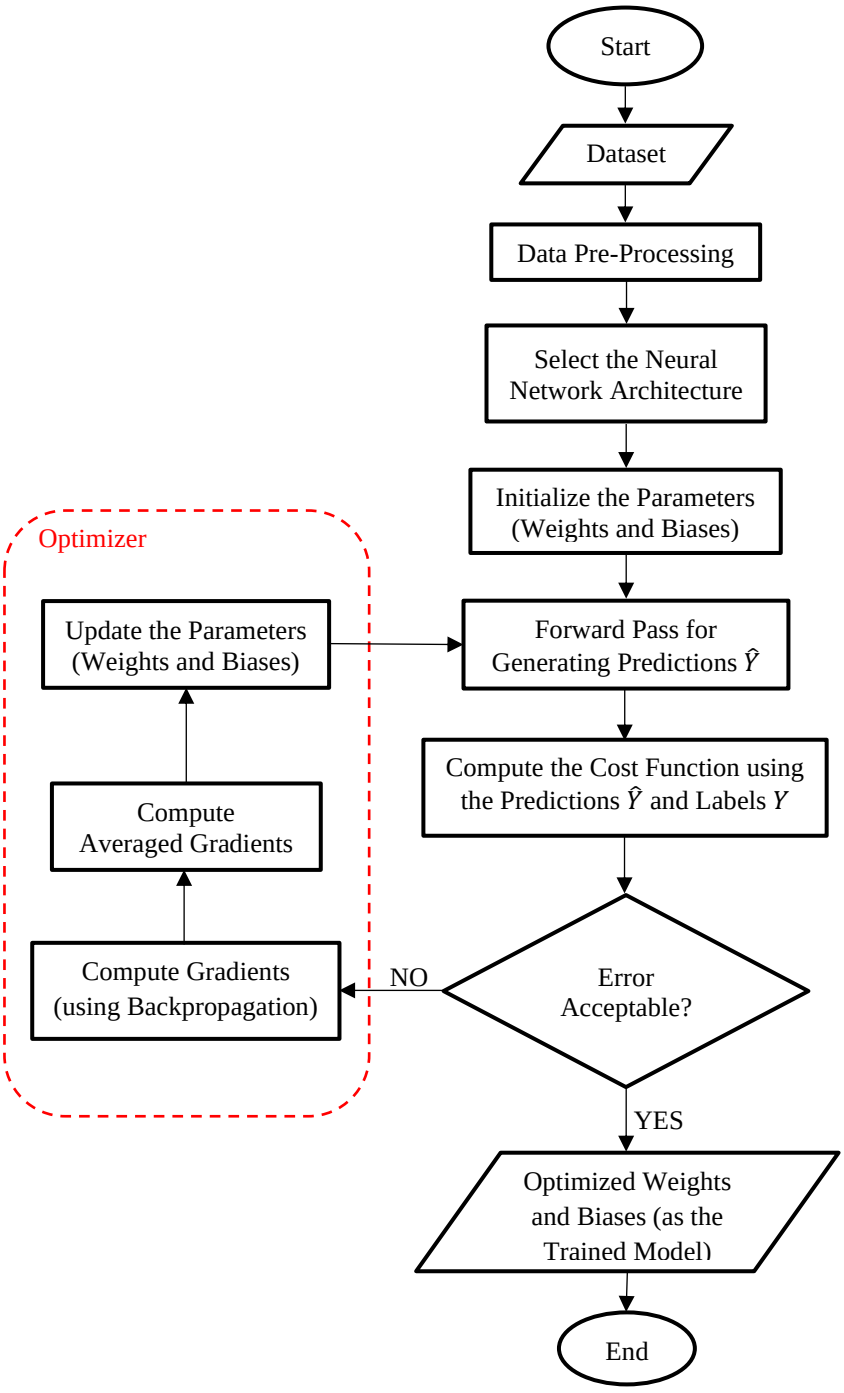


Fig. 5.1: A flow chart to illustrate the neural network model training process

6. Illustrating ANN Computations by Working Out a Classification Example

6.1. Problem Description

The dataset shown in Table 6.1 is for illustrative purposes. It is taken from a famous dataset for classifying a Titanic passenger as survived: Yes or No (kaggle-titanic, 2017). The dataset has three input (independent) features/attributes: Class: Passenger class (1 = First, 2 = Second, 3 = Third), Age (in years), and Fare. The target variable Survived is a Boolean variable indicating whether the passenger survived (1 = Yes, 0 = No). The dataset has instances p, where $p = 1,2,\cdots,6$.

Class	Age	Fare	Survived
3	22	7.25	0
1	38	71.25	1
3	26	8.5	1
2	35	53.1	1
2	32	48.5	0
3	35	12.5	0

Table 6.1: A small dataset of an example problem: Titanic Survivors

6.2. Standardization of the Dataset

For standardization, we replace each value x_j of a column/attribute with $(x_j - \mu)/\sigma^*$, where μ is the mean of the values in the respective column, and σ^* is the standard deviation of the respective attribute:

$$\mu = \frac{1}{N}\sum_{j=1}^{N}(x_j) \quad \text{and} \quad \sigma^* = \sqrt{\frac{1}{N}\sum_{j=1}^{N}(x_j - \mu)^2}$$

After standardization, the updated values are given in Table 6.2.

Class	Age	Fare	Survived
0.8944	−1.6703	−1.0452	0
−1.7888	1.1931	1.5015	1
0.8944	−0.9545	−0.9955	1
−0.4472	0.6562	0.7793	1
−0.4472	0.1193	0.5962	0
0.8944	0.6562	−0.8363	0

Table 6.2: Standardized Dataset

6.3. Selection of the Network Structure

For the Titanic Survivors classification problem, the structure selected as the neural work is shown in Fig. 6.1. There are three layers: the input layer is $l = 0$, the hidden layer is $l = 1$, and the output layer is $l = 2$. The number of neurons in the input layer is 3 ($n_0 = 3$), in the hidden layer is 3 ($n_1 = 3$), and in the output layer is 3 ($n_2 = 3$).

For illustrative purposes, we consider only the first instance, i.e., $p = 1$, from Table 2 and use it for elaborating on the computations of the forward pass, cost function, gradients using backpropagations, and the Gradient Descent optimization for updating the parameters. The input data is:

$$X^{[0]} = \begin{bmatrix} x_1^{[0]} & x_2^{[0]} & x_3^{[0]} \end{bmatrix}^T = [0.8944 \quad -1.6703 \quad -1.0452]$$

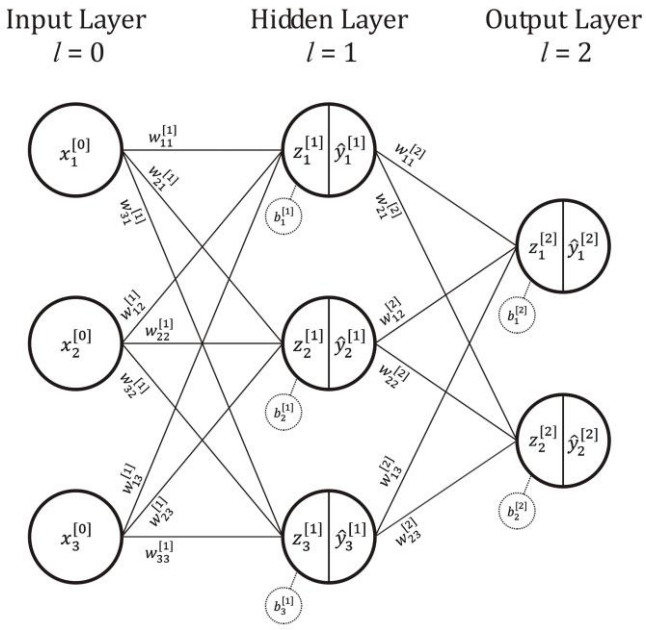


Fig. 6.1: The Structure of the neural network

6.4. Initializing Weights and Biases

For the hidden layer, $l = 1$:

$$W_1^{[1]} = \begin{bmatrix} w_{1,1}^{[1]} & w_{1,2}^{[1]} & w_{1,3}^{[1]} \end{bmatrix}^T = [0.01 \quad 0.02 \quad 0.1]$$

$$W_2^{[1]} = \begin{bmatrix} w_{2,1}^{[1]} & w_{2,2}^{[1]} & w_{2,3}^{[1]} \end{bmatrix}^T = [0.12 \quad 0.03 \quad 0.05]$$

$$W_3^{[1]} = \begin{bmatrix} w_{3,1}^{[1]} & w_{3,2}^{[1]} & w_{3,3}^{[1]} \end{bmatrix}^T = [0.06 \quad 0.02 \quad 0.03]$$

$$B^{[1]} = \begin{bmatrix} b_1^{[1]} & b_2^{[1]} & b_3^{[1]} \end{bmatrix}^T = [0.11 \quad 0.06 \quad 0.05]$$

For the output layer, $l = L = 2$:

$$W_1^{[2]} = \begin{bmatrix} w_{1,1}^{[2]} & w_{1,2}^{[2]} & w_{1,3}^{[2]} \end{bmatrix}^T = [0.04 \quad 0.03 \quad 0.3]$$

$$W_2^{[2]} = \begin{bmatrix} w_{2,1}^{[2]} & w_{2,2}^{[2]} & w_{2,3}^{[2]} \end{bmatrix}^T = [0.20 \quad 0.05 \quad 0.02]$$

$$B^{[2]} = \begin{bmatrix} b_1^{[2]} & b_2^{[2]} \end{bmatrix}^T = [0.06 \quad 0.04]$$

6.5. The Forward Pass (Finding the Neuron Values)

In the layer $l = 1$,

$$W_j^{[1]} \cdot X^{[0]} \quad = \quad \sum_{i=1}^{3} w_{j,i}^{[1]} x_i^{[0]} \quad = \quad \phi\left(z_j^{[1]}\right) \quad = \quad \hat{y}_j^{[1]}$$

Recall that each $\hat{y}_j^{[l]} \in (0,1)$, for $j = 1,2,\cdots,n_l$. is an estimated probability.

For the first neuron, $j = 1$:

$$\begin{aligned} W_1^{[1]} \cdot X^{[0]} &= \sum_{i=1}^{3} w_{1,i}^{[1]} x_i^{[0]} \\ &= w_{1,1}^{[1]} x_1^{[0]} + w_{1,2}^{[1]} x_2^{[0]} + w_{1,3}^{[1]} x_3^{[0]} \\ &= (0.01)(0.8944) + (0.02)(-1.6703) + (0.1)(-1.0452) \\ &= -0.1289 \end{aligned}$$

$$z_1^{[1]} = W_1^{[1]} \cdot X^{[0]} + b_1^{[1]} = -0.1289 + 0.11 = -0.0189$$

$$\hat{y}_1^{[1]} = \phi\left(z_1^{[1]}\right) = \frac{1}{1 + e^{-z_1^{[1]}}} = \frac{1}{1 + e^{+0.0189}} = 0.4953$$

For the second neuron, $j = 2$:

$$\begin{aligned} W_2^{[1]} \cdot X^{[0]} &= \sum_{i=1}^{3} w_{2,i}^{[1]} x_i^{[0]} \\ &= w_{2,1}^{[1]} x_1^{[0]} + w_{2,2}^{[1]} x_2^{[0]} + w_{2,3}^{[1]} x_3^{[0]} \\ &= (0.12)(0.8944) + (0.03)(-1.6703) + (0.05)(-1.0452) \\ &= 0.0049 \end{aligned}$$

$$z_2^{[1]} = W_2^{[1]} \cdot X^{[0]} + b_2^{[1]} = 0.0049 + 0.06 = 0.0649$$

$$\hat{y}_2^{[1]} = \phi\left(z_2^{[1]}\right) = \frac{1}{1 + e^{-z_2^{[1]}}} = \frac{1}{1 + e^{-0.0649}} = 0.5162$$

For the third neuron, $j = 3$:

$$\begin{aligned} W_3^{[1]} \cdot X^{[0]} &= \sum_{i=1}^{3} w_{3,i}^{[1]} x_i^{[0]} \\ &= w_{3,1}^{[1]} x_1^{[0]} + w_{3,2}^{[1]} x_2^{[0]} + w_{3,3}^{[1]} x_3^{[0]} \\ &= (0.06)(0.8944) + (0.02)(-1.6703) + (0.03)(-1.0452) \\ &= -0.0111 \end{aligned}$$

$$z_3^{[1]} = W_3^{[1]} \cdot X^{[0]} + b_3^{[1]} = -0.0111 + 0.05 = 0.0389$$

$$\hat{y}_3^{[1]} = \phi\left(z_3^{[1]}\right) = \frac{1}{1 + e^{-z_3^{[1]}}} = \frac{1}{1 + e^{-0.0389}} = 0.5097$$

In the layer $l = L = 2$,

$$W_j^{[2]} \cdot X^{[1]} = \sum_{i=1}^{3} w_{j,i}^{[2]} x_i^{[1]} = \sum_{i=1}^{3} w_{j,i}^{[2]} \hat{y}_i^{[1]} = \phi\left(z_j^{[2]}\right) = \hat{y}_j^{[1]}$$

For the first neuron, $j = 1$:

$$\begin{aligned} W_1^{[2]} \cdot X^{[1]} &= \sum_{i=1}^{3} w_{1,i}^{[2]} \hat{y}_i^{[1]} \\ &= w_{1,1}^{[2]} \hat{y}_1^{[1]} + w_{1,2}^{[2]} \hat{y}_2^{[1]} + w_{1,3}^{[2]} \hat{y}_3^{[1]} \\ &= (0.04)(0.4953) + (0.03)(0.5162) + (0.3)(0.5097) \\ &= 0.1882 \end{aligned}$$

$$z_1^{[2]} = W_1^{[2]} \cdot X^{[1]} + b_1^{[2]} = 0.1882 + 0.06 = 0.2482$$

$$\hat{y}_1^{[2]} = \phi\left(z_1^{[2]}\right) = \frac{1}{1 + e^{-z_1^{[2]}}} = \frac{1}{1 + e^{0.2482}} = 0.438$$

For the second neuron, $j = 2$:

$$\begin{aligned} W_2^{[2]} \cdot X^{[1]} &= \sum_{i=1}^{3} w_{2,i}^{[2]} \hat{y}_i^{[1]} \\ &= \hat{y}_1^{[1]} w_{2,1}^{[2]} + \hat{y}_2^{[1]} w_{2,2}^{[2]} + \hat{y}_3^{[1]} w_{2,3}^{[2]} \\ &= (0.20)(0.4953) + (0.05)(0.5162) + (0.02)(0.5097) \\ &= 0.1351 \end{aligned}$$

$$z_2^{[2]} = W_2^{[2]} \cdot X^{[1]} + b_2^{[2]} = 0.1351 + 0.04 = 0.1751$$

$$\begin{aligned} \hat{y}_2^{[2]} &= \phi\left(z_2^{[2]}\right) \\ &= \frac{1}{1 + e^{-z_2^{[2]}}} \\ &= \frac{1}{1 + e^{-0.1751}} \\ &= 0.544 \end{aligned}$$

On completion of the first forward pass, the vector of the output layer's neuron values (estimated probabilities) for instance p is as follows:

$$\hat{Y}^{[2]} = \begin{bmatrix} \hat{y}_1^{[2]} \\ \hat{y}_2^{[2]} \end{bmatrix} = \begin{bmatrix} 0.438 \\ 0.544 \end{bmatrix}$$

The vector of the target values:

$$Y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Recall that each $\hat{y}_j^{[L]} \in (0,1)$, for $j = 1,2,\cdots,n_L$ is a probability (estimated using the Sigmoid function) for the instance to belong to a particular class. The instance will belong to the class corresponding to the Kth neuron (Kth component in the vector of the target classes) such that

$$\hat{y}_K^{[L]} = \max_{1 \le j \le n_L} \left(\hat{y}_j^{[L]}\right)$$

6.6. Calculating the Cost Function (Error in the Calculated Output)

For the instance $p = 1$, the Binary-Cross Entropy function or error $c^{(p)}$ or c_p is calculated as:

$$\begin{aligned} c_p &= -\sum_{j=1}^{2} \left[y_j \ln \hat{y}_j^{[2]} + (1 - y_j) \ln\left(1 - \hat{y}_j^{[2]}\right)\right] \\ &= \left[y_1 \ln \hat{y}_1^{[2]} + (1 - y_1) \ln\left(1 - \hat{y}_1^{[2]}\right) + y_2 \ln \hat{y}_2^{[2]} + (1 - y_2) \ln\left(1 - \hat{y}_2^{[2]}\right)\right] \\ &= -[(0)\ln(0.438) + (1 + 0)\ln(1 - 0.438) + (1)\ln(0.544) + (0)\ln(1 - 0.544)] \\ &= -1.18506 \end{aligned}$$

6.7. Learning Phase: Computing the Gradients through Backpropagation

Gradient of the cost function with respect to the weights and biases used in the layer $l = L = 2$:

For the Sigmoid activation function together with the Cross-Entropy cost function, we have for $j = 1,2$ and $i = 1,2,3$:

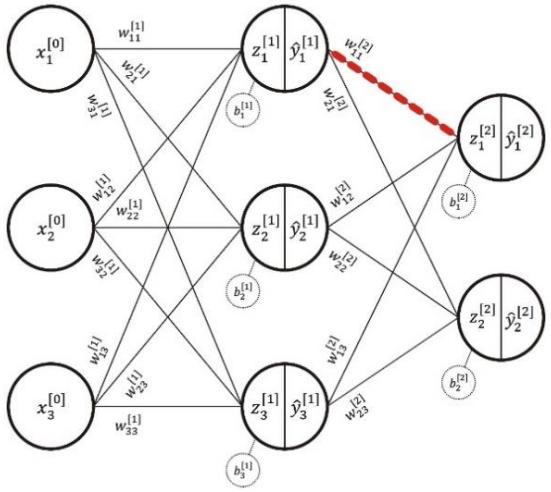
$$\begin{aligned} \frac{\partial c_p}{\partial w_{j,i}^{[2]}} &= \delta_j^{[2]} \times \hat{y}_i^{[1]} && \because \text{from Eq. } (B.9) \\ &= \frac{\hat{y}_j^{[2]} - y_j}{\left(1 - \hat{y}_j^{[2]}\right)\hat{y}_j^{[2]}} \times \phi'\left(z_j^{[2]}\right) \times \hat{y}_i^{[1]} && \because \text{from Eq. } (B.17) \\ &= \frac{\hat{y}_j^{[2]} - y_j}{\left(1 - \hat{y}_j^{[2]}\right)\hat{y}_j^{[2]}} \times \hat{y}_j^{[2]}\left(1 - \hat{y}_j^{[2]}\right) \times \hat{y}_i^{[1]} && \because \text{from Eq. } (B.18) \\ &= \left(\hat{y}_j^{[2]} - y_j\right) \times \hat{y}_i^{[1]} && ---(E.3) \end{aligned}$$

Also

$$\frac{\partial c_p}{\partial b_j^{[2]}} = \delta_j^{[2]} = \hat{y}_j^{[2]} - y_j \qquad ---(E.4)$$

Calculating gradients with respect to the weights using Eq. $(E.3)$:

For $j = 1, i = 1$:

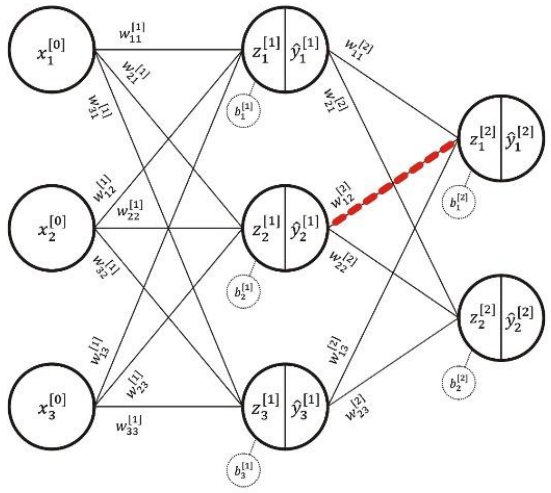


$$\frac{\partial c_p}{\partial w_{1,1}^{[2]}} = \left(\hat{y}_1^{[2]} - y_1\right) \times \hat{y}_1^{[1]}$$

$$= \left(0.438 - (0)\right) \times 0.4953$$

$$= 0.2169$$

For $j = 1, i = 2$:

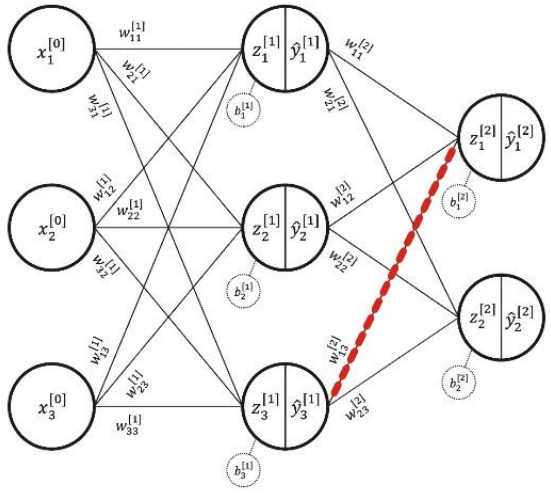


$$\frac{\partial c_p}{\partial w_{1,2}^{[2]}} = \left(y_1^{[2]} - y_1\right) \times \hat{y}_2^{[1]}$$

$$= \left(0.438 - (0)\right) \times 0.5162$$

$$= 0.2261$$

For $j = 1, i = 3$:

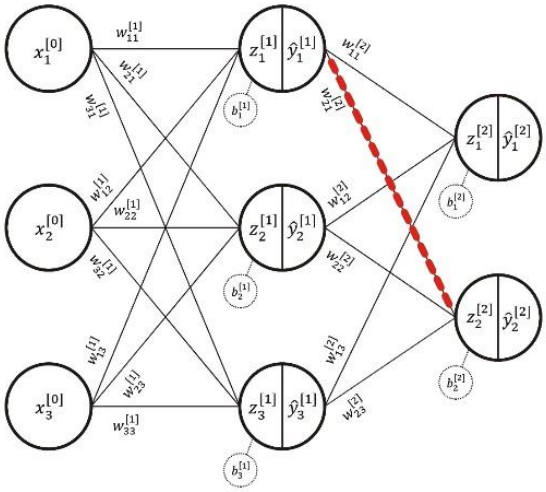


$$\frac{\partial c_p}{\partial w_{1,3}^{[2]}} = \left(\hat{y}_1^{[2]} - y_1\right) \times \hat{y}_3^{[1]}$$

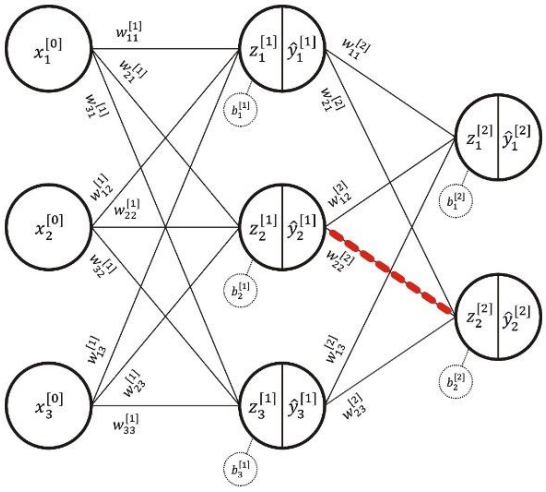
$$= \left(0.438 - (0)\right) \times 0.5097$$

$$= 0.2232$$

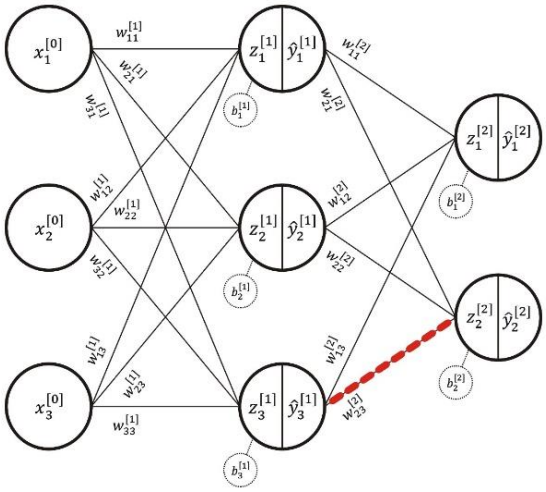
For $j = 2, i = 1$:

	$\frac{\partial c_p}{\partial w_{2,1}^{[2]}} = \left(\hat{y}_2^{[2]} - y_2\right) \times \hat{y}_1^{[1]}$ $= (0.544 - 1) \times 0.4953$ $= -0.2258$

For $j = 2, i = 2$:

	$\frac{\partial c_p}{\partial w_{2,2}^{[2]}} = \left(\hat{y}_2^{[2]} - y_2\right) \times \hat{y}_2^{[1]}$ $= (0.544 - 1) \times 0.5162$ $= -0.2354$

For $j = 2, i = 3$:

	$\frac{\partial c_p}{\partial w_{2,3}^{[2]}} = \left(\hat{y}_2^{[2]} - y_2\right) \times \hat{y}_3^{[1]}$ $= (0.544 - 1) \times 0.5097$ $= -0.2324$

Calculating gradients with respect to the biases using Eq. $(E.4)$:

