DL USING TENSORFLOW

Written by

AADIL HUSSAIN

Chapter 1
Introduction to Deep Learning & Tensorflow

1.1 Overview of Deep Learning

Deep Learning is a subset of AI , field fall under Machine Learning that uses multilayered neural networks, known as deep neural networks, to mimic the complex decision-making ability of the human brain. Most artificial intelligence (AI) applications in our lives today rely on some form of deep learning.

The main difference between deep learning and machine learning is the design of the underlying neural network. Traditional machine learning models use simple neural networks with one or two computational layers. In contrast, deep learning models use three or more layers, often including hundreds or thousands of layers to train the models.

These deep networks, inspired by the human brain, allow computers to perform tasks like image recognition and natural language processing without explicit programming.

The field draws inspiration from biological neuroscience and focuses on stacking artificial neurons into layers and training them to process data. The term “deep” refers to the multiple layers in the network, which can range from three to hundreds or thousands.

Definition:

- *Fundamentally, deep learning refers to a class of machine learning algorithms in which a hierarchy of layers is used to transform input data into a progressively more abstract and composite representation.*

The term “deep” in deep learning highlights the multilayered nature of data transformation within these models. Specifically, deep learning systems are characterized by a significant depth of the credit assignment path (CAP)—the sequence of transformations leading from input to

output. CAPs elucidate potential causal relationships between inputs and outputs. For feedforward neural networks, the CAP depth is determined by the number of hidden layers plus one, accounting for the output layer. In the case of recurrent neural networks, where signals may traverse through layers multiple times, the depth of the CAP can theoretically be infinite. While a universally accepted delineation between shallow and deep learning does not exist, experts generally concur that a CAP depth exceeding two signifies deep learning. Notably, a CAP of depth two has been demonstrated to possess universal approximation capabilities, capable of emulating any function. Beyond this point, additional layers do not enhance the function approximation potential of the network. However, deep models (with a CAP greater than two) tend to extract superior features compared to their shallow counterparts, illustrating the advantage of deeper architectures in effective feature learning.

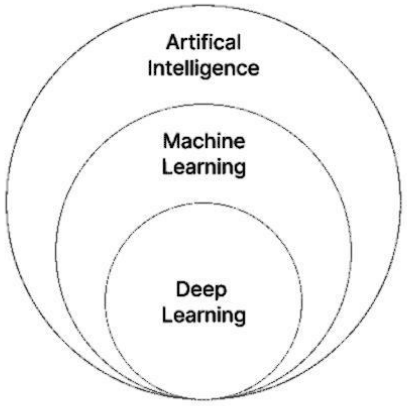


Fig 1.1 – Subsets of AI

A deep learning process can figure out the best features to use at different levels by itself. Before deep learning, machine learning methods often required manual feature engineering. This step transformed the data into a form that worked better for classification algorithms. In deep learning, features are not created by hand. Instead, the model automatically finds useful representations from the data. However, this doesn't remove the need for manual adjustments. For instance, changing the number of layers and their sizes can lead to different levels of abstraction.

Deep learning architectures can be constructed with a greedy layer-by-layer method. Deep learning helps to disentangle these abstractions and pick out which features improve performance.

Deep learning algorithms can be applied to unsupervised learning tasks. This is an important benefit because unlabeled data is more abundant than the labeled data. Examples of deep structures that can be trained in an unsupervised manner are deep belief networks

The concept of deep learning was brought to the attention of the machine learning community by Rina Dechter in 1986, and to the field of artificial neural networks by Igor Aizenberg and his team in 2000, specifically relating to Boolean threshold neurons. However, the timeline of its emergence seems to be more intricate.

Deep neural networks are typically understood through the lens of the universal approximation theorem or through probabilistic inference. The traditional universal approximation theorem addresses the ability of feedforward neural networks with a single hidden layer of finite size to represent continuous functions.

George Cybenko published the first proof in 1989 for sigmoid activation functions, which was later extended to multi-layer feedforward architectures by Kurt Hornik in 1991. More recent studies have also demonstrated that universal approximation applies to non-bounded activation functions like Kunihiko Fukushima's rectified linear unit. The universal approximation theorem specific to deep neural networks pertains to the capability of networks with limited width while allowing the depth to increase. Lu et al. established that if a deep neural network with ReLU activation has a width that is strictly greater than the input dimension, it can approximate any Lebesgue integrable function; conversely, if the width is less than or equal to the input dimension, such a deep neural network does not function as a universal approximator.

The probabilistic interpretation, rooted in machine learning, encompasses inference and optimization techniques related to training and testing, views activation nonlinearity as a cumula: History of Deep Learning tive distribution function, and has facilitated the emergence of dropout as a regularizer in neural networks, championed by researchers such as Hopfield, Widrow, and Narendra, with its significance highlighted in surveys like Bishop's.

1.2 : History of Deep Learning

Before 1980 , There are two categories of artificial neural networks (ANN): feedforward neural networks (FNN), also known as multilayer perceptrons (MLP), and recurrent neural networks (RNN). Unlike FNNs, which lack cycles in their connectivity structure, RNNs incorporate them. In the 1920s, Wilhelm Lenz and Ernst Ising developed the Ising model, which is fundamentally a non-learning RNN architecture made up of neuron-like threshold elements.