For the first neuron, $j = 1$:

$$\frac{\partial c_p}{\partial b_1^{[2]}} = \hat{y}_1^{[2]} - y_1$$

$$= 0.438 - 0$$

$$= 0.438$$

For the second neuron, $j = 2$:

$$\frac{\partial c_p}{\partial b_2^{[2]}} = \hat{y}_2^{[2]} - y_2$$

$$= 0.544 - 1$$

$$= -0.456$$

Gradient of the cost function with respect to the weights and biases used in the layer $l = 1$:

For the Sigmoid activation function together with the Cross-Entropy cost function, we have for $k = 1,2,3$ and $i = 1,2,3$:

$$\frac{\partial c_p}{\partial w_{k,i}^{[1]}} = \delta_k^{[1]} \times \hat{y}_i^{[0]} \qquad \because \text{from Eq. } (B.23)$$

$$= \left(\sum_{j=1}^{2} \delta_j^{[2]} \times w_{j,k}^{[2]} \times \phi' \left(z_k^{[1]} \right) \right) \times \hat{y}_i^{[0]} \qquad \because \text{from Eq. } (B.28)$$

$$= \left(\sum_{j=1}^{2} \left(\hat{y}_j^{[2]} - y_j \right) \times w_{j,k}^{[2]} \right) \times \phi' \left(z_k^{[1]} \right) \times \hat{y}_i^{[0]} \qquad \because \text{from Eq. } (E.3)$$

$$= \left(\sum_{j=1}^{2} \left(\hat{y}_j^{[2]} - y_j \right) \times w_{j,k}^{[2]} \right) \times \hat{y}_k^{[1]} \left(1 - \hat{y}_k^{[1]} \right) \times \hat{y}_i^{[0]} \qquad \because \text{from Eq. } (B.18)$$

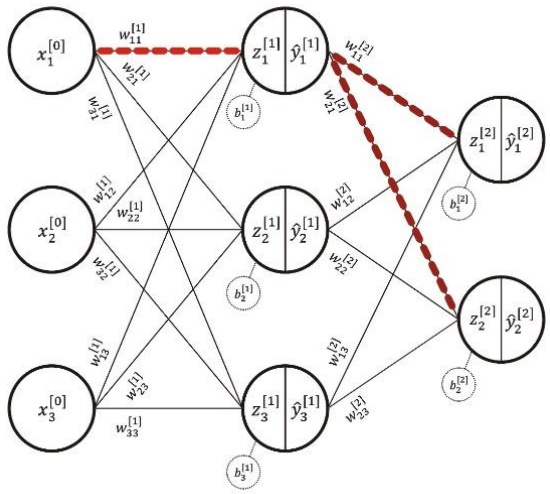
$$---(E.5)$$

Also

$$\frac{\partial c_p}{\partial b_k^{[1]}} = \delta_k^{[1]} = \left(\sum_{j=1}^{2} \left(\hat{y}_j^{[2]} - y_j \right) \times w_{j,k}^{[2]} \right) \times \hat{y}_k^{[1]} \left(1 - \hat{y}_k^{[1]} \right) \qquad ---(E.6)$$

Calculating gradients with respect to the weights using Eq. $(E.5)$:

For $k = 1, i = 1$:

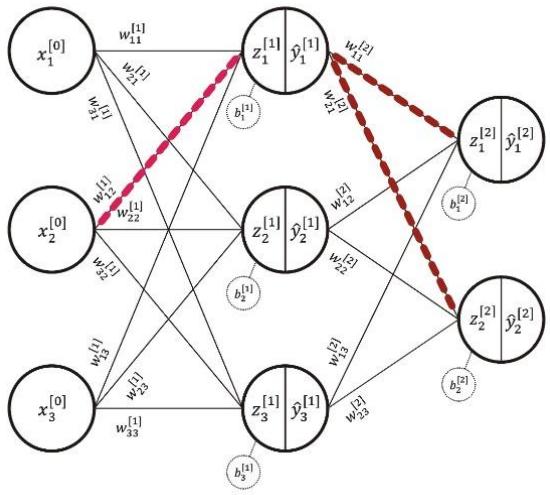


$$\frac{\partial c_p}{\partial w_{1,1}^{[1]}} = \left(\left(\hat{y}_1^{[2]} - y_1\right) \times w_{1,1}^{[2]} + \left(\hat{y}_2^{[2]} - y_2\right) \times w_{2,1}^{[2]}\right) \times \hat{y}_1^{[1]}\left(1 - \hat{y}_1^{[1]}\right) \times x_1^{[0]}$$

$$= ((0.438 - 0) \times 0.04 + (0.544 - 1) \times 0.20) \times 0.4953(1 - 0.4953) \times (0.8944)$$

$$= -0.0165$$

For $k = 1, i = 2$:

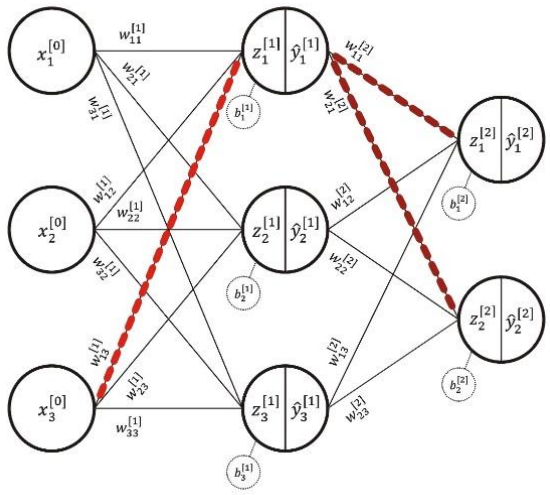


$$\frac{\partial c_p}{\partial w_{1,2}^{[1]}} = \left(\left(\hat{y}_1^{[2]} - y_1\right) \times w_{1,1}^{[2]} + \left(\hat{y}_1^{[2]} - y_2\right) \times w_{2,1}^{[2]}\right) \times \hat{y}_1^{[1]}\left(1 - \hat{y}_1^{[1]}\right) \times x_2^{[0]}$$

$$= ((0.438 - 0) \times 0.04 + (0.544 - 1) \times 0.20) \times 0.4953(1 - 0.4953) \times (-1.6703)$$

$$= 0.0556$$

For $k = 1, i = 3$:

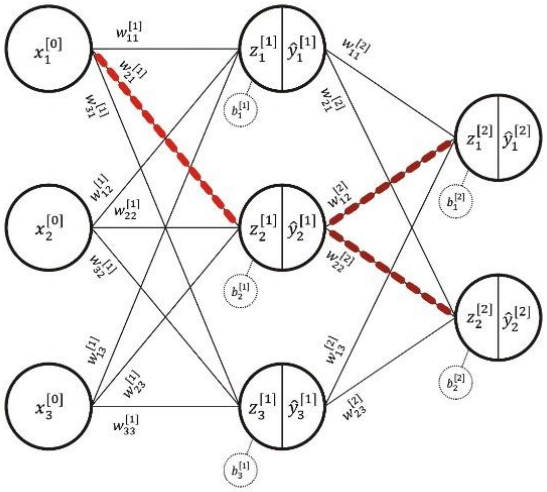


$$\frac{\partial c_p}{\partial w_{1,3}^{[1]}} = \left(\left(\hat{y}_1^{[2]} - y_1\right) \times w_{1,1}^{[2]} + \left(\hat{y}_2^{[2]} - y_2\right) \times w_{2,1}^{[2]}\right) \times \hat{y}_1^{[1]}\left(1 - \hat{y}_1^{[1]}\right) \times x_3^{[0]}$$

$$= ((0.438 - 0) \times 0.04 + (0.544 - 1) \times 0.20) \times 0.4953(1 - 0.4953) \times (-1.0452)$$

$$= 0.0414$$

For $k = 2, i = 1$:

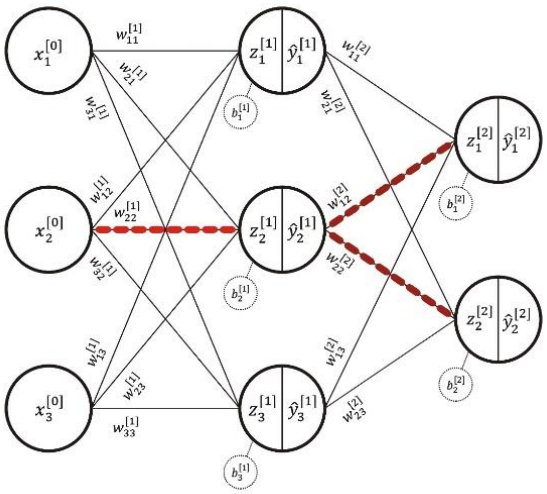


$$\frac{\partial c_p}{\partial w_{2,1}^{[1]}} = \left(\left(\hat{y}_1^{[2]} - y_1\right) \times w_{1,2}^{[2]} + \left(\hat{y}_2^{[2]} - y_2\right) \times w_{2,2}^{[2]}\right) \times \hat{y}_2^{[1]}\left(1 - \hat{y}_2^{[1]}\right) \times x_1^{[0]}$$

$$= ((0.438 - 0) \times 0.03 + (0.544 - 1) \times 0.05) \times 0.5162(1 - 0.5162) \times (0.8944)$$

$$= 0.0081$$

For $k = 2, i = 2$:

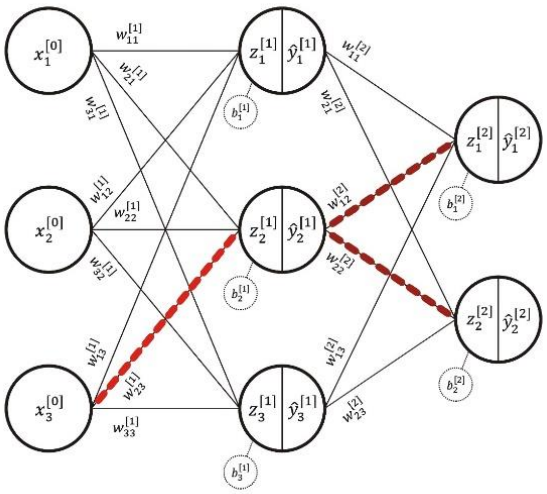


$$\frac{\partial c_p}{\partial w_{2,2}^{[1]}} = \left(\left(\hat{y}_1^{[2]} - y_1\right) \times w_{1,2}^{[2]} + \left(\hat{y}_2^{[2]} - y_2\right) \times w_{2,2}^{[2]}\right) \times \hat{y}_2^{[1]}\left(1 - \hat{y}_2^{[1]}\right) \times x_2^{[0]}$$

$$= ((0.438 - 0) \times 0.03 + (0.544 - 1) \times 0.05) \times 0.5162(1 - 0.5162) \times (-1.6705)$$

$$= -0.0226$$

For $k = 2, i = 3$:

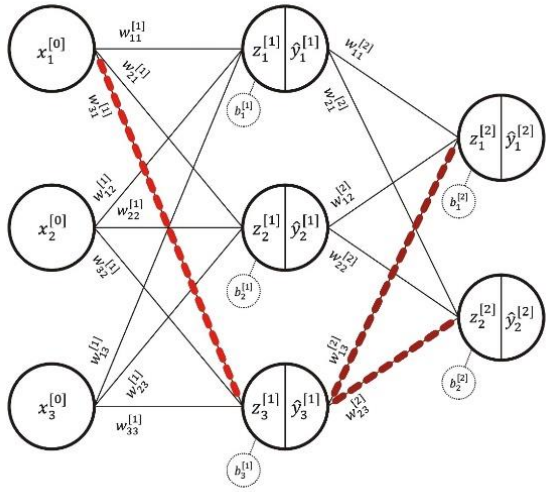


$$\frac{\partial c_p}{\partial w_{2,3}^{[1]}} = \left(\left(\hat{y}_1^{[2]} - y_1\right) \times w_{1,2}^{[2]} + \left(\hat{y}_2^{[2]} - y_2\right) \times w_{2,2}^{[2]}\right) \times \hat{y}_2^{[1]}\left(1 - \hat{y}_2^{[1]}\right) \times x_3^{[0]}$$

$$= ((0.438 - 0) \times 0.03 + (0.544 - 1) \times 0.05) \times 0.5162(1 - 0.5162) \times (-1.0452)$$

$$= -0.00191$$

For $k = 3, i = 1$:

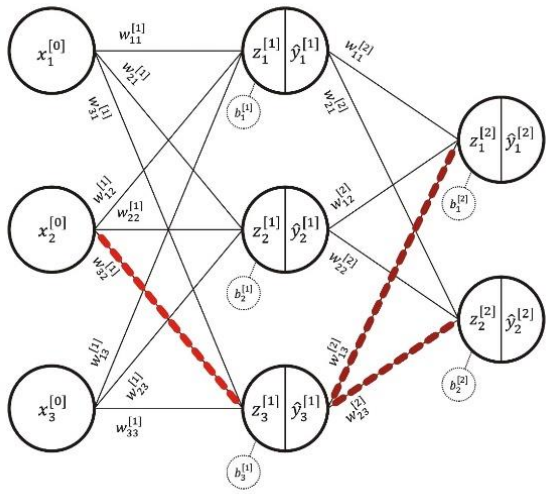


$$\frac{\partial c_p}{\partial w_{3,1}^{[1]}} = \left(\left(\hat{y}_1^{[2]} - y_1\right) \times w_{1,3}^{[2]} + \left(\hat{y}_2^{[2]} - y_2\right) \times w_{2,3}^{[2]}\right) \times \hat{y}_3^{[1]}\left(1 - \hat{y}_3^{[1]}\right) \times x_1^{[0]}$$

$$= ((0.438 - 0) \times 0.3 + (0.544 - 1) \times 0.02) \times 0.5097(1 - 0.5097) \times (0.8944)$$

$$= 0.1294$$

For $k = 3, i = 2$:

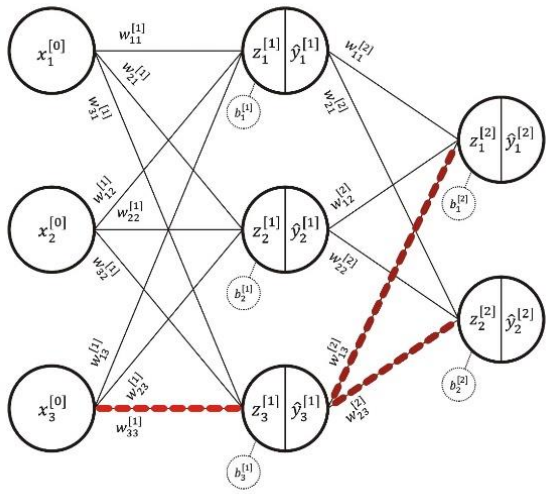


$$\frac{\partial c_p}{\partial w_{3,2}^{[1]}} = \left(\left(\hat{y}_1^{[2]} - y_1\right) \times w_{1,3}^{[2]} + \left(\hat{y}_2^{[2]} - y_2\right) \times w_{2,3}^{[2]}\right) \times \hat{y}_3^{[1]}\left(1 - \hat{y}_3^{[1]}\right) \times x_2^{[0]}$$

$$= ((0.438 - 0) \times 0.3 + (0.544 - 1) \times 0.02) \times 0.5097(1 - 0.5097) \times (-1.6703)$$

$$= 0.1352$$

For $k = 3, i = 3$:



$$\frac{\partial c_p}{\partial w_{3,3}^{[1]}} = \left(\left(\hat{y}_1^{[2]} - y_1\right) \times w_{1,3}^{[2]} + \left(\hat{y}_2^{[2]} - y_2\right) \times w_{2,3}^{[2]}\right) \times \hat{y}_3^{[1]}\left(1 - \hat{y}_3^{[1]}\right) \times x_3^{[0]}$$

$$= ((0.438 - 0) \times 0.3 + (0.544 - 1) \times 0.02) \times 0.5097(1 - 0.5097) \times (-1.0452)$$

$$= 0.1338$$

Calculating gradients with respect to the biases using Eq. $(E.6)$:

For first neuron, $k = 1$:

$$\begin{aligned}\frac{\partial c_p}{\partial b_1^{[1]}} &= \left[\left(\hat{y}_1^{[2]} - y_1\right) \times w_{1,1}^{[2]} + \left(\hat{y}_2^{[2]} - y_2\right) \times w_{2,1}^{[2]}\right] \times \hat{y}_1^{[1]}\left(1 - \hat{y}_1^{[1]}\right) \\ &= ((0.438 - 0) \times 0.04 + (0.544 - 1) \times 0.20) \times 0.4953(1 - 0.4953) \\ &= -0.0053\end{aligned}$$

For the second neuron, $k = 2$:

$$\begin{aligned}\frac{\partial c_p}{\partial b_2^{[1]}} &= \left[\left(\hat{y}_1^{[2]} - y_1\right) \times w_{1,2}^{[2]} + \left(\hat{y}_2^{[2]} - y_2\right) \times w_{2,2}^{[2]}\right] \times \hat{y}_2^{[1]}\left(1 - \hat{y}_2^{[1]}\right) \\ &= ((0.438 - 0) \times 0.03 + (0.544 - 1) \times 0.05) \times 0.5162(1 - 0.5162) \\ &= 0.0074\end{aligned}$$

For the third neuron, $k = 3$:

$$\begin{aligned}\frac{\partial c_p}{\partial b_3^{[1]}} &= \left[\left(\hat{y}_1^{[2]} - y_1\right) \times w_{1,3}^{[2]} + \left(\hat{y}_2^{[2]} - y_2\right) \times w_{2,3}^{[2]}\right] \times \hat{y}_3^{[1]}\left(1 - \hat{y}_3^{[1]}\right) \\ &= ((0.438 - 0) \times 0.3 + (0.544 - 1) \times 0.02) \times 0.5097(1 - 0.5097) \\ &= 0.1294\end{aligned}$$

6.8. Learning Phase: Updating the Parameters (Weights and Biases)

Using the Gradient Descent method, with the learning rate $\alpha = 0.5$, the parameters (weights and biases) are updated as follows.

$$w_{j,i}^{[l]} = w_{j,i}^{[l]} - \left(\alpha \times \frac{\partial c_p}{\partial w_{j,i}^{[l]}}\right) \quad ---(E.7)$$

$$b_j^{[l]} = b_j^{[l]} - \left(\alpha \times \frac{\partial c_p}{\partial b_j^{[l]}}\right) \quad ---(E.8)$$

For weight $w_{1,1}^{[2]}$:

$$w_{1,1}^{[2]} = w_{1,1}^{[2]} - \left(\alpha \times \frac{\partial c_p}{\partial w_{1,1}^{[2]}}\right) = 0.04 - \left(0.5 \times (0.7122)\right) = -0.0684$$

For weight $w_{1,2}^{[2]}$:

$$w_{1,2}^{[2]} = w_{1,2}^{[2]} - \left(\alpha \times \frac{\partial c_p}{\partial w_{1,2}^{[2]}}\right) = 0.03 - \left(0.5 \times (-0.7423)\right) = 0.0830$$

For weight $w_{1,3}^{[2]}$:

$$w_{1,3}^{[2]} = w_{1,3}^{[2]} - \left(\alpha \times \frac{\partial c_p}{\partial w_{1,3}^{[2]}}\right) = 0.3 - \left(0.5 \times (-0.7329)\right) = 0.1884$$

For weight $w_{2,1}^{[2]}$:

$$w_{2,1}^{[2]} = w_{2,1}^{[2]} - \left(\alpha \times \frac{\partial c_p}{\partial w_{2,1}^{[2]}}\right) = 0.2 - \left(0.5 \times (-0.2258)\right) = 0.3129$$

For weight $w_{2,2}^{[2]}$:

$$w_{2,2}^{[2]} = w_{2,2}^{[2]} - \left(\alpha \times \frac{\partial c_p}{\partial w_{2,2}^{[2]}}\right) = 0.05 - \left(0.5 \times (-0.2354)\right) = 0.1677$$

For weight $w_{2,3}^{[2]}$:

$$w_{2,3}^{[2]} = w_{2,3}^{[2]} - \left(\alpha \times \frac{\partial c_p}{\partial w_{2,3}^{[2]}}\right) = 0.02 - \left(0.5 \times (-0.2324)\right) = 0.1362$$

For weight $w_{1,1}^{[1]}$:

$$w_{1,1}^{[1]} = w_{1,1}^{[1]} - \left(\alpha \times \frac{\partial c_p}{\partial w_{1,1}^{[1]}}\right) = 0.01 - \left(0.5 \times (-0.0165\ )\right) = 0.0182$$

For weight $w_{1,2}^{[1]}$:

$$w_{1,2}^{[1]} = w_{1,2}^{[1]} - \left(\alpha \times \frac{\partial c_p}{\partial w_{1,2}^{[1]}}\right) = 0.02 - \left(0.5 \times (0.0556\ )\right) = 0.0078$$

For weight $w_{1,3}^{[1]}$:

$$w_{1,3}^{[1]} = w_{1,3}^{[1]} - \left(\alpha \times \frac{\partial c_p}{\partial w_{1,3}^{[1]}}\right) = 0.1 - \left(0.5 \times (0.0414)\right) = 0.0793$$

For weight $w_{2,1}^{[1]}$:

$$w_{2,1}^{[1]} = w_{2,1}^{[1]} - \left(\alpha \times \frac{\partial c_p}{\partial w_{2,1}^{[1]}}\right) = 0.12 - \left(0.5 \times (0.0081\ )\right) = 0.1159$$

For weight $w_{2,2}^{[1]}$:

$$w_{2,2}^{[1]} = w_{2,2}^{[1]} - \left(\alpha \times \frac{\partial c_p}{\partial w_{2,2}^{[1]}}\right) = 0.03 - \left(0.5 \times (\ 0.0226)\right) = 0.0187$$

For weight $w_{2,3}^{[1]}$:

$$w_{2,3}^{[1]} = w_{2,3}^{[1]} - \left(\alpha \times \frac{\partial c_p}{\partial w_{2,3}^{[1]}}\right) = 0.05 - \left(0.5 \times (0.0191)\right) = 0.0404$$

For weight $w_{3,1}^{[1]}$:

$$w_{3,1}^{[1]} = w_{3,1}^{[1]} - \left(\alpha \times \frac{\partial c_p}{\partial w_{3,1}^{[1]}}\right) = 0.06 - \left(0.5 \times (0.1294)\right) = -0.0047$$

For weight $w_{3,2}^{[1]}$:

$$w_{3,2}^{[1]} = w_{3,2}^{[1]} - \left(\alpha \times \frac{\partial c_p}{\partial w_{3,2}^{[1]}}\right) = 0.02 - \left(0.5 \times (0.1352)\right) = -0.0476$$

For weight $w_{3,3}^{[1]}$:

$$w_{3,3}^{[1]} = w_{3,3}^{[1]} - \left(\alpha \times \frac{\partial c_p}{\partial w_{3,3}^{[1]}}\right) = 0.03 - \left(0.5 \times (0.13378)\right) = -0.0369$$

For bias $b_1^{[2]}$:

$$b_1^{[2]} = b_1^{[2]} - \left(\alpha \times \frac{\partial c_p}{\partial b_1^{[2]}}\right) = 0.06 - \left(0.5 \times (0.438)\right) = 0.159$$

For bias $b_2^{[2]}$:

$$b_2^{[2]} = b_2^{[2]} - \left(\alpha \times \frac{\partial c_p}{\partial b_2^{[2]}}\right) = 0.04 - \left(0.5 \times (-0.456)\right) = 0.268$$

For bias $b_1^{[1]}$:

$$b_1^{[1]} = b_1^{[1]} - \left(\alpha \times \frac{\partial c_p}{\partial b_1^{[1]}}\right) = 0.11 - \left(0.5 \times (-0.005278)\right) = 0.1126$$

For bias $b_2^{[1]}$:

$$b_2^{[1]} = b_2^{[1]} - \left(\alpha \times \frac{\partial c_p}{\partial b_2^{[1]}}\right) = 0.06 - \left(0.5 \times (0.00745)\right) = 0.0563$$

For bias $b_3^{[1]}$:

$$b_3^{[1]} = b_3^{[1]} - \left(\alpha \times \frac{\partial c_p}{\partial b_3^{[1]}}\right) = 0.05 - \left(0.5 \times (0.1291)\right) = -0.01455$$

The updated weights and biases are given as follows.

For the layer $l = 1$:

$$W_1^{[1]} = \begin{bmatrix} w_{1,1}^{[1]} & w_{1,2}^{[1]} & w_{1,3}^{[1]} \end{bmatrix}^T = [0.0137 \quad 0.0130 \quad 0.0956]$$
$$W_2^{[1]} = \begin{bmatrix} w_{2,1}^{[1]} & w_{2,2}^{[1]} & w_{2,3}^{[1]} \end{bmatrix}^T = [0.1177 \quad 0.0342 \quad 0.0526]$$
$$W_3^{[1]} = \begin{bmatrix} w_{3,1}^{[1]} & w_{3,2}^{[1]} & w_{3,3}^{[1]} \end{bmatrix}^T = [0.0128 \quad -0.0681 \quad -0.0251]$$
$$B^{[1]} = \begin{bmatrix} b_1^{[1]} & b_2^{[1]} & b_3^{[1]} \end{bmatrix}^T = [0.1142 \quad 0.0574 \quad -0.0027]$$

For the layer $l = L = 2$:

$$W_1^{[2]} = \begin{bmatrix} w_{1,1}^{[2]} & w_{1,2}^{[2]} & w_{1,3}^{[2]} \end{bmatrix}^T = [-0.3161 \quad 0.4011 \quad 0.6664]$$
$$W_2^{[2]} = \begin{bmatrix} w_{2,1}^{[2]} & w_{2,2}^{[2]} & w_{2,3}^{[2]} \end{bmatrix}^T = [0.3129 \quad 0.1677 \quad 0.1362]$$
$$B^{[2]} = \begin{bmatrix} b_1^{[2]} & b_2^{[2]} \end{bmatrix}^T = [-0.659 \quad 0.268]$$

The updated parameters will be used for the next randomly chosen training instance and the process is repeated until all the training instances have been passed through the network. This completes one epoch. Such epochs are repeated until the desired accuracy of the error function is achieved.

7. Algorithm for ANN Computations with Vectorized Implementation

Vectorization is a technique used to optimize the implementation of machine learning algorithms, including neural networks. In a vectorized implementation, the computations of a neural network are represented as matrix operations. The input data, weights, biases, and activations of each layer of the network are stored as matrices or tensors, and the computations between layers are performed as matrix multiplications, additions, and other operations. The computations of the network are performed on entire batches of data simultaneously rather than performing element-wise operations using loops.

The computations of other types of layers, such as convolutional or recurrent layers, can also be implemented efficiently with vectorized implementation. In practice, most modern deep learning frameworks, such as TensorFlow and PyTorch, also provide highly optimized vectorized implementations of common neural network operations. The benefits of a vectorized implementation of a neural network include:

- Speed: Vectorized operations can be efficiently parallelized and optimized, leading to significant speedups compared to non-vectorized implementations.
- Memory efficiency: By processing multiple data points or batches simultaneously, vectorized implementations can achieve high throughput and reduce the memory requirements of the computations.
- Simplified code: Vectorized operations often require less code and are more intuitive to write and read than non-vectorized operations, leading to simpler and more maintainable code.
- Better use of hardware resources: Vectorized implementations can take advantage of specialized hardware such as GPUs, Vector registers, and Tensor-cores for efficiency.
- Scalability: Vectorized implementations can be easily scaled to handle large datasets and complex models, making them well-suited for deep learning applications.

7.1. Description of the Algorithm Variables

Input and Output Variables:

N: an integer as the number of training instances

m: an integer as the number of training instances in a step/iteration.

γ: an integer as the number of mini-batches.

> Note that $m = N$ (and $\gamma = 1$) for the Batch-Gradient Descent, $m = 1$ (and $\gamma = N$) for the Stochastic Gradient Descent method, and $1 < m < N$ for the Mini-Batch Gradient Descent.

tol: a scalar as the tolerance (or permissible error to be used as a stopping criterion)

MAX_{epochs}: an integer as the maximum number of *epochs*. Recall that there are γ mini-batches of m training instances each, such that $\gamma m = N$. So γ iterations/steps are required to complete one *epoch* (if the sampling is made *without replacement*).

L: an integer as the number of layers (other than the input layer) in the neural network.

$ldim$: a vector of $L + 1$ components such that lth component is n_l, representing the numbers of neurons in layer l, for $l = 0,1,2, \cdots, L$. Note that $l = 0$ is the input layer having n_0 neurons.

α: a scalar as the learning rate for the optimization method

$\boldsymbol{M}$: an $n_0 \times m$ matrix of m column-vectors, each representing a training instance having n_0 attributes (also called features or independent variables, represented by $x_j^{[0](p)}$). Each pth instance has the form:

$$X^{[0](p)} = \begin{bmatrix} x_1^{[0](p)} \\ x_2^{[0](p)} \\ \vdots \\ x_{n_0}^{[0](p)} \end{bmatrix}_{n_0 \times 1}$$

Thus, the matrix $\boldsymbol{M}$ has the form (horizontal stacking of m column vectors for vectorized implementation) (also see Fig. 7.1):

$$M = \begin{bmatrix} | & | & & | \\ X^{[0](1)} & X^{[0](2)} & \dots & X^{[0](m)} \\ | & | & & | \end{bmatrix}_{n_0 \times m}$$

If $m = 1$, then $\boldsymbol{M}$ has only one instance. If $m = N$, then $\boldsymbol{M}$ is the set of all training instances. In general, $\boldsymbol{M}$ is a subset (minibatch) of the whole set $\bar{\boldsymbol{M}}$ of N training instances. So, if we partition $\bar{\boldsymbol{M}}$, there can be γ disjoint minibatches in $\bar{\boldsymbol{M}}$, where $\gamma m = N$. That said,

$$\bar{\boldsymbol{M}} = \begin{bmatrix} — \boldsymbol{M}^{\{1\}} — & — \boldsymbol{M}^{\{2\}} — & \dots & — \boldsymbol{M}^{\{\gamma\}} — \end{bmatrix}_{n_0 \times N}$$

However, for the stochastic approach, any mini-batch $\boldsymbol{M}$ is a sample of randomly chosen m instances from $\bar{\boldsymbol{M}}$.

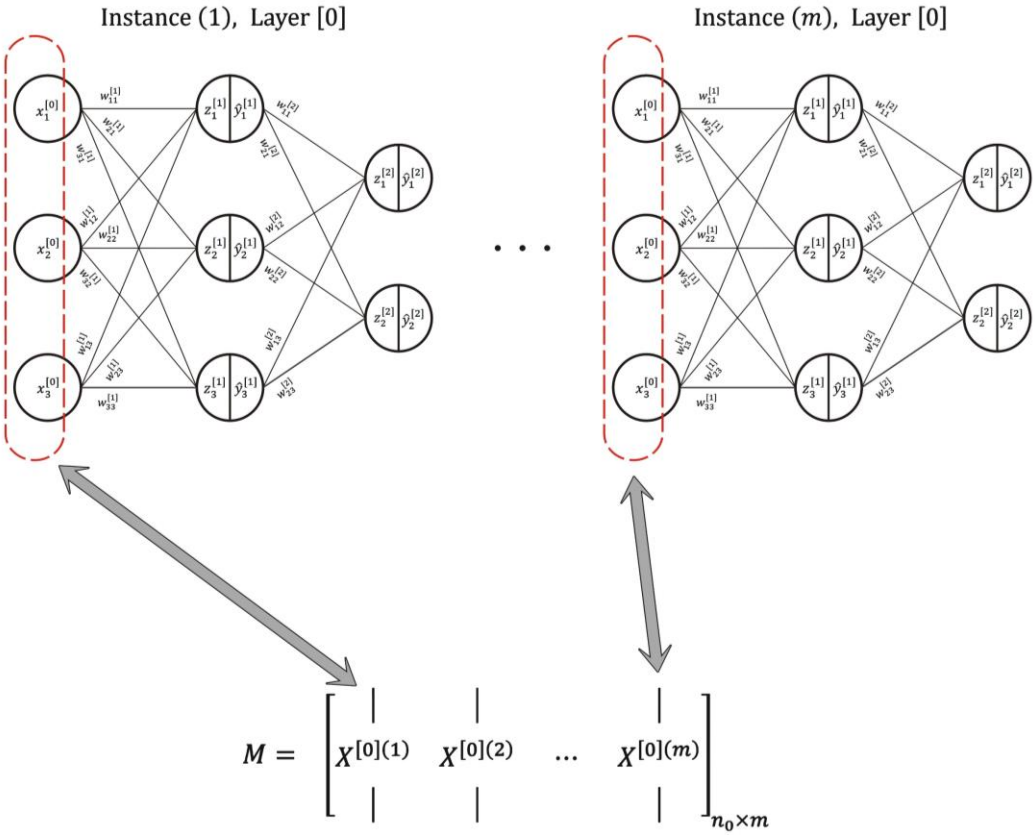


Fig. 7.1: An example of the structure of $\boldsymbol{M}$ matrix, for the input layer $l =$, $n_2 = 2$, and $n_1 = 3$.

$\boldsymbol{Y}$: an $n_L \times m$ matrix of true values/targets/labels, corresponding to the instances in $\boldsymbol{M}$, such that its pth column vector has the form:

$$Y^{(p)} = \begin{bmatrix} y_1^{(p)} \\ y_2^{(p)} \\ \vdots \\ y_{n_L}^{(p)} \end{bmatrix}_{n_L \times 1}$$

$Y^{(p)}$ is the vector of n_L true values/targets/labels $y_j^{(p)}$, for instance p represented by pth column in $\boldsymbol{M}$. Thus, the matrix $\boldsymbol{Y}$ has the form (horizontal stacking of m column vectors for vectorized implementation):

$$\boldsymbol{Y} = \begin{bmatrix} | & | & & | \\ Y^{(1)} & Y^{(2)} & \dots & Y^{(m)} \\ | & | & & | \end{bmatrix}_{n_L \times m}$$

In general, $\boldsymbol{Y}$ is a subset (minibatch) of the whole set $\overline{\boldsymbol{Y}}$ of N target labels corresponding to the instances in $\boldsymbol{M}$. Thus, if we partition $\overline{\boldsymbol{Y}}$, there can be γ disjoint subsets in $\overline{\boldsymbol{Y}}$, where $\gamma m = N$. That said,

$$\overline{\boldsymbol{Y}} = \begin{bmatrix} \_\_ \boldsymbol{Y}^{\{1\}} \_\_ & \_\_ \boldsymbol{Y}^{\{2\}} \_\_ & \dots & \_\_ \boldsymbol{Y}^{\{\gamma\}} \_\_ \end{bmatrix}_{n_L \times N}$$

A subset $\boldsymbol{Y}$ is sampled from $\overline{\boldsymbol{Y}}$ with the same indices as the corresponding mini-batch $\boldsymbol{M}$ is sampled from $\overline{\boldsymbol{M}}$.

$\overline{\boldsymbol{W}}^{[l]}$: an $n_l \times (n_{l-1})$ matrix of weights such that its jth row is the transpose of the column vector of the form:

$$W_j^{[l]} = \begin{bmatrix} w_{j,1}^{[l]} \\ w_{j,2}^{[l]} \\ \vdots \\ w_{j,n_{l-1}}^{[l]} \end{bmatrix}_{n_{l-1}\times 1}$$

$W_j^{[l]}$ is used for its dot product with the vector of n_{l-1} neurons of layer $l-1$ to generate jth neuron of layer l. It has n_{l-1} components because there are n_{l-1} neurons in layer $l-1$. Thus, the matrix $\overline{\boldsymbol{W}}^{[l]}$ has the form (vertical stacking of n_l column vectors for vectorized implementation) (also see Fig. 7.2):

$$\overline{\boldsymbol{W}}^{[l]} = \begin{bmatrix} -- & W_1^{[l]T} & -- \\ -- & W_2^{[l]T} & -- \\ & \vdots & \\ -- & W_{n_l}^{[l]T} & -- \end{bmatrix}_{n_l\times(n_{l-1})}$$

For each layer l, where $l = 1,2,3,\cdots,L$, a different $\overline{\boldsymbol{W}}^{[l]}$ matrix is required.

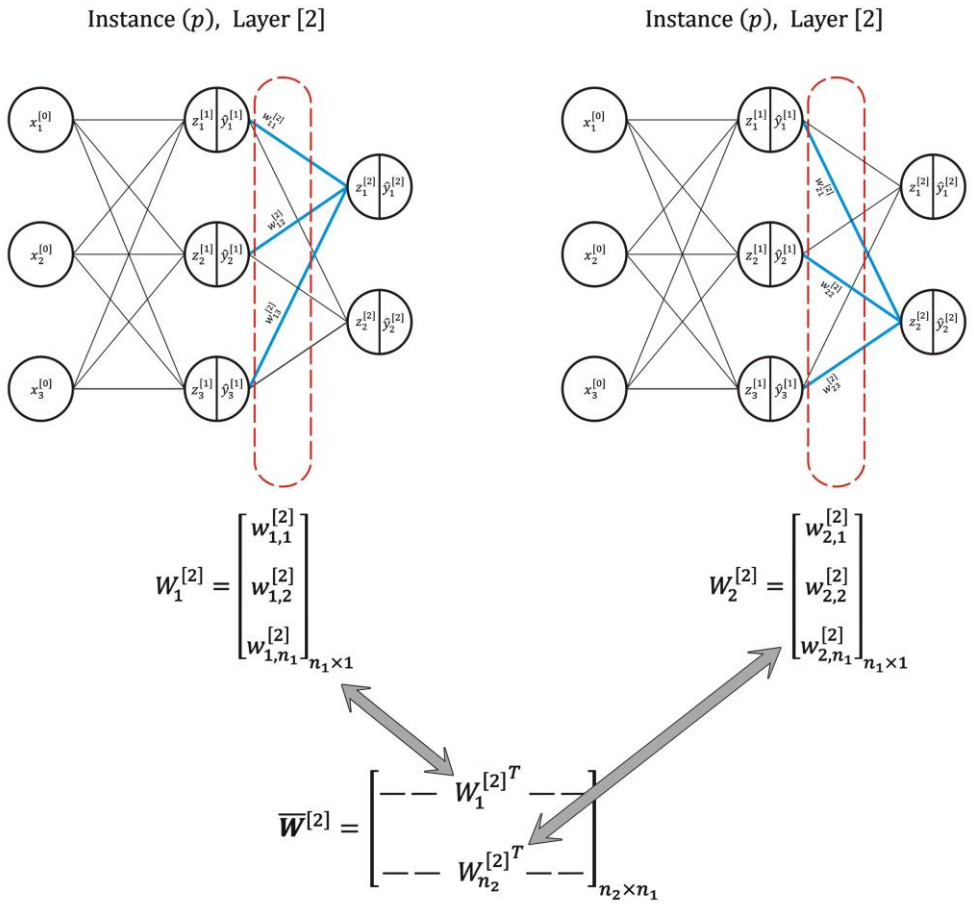


Fig. 7.2: An example of the structure of $\overline{\boldsymbol{W}}^{[l]}$ matrix, for layer $l = 2$, $n_2 = 2$, and $n_1 = 3$.

$B^{[l]}$: a vector of n_l biases such that its jth component is to be used for generating the jth neuron in layer l. $B^{[l]}$ has the form (also see Fig. 7.3):

$$B^{[l]} = \begin{bmatrix} b_1^{[l]} \\ b_2^{[l]} \\ \vdots \\ b_{n_l}^{[l]} \end{bmatrix}_{n_l \times 1}$$

For each layer l, where $l = 1,2,3,\cdots,L$, a different $B^{[l]}$ vector is required.

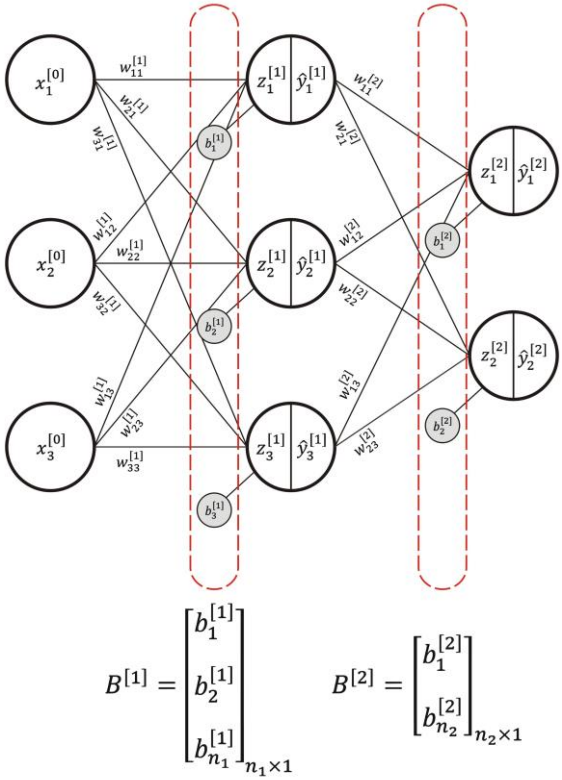


Fig. 7.3: An example of the structure of $B^{[l]}$ matrix, for layers $l = 1$ and 2.

Auxiliary Variables:

$\boldsymbol{Z}^{[l]}$: an $n_l \times m$ matrix of pre-activation neurons (linear neurons) such that its pth column vector has the form:

$$Z^{[l](p)} = \begin{bmatrix} z_1^{[l](p)} \\ z_2^{[l](p)} \\ \vdots \\ z_{n_l}^{[l](p)} \end{bmatrix}_{n_l \times 1} = \begin{bmatrix} W_1^{[l]^T} X^{[l-1](p)} + b_1^{[l]} \\ W_2^{[l]^T} X^{[l-1](p)} + b_2^{[l]} \\ \vdots \\ W_{n_l}^{[l]^T} X^{[l-1](p)} + b_{n_l}^{[l]} \end{bmatrix}_{n_l \times 1}$$

$Z^{[l](p)}$ is the vector of pre-activation neurons in layer l of the network for instance p. It has n_l components because there are n_l neurons in layer l. Thus, the matrix $\boldsymbol{Z}^{[l]} = \overline{\boldsymbol{W}}^{[l]}\, \widehat{\boldsymbol{Y}}^{[l-1]} + B^{[l]}$ has the form (horizontal stacking of m column vectors for vectorized implementation) (also see Fig. 7.4):

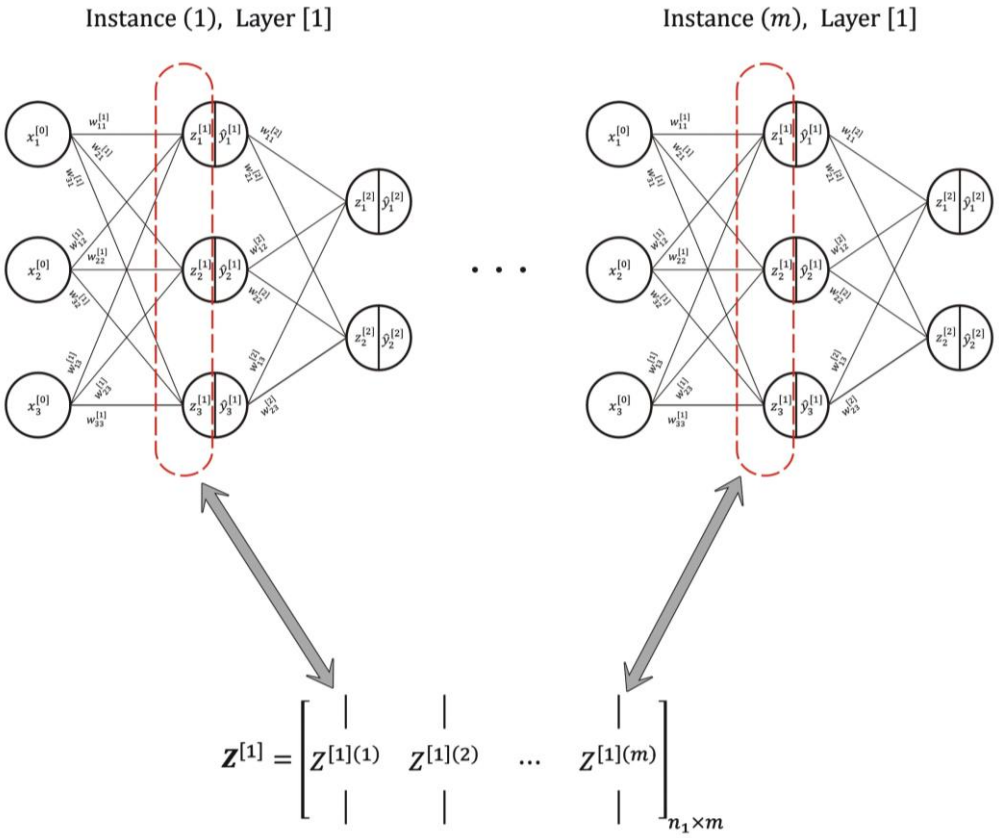


Fig. 7.4: An example of the structure of $\boldsymbol{Z}^{[1]}$ matrix, for layer $l = 1$.

$$\boldsymbol{Z}^{[l]} = \begin{bmatrix} | & | & & | \\ Z^{[l](1)} & Z^{[l](2)} & \dots & Z^{[l](m)} \\ | & | & & | \end{bmatrix}_{n_l \times m}$$

For each layer l, where $l = 1,2,3, \cdots, L$, a different $\boldsymbol{Z}^{[l]}$ matrix is required.

$\widehat{\boldsymbol{Y}}^{[l]}$: an $n_l \times m$ matrix of the post-activation neurons (non-linear neurons) of layer l such that its pth column vector has the form:

$$\hat{Y}^{[l](p)} = \begin{bmatrix} \hat{y}_1^{[l](p)} \\ \hat{y}_2^{[l](p)} \\ \vdots \\ \hat{y}_{n_l}^{[l](p)} \end{bmatrix}_{n_l \times 1} = \begin{bmatrix} \phi\left(z_1^{[l](p)}\right) \\ \phi\left(z_2^{[l](p)}\right) \\ \vdots \\ \phi\left(z_{n_l}^{[l](p)}\right) \end{bmatrix}_{n_l \times 1}$$

$\hat{Y}^{[l](p)}$ is the vector of post-activation neurons in layer l of the network for instance p. It has n_l components because there are n_l neurons in layer l. Thus, the matrix $\widehat{\boldsymbol{Y}}^{[l]} = \boldsymbol{\phi}\left(\boldsymbol{Z}^{[l]}\right)$ has the form (horizontal stacking of m column vectors):

$$\widehat{\boldsymbol{Y}}^{[l]} = \begin{bmatrix} | & | & & | \\ \hat{Y}^{[l](1)} & \hat{Y}^{[l](2)} & \dots & \hat{Y}^{[l](m)} \\ | & | & & | \end{bmatrix}_{n_l \times m}$$

For each layer l, where $l = 1,2,3, \cdots, L$, a different $\widehat{\boldsymbol{Y}}^{[l]}$ matrix is required. Moreover, $\widehat{\boldsymbol{Y}}^{[0]} = M$.

Note that for $l = L$, $\widehat{\boldsymbol{Y}}^{[L]}$ represents the final output neurons (predicted values) in the output layer L. Thus, for m instances,

$$\widehat{\boldsymbol{Y}}^{[L]} = \begin{bmatrix} | & | & & | \\ \hat{Y}^{[L](1)} & \hat{Y}^{[L](2)} & \dots & \hat{Y}^{[L](m)} \\ | & | & & | \end{bmatrix}_{n_L \times m}$$

Note that the pth column vector in the input matrix $\boldsymbol{M}$, the pth column vector in the output matrix $\widehat{\boldsymbol{Y}}^{[L]}$, and pth column vector in the target matrix $\boldsymbol{Y}$ corresponds to the pth instance in the training set.

It is pertinent to note the following:

- For regression problems, $n_L = 1$ is usually sufficient. The goal is to predict a single continuous numeric value. Hence, the single output neuron represents the prediction for the corresponding instance. Thus, each column in $\widehat{\boldsymbol{Y}}^{[L]}$ represents the predicted numeric value, and each column in $\boldsymbol{Y}$ represents a single target value.

  Using multiple neurons in the output layer for regression is possible in certain cases, such as there are multiple continuous target variables to predict simultaneously (multi-output regression). Each neuron in the output layer would then represent one of the target variables.

- For binary classification problems, $n_L = 1$, is usually sufficient. The single neuron in the output layer represents the predicted probability for the corresponding instance for belonging to one of the two classes. Thus, each column in $\widehat{\boldsymbol{Y}}^{[L]}$ represents the predicted probability, and each column in $\boldsymbol{Y}$ represents a single target value, either 0 or 1. To make the final prediction, the typical decision rule is as follows: If the predicted probability is greater than or equal to 0.5, then the corresponding instance is considered to belong to that class (called the positive class, labeled "1"), otherwise not (i.e., it belongs to the negative class, labeled "0").

- For multiclass classification problems, each neuron in the output layer represents a possible class, i.e., n_L equals the number of classes. Each column in $\widehat{\boldsymbol{Y}}^{[L]}$ represents the predicted probabilities on n_L neurons for the corresponding instance for belonging to each class. Further, each column in $\boldsymbol{Y}$ has n_L entries that are all 0s except the one entry (that is, 1), which corresponds to the particular class to which the corresponding instance belongs.

  To make the final prediction, a decision rule, such as choosing the class with the higher probability or comparing the probabilities against a threshold, can be applied. Even for binary classification problems, setting n_L as 2 can also work. In that case, each neuron represents the predicted probability or score for one of the two classes.

During training, the neural network learns to adjust its weights and biases to minimize the difference between the predicted output (neuron values in the output layer) of the network and the actual target values. For regression problems, the Mean-Squared Error Loss function is a more appropriate choice. For classification problems, the Binary/Multiclass Cross Entropy Loss function is a more appropriate choice.

C_m: a vector of m errors (loss functions) having the form:

$$C_m = \begin{bmatrix} c^{(1)} \\ c^{(2)} \\ \vdots \\ c^{(m)} \end{bmatrix}_{m\times 1} = \begin{bmatrix} E\left(Y^{(1)}, \hat{Y}^{[L](1)}\right) \\ E\left(Y^{(2)}, \hat{Y}^{[L](2)}\right) \\ \vdots \\ E\left(Y^{(m)}, \hat{Y}^{[L](m)}\right) \end{bmatrix}_{m\times 1}$$

Its pth component c_p or $c^{(p)}$, for $p = 1,2,3,\cdots,m$, represents the error, defined by an appropriate loss (or cost) function $E = E\left(Y^{(p)}, \hat{Y}^{[L](p)}\right)$, between the pth target vector in $\boldsymbol{Y}$ and the pth output vector in $\hat{\boldsymbol{Y}}^{[L]}$.