The 1980s and 90s marked a pivotal era for artificial intelligence and deep learning, transitioning from optimism to a sobering reality. Following Yann LeCun's groundbreaking

demonstration of backpropagation at Bell Labs in 1989, which utilized convolutional neural networks to decipher handwritten digits, the field faced a significant setback during the second AI winter from the late 1980s to the early 90s. The anticipations surrounding AI's immediate capabilities led to disillusionment among investors and researchers, rendering the term “Artificial Intelligence” nearly synonymous with pseudoscience. Nonetheless, dedicated researchers made noteworthy progress, culminating in the development of the support vector machine by Dana Cortes and Vladimir Vapnik in 1995, as well as the advent of long short-term memory (LSTM) in recurrent neural networks by Sepp Hochreiter and Juergen Schmidhuber in 1997. This decade set the stage for a technological breakthrough; by 1999, the introduction of faster computers and graphics processing units (GPUs) revolutionized data processing speeds, allowing neural networks to start outperforming traditional support vector machines. While the early 2000s presented challenges such as the Vanishing Gradient Problem—where lower layer features could not be effectively learned by upper layers—innovations in layer-by-layer pre-training and LSTM paved the way for overcoming these obstacles. The seminal META Group report in 2001 highlighted the burgeoning complexities of data growth, ushering in the age of Big Data. This foundational period culminated in 2009 with Fei-Fei Li’s launch of ImageNet, which amassed over 14 million labeled images, significantly advancing image recognition capabilities and energizing the field of deep learning for years to come.

2011-2020

By 2011, the speed of GPUs had increased significantly, making it possible to train convolutional neural networks “without” the layer-by-layer pre-training. With the increased computing speed, it became obvious deep learning had significant advantages in terms of efficiency and speed. One example is AlexNet, a convolutional neural network whose architecture won several international competitions during 2011 and 2012. Rectified linear units were used to enhance the speed and dropout.

Also in 2012, Google Brain released the results of an unusual project known as The Cat Experiment. The free-spirited project explored the difficulties of “unsupervised learning.” Deep learning uses “supervised learning,” meaning the convolutional neural net is trained using labeled data (think images from ImageNet). Using unsupervised learning, a convolutional neural net is given unlabeled data, and is then asked to seek out recurring patterns.

The Cat Experiment used a neural net spread over 1,000 computers. Ten million “unlabeled” images were taken randomly from YouTube, shown to the system, and then the training software was allowed to run. At the end of the training, one neuron in the highest layer was found to respond strongly to the images of cats. Andrew Ng, the project’s founder said, “We also found a neuron that responded very strongly to human faces.” Unsupervised learning remains a significant goal in the field of deep learnIng.

The Generative AdversarIal Neural Network (GAN) was introduced in 2014. GAN was created by Ian Goodfellow. With GAN, two neural networks play against each other in a game. The goal of the game is for one network to imitate a photo, and trick its opponent into believing it is real. The opponent is, of course, looking for flaws. The game is played until the near perfect photo tricks the opponent. GAN provides a way to perfect a product (and has also begun being used by scammers).

1.3 : Origin of Neural Networks - The building blocks

The evolution of deep learning started in 1940's 2 guys called Warren McCulloch and Walter Pitts proposed the concept of artificial neurons. They developed a mathematical model which based on the working of basic biological neuron. That artificial neuron is called Mcp-neuron.That laid the foundation for Deep Learning.

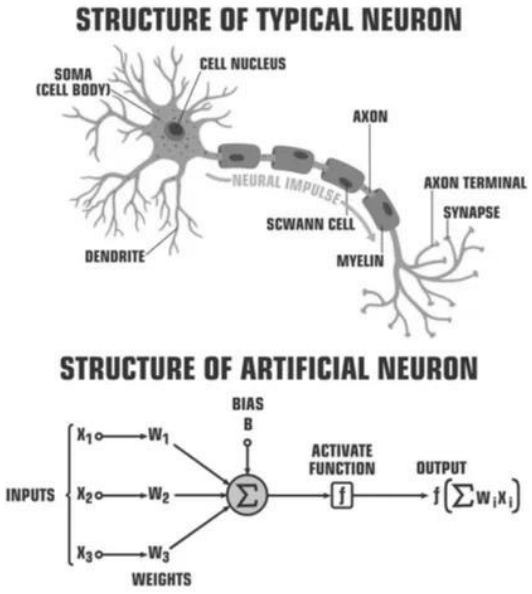


Fig 1.2 : Structure of Biological & Artificial Neural Networks

In 1957, Frank Rosenblatt introduced the perceptron, an innovative algorithm inspired by the structure and function of artificial neurons. This groundbreaking model demonstrated the ability to learn from experience, effectively adjusting its weights to enhance the accuracy of its predictions. However, the perceptron's capabilities were confined to solving linearly separable problems, which eventually contributed to a waning interest in neural networks. Following the

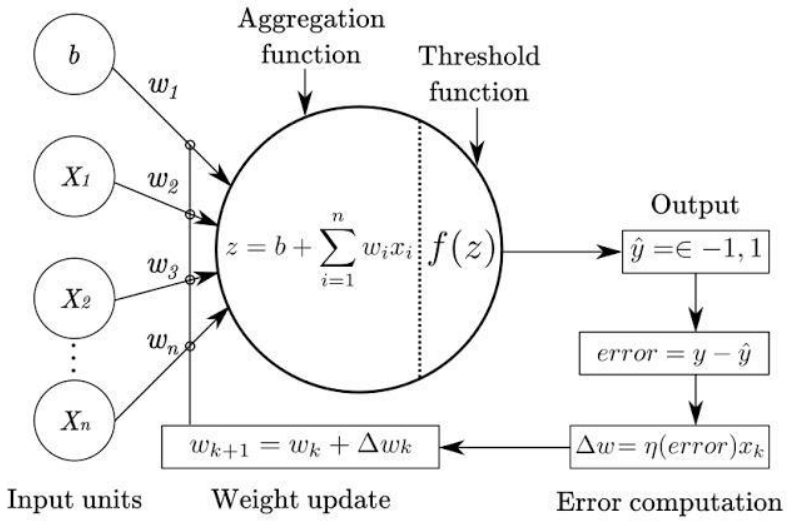
introduction of the perceptron, the field faced significant setbacks, culminating in what is known as the first AI winter—a period marked by stagnation in artificial intelligence research. During this time, funding dwindled, and enthusiasm for deep learning plummeted, particularly in the late 1960s, as skepticism grew regarding the efficacy of binary neurons and their capacity to tackle more complex problems. This decline in confidence stifled progress in the field and resulted in a nearly complete halt in deep learning research efforts, revealing the fragile nature of early advancements in artificial intelligence.

1.4 : The Perceptron

The perceptron is an algorithm that plays a foundational role in the field of supervised learning, particularly for developing binary classifiers—essentially decision-making tools that help determine whether a given input, represented as a vector of numerical values, fits into a particular category or class.

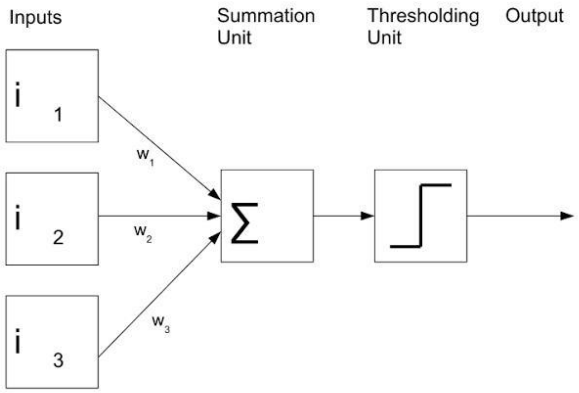
At its core, this algorithm leverages a methodology akin to that of a linear classifier, a sophisticated classification approach that generates predictions based on a linear predictor function. This function integrates a series of weights with the input feature vector, establishing a decision boundary to effectively segregate the different classes. The perceptron itself acts as a critical building block for more complex models and techniques in machine learning, highlighting its significance and versatility in various applications. By iteratively adjusting the weights based on the input data and corresponding labels during the training phase, the perceptron learns to minimize classification errors over time. This iterative learning process is essential for refining the model's performance, ultimately enabling it to classify new, unseen instances accurately. Given its simplicity, the perceptron can serve as an excellent starting point for novice data scientists, offering them an accessible introduction to the principles of machine learning and decision-making processes. As a result, the utility of the perceptron extends far beyond its basic form, influencing a plethora of advanced algorithms and systems designed for tackling intricate classification problems across diverse domains. Whether it is in the realm of image recognition, spam detection, or medical diagnosis, the underlying principles of the perceptron continue to inspire the development of robust machine learning architectures that push the boundaries of what is possible in automated decision-making processes.

Definition : In the modern sense, the perceptron is an algorithm for learning a binary classifier called a threshold function:



The original Perceptron, developed by Frank Rosenblatt in the late 1950s, serves as one of the foundational models in the field of artificial intelligence and machine learning. This mechanism is designed to take multiple binary inputs—essentially yes or no, or true or false—producing a single binary output, which can similarly be interpreted in this way: 0 or 1. The core idea revolves around the concept of assigning different weights to each input, signifying how important each piece of information is when making a decision. For instance, let's consider a scenario where we try to decide whether to attend a concert. You would assess factors such as the quality of the artist and the likelihood of good weather. Each of these factors would have a weight, reflecting their significance to your decision.

To implement the Perceptron, you start by establishing a threshold value, let's say 1.5. Then, for each input, you multiply the input by its respective weight. For example, if you have three inputs representing the artist's quality (x1), the weather (x2), and perhaps the ticket price (x3), you would apply their corresponding weights. So, if you were to evaluate x1 (which might be 1 for good quality), you multiply it by its weight (say, 0.7), yielding a score of 0.7. You would do the same for each factor: if the weather is poor, you might assign that input a value of 0 and a weight of 0.6, resulting in a score of 0. In total, if your artist's score is 0.7, your weather score is 0, and the ticket price is favorable at, say, a weight of 0.4 (1 * 0.4 = 0.4), when you sum these outcomes – 0.7 + 0 + 0 + 0.4 – you get a weighted sum of 1.6. Since this value exceeds your threshold of 1.5, your Perceptron will output a definitive "yes," confirming that you should indeed go to the concert! This process demonstrates the simple yet effective methodology of the Perceptron algorithm, where inputs can influence decisions based on their assigned significance, providing a powerful tool for binary classification tasks in various applications.