C: a scalar value as the cost function, which is the average of errors or loss function values present in the vector C_m. C can be expressed as:

$$C = \frac{1}{m}\sum_{p=1}^{m} c^{(p)} = \frac{1}{m}\sum_{p=1}^{m} E\left(Y^{(p)}, \hat{Y}^{[L](p)}\right)$$

It may be noted that the cost function C depends on all the weights and biases:

$$C = C(\boldsymbol{W}^{[1]}, B^{[1]}, \quad \boldsymbol{W}^{[2]}, B^{[2]}, \quad \cdots, \quad \boldsymbol{W}^{[L]}, B^{[L]})$$

$\boldsymbol{dZ}^{[l]}$: an $n_l \times m$ matrix of gradients of the cost function C with respect to the pre-activation neurons (linear neurons). $\boldsymbol{dZ}^{[l]}$ has the form (horizontal stacking of m column vectors for vectorized implementation):

$$\boldsymbol{dZ}^{[l]} = \begin{bmatrix} | & | & & | \\ dZ^{[l](1)} & dZ^{[l](2)} & \cdots & dZ^{[l](m)} \\ | & | & & | \end{bmatrix}_{n_l\times m}$$

Its pth component, $dZ^{[l](p)}$ is a column vector of the gradients of C with respect to the pre-activation neurons in layer l of the network for instance p. It has n_l components because there are n_l neurons in layer l. As $\boldsymbol{dZ}^{[l]}$ is formed for layer l. Therefore, for each $l = 1,2,3,\cdots,L$, a different $\boldsymbol{dZ}^{[l]}$ matrix is required. For the output layer, L, for instance p, $dZ^{[L](p)}$ is given by

$$dZ^{[L](p)} = \begin{bmatrix} \delta_1^{[L](p)} \\ \delta_2^{[L](p)} \\ \vdots \\ \delta_{n_L}^{[L](p)} \end{bmatrix}_{n_L\times 1} = \begin{bmatrix} \dfrac{\partial c^{(p)}}{\partial \hat{y}_1^{[L](p)}} \times \phi'\left(z_1^{[L](p)}\right) \\ \dfrac{\partial c^{(p)}}{\partial \hat{y}_2^{[L](p)}} \times \phi'\left(z_2^{[L](p)}\right) \\ \vdots \\ \dfrac{\partial c^{(p)}}{\partial \hat{y}_{n_L}^{[L](p)}} \times \phi'\left(z_{n_L}^{[L](p)}\right) \end{bmatrix}_{n_L\times 1}$$

For the layers, $l = L-1, L-2, \cdots, 1$, for instance p, $dZ^{[l](p)}$ is given by

$$dZ^{[l](p)} = \begin{bmatrix} \delta_1^{[l](p)} \\ \delta_2^{[l](p)} \\ \vdots \\ \delta_{n_l}^{[l](p)} \end{bmatrix}_{n_L \times 1} = \begin{bmatrix} \sum_{t=1}^{n_{l+1}} \delta_t^{[l+1](p)} \times w_{t,1}^{[l+1](p)} \times \phi'\left(z_1^{[l](p)}\right) \\ \sum_{t=1}^{n_{l+1}} \delta_t^{[l+1](p)} \times w_{t,2}^{[l+1](p)} \times \phi'\left(z_2^{[l](p)}\right) \\ \vdots \\ \sum_{t=1}^{n_{l+1}} \delta_t^{[l+1](p)} \times w_{t,n_l}^{[l+1](p)} \times \phi'\left(z_{n_l}^{[l](p)}\right) \end{bmatrix}_{n_l \times 1}$$

$\boldsymbol{d\hat{Y}}^{[L]}$: an $n_L \times m$ matrix of gradients of the cost function C with respect to the post-activation neurons (non-linear neurons) at the output layer, L, such that its pth column vector has the form:

$$d\hat{Y}^{[L](p)} = \begin{bmatrix} \frac{\partial c^{(p)}}{\partial \hat{y}_1^{[L](p)}} \\ \frac{\partial c^{(p)}}{\partial \hat{y}_2^{[L](p)}} \\ \vdots \\ \frac{\partial c^{(p)}}{\partial \hat{y}_{n_L}^{[L](p)}} \end{bmatrix}_{n_L \times 1} = \begin{bmatrix} E'\left(y_1^{(p)}, \hat{y}_1^{[L](p)}\right) \\ E'\left(y_2^{(p)}, \hat{y}_2^{[L](p)}\right) \\ \vdots \\ E'\left(y_{n_L}^{(p)}, \hat{y}_{n_L}^{[L](p)}\right) \end{bmatrix}_{n_L \times 1}$$

$d\hat{Y}^{[L](p)}$ is the vector of the gradients of C with respect to the post-activation neurons in the output layer L of the network for instance p. It has n_L components because there are n_L neurons in layer L. Thus, the matrix $\boldsymbol{d\hat{Y}}^{[L]}$ has the form (horizontal stacking of m column vectors for vectorized implementation):

$$\boldsymbol{d\hat{Y}}^{[L]} = \begin{bmatrix} | & | & & | \\ d\hat{Y}^{[L](1)} & d\hat{Y}^{[L](2)} & \cdots & d\hat{Y}^{[L](m)} \\ | & | & & | \end{bmatrix}_{n_L \times m}$$

$\boldsymbol{d\overline{W}}^{[l]}$: an $n_l \times (n_{l-1})$ matrix of gradients of the cost function C with respect to each of the weights such that its jth row is the transpose of the column vector of the form:

$$dW_j^{[l]} = \begin{bmatrix} \frac{\partial C}{\partial w_{j,1}^{[l]}} \\ \frac{\partial C}{\partial w_{j,2}^{[l]}} \\ \vdots \\ \frac{\partial C}{\partial w_{j,n_{l-1}}^{[l]}} \end{bmatrix}_{n_{l-1} \times 1} = \begin{bmatrix} \frac{1}{m}\sum_{p=1}^{m} \frac{\partial c^{(p)}}{\partial w_{j,1}^{[l]}} \\ \frac{1}{m}\sum_{p=1}^{m} \frac{\partial c^{(p)}}{\partial w_{j,2}^{[l]}} \\ \vdots \\ \frac{1}{m}\sum_{p=1}^{m} \frac{\partial c^{(p)}}{\partial w_{j,n_{l-1}}^{[l]}} \end{bmatrix}_{n_{l-1} \times 1} = \begin{bmatrix} \frac{1}{m}\sum_{p=1}^{m} \left(\delta_j^{[l](p)} \times \hat{y}_1^{[l-1](p)}\right) \\ \frac{1}{m}\sum_{p=1}^{m} \left(\delta_j^{[l](p)} \times \hat{y}_2^{[l-1](p)}\right) \\ \vdots \\ \frac{1}{m}\sum_{p=1}^{m} \left(\delta_j^{[l](p)} \times \hat{y}_{n_{l-1}}^{[l-1](p)}\right) \end{bmatrix}_{n_{l-1} \times 1}$$

$dW_j^{[l]}$ is the vector of gradients of C with respect to each of n_{l-1} weights that were used for generating the jth neuron of layer l. Thus, the matrix $\boldsymbol{d\overline{W}}^{[l]}$ has the form (vertical stacking of n_l column vectors for vectorized implementation):

$$\boldsymbol{d\overline{W}}^{[l]} = \begin{bmatrix} — — \ dW_1^{[l]^T} — — \\ — — \ dW_2^{[l]^T} — — \\ \vdots \\ — — \ dW_{n_l}^{[l]^T} — — \end{bmatrix}_{n_l \times (n_{l-1})}$$

For each layer l, where $l = 1,2,3,\cdots,L$, a different $\boldsymbol{d\overline{W}}^{[l]}$ matrix is required.

$dB^{[l]}$: a $n_l \times 1$ vector of gradients of C with respect to each of n_l biases such that its jth component is the gradient of C with respect to the bias b_j that was used for generating the jth neuron in layer l.

$$dB^{[l]} = \begin{bmatrix} \frac{\partial C}{\partial b_1^{[l]}} \\ \frac{\partial C}{\partial b_2^{[l]}} \\ \vdots \\ \frac{\partial C}{\partial b_{n_l}^{[l]}} \end{bmatrix}_{n_l \times 1} = \begin{bmatrix} \frac{1}{m}\sum_{p=1}^{m} \frac{\partial c^{(p)}}{\partial b_1^{[l]}} \\ \frac{1}{m}\sum_{p=1}^{m} \frac{\partial c^{(p)}}{\partial b_2^{[l]}} \\ \vdots \\ \frac{1}{m}\sum_{p=1}^{m} \frac{\partial c^{(p)}}{\partial b_{n_l}^{[l]}} \end{bmatrix}_{n_l \times 1} = \begin{bmatrix} \frac{1}{m}\sum_{p=1}^{m} \left(\delta_1^{[l](p)}\right) \\ \frac{1}{m}\sum_{p=1}^{m} \left(\delta_2^{[l](p)}\right) \\ \vdots \\ \frac{1}{m}\sum_{p=1}^{m} \left(\delta_{n_l}^{[l](p)}\right) \end{bmatrix}_{n_l \times 1}$$

For each layer l, where $l = 1,2,3,\cdots,L$, a different $dB^{[l]}$ vector is required.

7.2. The Procedure

Set N, the total number of training instances

Set γ, as the number of mini-batches

Set $m = \frac{N}{\gamma}$, the number of training instances in a minibatch

Set tol, as the tolerance

Set MAX_{epochs} as the maximum number of *epochs*

Set L, the number of layers (other than the input layer) in the neural network

Set $ldim$, with lth component n_l as the number of neurons in layer l, for $l = 0,1,2,\cdots,L$

Set α, the learning rate

Set $\overline{\boldsymbol{M}}$, the matrix of order $n_0 \times N$ having feature values of all the training instances

Set $\overline{\boldsymbol{Y}}$, the matrix of order $n_L \times N$ having target values of all the training instances (as given in $\overline{\boldsymbol{M}}$).

Declare L data structures for each of the following with respective sizes:

$\left(\overline{\boldsymbol{W}}^{[l]}\right)_{n_l\times(n_{l-1})}$, $\left(B^{[l]}\right)_{n_l\times 1}$, $\left(\boldsymbol{Z}^{[l]}\right)_{n_l\times m}$, $\left(\widehat{\boldsymbol{Y}}^{[l]}\right)_{n_l\times m}$

Declare one data structure for each of the following with respective sizes:

$(\boldsymbol{Y})_{n_L\times m}$, $\left(\boldsymbol{d}\widehat{\boldsymbol{Y}}^{[L]}\right)_{n_L\times m}$, $(C_m)_{m\times 1}$

Declare L data structures for each of the following with respective sizes:

$\left(\boldsymbol{dZ}^{[l]}\right)_{n_l\times m}$, $\left(\boldsymbol{d}\overline{\boldsymbol{W}}^{[l]}\right)_{n_l\times(n_{l-1})}$, $\left(dB^{[l]}\right)_{n_l\times 1}$

Initialize weights, $\overline{\boldsymbol{W}}^{[l]}$, for layer $l = 1,2,\cdots,L$ (using an appropriate approach).

Initialize biases, $B^{[l]}$, for layer $l = 1,2,\cdots,L$ (using an appropriate approach).

7.2.1. Scalar Implementation in Component-Form

Define a subroutine for scalar implementation of the activation function, $\phi(\blacksquare)$

Example 1: The Sigmoid activation function for implementation in one line is as follows:

Return scalar $\phi(x) = 1/(1+e^{-x})$

Example 2: The ReLU activation function for implementation in one line is as follows:

Return scalar $\phi(x) = \max(0,x)$

Define a subroutine for scalar implementation of the derivative of the activation function, $\phi'(\blacksquare)$

Example 1: The Sigmoid activation function for implementation in one line is as follows:

Return scalar $\phi'(x) = \phi(x)\big(1-\phi(x)\big)$

Example 2: The ReLU activation function for implementation in one line is as follows:

Return scalar $\phi'(x) = \begin{cases} 0, & x \le 0 \\ 1, & x > 0 \end{cases}$

Define a subroutine for the cost function, receiving two vectors and returning a scalar, $E(\blacksquare,\blacksquare)$

Example 1: The Quadratic cost function for implementation is as follows:

Function $E(P,Q)$

$sum = 0.0$

For $j = 1,2,\cdots,length(P)$

$sum = sum + \big(P(j)-Q(j)\big)\times\big(P(j)-Q(j)\big)$

End for j

Return scalar $sum/2.0$

Example 2: The Binary Cross-Entropy function for implementation is as follows:

Function $E(P,Q)$

$sum = 0.0$

For $j = 1,2,\cdots,length(P)$

$sum = sum + \big[P(j)\ln\big(Q(j)\big) + \big(1-P(j)\big)\ln\big(1-Q(j)\big)\big]$

End for j

Return scalar sum

Define a subroutine for the derivate of the cost function, receiving two scalars and returning a scalar, $E'(\blacksquare, \blacksquare)$

Example 1: The Quadratic cost function for implementation in one line is as follows:

Return scalar $E'(p,q) = q - p$

Example 2: The Binary Cross-Entropy function for implementation in one line is as follows:

Return scalar $E'(p,q) = (q-p)/(1-q)q$

The main processing code is as follows:

For $k = 1,2,\cdots, MAX_{epochs}$

Set $shuffling$, as a random permutation/shuffling of integers 1 to N (for the Stochastic GD approach)
Set $\bar{\boldsymbol{M}}_{\boldsymbol{s}}$, as a rearrangement of the columns of $\bar{\boldsymbol{M}}$ according to $shuffling$
Set $\bar{\boldsymbol{Y}}_{\boldsymbol{s}}$, as a rearrangement of the target values in $\bar{\boldsymbol{Y}}$ according to $shuffling$

For $g = 1,2,\cdots,\gamma$

$\boldsymbol{M} = \bar{\boldsymbol{M}}_{\boldsymbol{s}}[\ :\ ,\ \ (g-1)\times m + 1 : g \times m]$ (sampling *without replacement*)
$\boldsymbol{Y} = \bar{\boldsymbol{Y}}_{\boldsymbol{s}}[\ :\ ,\ \ (g-1)\times m + 1 : g \times m]$ (sampling *without replacement*)

Forward Pass:

$C = 0$
For each $p = 1,2,\cdots, m$
$Xinp = X^{[0](p)}$ (here, $X^{[0](p)}$ is pth column of $\boldsymbol{M}$)
For each $l = 1,2,\cdots, L$
For $j = 1,2,\cdots, n_l$
$sum = 0$
For $i = 1,2,\cdots, n_{l-1}$
$sum = sum + w_{j,i}^{[l]} \cdot Xinp[i]$
End for i
$z_j^{[l](p)} = sum + b_j^{[l]}$
$\hat{y}_j^{[l](p)} = \phi\left(z_j^{[l](p)}\right)$
End for j
Xinp $= \hat{Y}^{[l](p)}$
End for l
$c^{(p)} = E\left(Y^{(p)}, \hat{Y}^{[L](p)}\right)$ (here, $Y^{(p)}$is pth column of $\boldsymbol{Y}$)
$C = C + c^{(p)}$
End for p
$C = \dfrac{C}{m}$

Computation of Gradients through Backpropagation:

For each $p = 1,2,\cdots, m$
For $j = 1,2,\cdots, n_L$
$\dfrac{\partial c^{(p)}}{\partial \hat{y}_j^{[L](p)}} = E'\left(y_j^{(p)}, \hat{y}_j^{[L](p)}\right)$ forming $\boldsymbol{d\hat{Y}^{[L]}}$ matrix
$\dfrac{\partial c^{(p)}}{\partial z_j^{[L](p)}} = \dfrac{\partial c^{(p)}}{\partial \hat{y}_j^{[L](p)}} \times \phi'\left(z_j^{[L](p)}\right)$ forming $\boldsymbol{dZ^{[L]}}$ matrix

End for p
End for j

For each $p = 1,2,\cdots,m$
For each $l = L-1, L-2, \cdots, 1$
For $j = 1,2,\cdots,n_l$
$sum = 0$
For $t = 1,2,\cdots,n_{l+1}$

$$sum = sum + \frac{\partial c^{(p)}}{\partial z_t^{[l+1](p)}} \cdot w_{t,j}^{[l+1]}$$

End for t

$$\frac{\partial c^{(p)}}{\partial z_j^{[l](p)}} = sum \times \phi'\left(z_j^{[l](p)}\right)$$ forming $\boldsymbol{dZ}^{[l]}$ matrix

End for j
End for l
End for p

For each $l = L, L-1, L-2, \cdots, 1$
For $j = 1,2,\cdots,n_l$
For $i = 1,2,\cdots,n_{l-1}$
$sum = 0$
For $p = 1,2,\cdots,m$

$$sum = sum + \frac{\partial c^{(p)}}{\partial z_j^{[l](p)}} \times \hat{y}_i^{[l-1](p)}$$

End for p

$$\frac{\partial C}{\partial w_{j,i}^{[l]}} = \frac{sum}{m}$$ forming $\boldsymbol{d\overline{W}}^{[l]}$ matrix

End for i
End for j
End for l

For each $l = L, L-1, L-2, \cdots, 1$
For $j = 1,2,\cdots,n_l$
$sum = 0$
For $p = 1,2,\cdots,m$

$$sum = sum + \frac{\partial c^{(p)}}{\partial z_j^{[l](p)}}$$

End for p

$$\frac{\partial C}{\partial b_j^{[l]}} = \frac{sum}{m}$$ forming $\boldsymbol{dB}^{[l]}$ vector

End for j
End for l

Updating the Parameters (Weights and Biases)

For each $l = L, L-1, L-2, \cdots, 1$
For $j = 1,2,\cdots,n_l$
For $i = 1,2,\cdots,n_{l-1}$

$$w_{j,i}^{[l]} \quad = \quad w_{j,i}^{[l]} - \alpha\left(\frac{\partial C}{\partial w_{j,i}^{[l]}}\right)$$

End for i

End for j

End for l

For each $l = L, L-1, L-2, \cdots, 1$

For $j = 1,2, \cdots, n_l$

$$b_j^{[l]} \quad = \quad b_j^{[l]} - \alpha\left(\frac{\partial C}{\partial b_j^{[l]}}\right)$$

End for j

End for l

End for g

If $(C \leq tol)$, then terminate the loop over k.

End for k

Output $\overline{\boldsymbol{W}}^{[l]}$ and $B^{[l]}$ as the learned/optimized model parameters.

7.2.2. Vectorized Implementation in Matrix-Form

Define a subroutine for scalar implementation of the activation function, $\phi(\blacksquare)$

Example 1: The Sigmoid activation function for implementation in one line is as follows:

Return matrix $\phi(\boldsymbol{Z}) = 1/\left(1 + e^{-\boldsymbol{Z}}\right)$

Example 2: The ReLU activation function for implementation in one line as follows:

Return matrix $\phi(\boldsymbol{Z}) = \max(0, \boldsymbol{Z})$

Define a subroutine for scalar implementation of the derivative of the activation function, $\phi'(\blacksquare)$

Example 1: The Sigmoid activation function for implementation in one line is as follows:

Return matrix $\phi'(\boldsymbol{Z}) = \widehat{\boldsymbol{Y}}^{[L]} \otimes \left(1 - \widehat{\boldsymbol{Y}}^{[L]}\right)$

Example 2: The ReLU activation function for implementation in one line is as follows:

Return matrix $\phi'(\boldsymbol{Z}) = \begin{cases} 0, & \boldsymbol{Z} \leq 0 \\ 1, & \boldsymbol{Z} > 0 \end{cases}$

Define a subroutine for the cost function, receiving two vectors and returning a scalar, $E(\blacksquare, \blacksquare)$

Example 1: The Quadratic cost function for implementation is as follows:

Return scalar $E(\boldsymbol{P}, \boldsymbol{Q}) = \frac{1}{2}\|\boldsymbol{P} - \boldsymbol{Q}\|_2^2$

Example 2: The Binary Cross-Entropy function for implementation is as follows:

Return scalar $E(\boldsymbol{P}, \boldsymbol{Q}) = -[\boldsymbol{P}\ln(\boldsymbol{Q}) + (I - \boldsymbol{P})\ln(I - \boldsymbol{Q})]$

Define a subroutine for the derivate of the cost function, receiving two scalars and returning a scalar, $E'(\blacksquare, \blacksquare)$

Example 1: The derivative of the Quadratic cost function for implementation in one line as follows:

Return matrix $E'(\boldsymbol{P}, \boldsymbol{Q}) = \boldsymbol{Q} - \boldsymbol{P}$

Example 2: The derivative of the Binary Cross-Entropy function for implementation in one line as follows:

Return matrix $E'(\boldsymbol{P}, \boldsymbol{Q}) = (\boldsymbol{Q} - \boldsymbol{P})/(1 - \boldsymbol{Q}) \otimes \boldsymbol{Q}$

The main processing code is as follows:

For $k = 1,2, \cdots, MAX_{epochs}$

Set $shuffling$, as a random permutation/shuffling of integers 1 to N (for the Stochastic GD approach)
Set $\bar{\boldsymbol{M}}_{\boldsymbol{s}}$, as a rearrangement of the columns of $\bar{\boldsymbol{M}}$ according to $shuffling$
Set $\bar{\boldsymbol{Y}}_{\boldsymbol{s}}$, as a rearrangement of the target values in $\bar{\boldsymbol{Y}}$ according to $shuffling$
For $g = 1,2, \cdots, \gamma$

$\boldsymbol{M} = \bar{\boldsymbol{M}}_{\boldsymbol{s}}[\ :\ ,\ \ (g-1) \times m + 1 : g \times m]$ (sampling *without replacement*)
$\boldsymbol{Y} = \bar{\boldsymbol{Y}}_{\boldsymbol{s}}[\ :\ ,\ \ (g-1) \times m + 1 : g \times m]$ (sampling *without replacement*)

Forward Pass:

$C = 0$
$\hat{\boldsymbol{Y}}^{[0]} = M$
For each $l = 1,2, \cdots, L$
$\boldsymbol{Z}^{[l]} = \bar{\boldsymbol{W}}^{[l]}\, \hat{\boldsymbol{Y}}^{[l-1]} + B^{[l]}$
$\hat{\boldsymbol{Y}}^{[l]} = \phi\left(\boldsymbol{Z}^{[l]}\right)$
End for l
For each $p = 1,2, \cdots, m$
$c^{(p)} = E\left(Y^{(p)}, \hat{Y}^{[L](p)}\right)$ (here, $Y^{(p)}$is pth column of $\boldsymbol{Y}$)
$C = C + c^{(p)}$
End for p
$C = \dfrac{C}{m}$

Computation of Gradients through Backpropagation:

$\boldsymbol{d}\hat{\boldsymbol{Y}}^{[L]} = E'\left(\bar{\boldsymbol{Y}}, \hat{\boldsymbol{Y}}^{[L]}\right)$
$\boldsymbol{dZ}^{[L]} = \boldsymbol{d}\hat{\boldsymbol{Y}}^{[L]} \otimes \phi'\left(\boldsymbol{Z}^{[L]}\right)$ Hadamard (or componentwise) product

For each $l = L-1, L-2, \cdots, 1$
$\boldsymbol{dZ}^{[l]} = \left(\bar{\boldsymbol{W}}^{[l+1]^T} \boldsymbol{dZ}^{[l+1]}\right) \otimes \phi'\left(\boldsymbol{Z}^{[l]}\right)$

For each $l = L, L-1, L-2, \cdots, 1$
$\boldsymbol{d}\bar{\boldsymbol{W}}^{[l]} = \dfrac{1}{m} \times \boldsymbol{dZ}^{[l]} \hat{\boldsymbol{Y}}^{[l-1]^T}$
End for l

For each $l = L, L-1, L-2, \cdots, 1$
$dB^{[l]} = \dfrac{1}{m} \times rowsum\ (\boldsymbol{dZ}^{[l]})$
End for l

Updating the Parameters (Weights and Biases)

For each $l = L, L-1, L-2, \cdots, 1$

$$\overline{\boldsymbol{W}}^{[l]} = \overline{\boldsymbol{W}}^{[l]} - \alpha \boldsymbol{d}\overline{\boldsymbol{W}}^{[l]}$$

End for l

For each $l = L, L-1, L-2, \cdots, 1$

$$B^{[l]} = B^{[l]} - \alpha dB^{[l]}$$

End for l

End for g

If $(C \leq tol)$, then terminate the loop over k.

End for k

Output $\overline{\boldsymbol{W}}^{[l]}$ and $B^{[l]}$ as the learned/optimized model parameters.

This completes the training algorithm of ANN.