A perceptron, often hailed as an essential art“fici”l neuron, mirrors the functioning of biological neurons and serves as a foundational element of artificial intelligence. This computational unit processes information by receiving inputs, termed nodes, each endowed with a binary value (1 for true, 0 for false) and a corresponding weight that reflects its significance in the overall calculation. In our example, the node values are 1, 0, 1, 0, and 1, with weights of 0.7, 0.6, 0.5, 0.3, and 0.4 respectively. The perceptron calculates a weighted sum by multiplying each input value by its weight . For the perceptron to produce an output, this sum must exceed a predetermined threshold value—in this case, 1.5. Once the summation is complete, an activation function evaluates the result to determine whether the perceptron “fires,” indicating a successful prediction. Beyond simple computation, perceptrons are capable of learning from experience through a training process that fine-tunes their weights based on feedback from labeled examples. By correcting its outputs, the perceptron enhances its accuracy and adaptability, making it a powerful tool for predictive modeling in various applications.

Basic Components of Perceptron

1. Input Layer:

- Comprised of one or more input neurons.

- Responsible for receiving input signals, either from external sources or from other layers within the neural network.

2. Weights:

- Each input neuron has an associated weight.

- Represents the strength of the connection between the input neuron and the output neuron.

- Weights are crucial in determining how much influence each input has on the final output.

3. Bias:

- A bias term is introduced alongside the input layer.

- Provides additional flexibility, allowing the perceptron to model complex patterns not captured solely by input signals.

- Acts as an offset, ensuring that the model can make predictions even when all input features are zero.

4. Activation Function:

- Determines the perceptron's output based on the weighted sum of inputs plus the bias.

- Common types of activation functions include:

- **Step Function**: Produces a binary output based on a threshold.

- **Sigmoid Function**: Maps input values to a range between 0 and 1, useful for probability estimation.

- ReLU Function: Outputs zero for negative inputs and passes positive inputs unchanged, enhancing computational efficiency.

5. Output:

- The perceptron delivers a single binary output, either 0 or 1.

- Represents the predicted class or category for the given input data based on learned criteria during training.

6. Training Algorithm :

- The perceptron is usually trained using a supervised learning approach.

- Algorithms such as the perceptron learning algorithm or backpropagation facilitate this training process.

- During training, weights and biases are iteratively adjusted to minimize the discrepancy between predicted and actual outputs for a training dataset, ensuring improved accuracy over time.

DL using Tensorflow

This overview illustrates the essential components of a perceptron, forming the backbone of many foundational machine learning models. Understanding these elements is crucial for anyone looking to delve into artificial neural networks and their applications.

Biological Neuron	Artificial Neuron
Cell Nucleus	Nodes
Dendrites	Input
Synapse	Weights
Axon	Output

1.5 : The Neural Networks

A neural network is a sophisticated computational model inspired by the intricacies of the human brain, consisting of interconnected units referred to as artificial neurons. These neurons, although designed to mimic their biological counterparts, have been subject to extensive research aimed at enhancing accuracy and efficiency. Recent investigations into artificial neuron models that closely resemble the biological neurons have yielded impressive results, showcasing significant performance improvements across various applications, from image recognition to natural language processing. Each artificial neuron operates by receiving inputs — signals from its connected neurons — which represent real numbers. These inputs are then processed through a mathematical construct known as an activation function, which introduces a non-linear transformation. This non-linearity is critical, as it allows the network to capture complex patterns within the data.

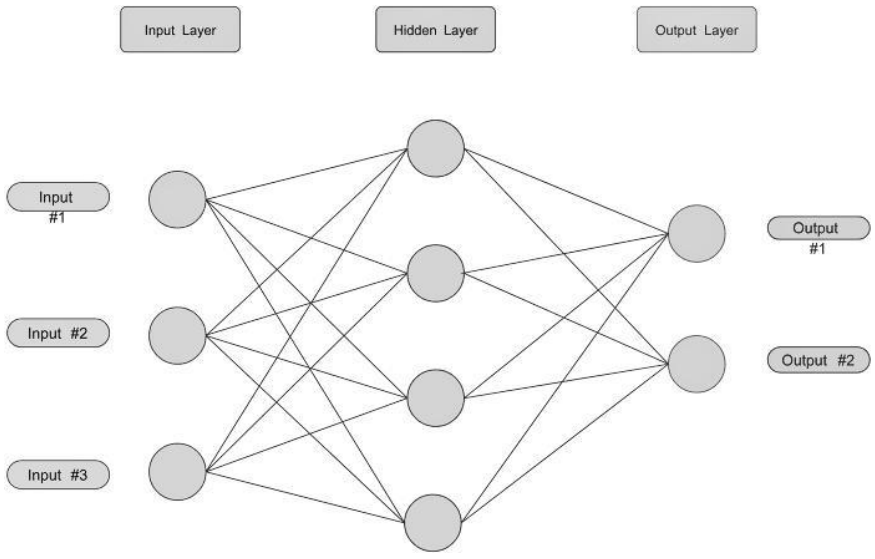


Fig 1.4 : Neural Network Diagram

Connections between neurons, called edges, symbolize the synapses found in biological neural networks, and the strength of these connections is modulated by weights that are fine-tuned during the training phase of the neural network. This

learning process is what distinguishes a neural network from traditional programming, as it enables the model to adapt and improve its performance over time based on the data it encounters. Typically, neurons are organized into layers, with each layer serving a distinct purpose in the processing pipeline. The first layer, known as the input layer, receives the initial data, which then propagates through subsequent layers — often referred to as hidden layers — before reaching the final output layer that delivers the network's predictions or classifications.

In deep neural networks, which are characterized by having at least two hidden layers, the complexity of the tasks they can address is greatly enhanced. Each hidden layer can learn to detect various features or patterns based on the representations learned from the previous layers. For instance, in image processing, the first hidden layer may identify simple edges, while deeper layers may learn to recognize more complex shapes and eventually entire objects. This hierarchical learning enables deep networks to perform exceptionally well on intricate tasks that require a nuanced understanding of vast amounts of data.