7.3. Test Case: Predicting Stock Price Using Time Series Data in Python

7.3.1. Vectorized Implementation of the ANN Algorithm in Python

We consider the share price history of 22 years (date-wise) of Pak Suzuki Motor Company Limited (Pakistan) available from (Kaggle - Pakistan Stock Exchange Data, 2022). We address the problem of predicting the stock price of the company as a regression problem through supervised learning using historical data with the ANN. We solve the problem using the ANN algorithm discussed in Section 7. First, we present a Python code for the vectorized implementation of the algorithm, as discussed in Section 7.2.2. Next, we demonstrate that similar performance results are obtained using the TensorFlow (Abadi et al., 2016) library functions for similar algorithm of ANN model building and prediction (Emmert-Streib et al., 2020). The algorithm presented for the ANN model is the basic one. Although rich and powerful in methodologies and performance capabilities, deep neural network models can be helpful only when used judiciously with appropriate hyperparameters. The models' efficiency and performance can be improved by paying attention to practical considerations, including hyperparameter tuning, regularization, batch normalizations (discussed in Section 8), and optimization techniques (discussed in Section 9). The vectorized implementation of the algorithm is given as follows.

```
# Loading the dataset suzuki_data.csv with dimensions: 5252x14
# Stock price data of Pak Suzuki Motor Company Limited (Pakistan)
# Stock prices daily data from 01 Jan 2001 to 17 Nov 2022

import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler

df = pd.read_csv('suzuki_data.csv')

## Setting Hyperparameters
MAX_epochs = 100        # Maximum number of epochs
tol = 0.000001          # Tolerance
alpha = 0.01            # Learning rate
m = 100                 # Number of instances in a minibatch
window_size = 10
train_test_ratio = 0.95

```

```
# Setting Layers and Number of neurons in each layer
ldim = [window_size, 20, 30, 10, 5, 1]
L = len(ldim) - 1    # Number of layers (other than the input layer)

ldim = np.array(ldim, dtype=int)
np.random.seed(1)

print("Original Dataset Head:")
print(df.head())

print("Original Dataset Tail:")
print(df.tail(5))

# Loading and making the data ready in desired format

df = df.dropna()                # data cleaning as desired

# Selecting the relevant column for training and prediction
df_main = df['Close']
df_main = np.array(df_main).reshape(-1,1)

# normalizing the data between 0 to 1
normalizing = MinMaxScaler(feature_range=(0,1))
normalized_data = normalizing.fit_transform(df_main)

## Spliting the time series data into train and test sets
## Temporal order of the data is preserved

train_size = int(len(normalized_data) * train_test_ratio)
train_data = normalized_data[:train_size]
test_data  = normalized_data[train_size:]

# Function for creating datasets using Sliding Window Technique
# This is a common technique for time series prediction using FNN
# Slider window size is the time step

def dataset_creator(window_size, data):
    X, Y = [], []
    for i in range(window_size, len(data)):
        X.append(data[i-window_size:i, 0])
        Y.append(data[i, 0])
    return np.array(X), np.array(Y)

# Creating the training and test sets
X_train, Y_train = dataset_creator(window_size, train_data)
X_test, Y_test = dataset_creator(window_size, test_data)

```

```
Original Dataset Head:
         Date   Open   High    Low  Close   Volume
0  2001-01-01  10.25  10.25  10.25  10.25   1000.0
1  2001-01-02  10.25  11.50  10.70  11.30  10500.0
2  2001-01-03  11.30  11.30  10.75  10.75   6500.0
3  2001-01-04  10.75  11.30  11.25  11.25   3000.0
4  2001-01-05  11.25  11.30  11.05  11.05   6500.0
```

```
Original Dataset Tail:
```

	Date	Open	High	Low	Close	Volume
5247	2022-11-11	161.50	162.98	159.00	160.84	18222.0
5248	2022-11-14	159.00	159.85	156.94	157.25	52296.0
5249	2022-11-15	158.50	164.50	157.01	162.27	151394.0
5250	2022-11-16	163.00	164.00	160.00	160.29	49327.0
5251	2022-11-17	161.69	161.70	158.60	159.21	43334.0

```
##============= Formatting the training set =============##

N = X_train.shape[0]   # Total number of training instances
gamma = int( np.ceil(N/m) )

## converting Pandas objects to NumPy arrays and desired dimensions
MM = X_train.T
YY = Y_train.reshape(1, -1)

print("Training Set Features ready with dimensions: " , MM.shape)
print("Training Set Target ready with dimensions: " , YY.shape)

##------------- Formatting the training set -------------##


##============= Formatting the test set =============##

Nt = X_test.shape[0]   # Total number of training instances

## converting pandas objects to numpy arrays and desired dimensions
Mt = X_test.T
Yt = Y_test.T

print("Test Set Features ready with dimensions: " , Mt.shape )
print("Test Set Target ready with dimensions: " , Yt.shape )

##------------- Formatting the test set -------------##

```

```
Training Set Features ready with dimensions:  (10, 4979)
Training Set Target ready with dimensions:  (1, 4979)
Test Set Features ready with dimensions:  (10, 253)
Test Set Target ready with dimensions:  (253,)
```

```
## Defining activitation functions and their derivatives

def sigmoid(Z):
    return 1 / (1 + np.exp(-Z))

def sigmoid_derivative(Z):
    return sigmoid(Z) * (1 - sigmoid(Z))

def relu(Z):
    return np.maximum(0, Z)

def relu_derivative(Z):
    return np.where(Z <= 0, 0, 1)

```

```
## Defining loss functions and their derivatives

def quadratic_cost(P, Q):
    return 0.5 * np.linalg.norm(P - Q) ** 2

def quadratic_cost_derivative(P, Q):
    return Q - P

def binary_cross_entropy(P, Q):
    err = -(np.dot(P,np.log(Q)) + np.dot((1-P),np.log(1-Q)))
    #err = -(P*np.log(Q) + (1-P)*np.log(1-Q))
    #err = np.squeeze(err)
    return  err

def binary_cross_entropy_derivative(P, Q):
    return (Q-P)/((1-Q) * Q)
```

```
## Defining the function that trains/optimizes model parameters

def trainer(L, N, m, gamma, alpha, Nt, MAX_epochs, ldim, MM, YY):

  ##  Initializing the parameters (weights and biases)
    W = np.zeros( (L+1, np.max(ldim) , np.max(ldim)) )
    B = np.zeros( (L+1, np.max(ldim) , 1))
    dW = np.zeros( (L+1, np.max(ldim) , np.max(ldim)) )
    dB = np.zeros( (L+1, np.max(ldim) , 1) )

    for l in range(1, L + 1):
        W[ l , 0:ldim[l] , 0:ldim[l-1] ] = np.random.randn(
                                        ldim[l], ldim[l-1] ) * 0.01
        B[ l , 0:ldim[l] , : ] = np.zeros( (ldim[l], 1) )

    ##  Creating auxiliary data structures

    Y_hat = np.zeros( (L+1, np.max(ldim), m) )
    Z     = np.zeros( (L+1, np.max(ldim), m) )
    dZ    = np.zeros( (L+1, np.max(ldim), m) )

    ## Loop over epochs

    for k in range(1, MAX_epochs+1):

        # print('\n======================== Epoch number: ', k)
        # Making a random rearrangement of training instances
        permut = np.random.permutation(N)
        # permut = np.array([i for i in range(0,N)])
        MM = MM[:, permut]
        YY = YY[:, permut]

        ## Loop over iterations / steps

        start_idx = -m
        end_idx = 0
```

```
        for g in range(1, gamma+1):

            start_idx = start_idx + m
            end_idx = end_idx + m

            if end_idx>N:
                end_idx = N

            mc = end_idx-start_idx

            # M (n0xm) is sampled from MM (n0xN)
            M = MM[:, start_idx:end_idx]
            # Y (1xm) is sampled from YY (1xN)
            Y = YY[:, start_idx:end_idx]

            ## Carrying out computtaions of the Forward Pass
            C = 0
            Y_hat[0, 0:ldim[0], 0:mc] = M

            for l in range(1, L + 1):
                wx = W[l, 0:ldim[l], 0:ldim[l-1] ]
                yold = Y_hat[l-1, 0:ldim[l-1], 0:mc]
                bx = B[l, 0:ldim[l], :]

                zx = np.dot(wx, yold) + bx
                Z[l, 0:ldim[l], 0:mc] = zx

                #if l==L:
                yx = sigmoid(zx)
                #else:
                #    yx = relu(zx)

                Y_hat[l, 0:ldim[l], 0:mc] = yx

            ## computing cost function
            for p in range(0,mc):
                cp = quadratic_cost(Y[0:ldim[L],p], Y_hat[L,0:ldim[L], p])
                #cp = binary_cross_entropy( Y[0:ldim[L], p],
                                                #Y_hat[L, 0:ldim[L], p])
                C += cp

            C /= mc


            ## Computing gradients through backpropagation

            dY_hat_L = quadratic_cost_derivative(Y[0:ldim[L], 0:mc],
                                                 Y_hat[L, 0:ldim[L], 0:mc])
            #dY_hat_L = binary_cross_entropy_derivative(Y[0:ldim[L], 0:mc],
                                                #Y_hat[L, 0:ldim[L], 0:mc])

            dZ[L, 0:ldim[L], 0:mc] = dY_hat_L * sigmoid_derivative(
                                                    Z[L, 0:ldim[L], 0:mc])

```

```
            for l in range(L-1, 0, -1):
                wt = W[l+1, 0:ldim[l+1], 0:ldim[l]].T
                dzt = dZ[l+1, 0:ldim[l+1],  0:mc]
                zt = Z[l, 0:ldim[l], 0:mc]
                #dZ[l,0:ldim[l],0:mc]= np.dot(wt,dzt)*relu_derivative(zt)
                dZ[l,0:ldim[l],0:mc]= np.dot(wt,dzt)*sigmoid_derivative(zt)

            for l in range(L, 0, -1):
                dZl = dZ[l, 0:ldim[l], 0:mc]
                Y_prev = Y_hat[l-1, 0:ldim[l-1], 0:mc]
                dW[l, 0:ldim[l] , 0:ldim[l-1]] = (np.dot(dZl, Y_prev.T))/mc
                dB[l, 0:ldim[l] , :] =  (np.sum(dZl,
                                axis=1, keepdims=True))/mc

            ## Updating parameters using the Gradient Descent method
            for l in range(L, 0, -1):
                W[ l, 0:ldim[l] , 0:ldim[l-1] ] = W[ l, 0:ldim[l] ,
                    0:ldim[l-1] ] - alpha*dW[ l, 0:ldim[l] , 0:ldim[l-1] ]
                B[ l, 0:ldim[l] , : ] = B[ l, 0:ldim[l] , : ] \
                                            - alpha*dB[ l, 0:ldim[l] , : ]

        ## Loop terminated, over iterations / steps

    ## Loop terminated, over epochs

    print(f'After {k} epochs, the training Error (Cost function)= {C}')

    return W, B
    # function trainer() ended
```

```
## Defining the function that make prediction

def predicting(L, ldim, W, B, Mt, Nt):

    C = 0
    Y_hat1 = np.zeros( (L+1, np.max(ldim), Nt) )
    Y_hat1[0, 0:ldim[0], 0:Nt] = Mt

    for l in range(1, L + 1):
            wx = W[l, 0:ldim[l], 0:ldim[l-1] ]
            yold = Y_hat1[l-1, 0:ldim[l-1], 0:Nt]
            bx = B[l, 0:ldim[l], :]

            zx = np.dot(wx, yold) + bx
            #Z1[l, 0:ldim[l], 0:Nt] = zx
            #if l==L:
            yx = sigmoid(zx)
            #else:
            #    yx = relu(zx)

            Y_hat1[l, 0:ldim[l], 0:Nt] = yx

    return yx
    # function predicting() ended
```

```
## Defining the function for calculating MAE and RMSE

def  reg_errors1(Y_test, Y_predict_normalized):

    Y_predict = normalizing.inverse_transform(Y_predict_normalized)
    Y_test1 = normalizing.inverse_transform(Y_test.reshape(-1,1))

    from sklearn.metrics import mean_absolute_error
    from sklearn.metrics import mean_squared_error

    mae = mean_absolute_error(Y_test1,Y_predict.T)
    mse = mean_squared_error(Y_test1,Y_predict.T)
    rmse = np.sqrt(mse)

    print("MAE in the prediction for the test data:", mae)
    print("RMSE in the prediction for the test data: ", rmse)

    # function reg_errors1() ended
```

```
print('\n========== Training epochs started ==========\n')
W, B = trainer(L, N, m, gamma, alpha, Nt, MAX_epochs, ldim, MM, YY)
print('\n========== Training epochs completed ==========')
print("\n")


## using the optimized parameters for making predictions
## The Forward Pass with the test data
print("\n~~~~~ Forward Pass with the test data started ~~~~~")
Y_predict_normalized = predicting(L, ldim, W, B, Mt, Nt)
print("\n================== Prediction Completed ==================")


print("\n======================== Test Errors ========================")
reg_errors1(Y_test, Y_predict_normalized)
```

```
========== Training epochs started ==========

After 100 epochs, the training Error (Cost function) = 0.01797601542125484

========== Training epochs completed ==========


~~~~~ Forward Pass with the test data started ~~~~~

================== Prediction Completed ==================

======================== Test Errors ========================
MAE in the prediction for the test data: 30.410900839424134
RMSE in the prediction for the test data:  37.43117805243188
```

7.3.2. Vectorized Implementation of the ANN Algorithm in Python using TensorFlow

A Python based implementation of the basic ANN algorithm using the TensorFlow functions is as follows. The starting part for data loading, transformation, and preparation of the training and test sets is the same as those presented for the vectorized implementation. The subsequent part is given below.

```python
import tensorflow as tf
from sklearn.model_selection import train_test_split
from tensorflow.keras.optimizers import SGD
from tensorflow import keras
from tensorflow.keras import initializers
from tensorflow.keras import layers

model = keras.models.Sequential()
model.add(tf.keras.Input(shape=(ldim[0],)))
model.add(layers.Dense(units=ldim[1],
                       kernel_initializer=initializers.RandomNormal(),
                       bias_initializer=initializers.Zeros(),
                       activation='sigmoid'))
model.add(layers.Dense(units=ldim[2],
                       kernel_initializer=initializers.RandomNormal(),
                       bias_initializer=initializers.Zeros(),
                       activation='sigmoid'))
model.add(layers.Dense(units=ldim[3],
                       kernel_initializer=initializers.RandomNormal(),
                       bias_initializer=initializers.Zeros(),
                       activation='sigmoid'))
model.add(layers.Dense(units=ldim[4],
                       kernel_initializer=initializers.RandomNormal(),
                       bias_initializer=initializers.Zeros(),
                       activation='sigmoid'))
model.add(layers.Dense(units=ldim[5],
                       kernel_initializer=initializers.RandomNormal(),
                       bias_initializer=initializers.Zeros(),
                       activation='sigmoid'))

model.compile(loss='mean_squared_error',
                            optimizer= SGD(learning_rate=alpha))

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 20)                220

 dense_1 (Dense)             (None, 30)                630

 dense_2 (Dense)             (None, 10)                310

 dense_3 (Dense)             (None, 5)                 55

 dense_4 (Dense)             (None, 1)                 6

=================================================================
Total params: 1,221
Trainable params: 1,221
Non-trainable params: 0
_________________________________________________________________
```

```python
## Defining the function for calculating MAE and RMSE

def  reg_errors2(Y_test, Y_predict_normalized):

    Y_predict = normalizing.inverse_transform(Y_predict_normalized)
    Y_test1 = normalizing.inverse_transform(Y_test.reshape(-1,1))

    from sklearn.metrics import mean_absolute_error
    from sklearn.metrics import mean_squared_error

    mae = mean_absolute_error(Y_test1,Y_predict)
    mse = mean_squared_error(Y_test1,Y_predict)
    rmse = np.sqrt(mse)

    print("MAE in the prediction for the test data:", mae)
    print("RMSE in the prediction for the test data: ", rmse)

    # function reg_errors2() ended
```

```python
# Training the model

print('\n========== Training process started ==========\n')
model.fit(X_train, Y_train, epochs=MAX_epochs, batch_size=m)
print('\n========== Training process completed ==========')
print("\n")


## using the optimized parameters for making predictions
## The Forward Pass with the test data
print('\n========== Prediction process started ==========')
Y_predict_normalized = model.predict(X_test)
print('\n========== Prediction process completed ==========')


print("\n======================== Test Errors ========================")
reg_errors2(Y_test, Y_predict_normalized)
```

```
========== Training process started ==========
Epoch 1/100
50/50 [==============================] - 1s 2ms/step - loss: 0.0899
Epoch 2/100
50/50 [==============================] - 0s 2ms/step - loss: 0.0768
...
Epoch 100/100
50/50 [==============================] - 0s 2ms/step - loss: 0.0369

========== Training process completed ==========

========== Prediction process started ==========
8/8 [==============================] - 0s 2ms/step

========== Prediction process completed ==========

======================== Test Errors ========================
MAE in the prediction for the test data: 28.918479638382383
RMSE in the prediction for the test data:  35.91955904463751
```

Remark: The model's performance will be improved if the Adam optimizer is used in the given code. Details about the Adam optimizer are provided in section 9.3.4. The performance may be enhanced further by adjusting the hyperparameters. The subsequent section delves into hyperparameters and hyperparameter tuning in detail.

8. Practical Considerations for Improved Neural Network Training

8.1. Issues in Training the Neural Network

Generalization refers to the ability of a model to accurately predict outcomes for new, unseen data that is not included in the training set. A model that can generalize well has learned the underlying patterns and relationships in the training data without memorizing it and can apply it to new data. Despite various issues, deep learning with neural networks has offered so much potential of generalization. Kawaguchi et al. (2022) presented a theoretical treatment of generalization, even proposed new open problems. While training neural networks, there can be several issues that can result in poor generalization. Marcus, (2018) critically discussed some of the aspect related to issues with the neural networks.

Some common issues related to the neural networks are discussed below:

1. **Overfitting**: If the training error (the error your model makes on the training set) is low, but the generalization error (the error your model makes on the test set) is high, then it means that the model is exhibiting high variance and overfitting the training data. The strategies include those for appropriate data collection, data pre-processing, network designing, parameter initialization, hyperparameter tuning, cross-validation, model complexity, regularization, and ensemble methods (those combine the predictions of multiple models for improved performance) are planned for avoiding overfitting.

2. **Underfitting**: Underfitting is when a machine learning model is too simple and cannot capture the underlying patterns in the data. This can result in high bias and poor training and test data performance. Increasing the complexity of the model, and adding more layers or neurons, can help address underfitting. The strategies, such as those for appropriate data pre-processing, network designing, model complexity, regularization, and ensemble methods, are planned to avoid underfitting.

3. **Vanishing and Exploding Gradients**: Vanishing and exploding gradients are two common problems that can occur during the training of deep neural networks, in particular, the backpropagation phase. *Vanishing of the gradients* occurs when they become minimal, often approaching zero, as they propagate back through the layers of the network. This can cause the weights to update very slowly or not at all, which can significantly slow down or even prevent the convergence of the network during training. In extreme cases, vanishing gradients can also cause the network to become stuck in a local minimum, leading to poor overall performance. *Exploding of the gradients*, on the other hand, occurs when the gradients become very large, often reaching extremely high values, as they propagate back through the network layers. Again, this can cause the weights to update rapidly and lead to instability in the training process, which may result in divergence.

Vanishing and exploding gradients are more likely to occur in deep neural networks with certain types of activation functions, such as the sigmoid or hyperbolic tangent function, because the derivatives of these functions can become very small or very large, resulting in extreme gradients. Other activation functions, such as ReLU and its variants, are often considered to reduce these problems. Moreover, using various techniques of parameter initialization and batch normalization can also help stabilize the gradients during training.

4. **High Computational Cost (Limited Computational Resources)**: Training large neural networks can be computationally expensive and require high-end hardware, such as GPUs or TPUs. Training can sometimes take days or weeks to complete, making it challenging to accomplish within a reasonable timeframe. Efficient algorithms and state-of-the-art computational parallel computing resources are acquired to address the challenge (Ali, 2013).

5. **Slow Convergence for Optimization**: Due to the versatility and complexity of the cost functions to be optimized for training the neural network model, the iterative convergence of the optimization algorithm (usually the Gradient Descent method) becomes slow. A handful of the mathematical theory of optimization methods has grown to increase the convergence rate. Some advanced strategies for the Gradient Descent method include the Stochastic Gradient method, the Stochastic Gradient method with momentum or Nestrove momentum, the Gradient Descent with Adaptive Learning Rates (e.g., AdaGrad, RMSProp, AdaDelta, Adam), and many more.

The subsequent sections discuss crucial practical considerations at various phases to avoid neural network training issues for a successful deep learning activity.

8.2. Data Collection (or Data Generation)

The very first activity for any deep learning (or any machine learning) project is acquiring/generating the dataset to be used for learning. The size of the dataset should be large enough so that meaningful generalization (a well-trained model) can be made and the model will not overfit the training data. Having a large enough dataset is one of the primary ways to avoid overfitting. Moreover, the data to be used for training should be representative of the overall population so that the trained model will judiciously generalize future data. Otherwise, the trained model will not be able to perform reasonably (e.g., in making predictions). Even large datasets cannot perform if the data has *sampling bias*: non-representativeness due to biasing in data selection. On the other hand, smaller data sizes have a high probability of missing a variety of representative examples from the population. This is called *sampling noise*.

8.3. Data Pre-processing and Feature Engineering

Firstly, the dataset should be *clean*, not *dirty*. It should be made free from errors or *noise*, such as missing values, *outliers*, etc. Having a clean dataset is also one of the primary ways to avoid overfitting. The rows/columns with errors can be removed from the dataset, or the erroneous value may be imputed/replaced by some reasonable approximation (e.g., the median of the column entries). The data *pre-processing* may also involve data wrangling, data integration, data transformation (such as normalization), data reduction (such as dimensionality reduction), and data discretization. With a poor-

quality dataset, a well-trained model cannot be anticipated. In practice, a significant proportion of the total time of a data science activity is spent on pre-processing the data.

The data pre-processing activities related to features/variables are also termed ***feature engineering***. This typically involves activities such as appropriately representing data (converting from categorical/text/image values to numeric form through one-hot encoding, label encoding, or ordinal encoding as appropriate), feature selection, feature extraction/derivation, and feature scaling. These activities are performed before feeding the data into the learning algorithm as a way to preprocess and transform the raw data into meaningful inputs for the model. Having better features (as a result of some feature engineering) may help avoid high bias, which avoids underfitting.

Some of the critical feature engineering activities are discussed below:

1. ***Feature scaling*** or ***normalization*** (***standardization***): It refers to scaling numerical features to a common scale, a specific range, or distribution to ensure that they have similar statistical properties for learning efficiency. The techniques include the Min–Max normalization, Z-score normalization, and Normalization by decimal scaling.

2. ***Feature selection***: Selecting a subset of the essential features from the available set of features to reduce dimensionality for relevance and efficiency. Having fewer features/attributes produces a simpler model, which may help avoid overfitting. Conventional machine learning techniques requires feature selection manually, whereas neural networks perform this activity automatically as part of the model's working.

3. ***Feature extraction***: Creating new features by extracting relevant information from existing features, such as aggregating data, creating interaction terms, or applying mathematical transformations. This is highly dependent on the problem or domain-specific knowledge or insights. Conventional machine learning techniques requires feature extraction manually, whereas neural networks perform this activity automatically as part of the model's working.

8.4. Designing the Neural Network Architecture

In the **single-layer** neural network, the vector of inputs generates the output. In a **multi-layer** or **deep** neural network, the input layer is followed by a sequence of hidden layers to perform computations one after the other and ultimately generate the output layer. The neurons in the input layer are features (attributes/columns) of the dataset under consideration. The neurons within a layer are independent of each other. In a fully connected neural network, each neuron in layer l is generated using all the neurons in the previous layer $l-1$. The design of a **fully connected** neural network is based on the number of layers (L) and the number of neurons/units in a layer (n_l), including the input layer. This architecture is called **feed-forward** because the layer feeds its computed values into the next layer, in the forward direction, from the input layer to the output layer. Thus, a neural network architecture acts as a composite function Φ of the set of input features $X^{[0]}$, in which each of the L layers acts as a function that is applied to the output of the previous layer, as expressed in Eq. $(F.7)$. This way, the neural network serves as a computational framework.

The number of layers L and neurons per layer n_l for a feed-forward deep neural network can significantly impact the model's performance and accuracy. These two quantities determine the model's ***capacity***, that is, the ability of the model to lean and represent (or store and process) the variety and

complexity of the patterns in the data. In case of underfitting (getting not enough signal), a network can be made ***deeper*** (by increasing L) and wider (by increasing n_l). However, there is no one-size-fits-all approach for determining the best architecture for a given problem. It often requires a combination of intuition, experimentation, and domain expertise. Therefore, it is recommended to start with a simpler architecture and gradually increase the complexity of the model, observing the complexity of the problem. For example, problems that require more complex decision boundaries may benefit from deeper architectures with more neurons per layer. Sometimes, the average of the number of neurons in the input layer and the number of neurons in the output layer can serve as a starting point for setting the number of neurons per layer. Ultimately, the best way to determine the optimal architecture for a given problem is through experimentation while considering other factors (hyperparameters).

Wider networks tend to capture *linear* relationships among the features more easily, whereas *deeper* networks tend to capture nonlinear relationships more easily. Another pertinent point is that the initial layers capture **low-level features**, such as edges and textures. In contrast, the deeper layers capture **high-level features**, such as shapes and objects, in a neural network. The initial layers are designed to capture the low-level features. As the information moves through the network, these low-level features are combined and transformed into more complex features. As a result, the deeper layers have access to a larger receptive field, meaning they can consider a broader range of input data. Additionally, the deeper layers often have more neurons, which allows them to capture more nuanced and intricate relationships between the features. Thus, the hierarchical approach to feature extraction allows neural networks to learn increasingly abstract representations of the input data as it passes through the network, making them powerful tools for tasks such as image and speech recognition.

Besides the number of layers and the number of neurons in each layer, another critical task for the computational framework of a neural network is to make choices for the activation function/s and the loss function, mainly depending on the problem or domain under consideration. Section 2 has mentioned various possible choices for these in Tables 2.1 and 2.2.

8.5. Initialization Techniques

Initialization of the parameters (weights and biases) is crucial in training neural networks. Proper initialization can help address the problem of vanishing and exploding gradients (that can occur during the backpropagation phase), accelerate the convergence of the optimization algorithm, and improve the overall performance of the neural network. The choice of initialization method depends on the specific neural network architecture and activation function used. A good practice is experimenting with different initialization methods to see which works best for the specific problem at hand. Following are some well-known techniques for parameter initialization in neural networks for each layer, $l = 1,2,3,\cdots$. Python-like pseudo-codes using the NumPy library (with alias np) are also given.

1. ***Zero Initialization***: In this technique, all the weights and biases in the network are set to zero during initialization. A Python-like code using the NumPy library for zero initializing might be:

$$W^{[l]} = \text{np.zeros}(n_l, n_{l-1})$$
$$B^{[l]} = \text{np.zeros}(n_l, 1)$$

   With zero initialization, the derivatives with respect to the loss function for each weight in the layer will be identical. Consequently, in subsequent layers, all the weights will retain the same value, resulting in symmetric hidden layers. This symmetry will persist for all the layers,

causing the network to perform no better than a linear model. However, the biases set to zero will not pose any issues as the non-zero weights break the symmetry. Therefore, even if the biases are zero, the values in each neuron will be distinct.

2. ***Random Initialization***: In this technique, the weights and biases are randomly initialized to values from a uniform or normal/gaussian distribution with zero mean and a small standard deviation. This approach helps break the network's symmetry and can lead to better performance. This approach is commonly used in deep learning frameworks such as TensorFlow and PyTorch due to its simplicity and effectiveness. A Python-like code for randomly initializing weights to values from a uniform distribution is:

$$W^{[l]} = \text{np.random.rand}(n_l, n_{l-1})$$

A Python-like code for randomly initializing weights to values from a normal/gaussian distribution is:

$$W^{[l]} = \text{np.random.randn}(n_l, n_{l-1})$$

3. ***Xavier Initialization*** (***Glorot Initialization***): In this technique, the weights are initialized to values from a Gaussian/normal distribution with zero mean and a standard deviation of $\sqrt{2.0/(n_{l-1}+n_l)}$. This technique benefits activation functions like hyperbolic tangent, logistic, and softmax (Xavier, 2010). A Python-like code for randomly initializing weights to values from Gaussian/normal distribution is:

$$W^{[l]} = \text{np.random.randn}(n_l, n_{l-1}) \times \text{np.sqrt}\big(2.0/(n_{l-1}+n_l)\big)$$

4. ***He Initialization***: This technique is similar to Xavier initialization but is used for ReLU activation functions. The weights are initialized to values from a Gaussian/ Normal distribution with zero mean and a standard deviation of $\sqrt{2/n_{l-1}}$ (He et al., 2015). A Python-like code for He initialization technique to values from a Gaussian/normal distribution is:

$$W^{[l]} = \text{np.random.randn}(n_l, n_{l-1}) \times \text{np.sqrt}(2.0/n_{l-1})$$

5. ***LeCun Initialization***: This technique is used for fully connected layers and convolutional layers with hyperbolic tangent activation function. The weights are initialized to values from a uniform distribution with zero mean and a standard deviation of $\sqrt{1/n_{l-1}}$ (LeCun et al., 2015). A Python-like code for the LeCun initialization technique to values from a Gaussian/normal distribution is:

$$W^{[l]} = \text{np.random.randn}(n_l, n_{l-1}) \times \text{np.sqrt}(1.0/n_{l-1})$$

6. ***Pretrained Initialization*** (***Transfer learning***): This technique involves initializing the weights to the values that are learned from a pre-training step on a similar task or dataset. This approach can be helpful in cases where the size of the labeled data is small. This is commonly used in computer vision and natural language processing tasks (Yosinski et al., 2014).

8.6. Hyperparameter Tuning and Model Validation

A machine learning or deep learning model (i.e., the learning function) has some ***model parameters*** (such as slope and intercept in a linear regression model and weights and biases in a deep learning

model) that are crucial to the model's performance for making predictions. A learning algorithm tries to find optimal values for the model parameters so that the model generalizes well to new instances.

On the other hand, a ***hyperparameter*** is a parameter of the learning algorithm itself, not of the trained model. Hyperparameters are set prior to the start of the training process, and they remain unaltered during the model training. As hyperparameters dictate the performance of the learning algorithm, they are tuned to obtain better generalization. The number of layers (L) and the number of neurons/units (n_l) in a layer l are the most basic hyperparameters for a neural network architecture.

A list of hyperparameters (both mandatory and situational) for fully connected neural networks is as follows:

1. Number of layers other than the input layer (L)
2. Number of neurons/units in a layer l (n_l)
3. Choice of the activation function/s
4. Choice of the loss function
5. Size of mini-batches of the training set for iterations of the optimization algorithm (m), or equivalently the number of mini-batches of the training set (γ)
6. Number of epochs $\left(MAX_{epochs}\right)$
7. Choice of the optimization algorithm
8. Learning rate of the optimization algorithm (α)
9. Regularization parameter (λ)
10. Value of the momentum parameter β for the optimization algorithm (if needed)
11. RMSProp hyperparameter β_1 (if needed)
12. Adam parameters β_1, β_2 and ε (if needed)
13. Dropout rate for the neurons of a layer
14. Learning rate decay

There are various libraries, such as Bayesian optimization, Hyperopt, Spearmint, SMAC, GridSearchCV, RandomizedSearchCV, Ray Tune, and Optuna, for choosing different hyperparameters. These libraries offer a range of techniques for hyperparameter tuning, from exhaustive searches to more sophisticated Bayesian methods. Choosing the appropriate library depends on the specific requirements of a project, such as the size of the search space, computational resources, and the desired level of optimization.

An effective and efficient approach for choosing or **tuning hyperparameters** (as well as for best model selection) is to try out some random/heuristic guesses and then narrow down the choice for best values. However, the exhaustive (brute-force) or grid search is slow and resource-consuming.

To validate a particular selection of a set of hyperparameters (or tuning the hyperparameters, in general), there are two well-known approaches: the ***Holdout Validation*** technique and ***k-Fold Cross-Validation*** technique.

When training a model, first, we typically divide the available data into two sets: the ***training set*** (around 80-98% of the whole dataset) and the ***test set*** (around 2-20% of the whole dataset). The percentages of divisions are decided depending on the dataset sizes. Typically, 1000 instances in the test set are assumed sufficient to serve the purpose. For simplicity, we assume the division as depicted in Fig. 8.1. During the learning phase, the ***training error*** is used to dictate the progress. The training error is the error of the model on the training set, which measures how well the model fits the training data. It is calculated by comparing the model's predicted output with the training data's actual/target output. During the learning/training process, the error between the predicted and target values is computed using the **loss function** for each training instance. The training error (also called the **cost function**) is

an average (or mean) of the errors calculated for all the instances in a batch of training instances under consideration.

Once the model is ready, it is measured how well the model performs on new, unseen data (the test set) by finding the ***generalization error***. The generalization error is calculated as the difference between the predicted outputs and the target outputs of the test set. The generalization error is the most important metric for evaluating the performance of the model, as it measures how well the model will perform in real-world situations.

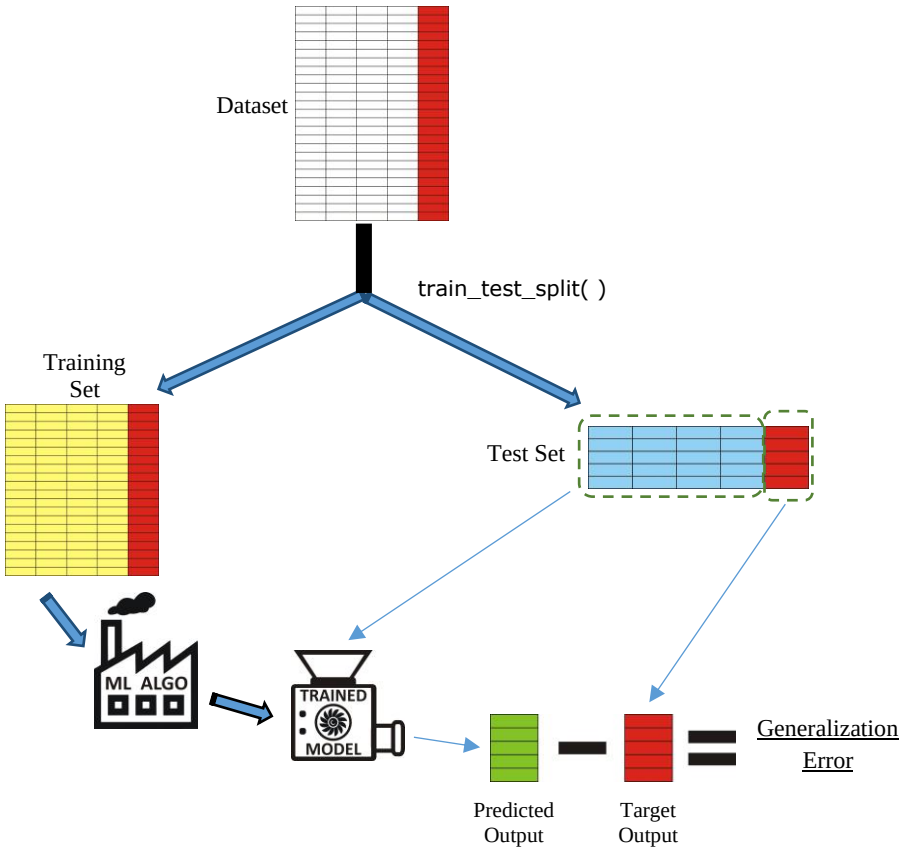


Fig. 8.1: A basic approach train-test splitting and model building and testing

8.6.1. Holdout Validation Technique

When training a model, first, we typically divide the available data into two sets: the training set and the test set, as discussed earlier. Next, for the training phase, the training set is further divided into two sets: the reduced training set and the ***validation set*** (also called the ***development set***, ***dev set***, or ***holdout set***). The validation set might have 1-50% of the training set, depending upon the training set size. For large training sets, a lower percentage of the training set is sufficient for making the validation set.

The ***validation error*** is the error of the model on the validation set, which measures how well the model generalizes to new data. The model is trained on the training set. Once the model is ready, the validation error is then calculated as the difference between the predicted outputs and the target outputs of the validation set. The validation error is used to tune the model's hyperparameters, such as the learning rate or the number of layers, to improve the model's performance on new data.

The reduced training set is used to train multiple models with various choices of hyperparameters. The model that gives the best accuracy on the validation set (i.e., lowest validation error) is selected. This way, tuning of the hypermeter is carried out. Then, the selected model is trained on the whole training set. This produces the final model. This approach serves as a technique to generalize better the training set by avoiding overfitting so that the model will perform well on the test set (unseen data). Next, the final model is evaluated on the test set, and the generalization error is computed. This process is depicted in Fig. 8.2.

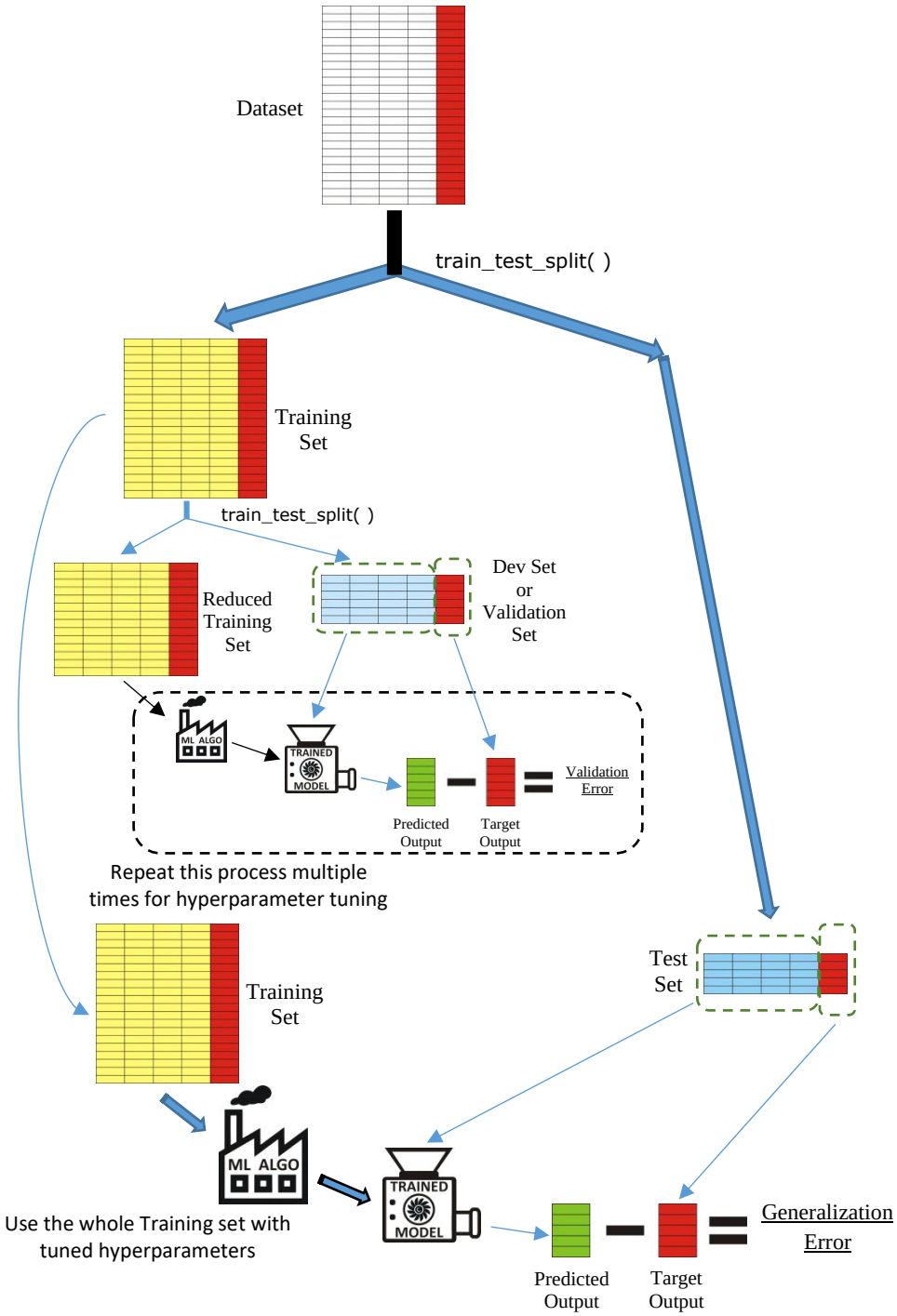


Fig. 8.2: Model training with Holdout Validation approach

The validation error is used to ensure the validity of the model. It checks whether our model is too simple to learn underlying patterns or too complex to learn everything in the training set. The learning algorithm that performs poorly on the training set by producing large training error (with reference to a baseline/human-level performance) is said to underfit. This indicates high bias, which comes from developing a simpler model using fewer parameters and lesser degrees of freedom. The learning algorithm that performs well on the training data (with reference to a baseline/human-level performance), but not well on the validation set is said to overfit the training data. This indicates high variance, which comes from developing a complex model using too many parameters and higher degrees of freedom. Thus, the bias-variance tradeoff is to be taken care of for a balanced learning. That said, it is vital to strike a balance between model complexity and performance on the unseen data. Fig. 8.3 shows the performance of three different training example and explains how to detect overfitting and underfitting.

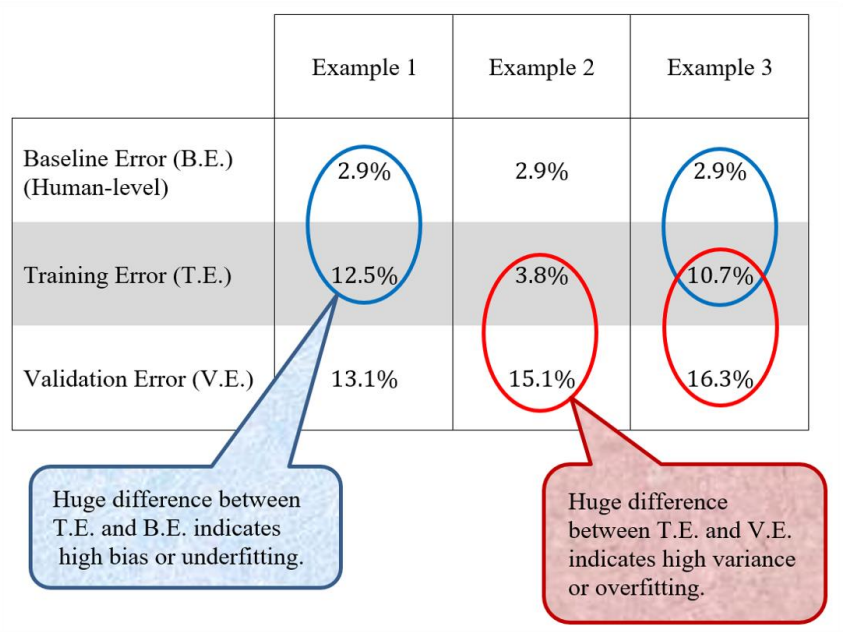


Fig. 8.3: Identifying high bias and high variance using the training and validation errors.

The holdout validation technique usually works reasonably well. However, there are some flaws with it. The results obtained from holdout validation can be highly dependent on the specific way in which the data is split into training and validation sets. This can lead to variability in the performance of the model and make it difficult to compare different hyperparameters. Evaluating the model performance with a small validation set may to lead to select suboptimal choices of hyperparameters. A larger validation set makes the reduced training set smaller. In that case, a smaller training set will be used for training the model. The hyperparameters tuned on a smaller set will then be used for model training on the original dataset. This issue is addressed by the k-fold cross validation technique.

8.6.2. k-fold Cross-Validation Technique

In k-fold cross-validation, the reduced training set is split into k equally sized folds/subsets. The model is then trained on $k-1$ of the folds and validated on the one remaining fold. This process is repeated k times (k trials), with each of the k folds being used as the validation set exactly once. The performance of the model is then calculated as the average performance in k trials. The multiple evaluations are averaged to get a better measure of the performance. Based on this estimate, the best

hyperparameter values are selected or obtained by trying out different values in the next iteration of cross-validation. The major drawback with this technique is manifold increment in the computation time. For $k = 5$, the five-fold cross-validation is depicted in Fig. 8.4. Note that the model averaging is typically applied during the evaluation and selection phase of model development, and should not be confused with **model ensemble techniques**, where multiple models are combined during the training phase to create a single, more powerful model. Model averaging in the context of k-fold cross-validation is used to obtain a more accurate estimate of a model's performance and to aid in model selection, rather than directly influencing the training of the models themselves.

8.7. Regularization

Regularization is a technique used to balance the trade-off between bias and variance. An effective regularization approach aims to minimize the variance without significantly increasing the bias. This means that the ideal regularization should achieve a considerable reduction in variance while keeping the increase in bias to a minimum. During the training process, the regularization method adds an additional constraint or penalty term to the cost function that the neural network is attempting to optimize. The penalty term is a function of the model parameter weight, and it can take different forms depending on the type of regularization used, such as L1 regularization or L2 regularization (Goodfellow, 2016; Aggarwal, 2018). The addition of a regularization term to the cost function affects updating of the model parameters by modifying the gradient of the cost function with respect to the weights. The gradients will now include the derivative of the regularization term, that penalizes large weights.

The primary objective of adding a penalty term in regularization techniques is to impose a constraint on a neural network model from having excessively large weight values, and instead, to have smaller weight values that are better suited for generalization, thereby enhancing the model's capacity to handle new data.

Following are some commonly used regularization methods (Nusrat and Jang, 2018; Dawani, 2020).