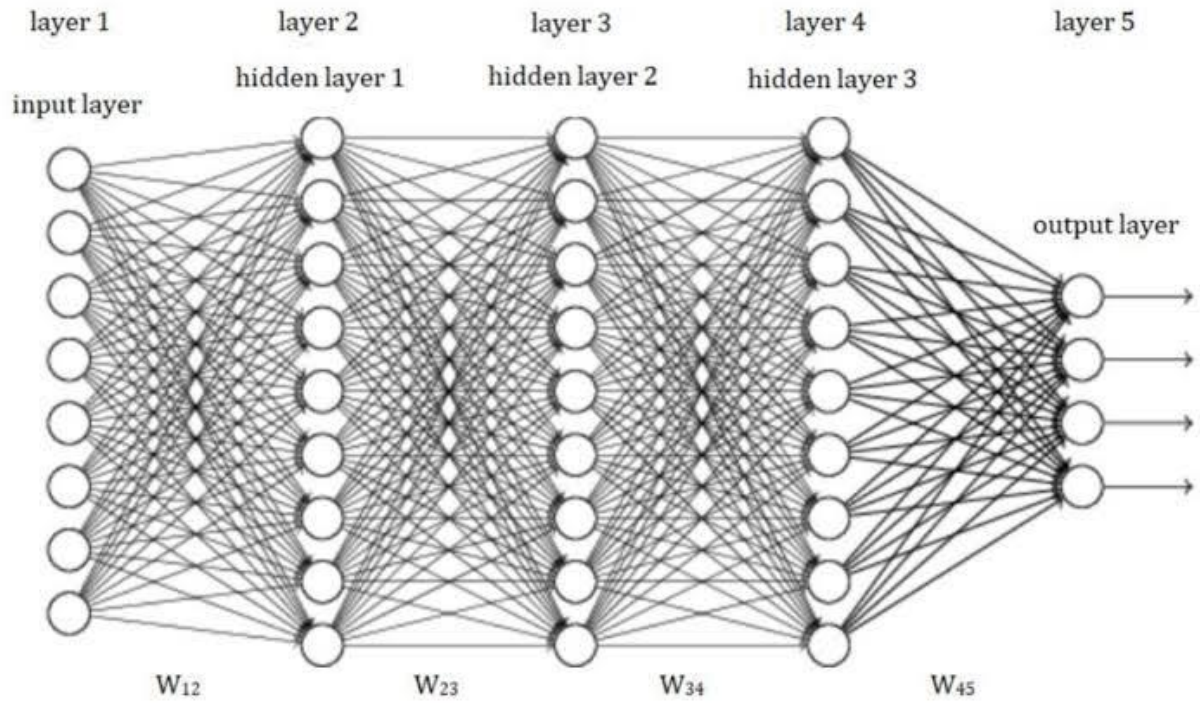


Fig 1.6 : Deep Neural Networks

Moreover, the training of these networks is supported by a process known as backpropagation, which effectively adjusts the weights of the connections to minimize errors in predictions. This optimization is essential in enhancing the overall functionality of the network, allowing it to generalize better to new, unseen data. The combination of multi-layered architectures and sophisticated training techniques is what empowers deep learning as a vital sector within the field of artificial intelligence. As research progresses and computational power increases, the potential applications of neural networks expand significantly, influencing industries such as healthcare, finance, and autonomous systems. Their ability to learn from experience positions them as a revolutionary tool in modeling complex real-world phenomena, thus shaping the future of technology and our understanding of intelligence itself.

1.6 : Training of Neural Networks

Neural networks, a cornerstone of modern machine learning, are primarily trained through a concept known as empirical risk minimization. This methodology revolves around the optimization of the network's parameters to reduce the discrepancy between the predicted outputs generated by the network and the actual target values sourced from a given dataset.

In essence, it seeks to fine-tune the model so that its predictions become increasingly accurate. To achieve this optimization, gradient-based methods are frequently employed, with backpropagation being the most widely recognized technique. This approach utilizes the gradients of the loss function – a quantitative measure of prediction error – to guide the adjustments of the network's weights and biases toward a minimum loss. During the training phase, artificial neural networks (ANNs) engage in a learning process characterized by their interaction with labeled training data. This data contains both input features and corresponding target outputs, allowing the network to learn the underlying relationships within the data. By iteratively updating their parameters through a series of forward and backward passes, ANNs refine their predictions, systematically minimizing the defined loss function. Each iteration represents a step in the training Journey, as the

network internalizes patterns, generalizes from examples, and gradually improves its performance. Through this rigorous training process, ANNs become capable of making reliable predictions on unseen data, thus demonstrating their potential to solve complex problems across various domains such as image recognition, natural language processing, and more. Ultimately, the interplay of empirical risk minimization and sophisticated optimization techniques like backpropagation underscores the elegance and efficiency of neural networks in learning from data, highlighting their remarkable adaptability and effectiveness as models in today's data-driven landscape.

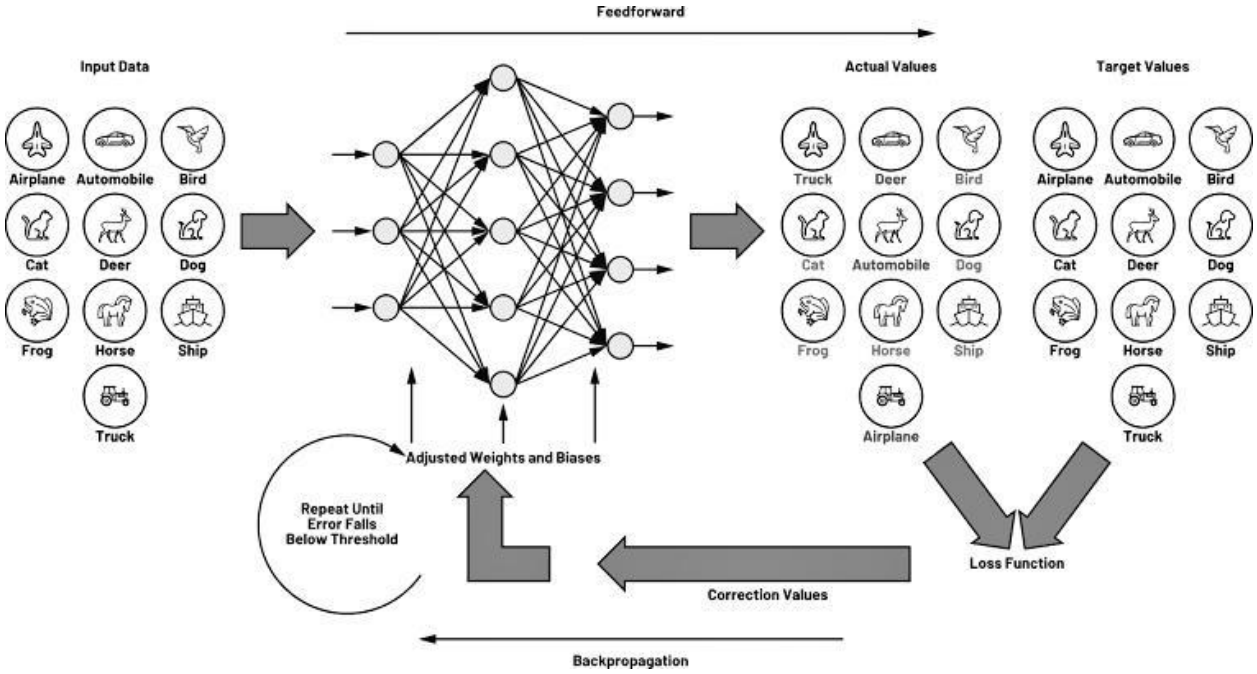


Fig 1.7 : Training of Neutral Networks

1.7 : Backpropagation – Neural Networks Training

Backpropagation is undoubtedly the cornerstone of training in neural networks, highly regarded as the most prevalent algorithm in this domain. At its core, it

facilitates the practical application of gradient descent in the complex landscape of multi-layered neural networks. For those working in machine learning, popular libraries such as Keras offer the luxury of automatic backpropagation, eliminating the need for developers to delve into the intricate details of the underlying calculations themselves. In a strictly technical sense, backpropagation specifically refers to the algorithm designed to compute gradients efficiently. However, the terminology is often stretched to encompass the entire training process, which includes the crucial step of adjusting model parameters in the direction opposite to the gradient. This adjustment is typically executed via methods like stochastic gradient descent or incorporated into more sophisticated optimizers such as Adaptive Moment Estimation (Adam). The historical development of backpropagation is rich and nuanced, featuring numerous discoveries, partial unveilings, and a somewhat convoluted terminological evolution. For instance, those familiar with the jargon of automatic differentiation may recognize backpropagation by its alternative names, including "reverse mode of automatic differentiation" or "reverse accumulation." The significance of backpropagation cannot be overstated, as it serves as the backbone for modern deep learning and neural network training methodologies, driving advancements across various applications, from natural language processing to computer vision. Understanding this algorithm's intricacies not only aids in the practical application of machine learning models but also deepens one's appreciation for the foundational concepts that power the impressive technological developments we see in today's digital landscape. Thus, mastering backpropagation and its associated techniques unlocks the potential for innovation and ensures that practitioners are well-equipped to tackle even the most complex challenges in the realm of artificial intelligence and beyond.

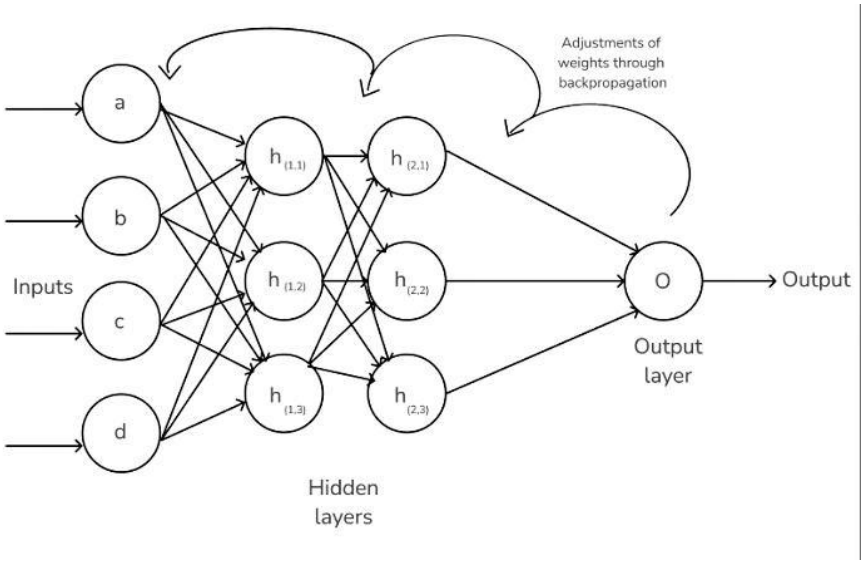


Fig 1.8 : Example of Backpropagation

1.8 : Advantages of using Backpropagation

Before getting into the intricate details of backpropagation in neural networks, it's essential to acknowledge the considerable significance this algorithm holds in the realm of machine learning and artificial intelligence.