8.7.1. L2-Regularization

The ***L2-Regularization*** (based on the l_2-norm) also known as the ***Tikhonov regularization***, involves adding a constraint or penalty term to the cost function of the model, which is proportional to the sum of the squares of the weights. This results in smaller weight updates during backpropagation, which, in turn, leads to a smoother optimization and a better chance of finding a global minimum. Therefore, L2-regularization is also sometimes called weight decay.

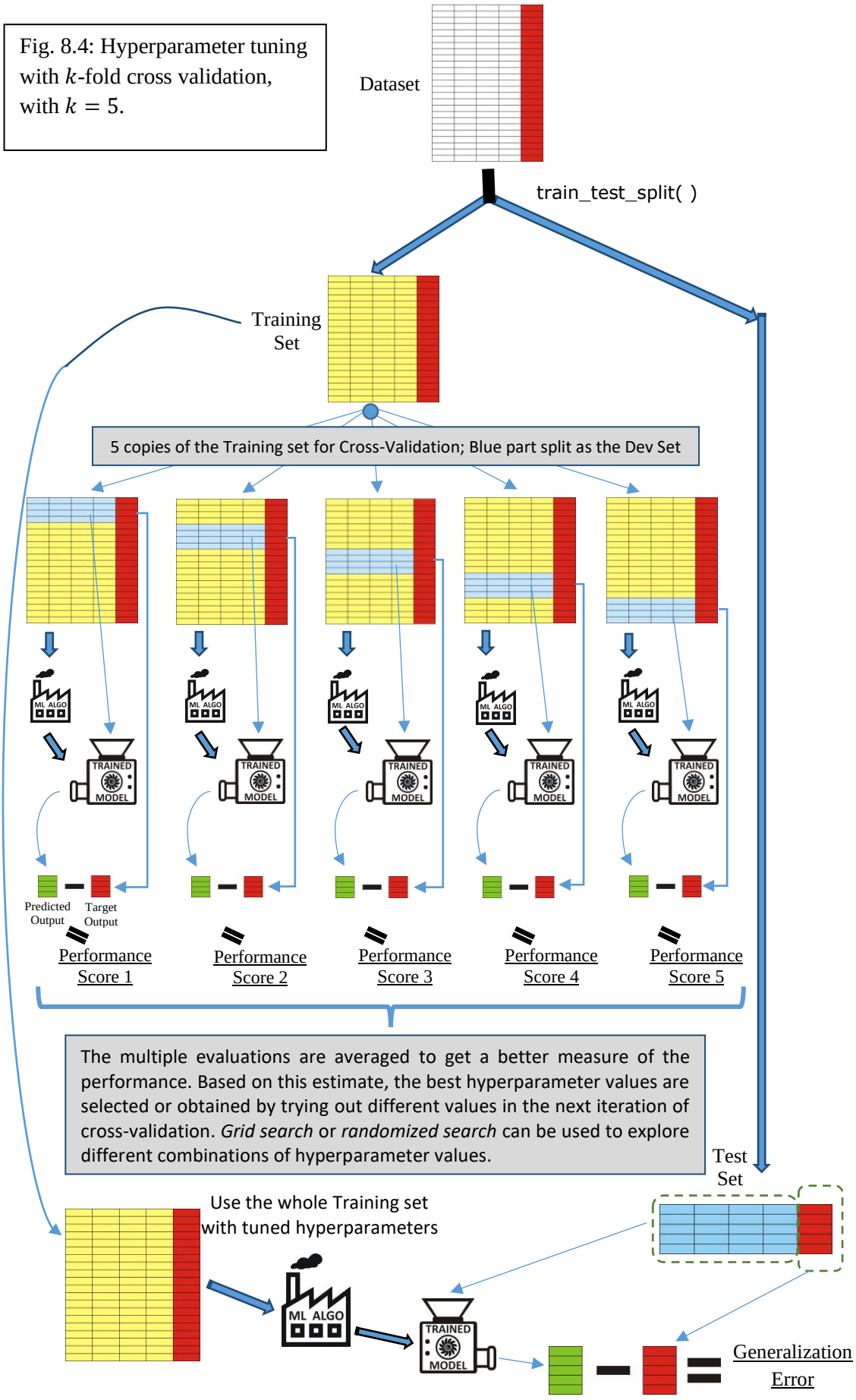
The cost function with L2-regularization term can be expressed in component form as:

$$c^{(p)} = \frac{1}{2}\sum_{j=1}^{n_L}\left(y_j^{(p)} - \hat{y}_j^{[L](p)}\right)^2 + \frac{\lambda}{2}\sum_{l=1}^{L}\left(\sum_{j=1}^{n_l}\sum_{i=1}^{n_{l-1}}\left(w_{j,i}^{[l]}\right)^2\right)$$

This can be expressed in vector form, using the Frobenius norm of matrices, as:

$$c^{(p)} = \frac{1}{2}\left\|Y^{(p)} - \hat{Y}^{[L](p)}\right\|_2^2 + \frac{\lambda}{2}\sum_{l=1}^{L}\left(\left\|\overline{\boldsymbol{W}}^{[l]}\right\|_F\right)^2$$

Fig. 8.4: Hyperparameter tuning with k-fold cross validation, with $k = 5$.



Here, F in the subscript shows the Frobenius norm of the matrix of all the weights of the model. λ is the regularization parameter, which controls the strength of the regularization penalty. The gradient of the penalty term with respect to the weights is $\lambda\left(w_{j,i}^{[l]}\right)$ and is included in the relevant term of Eq. $(G.3)$ for updating the weights. Rearrangement gives $(1-\alpha\lambda)$ as the coefficient of previous weight and is termed as weight decay factor.

8.7.2. L1-Regularization

The ***L1-Regularization*** (based on the l_1-norm) adds a penalty term to the cost function proportional to the absolute value of the weights which encourages sparsity. This leads to some of the weights becoming zero during training, which can lead to a simpler and more interpretable model. The cost function with L1-regularization term can be expressed in component form as:

$$c^{(p)} = \frac{1}{2}\sum_{j=1}^{n_L}\left(y_j^{(p)} - \hat{y}_j^{[L](p)}\right)^2 + \frac{\lambda}{2}\sum_{l=1}^{L}\left(\sum_{j=1}^{n_l}\sum_{i=1}^{n_{l-1}}\left|w_{j,i}^{[l]}\right|\right)$$

This can be expressed in vector form, using the l_1-norm of vectors, as:

$$c^{(p)} = \frac{1}{2}\left\|Y^{(p)} - \hat{Y}^{[L](p)}\right\|_2^2 + \frac{\lambda}{2}\sum_{l=1}^{L}\sum_{j=1}^{n_l}\left\|W_j^{[l]}\right\|_1$$

The gradient of the penalty term with respect to the weights is $\lambda t_{j,i}$, where

$$t_{j,i} = \begin{cases} -1, & w_{j,i}^{[l]} < 0 \\ 1, & w_{j,i}^{[l]} > 0 \end{cases}$$

The gradient $\lambda t_{j,i}$ is included in the relevant term of Eq. $(G.3)$ for updating the weights. An issue with the L1-regularization term is that it is based on absolute value terms, which are not differentiable when $w_{i,j}$ is exactly equal to zero. To address this issue of zero weights, $t_{j,i}$ can be set to zero, or chosen stochastically from $\{-1,1\}$. However, note that, the weights often do not become zero in practice because the numerical errors are typically sufficient to prevent these to be exactly zero.

The L2-regularization usually outperforms L1-regularization in producing accurate results, hence is a preferable choice. There is another regularization, called the ***Elastic-Net Regularization***, which adds the penalty terms of both the L1 and L2-regularization method to achieve the strengths of both methods. The Elastic-Net regularization can be particularly useful in situations where there are many input features (i.e., high-dimensional data) (Zou and Hastie, 2005; Hastie et al., 2016).

8.7.3. Dropout Regularization

The Dropout regularization method involves randomly dropping out (i.e., setting to zero) some of the units (neurons) in a neural network during training. This technique was first introduced by Srivastava et al. (2014) and used frequently in deep learning. The basic idea is to reduce the co-dependency between the neurons in the network by randomly dropping out a certain percentage of neurons during training.

During each training iteration, a certain percentage or probability p of neurons are randomly selected to be dropped out. Therefore, the remaining neurons have to learn to represent the input data without relying on the dropped-out neurons. This helps to prevent the network from memorizing the training data and instead learn more robust and generalizable features.

The probability p of dropout is a hyperparameter that is typically set through experimentation. A higher probability of dropout can lead to better generalization performance, but too high probability can cause underfitting. On the other hand, a lower probability of dropout can lead to better training performance, but too low probability can cause overfitting. Therefore, probability 30% to 50% is usually used in literature. After training, the dropout technique is usually turned off, and the full network is used for making predictions.

8.7.4. Early Stopping Regularization

Interestingly, the information that is extracted from a dataset during generalization can be thought of as two kinds: *signal* and *noise*. The signal helps in learning for better generalization of unseen data. In contrast, the noise makes the model memorize patterns for accuracy only on the training data but not the unseen data. **Underfitting** the training data occurs when the model has not learned enough signals. Therefore, the validation error is not lowered to a possible or desired level. **Overfitting** the training data occurs when the model has learned significant noises. Therefore, the validation error is not lowered to a possible or desired level (Rhnyewale, 2020). A deep learning practitioner attempts to find a balance or trade-off.

Ideal learning creates a trained model that learns all of the signals (avoiding underfitting perfectly) and none of the noises (avoiding underfitting perfectly). This does not happen in reality, however. So we allow the model to learn both the signals and noises as long as the validation error drops. After a certain point, the validation error tends to increase again. Having the lowest validation error is the point where the early stopping of the learning process can be made.

The ***Early Stopping*** is kind of regularization that is used for iterative learning algorithms. In this technique, the performance of the model is monitored during the validation process using the development set. While training a model, the graphs of the training error and the validation error can be plotted after each epoch. These graphs are called as the ***learning curves***. The trends of these graphs can help detect overfitting and underfitting. The epochs of the training are stopped when the validation error stops decreasing over a number of epochs. The model parameters, only those for which the validation error gets reduced, are saved. When the iterative process is terminated the lasted saved effective model parameters are used. Unlike the Batch Gradient Descent method, noticing the attainment of minimum level of validation error is not easy in case of the Stochastic and Mini-batch Gradient Descent method, because the error curves are not smooth. For such cases, once the error somewhat rises again over the minimum level for several epochs, the iterative process is stopped. Fig. 8.5 depicts this technique.

Note that the learning curves can also be plotted as the losses/errors as functions of the training set size to detect the overfitting. The overfitting can be vanished by increasing the size of data set, however, after a certain point there no much benefit of increasing the data size for better performance. This is depicted in Fig. 8.6.

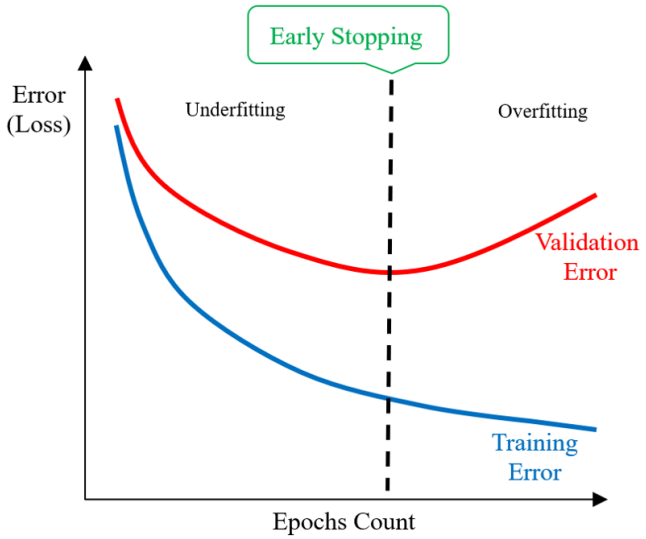


Fig. 8.5: Using Learning curves to detect a point for the Early Stopping.

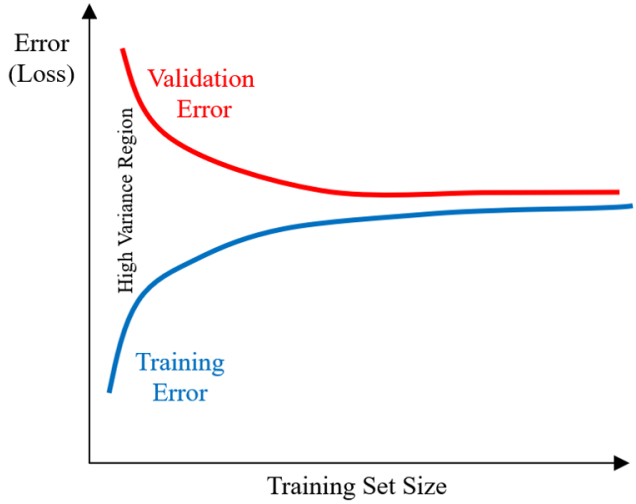


Fig. 8.6: Using Learning curves to detect the overfitting.

8.7.5. Data Augmentation (Dataset Augmentation)

Deep feedforward networks require a substantial amount of data to learn underlying data distributions and make accurate predictions on new data. However, gathering large, high-quality labelled datasets can be challenging and costly for certain problems. To circumvent the scarcity of training data, one effective method is data augmentation. This technique involves generating synthetic data and using it to train deep neural networks. The process typically utilizes generative models to learn the underlying dataset distribution. The generative model then creates synthetic data resembling ground-truth data, making it appear as if it originates from the same dataset. By artificially increasing the size of the training dataset, the data augmentation serves as a regularization technique to prevent overfitting.

Enhancing Robustness with Variations

To further enhance robustness and generalization to unseen data, various augmentation strategies are employed, particularly in computer vision tasks:

- **Image Cropping and Rotation:** These techniques involve cropping segments of input images or rotating them by specific angles. They have been proven to improve model performance by exposing it to diverse viewpoints.
- **Noise Injection:** Injecting noise into data can enhance the model's resilience to real-world noise. Synthetic data can be corrupted, blurred, or infused with Gaussian noise. These modifications help the model generalize better in noisy environments.

Challenges and Limitations

While data augmentation is highly effective, there are situations where it may face limitations:

- **Optical Character Recognition (OCR):** In OCR tasks, certain transformations like horizontal flips or 180-degree rotations can alter class labels. For instance, 'b' may become 'd,' and '6' may become '9.' In such cases, careful consideration is required when applying augmentation.
- **Specialized Domains:** Some domains, such as medical imaging with MRI and CT scans, may not permit traditional data augmentation. However, alternatives like affine transformations (e.g., rotations and translations) can be applied.

Noise Injection Strategies

Noise injection can be a powerful regularization technique. There are two primary ways to inject noise:

- **Injecting Noise to Input Data:** Adding noise directly to input data can improve the model's robustness by making it more resilient to variations and disturbances.
- **Injecting Noise into Hidden Units:** Injecting noise into hidden units during training is found to be an effective regularizer. It encourages stability and prevents overfitting, often outperforming traditional parameter shrinking techniques.

In summary, data augmentation is a valuable tool to enhance the generalization and robustness of deep neural networks, especially when dealing with limited or noisy training data. It complements other regularization techniques and helps mitigate the challenges associated with data-hungry models.

8.8. Batch Normalization of Each Layer

Batch normalization is a technique used in neural networks to prevent arising of the issues of vanishing and exploding gradients, as well as internal covariate shifts. Internal covariate shift occurs when the distribution of the inputs to each layer changes during training which can slow down the training process and make it more difficult for the model to converge to an optimal solution (Ioffe and Szegedy, 2015). By normalizing the inputs of each hidden layer, batch normalization helps the model learn more quickly with greater stability. This improves the performance of deep neural networks. Faster convergence is especially beneficial for larger datasets. Batch normalization can also act as a *regularizer*, thus reducing overfitting and improving generalization. It can make neural networks less sensitive to the choice of hyperparameters, such as learning rate and weight initialization, which can simplify the hyperparameter tuning process. Batch normalization also enables faster convergence with stability, allowing for higher learning rates. It has become a standard technique in deep learning and is widely used in many state-of-the-art architectures (Goodfellow, 2016; Aggarwal, 2018).

Batch normalization is applied layer-wise, while the neuron values for a layer l for each of the instance in the mini-batch are being computed. The batch normalization is applied either on the linearly transformed inputs (that is, $z_j^{[l](p)}$), or on the result of application of the activation function (that is,

$\hat{y}_j^{[l](p)}$). Here, we discuss the former approach as it is argued to be more advantageous. First, the normalization of the linearly transformed input for each of the neuron across all the instances in the mini-batch under consideration is performed and then resulting values are scaled and shifted to by learnable parameters, which allows the model to learn an optimal scale and shift for each feature. The steps of the batch normalization are explained below.

For the batch normalization of a layer l, across m instances of the mini-batch, the values at jth neuron, for $j = 1,2,3,\cdots,n_l$, of pth instances are treated as follows:

1. Compute the mean μ_j and variance σ_j^2:

$$\mu_j = \frac{1}{m}\left(\sum_{p=1}^{m} z_j^{[l](p)}\right), \qquad \text{for } j = 1,2,3,\cdots,n_l$$

$$\sigma_j^2 = \frac{1}{m}\left(\sum_{p=1}^{m}\left(z_j^{[l](p)} - \mu_j\right)\right)^2, \qquad \text{for } j = 1,2,3,\cdots,n_l$$

2. Normalize the activations by subtracting the mean and dividing by the standard deviation:

$$\hat{z}_j^{[l](p)} = \frac{z_j^{[l](p)} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

   Here, ε is a small constant added for numerical stability.

3. Scale and shift the normalized values using trainable parameters γ_j and β_j, respectively.

$$z_j^{[l](p)} = \gamma_j\left(\hat{z}_j^{[l](p)}\right) + \beta_j$$

   The scaling parameter γ_j scales the normalized value of the feature, and allows the model to learn the optimal scale for each feature dimension. The shifting parameter β_j shifts the normalized value of the feature, and allows the model to learn the optimal mean for each feature dimension.

Once the batch normalized values are ready, activation function is applied to obtain the output neuron values in layer l, which are then used as input for the next layer. When batch normalization is applied, the bias term is not needed, because the shifting parameter can accommodate for the role of the bias term.

9. Some Advanced Gradient Descent Strategies for Efficient Training

9.1. Gradient Descent with Momentum

As discussed earlier, Gradient Descent (GD) is a commonly used optimization method, but it can sometimes be slow, particularly when dealing with minor or noisy gradients. To address this issue, the

technique of using a ***momentum term*** with the GD method, also known as Polyak's Heavy Ball method, was introduced by Polyak (1964). It accelerates the learning process and prevents the optimizer from getting stuck in a local minimum.

The method introduces a variable v, i.e., the velocity, which represents the direction and speed at which the parameters (weights and biases) move through the parameter space. It facilitates the optimizer to take larger steps or gain momentum in the direction of the optimal solution. The value of v at each iteration depends on the previous value of v as well as the gradient of the cost function at the current estimate of the minimum:

$$v_{j,i}^{[l]} = \beta v_{j,i}^{[l]} - \alpha \frac{\partial C}{\partial w_{j,i}^{[l]}} \quad ---(G.5)$$

Here, $\beta \in (0,1)$ is a ***momentum*** or ***friction hyperparameter***. Thus, Eq. $(G.3)$ assumes the following form for updating the weights:

$$w_{j,i}^{[l]} = w_{j,i}^{[l]} + v_{j,i}^{[l]} \quad ---(G.6)$$

It is understood that the left-hand side value is the newer one, obtained by using the older value of the variable on the right side of the equation. Similar modifications for updating the biases can be made. Here, for layer l,

$$V_j^{[l]} = \begin{bmatrix} v_{j,1}^{[l]} \\ v_{j,2}^{[l]} \\ \vdots \\ v_{j,n_{l-1}}^{[l]} \end{bmatrix}_{n_{l-1}\times 1}$$

Thus, the matrix $\overline{\boldsymbol{V}}^{[l]}$ has the form (vertical stacking of n_l column vectors for vectorized implementation):

$$\overline{\boldsymbol{V}}^{[l]} = \begin{bmatrix} -- & {V_1^{[l]}}^T & -- \\ -- & {V_2^{[l]}}^T & -- \\ & \vdots & \\ -- & {V_{n_l}^{[l]}}^T & -- \end{bmatrix}_{n_l\times(n_{l-1})}$$

In the vectorized form, the Eqs. $(G.5)$ and $(G.6)$ can be written as,

$$\boldsymbol{V}^{[l]} = \beta \boldsymbol{V}^{[l]} - \alpha \boldsymbol{d}\overline{\boldsymbol{W}}^{[l]} \quad ---(G.7)$$

$$\overline{\boldsymbol{W}}^{[l]} = \overline{\boldsymbol{W}}^{[l]} + \boldsymbol{V}^{[l]} \quad ---(G.8)$$

The hyperparameter β controls the impact of the previous gradients on the current updating step. Specifically, a higher β value gives more weight to the previous gradients, leading to a smoother and more stable updating process that helps to avoid getting stuck in local optima. On the other hand, a lower β value gives less weight to the previous gradients, leading to more oscillations and potentially faster convergence but with a higher risk of overshooting the minimum. Therefore, choosing the proper

value of β is essential for achieving good performance in gradient descent with momentum. In literature, $\beta = 0.9$ is considered effective in practice (Srinivasan et al., 2018; Chen et al., 2022). Nakerst et al. (2020) proposed a scheme to further accelerate the GD with momentum and demonstrated it to be usable with other adaptive learning rate techniques.

9.2. Gradient Descent with Nesterov Momentum

Nesterov acceleration is a method of accelerating the convergence of the GD algorithms. It was introduced by Nesterov (1983, 2004). The basic idea of Nesterov acceleration is to use a momentum term with the previous update direction. This helps to avoid oscillations and overshooting and may perform better than the GD with momentum term with no acceleration.

Sutskever et al. (2013) proposed a modification to the momentum algorithm that was influenced by Nesterov's accelerated gradient method. The updating term for the model parameter to be used in Eq. $(G.5)$ is given by,

$$v_{j,i}^{[l]} = \beta v_{j,i}^{[l]} - \alpha \frac{1}{m} \sum_{p=1}^{m} \frac{\partial c^{(p)}\left(w_{j,i}^{[l]} + \beta v_{j,i}^{[l]}, b_j^{[l]}\right)}{\partial w_{j,i}^{[l]}} \quad ---(G.9)$$

This is accomplished by adding $\beta v_{j,i}^{[l]}$ to each weight $w_{j,i}^{[l]}$ and then performing the computations of the forward pass, loss function, and their gradients. In this way, we have the cost function C as:

$$C = C(\overline{\boldsymbol{W}}^{[1]} + \beta \boldsymbol{V}^{[1]}, B^{[1]}, \quad \overline{\boldsymbol{W}}^{[2]} + \beta \boldsymbol{V}^{[2]}, B^{[2]}, \quad \cdots, \quad \overline{\boldsymbol{W}}^{[L]} + \beta \boldsymbol{V}^{[l]}, B^{[L]})$$

instead of

$$C = C(\overline{\boldsymbol{W}}^{[1]}, B^{[1]}, \quad \overline{\boldsymbol{W}}^{[2]}, B^{[2]}, \quad \cdots, \quad \overline{\boldsymbol{W}}^{[L]}, B^{[L]})$$

In the vectorized form, the Eq. $(G.9)$ can be written as,

$$\boldsymbol{V}^{[l]} = \beta \boldsymbol{V}^{[l]} - \alpha \frac{\partial C\left(\overline{\boldsymbol{W}}^{[l]} + \beta \boldsymbol{V}^{[l]}, B^{[1]}\right)}{\partial \overline{\boldsymbol{W}}^{[l]}} \quad ---(G.10)$$

Eq. $(G.10)$ is used in Eq. $(G.8)$ for updating the weights. Similar modifications for updating the biases can be made. Including the momentum term in the gradient calculations requires extra computational resources.

9.3. Gradient Descent with Adaptive Learning Rates

9.3.1. AdaGrad (The Adaptive Gradient Technique)

In traditional gradient descent, the learning rate is fixed and is the same for all parameters. However, this can sometimes lead to slow convergence or even divergence. The AdaGrad technique, introduced by Duchi et al. (2011), can converge rapidly to a solution in high-dimensional, sparse parameter spaces. The AdaGrad algorithm adapts the learning rate of each parameter based on its historical gradients and computes a different learning rate for each parameter. The learning rate is inversely proportional to the square root of the sum of the squared gradients for that parameter up to the current iteration. The updating term for the model parameter is given by,

$$a_{j,i}^{[l]} = a_{j,i}^{[l]} + \left(\frac{\partial C}{\partial w_{j,i}^{[l]}}\right)^2 \quad ---(G.11)$$

Here, $a_{j,i}^{[l]}$ is the *aggregate* value of the ith parameter. $a_{j,i}^{[l]}$ is initialized with zero. Eq. $(G.3)$ assumes the following form for updating the weights:

$$w_{j,i}^{[l]} = w_{j,i}^{[l]} - \frac{\alpha}{\sqrt{a_{j,i}^{[l]}}}\frac{\partial C}{\partial w_{j,i}^{[l]}} \quad ---(G.12)$$

A small positive value such as $\epsilon = 10^{-8}$ can be added into $a_{j,i}^{[l]}$ for stabilization of the divisions involving small values. Similar formulas for updating the biases can be made. In the vectorized form, Eqs. $(G.11)$ and $(G.12)$ can be written as,

$$\boldsymbol{A}^{[l]} = \boldsymbol{A}^{[l]} + \boldsymbol{d\overline{W}}^{[l]} \otimes \boldsymbol{d\overline{W}}^{[l]} \quad ---(G.13)$$

$$\boldsymbol{\overline{W}}^{[l]} = \boldsymbol{\overline{W}}^{[l]} - \alpha\left(1 \oslash \sqrt[\boxdot]{\boldsymbol{A}^{[l]}}\right) \otimes \boldsymbol{d\overline{W}}^{[l]} \quad ---(G.14)$$

Here, also in the subsequent sections, $\sqrt[\boxdot]{\ }$ denotes element-wise square root, $\oslash$ denotes element-wise reciprocal, and $\otimes$ denotes element-wise multiplication operations for the matrices.

9.3.2. RMSProp (The Root Mean Square Propagation Technique)

Geoffrey Hinton (2012) first introduced the RMSProp algorithm in his Coursera lecture series as a modification to the AdaGrad algorithm. One drawback of AdaGrad is that $a_{j,i}^{[l]}$ becomes very large over time, and the learning rate becomes very small and eventually decreases to zero for non-convex functions. To address this issue, RMSProp uses an exponentially weighted moving average of the squared gradients and discards history from the extreme past. This allows for a more adaptive learning rate, scaling the updating parameters based on the size of gradients in the current updates. It uses a decay factor $\beta_2 \in (0,1)$ that controls the length scale of exponentially moving average. The updating term for the model parameter is given by,

$$a_{j,i}^{[l]} = \beta_2 a_{j,i}^{[l]} + (1-\beta_2)\left(\frac{\partial C}{\partial w_{j,i}^{[l]}}\right)^2 \quad ---(G.16)$$

Eq. $(G.16)$ is used in Eq. $(G.12)$ to update the weights. A small positive value such as $\epsilon = 10^{-8}$ can be added into $a_{j,i}^{[l]}$ for stabilization of the divisions involving small values. Similar formulas for updating the biases can be made. In the vectorized form, Eq. $(G.16)$ can be written as,

$$\overline{\boldsymbol{A}}^{[l]} = \beta_2 \overline{\boldsymbol{A}}^{[l]} + (1-\beta_2)\left(\boldsymbol{d\overline{W}}^{[l]} \otimes \boldsymbol{d\overline{W}}^{[l]}\right) \quad ---(G.17)$$

Here, $\overline{\boldsymbol{A}}^{[l]}$ is matrix having dimensions $n_l \times (n_{l-1})$, just like for $\boldsymbol{d\overline{W}}^{[l]}$ for any layer l. Eq. $(G.17)$ is used in Eq. $(G.14)$ to update the weights.

9.3.3. AdaDelta (The Adaptive Delta Technique)

AdaDelta is an adaptive learning rate optimization algorithm that was introduced by Zeiler (2012). It replaces the global learning rate with a ratio that depends on the root mean square of gradients. The updating term for the model parameter is given by,

$$\psi_{j,i}^{[l]} = \beta_2 \psi_{j,i}^{[l]} + (1-\beta_2)\left(\frac{\partial C}{\partial w_{j,i}^{[l]}}\right)^2 \quad ---(G.18)$$

The value of $\psi_{j,i}^{[l]}$ is based on the past gradients. $\psi_{j,i}^{[l]}$ is initialized with zero. Eq. $(G.3)$ assumes the following form for updating the weights:

$$w_{j,i}^{[l]} = w_{j,i}^{[l]} - \sqrt{\frac{\psi_{j,i}^{[l]}}{a_{j,i}^{[l]}}}\frac{\partial C}{\partial w_{j,i}^{[l]}} \quad ---(G.19)$$

$a_{j,i}^{[l]}$ is the aggregate value of the ith parameter, to be taken from Eq. $(G.16)$. A small positive value such as $\epsilon = 10^{-8}$ can be added into $a_{j,i}^{[l]}$ for stabilization of the divisions involving small values. Similar formulas for updating the biases can be made. In the vectorized form, Eqs. $(G.18)$ and $(G.19)$ can be written as,

$$\overline{\boldsymbol{\psi}}^{[l]} = \beta_2 \overline{\boldsymbol{\psi}}^{[l]} + (1-\beta_2)\left(d\overline{\boldsymbol{W}}^{[l]} \otimes d\overline{\boldsymbol{W}}^{[l]}\right) \quad ---(G.20)$$

$$\overline{\boldsymbol{W}}^{[l]} = \overline{\boldsymbol{W}}^{[l]} - \sqrt[\square]{\overline{\boldsymbol{\psi}}^{[l]}} \otimes \left(\frac{1}{\sqrt[\square]{\overline{\boldsymbol{A}}^{[l]}}}\right) \otimes d\overline{\boldsymbol{W}}^{[l]} \quad ---(G.21)$$

Here, $\overline{\boldsymbol{\psi}}^{[l]}$ is matrix having dimensions $n_l \times (n_{l-1})$, just like $d\overline{\boldsymbol{W}}^{[l]}$ for any layer l.

9.3.4. Adam (The Adaptive Moment Estimation Technique)

Adam is an optimization algorithm that combines ideas from RMSProp and momentum. It was introduced by Kingma & Ba (2017). It has become one of the most popular optimization algorithms in deep learning.

Adam maintains two moving averages of the gradient; one is exponential smoothing, and the second is exponential moving average. The first moment $f_{j,i}^{[l]}$, referred to as the ***mean*** or ***velocity***, is an exponentially decaying average of past gradients. It keeps track of the gradients' direction and helps smooth the gradient updates. The purpose of this term is similar to the momentum term in other optimization algorithms, such as gradient descent with momentum. The second moment $a_{j,i}^{[l]}$, also known as the ***uncentered variance***, is an exponentially decaying average of the squared gradients. It measures the variance or uncertainty of the gradients. By adapting the learning rate for each weight based on the variance of the gradients, the Adam algorithm can dynamically adjust the step size for different weights. This adaptive learning rate improves the optimization process by providing larger

updates for less frequently changing weights and smaller updates for frequently changing weights. The expression for the first moment $f_{j,i}^{[l]}$ is given by,

$$f_{j,i}^{[l]} = \beta_1 f_{j,i}^{[l]} + (1-\beta_1)\left(\frac{\partial C}{\partial w_{j,i}^{[l]}}\right) \quad ---(G.22)$$

Thus, Eq. $(G.3)$ assumes the following form for updating the weights:

$$w_{j,i}^{[l]} = w_{j,i}^{[l]} - \frac{\alpha^t}{\sqrt{a_{j,i}^{[l]}}} f_{j,i}^{[l]} \quad ---(G.23)$$

A small positive value such as $\epsilon = 10^{-8}$ can be added into $a_{j,i}^{[l]}$ for stabilization of the divisions involving small values. Similar formulas for updating the biases can be made. Here, α^t is the bias correction factor, with the value,

$$\alpha^t = \alpha\frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t} \quad ---(G.24)$$

In the vectorized form, Eqs. $(G.22)$ and $(G.23)$ can be written as,

$$\overline{\boldsymbol{F}}^{[l]} = \beta_1\overline{\boldsymbol{F}}^{[l]} + (1-\beta_1)\boldsymbol{d}\overline{\boldsymbol{W}}^{[l]} \quad ---(G.25)$$

$$\overline{\boldsymbol{W}}^{[l]} = \overline{\boldsymbol{W}}^{[l]} - \alpha^t\left(\frac{1}{\boxdot\sqrt{\overline{\boldsymbol{A}}^{[l]}}}\right)\otimes\overline{\boldsymbol{F}}^{[l]} \quad ---(G.26)$$

Here, $\overline{\boldsymbol{F}}^{[l]}$ is matrix having dimensions $n_l \times (n_{l-1})$, just like $\boldsymbol{d}\overline{\boldsymbol{W}}^{[l]}$ for any layer l.

The Adam technique offers a delicate balance between adaptive learning rates and momentum. Its ability to dynamically adjust the learning rate based on the magnitude of gradients, coupled with the incorporation of historical gradient information, fosters more efficient and stable convergence. By seamlessly navigating the complex optimization landscape, Adam empowers neural networks to transcend conventional boundaries and unlock their true potential for solving intricate real-world problems (Goodfellow, 2016; Aggarwal, 2018).

10. Enhancing the Model Performance for the Test Case

Tuning various hyperparameters, specially making a judicious choice of the cost function optimization method, as discussed in Section 8 and 9, the performance of the code can be improved significantly. Following are the modifications made in the code given in Section 7.3.2. This results in a neural network model that performs much better.