Backpropagation is integral not only because it vastly enhances the performance of a neural network, but it also serves numerous other valuable purposes, making it a popular choice among practitioners and researchers alike. One of its most notable advantages is that it requires no prior knowledge of the structure or functionality of a neural network, which substantially lowers the barrier to entry for newcomers in the field. This accessibility means that individuals from diverse backgrounds—whether they are seasoned data scientists or enthusiastic amateurs—can implement backpropagation effectively without needing a comprehensive understanding of the underlying mathematical concepts.

Moreover, the algorithm is straightforward to program, as it primarily revolves around the inputs provided to the neural network. Unlike other complex algorithms that may involve multiple parameters and settings, backpropagation focuses chiefly on the relationships between the inputs and outputs, streamlining the coding process. This simplicity not only boosts the efficiency of implementation but also reduces the likelihood of errors during programming, making it an attractive option for those looking to quickly prototype or iterate on machine learning models.

In addition to its user-friendly nature, backpropagation significantly speeds up the learning process of a neural network. Unlike some traditional methods that require extensive feature engineering or upfront learning of the function's characteristics, backpropagation allows the model to adapt seamlessly based on its performance feedback. This means that, as the model receives data and makes predictions, it can continually refine its parameters to improve accuracy without unnecessary delays. The result is a more responsive and agile learning system that excels in environments where time and adaptability are critical.

Finally, the inherent flexibility of backpropagation cannot be overstated. Its simplicity allows for easy modifications and customizations, making it applicable to a wide range of scenarios, from simple classification tasks to complex regression problems. Whether one

is working on image recognition, natural language processing, or any other application of neural networks, backpropagation adapts well to various architectures and datasets. This versatility ensures that practitioners can leverage the algorithm effectively across different projects, enhancing their productivity and the quality of their outcomes. In essence, backpropagation stands as a vital cornerstone in the development and optimization of neural networks, offering a blend of accessibility, efficiency, and practicality that drives innovation in the field of artificial intelligence.

1.9 : Limitation of Backpropagation

Backpropagation is a widely used algorithm for training neural networks, but it's essential to recognize that it isn't a one-size-fits-all solution for every neural network scenario. One of the fundamental limitations of backpropagation is its heavy reliance on the quality of training data. In order to produce reliable and accurate models, researchers and practitioners must ensure that the data fed into the algorithm is not only abundant but also high-quality. If the training data is full of inconsistencies or noise, the model may learn incorrect patterns, leading to suboptimal performance. This reliance on clean, well-curated data cannot be overstated, as it directly impacts the model's ability to generalize to new data, which is a critical goal in machine learning.

Moreover, training a backpropagation model can be a time-consuming endeavor. Depending on the complexity of the neural network architecture and the amount of data involved, training times can range from minutes to days or even longer. This requirement for extended computational time can be a significant drawback, especially when quick iterative testing and refining are necessary for projects with tight deadlines.

Another important consideration is that backpropagation inherently follows a matrix-based approach, which can sometimes complicate implementation. Calculating gradients through layers involves matrix multiplications that can introduce numerical instability or performance bottlenecks if not handled carefully. This can lead to additional challenges in model optimization and tuning, requiring practitioners to have a good grasp of linear algebra principles to effectively navigate these issues.

Despite these limitations, backpropagation remains a powerful and effective method for training neural networks. Its ability to adjust weights based on the error of the predicted output compared to the actual output allows for continuous learning and refinement of model performance. Furthermore, the widespread adoption of backpropagation has led to significant advancements in neural network research and development, making it a cornerstone of modern machine learning practice.

As we delve deeper into the mechanics and nuances of backpropagation, we will explore not just its operational intricacies but also its implications for different architectures and applications in neural networks. Understanding these factors will help us leverage backpropagation effectively, allowing for the development of robust models capable of tackling complex real-world problems. Whether you're implementing a simple feedforward network or a more advanced structure like a convolutional neural network, having a comprehensive understanding of backpropagation will enhance your capability to test, refine, and ultimately succeed in your deep learning projects. As we proceed, we will dissect various techniques, best practices, and even alternative strategies that complement backpropagation, ensuring a well-rounded approach to neural network training.

1.10 : How to set Components for Backpropagation in Neutral Networks

In this project, we are focused on developing and training a deep neural network that effectively learns the XOR functionality using two inputs and three hidden units. Our training set, represented as a truth table, clearly illustrates the desired output for the combinations of inputs X1 and X2. Specifically, for the combinations where both inputs are 0, the output (Y) should reflect 0; when one input is 1 and the other is 0, the output should be 1; conversely, when both inputs are 1, the output should revert to 0. Given this structure, we will implement an identity activation function defined as f(a) = a, allowing us to maintain the values during the activation process without alteration.