```python
# Loading the dataset suzuki_data.csv with dimensions: 5252x14
# Stock price data of Pak Suzuki Motor Company Limited (Pakistan)
# Stock prices daily data from 01 Jan 2001 to 17 Nov 2022

import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler

import matplotlib.pyplot as plt
%matplotlib inline

df = pd.read_csv('suzuki_data.csv')

## Setting Hyperparameters
MAX_epochs = 20         # Maximum number of epochs
tol = 0.000001          # Tolerance
alpha = 0.01            # Learning rate
m = 50                 # Number of instances in a minibatch
window_size = 10

train_test_ratio = 0.95
train_validation_ratio = 0.8

# Setting Layers and Number of neurons in each layer
ldim = [window_size, 20, 30, 10, 5, 1]
L = len(ldim) - 1    # Number of layers (other than the input layer)

ldim = np.array(ldim, dtype=int)
np.random.seed(1)

print("Original Dataset Head:")
print(df.head())

print("Original Dataset Tail:")
print(df.tail(5))

# Loading and making the data ready in desired format

df = df.dropna()                # data cleaning as desired

# Selecting the relevant column for training and prediction
df_main = df['Close']
df_main = np.array(df_main).reshape(-1,1)

# normalizing the data between 0 to 1
normalizing = MinMaxScaler(feature_range=(0,1))
normalized_data = normalizing.fit_transform(df_main)

```

```python
## Spliting the time series data into train and test sets
## Temporal order of the data is preserved

train_size = int(len(normalized_data) * train_test_ratio)
train_data = normalized_data[:train_size]
test_data  = normalized_data[train_size:]

# Function for creating datasets using Sliding Window Technique
# This is a common technique for time series prediction using FNN
# Slider window size is the time step

def dataset_creator(window_size, data):
    X, Y = [], []
    for i in range(window_size, len(data)):
        X.append(data[i-window_size:i, 0])
        Y.append(data[i, 0])
    return np.array(X), np.array(Y)


# Calculate the split index based on the ratio
splitter = int(len(train_data) * train_validation_ratio)

# Creating the training, validation and test sets
X_train, Y_train = dataset_creator(window_size, train_data[:splitter])
X_val, Y_val = dataset_creator(window_size, train_data[splitter:])
X_test, Y_test = dataset_creator(window_size, test_data)

```

Original Dataset Head:

	Date	Open	High	Low	Close	Volume
0	2001-01-01	10.25	10.25	10.25	10.25	1000.0
1	2001-01-02	10.25	11.50	10.70	11.30	10500.0
2	2001-01-03	11.30	11.30	10.75	10.75	6500.0
3	2001-01-04	10.75	11.30	11.25	11.25	3000.0
4	2001-01-05	11.25	11.30	11.05	11.05	6500.0

Original Dataset Tail:

	Date	Open	High	Low	Close	Volume
5247	2022-11-11	161.50	162.98	159.00	160.84	18222.0
5248	2022-11-14	159.00	159.85	156.94	157.25	52296.0
5249	2022-11-15	158.50	164.50	157.01	162.27	151394.0
5250	2022-11-16	163.00	164.00	160.00	160.29	49327.0
5251	2022-11-17	161.69	161.70	158.60	159.21	43334.0

```
import tensorflow as tf
from sklearn.model_selection import train_test_split
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.optimizers import Adam
from tensorflow import keras
from tensorflow.keras import initializers
from tensorflow.keras import layers

act = 'ReLU'        # Activation function
# Some other choices include 'sigmoid' , 'tanh' , and 'swish'


err = 'mean_squared_error'
dropout_rate = 0.2

model = keras.models.Sequential()
model.add(tf.keras.Input(shape=(ldim[0],)))
#model.add(layers.Dropout(dropout_rate))

model.add(layers.Dense(units=ldim[1],
                       kernel_initializer=initializers.RandomNormal(),
                       bias_initializer=initializers.Zeros(),
                       activation=act))
#model.add(layers.Dropout(dropout_rate))

model.add(layers.Dense(units=ldim[2],
                       kernel_initializer=initializers.RandomNormal(),
                       bias_initializer=initializers.Zeros(),
                       activation=act))
#model.add(layers.Dropout(dropout_rate))

model.add(layers.Dense(units=ldim[3],
                       kernel_initializer=initializers.RandomNormal(),
                       bias_initializer=initializers.Zeros(),
                       activation=act))
#model.add(layers.Dropout(dropout_rate))

model.add(layers.Dense(units=ldim[4],
                       kernel_initializer=initializers.RandomNormal(),
                       bias_initializer=initializers.Zeros(),
                       activation=act))
#model.add(layers.Dropout(dropout_rate))

model.add(layers.Dense(units=ldim[5],
                       kernel_initializer=initializers.RandomNormal(),
                       bias_initializer=initializers.Zeros(),
                       activation=act))

```

```
#model.compile(loss= err , optimizer= SGD(learning_rate=alpha))

model.compile(loss= err, optimizer= Adam(learning_rate=alpha))

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 20)                220

 dense_1 (Dense)             (None, 30)                630

 dense_2 (Dense)             (None, 10)                310

 dense_3 (Dense)             (None, 5)                 55

 dense_4 (Dense)             (None, 1)                 6

=================================================================
Total params: 1,221
Trainable params: 1,221
Non-trainable params: 0
_________________________________________________________________
```

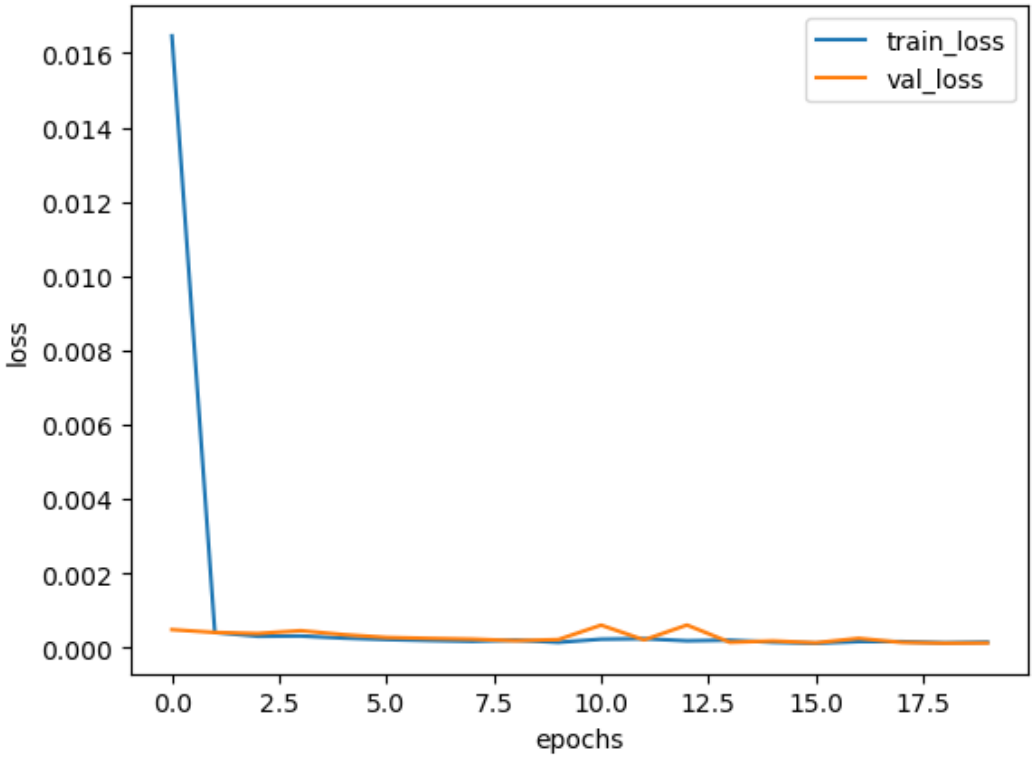
```
## Defining the function for comparing predicted and true values

def  pred_compare(Y_test1, Y_predict):

    xvals = list(range(0, len(Y_test1)))

    # Plot the true values and predicted values

    plt.figure(figsize=(12, 6))
    plt.plot(xvals, Y_test1, label='True Values',
             marker='o', linestyle='-', color='b')
    plt.plot(xvals, Y_predict, label='Predicted Values',
             marker='x', linestyle='-', color='r')

    plt.xlabel('Predicted Values')
    plt.ylabel('Stock Price')
    plt.legend()
    plt.tight_layout()
    plt.show()

    # function pred_compare() ended

```

```
## Defining the function for calculating MAE and RMSE

def  reg_errors(Y_test1, Y_predict):

#    Y_predict = normalizing.inverse_transform(Y_predict_normalized)
#    Y_test1 = normalizing.inverse_transform(Y_test.reshape(-1,1))
#    #print("shape ypredict",Y_test1.shape)

    from sklearn.metrics import mean_absolute_error
    from sklearn.metrics import mean_squared_error

    mae = mean_absolute_error(Y_test1,Y_predict)
    mse = mean_squared_error(Y_test1,Y_predict)
    rmse = np.sqrt(mse)

    print("MAE in the prediction for the test data:", mae)
    print("RMSE in the prediction for the test data: ", rmse)

    # function reg_errors() ended

```

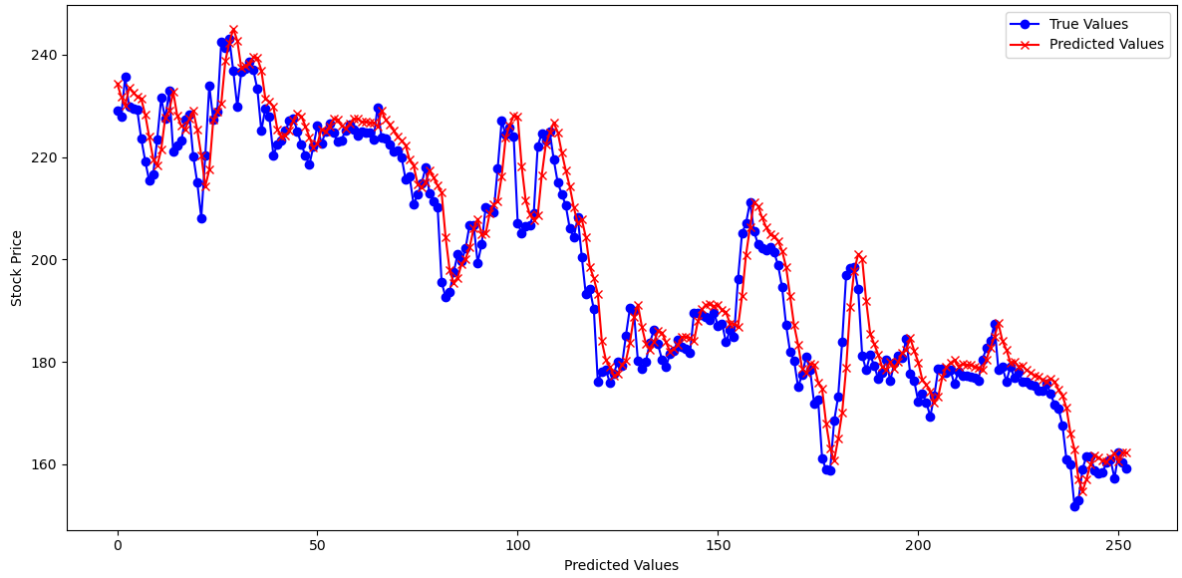
```
# Training the model

print('\n========== Training process started ==========\n')
tm = model.fit(X_train, Y_train, epochs=MAX_epochs, batch_size=m,
                                        validation_data=(X_val, Y_val))
print('\n========== Training process completed ==========')
print("\n")


plt.plot(tm.history['loss'])
plt.plot(tm.history['val_loss'])
plt.ylabel('loss')
plt.xlabel('epochs')
plt.legend(['train_loss','val_loss'], loc = 'upper right')
print(plt.show())


## using the optimized parameters for making predictions
## The Forward Pass with the test data
print('\n========== Prediction process started ==========')
Y_predict_normalized = model.predict(X_test)
print('\n========== Prediction process completed ==========')


# Recovering original stock proces from the normalized ones
Y_predict = normalizing.inverse_transform(Y_predict_normalized)
Y_test1 = normalizing.inverse_transform(Y_test.reshape(-1,1))


```

```
print("\n========= Comparing the predicted and true values =========")
pred_compare(Y_test1, Y_predict)


print("\n======================= Test Errors ======================")
reg_errors(Y_test1, Y_predict)

```

```
========== Training process started ==========

Epoch 1/20
80/80 [==============================] - 1s 5ms/step - loss: 0.0165 - val_l
oss: 4.7995e-04
Epoch 2/20
80/80 [==============================] - 0s 2ms/step - loss: 4.0469e-04 - v
al_loss: 4.0412e-04
...
Epoch 20/20
80/80 [==============================] - 0s 2ms/step - loss: 1.4306e-04 - v
al_loss: 1.2029e-04


========== Training process completed ==========
```




```
========== Prediction process started ==========

========== Prediction process completed ==========

========= Comparing the predicted and true values =========
```




```
======================== Test Errors ========================

MAE in the prediction for the test data: 4.580867688326025

RMSE in the prediction for the test data:  6.0117266939831575
```

Conclusion

This article provides a concise and comprehensive overview of deep learning with Artificial Neural Networks (ANNs), offering valuable insights into their mathematical foundations, training procedures, and optimization techniques. It employs graphical illustrations to demystify complex concepts, including the formulation of backpropagation and optimization approaches. The paper extensively presents pseudo-codes in both element-wise and vectorized forms. Furthermore, a Python-based vectorized implementation of ANN algorithms is showcased, delivering performance results akin to those achieved with established frameworks like TensorFlow. Next, the article delves into techniques for enhancing model generalization, tackling training challenges, and optimizing network performance. It underscores the significance of hyperparameter tuning by demonstrating substantial improvements in the neural network model's performance for a test case problem. By equipping readers with this knowledge and practical implementations, this article seeks to establish a robust foundation that fosters enduring progress in deep learning. It is a valuable resource for both students and practitioners alike, serving as a stepping stone towards mastering the intricate field of deep learning.

Acknowledgements

The authors are highly thankful to Prof. Dr. Muhammad Ali Ismail (Principal Investigator) and Mr. Uzair Abid (Team Lead) at National Center for Big Data & Cloud Computing, NED University of Engineering and Technology, Karachi, Pakistan, for their continuous guidance and support. The authors are thankful to Mr. Muneeb Rashid and Mr. Muhammad Jameel (Data Science @ FAST-NUCES, Islamabad, Pakistan), for their support and assistance on the subject.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A. K., Dean, J., & Zheng, X. (2016). TensorFlow: A System for Large-Scale Machine Learning, Operating Systems Design and Implementation (pp. 265–283). https://doi.org/10.5555/3026877.3026899

Aggarwal, C. C. (2018). Neural Networks and Deep Learning: A Textbook. Springer.

Agostinelli, F., Hoffman, M., Sadowski, P., Baldi, P. (2015). Learning Activation Functions to Improve Deep Neural Networks, arXiv:1412.6830v3. https://arxiv.org/abs/1412.6830v3

Ali, A., Kazmi, S. Z., Shahzadi, G., Rashid, M., & Ahsan, M. (2023). Demystifying CNN with Mathematical Insights: A Prelude with Application to an AI-based Sustainable Solution for Diabetic Retinopathy Diagnosis. Research Square. https://doi.org/10.21203/rs.3.rs-3338196/v1

Ali, A., & Syed, K. S. (2013). An Outlook of High Performance Computing Infrastructures for Scientific Computing, Advances in Computers, 91, 87-118. https://doi.org/10.1016/b978-0-12-408089-8.00003-3

Alexender, A. (2023). MIT 6.S191: Introduction to Deep Learning. YouTube. https://youtube.com/playlist?list=PLtBw6njQRU-rwp5__7C0oIVt26ZgjG9NI. http://introtodeeplearning.com

Bishop, C. M. (2006). Pattern Recognition and Machine Learning. Springer Verlag.

Chen, J. J., Wolfe, C. R., Li, Z., & Kyrillidis, A. (2022). Demon: Improved Neural Network Training with Momentum Decay. In ICASSP 2022 - 2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). https://doi.org/10.1109/icassp43922.2022.9746839

Chollet, F., et al. (2015) Keras, GitHub. https://github.com/fchollet/keras

Clevert, D. A., Unterthiner, T., & Hochreiter, S. (2016). Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs), arXiv:1511.07289v5. https://arxiv.org/abs/1511.07289v5

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. Mathematics of Control, Signals, and Systems, 2(4), 303-314.

Dawani, J. (2020). Hands-On Mathematics for Deep Learning: Build a solid mathematical foundation for training efficient deep neural networks. Packt Publishing Ltd.

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization, Journal of Machine Learning Research, 12(61), 2121−2159.

Dugas, C., Bengio, Y., Bélisle, F., Nadeau, C., & Garcia, R. (2001). Incorporating Second-Order Functional Knowledge for Better Option Pricing. Advances in Neural Information Processing Systems, 13, 330-336.

Emmert-Streib, F., Yang, Z., Han, F., Tripathi, S., & Dehmer, M. (2020). An introductory review of deep learning for prediction models with big data. Frontiers in Artificial Intelligence, 3. https://doi.org/10.3389/frai.2020.00004

Glorot, X., & Bengio, Y. (2010). Understanding the Difficulty of Training Deep Feedforward Neural Networks, In JMLR W&CP: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2010), 9, 249-256.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press. https://www.deeplearningbook.org/

Hagan, M., Demuth, H., Beale, M., & De Jesus, O. (2014). Neural Network Design, Second Edition. Martin Hagan.

Hahnloser, R. H., Sarpeshkar, R., Mahowald, M. A., Douglas, R. J., & Seung, H. S. (2000). Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. Nature, 405(6789), 947-951

Hastie, T., Tibshirani, R., & Friedman, J. H. (2016). The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition. Springer.

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In Proceedings of the IEEE international conference on computer vision. 1026-1034

Hendrycks, D., & Gimpel, K. (2020). Gaussian Error Linear Units (GELUs), arXiv:1606.08415v4. https://arxiv.org/abs/1606.08415v4

Higham, C. F., & Higham, D. J. (2019). Deep Learning: An Introduction for Applied Mathematicians. Siam Review, 61(3), 860–891. https://doi.org/10.1137/18m1165748

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. Neural Networks, 4(2), 251–257. https://doi.org/10.1016/0893-6080(91)90009-t

Ioffe, S., Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, arXiv:1502.03167v3. https://arxiv.org/abs/1502.03167v3

kaggle-titanic (2017). Github. https://github.com/agconti/kaggle-titanic

Kawaguchi, K., Bengio, Y., & Kaelbling, L. (2022). Generalization in Deep Learning. In P. Grohs & G. Kutyniok (Eds.), Mathematical Aspects of Deep Learning, 112-148, Cambridge University Press.

Kingma, D.P., Ba, J. (2017). Adam: A Method for Stochastic Optimization, arXiv:1412.6980v9. https://arxiv.org/abs/1412.6980v9

Kutyniok, G. (2022). The Mathematics of Artificial Intelligence, arXiv:2203.08890. https://doi.org/10.48550/arxiv.2203.08890

LeCun, Y., Bengio, Y. & Hinton, G. (2015). Deep learning, Nature, **521**, 436-444. https://doi.org/10.1038/nature14539

Marcus, G. (2018). Deep Learning: A Critical Appraisal. arXiv:1801.00631, https://doi.org/10.48550/arXiv.1801.00631

Mhaskar, H. N., Liao, Q., & Poggio, T. (2016). Learning functions: When is deep better than shallow. arXiv: 01603.00988v4

McCulloch, W. S., & Pitts, W. (1943). A Logical Calculus of The Ideas Immanent in Nervous Activity. The Bulletin of Mathematical Biophysics, 5, 115–133. https://doi.org/10.1007/bf02478259

MIT 6.S191: Introduction to Deep Learning
https://youtube.com/playlist?list=PLtBw6njQRU-rwp5__7C0oIVt26ZgjG9NI

Nakerst, G., Brennan, J. D., & Haque, M. (2020). Gradient Descent with Momentum --- to Accelerate or to Super-Accelerate? arXiv:2001.06472. https://doi.org/10.48550/arxiv.2001.06472

Nesterov, Y. (1983). A Method for Solving a Convex Programming Problem with Convergence Rate O(1/K^2), Soviet Mathematics Doklady, 27, 372-367.

Nesterov, Y. (2004). Introductory Lectures on Convex Optimization, In Serries: Applied Optimization, Springer. https://doi.org/10.1007/978-1-4419-8853-9

Nielsen, M. A. (2015). Neural Networks and Deep Learning. http://neuralnetworksanddeeplearning.com/

Nusrat, I., & Jang, S. (2018). A Comparison of Regularization Techniques in Deep Neural Networks. Symmetry, 10(11), 648. https://doi.org/10.3390/sym10110648

Kaggle - Pakistan stock exchange data (All Companies), (2022). https://www.kaggle.com/datasets/mukhazarahmad/pakistan-stock-exchange-data-all-companies

Polyak, B. T. (1964). Some Methods of Speeding Up the Convergence of Iteration Methods. U.S.S.R. Computational Mathematics And Mathematical Physics, 4(5), 1–17. https://doi.org/10.1016/0041-5553(64)90137-5

Ramachandran, P. (2017). Searching for Activation Functions, arXiv:1710.05941v2. https://arxiv.org/abs/1710.05941

Rhnyewale. (2020). Introduction to Deep Learning Using Keras and TensorFlow — Part2 - The Startup - Medium. https://rhnyewale.medium.com/introduction-to-deep-learning-using-keras-and-tensorflow-part2-e3c6d342ada8

Rolnick, D., & Tegmark, M. (2018). The power of deeper networks for expressing natural functions. IarXiv: 1705.05502

Rosenblatt, F. (1958). The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain. Psychological Review, 65(6), 386–408. https://doi.org/10.1037/h0042519

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning Representations by Back-propagating Errors, Nature, 323(6088), 533–536. https://doi.org/10.1038/323533a0

Sarker, I. H. (2021). Deep Learning: a comprehensive overview on techniques, taxonomy, applications and research directions. SN Computer Science, 2(6). https://doi.org/10.1007/s42979-021-00815-1

Schmidhuber, J. (2015). Deep learning in neural networks: An overview, Neural Networks, 61, 85–117. https://doi.org/10.1016/j.neunet.2014.09.003

Srinivasan, V., Sankar, A. R., & Balasubramanian, V. N. (2018). ADINE: An Adaptive Momentum Method for Stochastic Gradient Descent. https://doi.org/10.1145/3152494.3152515

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Journal of Machine Learning Research, 15(56), 1929–1958.

Strang, G. (2019). Linear Algebra and Learning from Data. Wellesley-Cambridge Press.

Vidal, R., Bruna, J., Giryes, R., & Soatto, S. (2017). Mathematics of Deep Learning. arXiv:1712.04741. https://doi.org/10.48550/arxiv.1712.04741

Xu, B., Wang, N., Chen, T., & Li, M. (2015). Empirical evaluation of rectified activations in convolutional network. arXiv preprint arXiv:1505.00853

Yosinski, J., Clune, J., Bengio, Y., & Lipson, H. (2014). How Transferable are Features in Deep Neural Networks? arXiv:1411.1792. https://arxiv.org/abs/1411.1792

Zeiler, M.D. (2012). ADADELTA: An Adaptive Learning Rate Method, arXiv:1212.5701. http://arxiv.org/abs/1212.5701

Zou, H. & Hastie T. (2005). Regularization and Variable Selection via the Elastic Net. Journal of the Royal Statistical Society, B. 67(2), 301-302.

Zhou, D. (2020). Universality of deep convolutional neural networks. arXiv: 1805.10769

Conflicts of Interest: The authors have *no conflicts* of interest to declare that are relevant to the content of this article.