To facilitate the transformation of inputs into outputs within our neural network, we will also adopt a hypothesis function of the form h(X) = W0.X0 + W1.X1 + W2.X2. This model outlines how weights and inputs combine to produce the output before applying the activation function. Alternatively, we could express this with a summation notation, h(X) = sigma(W.X) for all weights W and inputs X, encapsulating the essence of our computation succinctly. Crucially, we will employ the usual cost function associated with logistic regression as our loss function, which, despite its complexity, is integral for evaluating our model's performance.

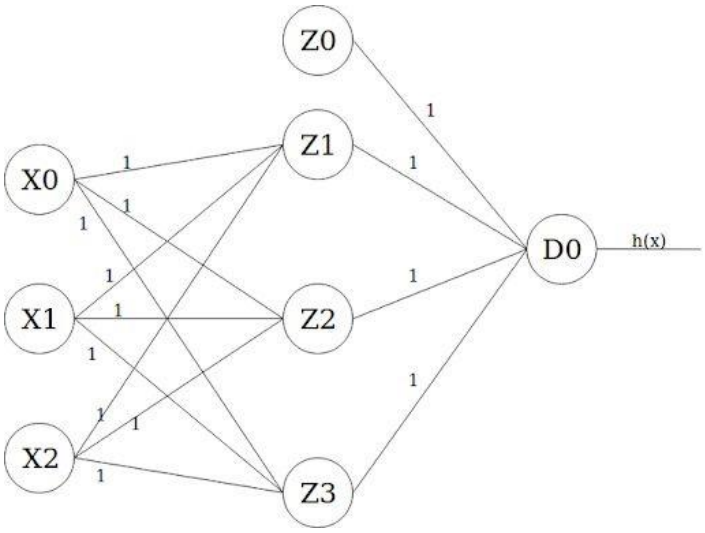
As we progress with training, we'll utilize the batch gradient descent optimization technique, which will guide us in adjusting the weights in a manner that minimizes our loss. This iterative process will unfold in discrete steps, each one informed by the gradients of our loss function relative to the weights, helping us hone in on the optimal set of weights. To begin with, we will establish a learning rate of 0.1, which strikes a balance between the speed of convergence and stability during training. To ensure that our model begins with a consistent foundation, all weights will be initialized to one, creating a baseline from which we can dynamically adjust through our training epochs.

Throughout this process, we will meticulously monitor the changes in the loss function as we update the weights based on the inputs. By keeping our focus on effectively learning the XOR

function, we anticipate that our neural network will demonstrate the ability to generalize well from the training set. To achieve this, we will likely run multiple iterations, continuously refining our approach until we see a marked improvement in accuracy. With perseverance and careful tuning of our parameters, we aim to have our model successfully replicating the intricacies of the XOR operation, effectively showcasing the power and capability of deep learning methodologies. As we proceed, collaboration and discussion within our team will ensure that we not only reach our training goals but also understand the underlying mechanisms that drive the learning processes of our neural network.

1.11 Building a Neural Networks

Let's finally draw a diagram of our long-awaited neural net. It should look something like this:



The leftmost layer is the input layer, which takes X0 as the bias term of value one, and X1 and X2 as input features. The layer in the middle is the first hidden layer, which also takes a bias term Z0 value of one. Finally, the output layer has only one output unit D0 whose activation value is the actual output of the model (i.e. h(x).)

1.12 How Forward Propagation Works

As we proceed with the neural network's forward propagation, it's crucial to delve deeper into how information moves through the layers, specifically focusing on the calculations conducted at each individual node or unit. This process unfolds in two distinct yet integral steps at every active node, enabling the transition of data from one layer to the next.

First and foremost, we compute the weighted sum of the inputs for a particular unit using the h(x) function, as defined in the previous stages of our discussion. This involves our weights, represented as W0, W1, and W2, multiplied by their corresponding inputs, X0, X1, and X2. For instance, in Unit Z1, this calculation results in h(x) = W0.X0 + W1.X1 + W2.X2, which simplifies to 1 * 1 + 1 * 0 + 1 * 0, yielding a sum of 1, noted as 'a.'

Once we have derived this value, we transition to the next step: applying the activation function. In our case, we're utilizing a linear activation function where f(a) = a, meaning that the output of our activation function will directly mirror the weighted sum we computed in the first step. Hence, we find that for Unit Z1, z = f(a) = f(1) = 1. This same calculation holds true for other nodes in the network that are not isolated.

Specifically, the other units follow suit in their computation. For Unit Z2, the weighted input also remains the same, yielding another output of 1, and similarly for Unit Z3, we observe another consistent output of 1 upon applying the activation function. These operations reflect the collaborative nature of the input features, where the nodes that are actively connected contribute to the overall data flow.

Moving further, we analyze Unit D0, which aggregates inputs from the preceding units, Z0, Z1, Z2, and Z3. The h(x) for this unit reveals a more complex summation: h(x) = W0.Z0 + W1.Z1 + W2.Z2 + W3.Z3 translates into 1 * 1 + 1 * 1 + 1 * 1 + 1 * 1, yielding a total of 4 for 'a.' The subsequent application of our linear activation function results in z = f(a) = f(4) = 4, thereby defining the output of our network for this iteration.

At this juncture, it's pertinent to note that the value of z for the final unit, D0, signifies the overall prediction of our model based on the input features provided. In our particular case, the model has determined an output of 4 for the input set {0, 0}. Following this, the next logical step is to calculate the loss or cost of the current iteration, which gives insight into the model's performance.

The loss calculation is straightforward, defined by the formula: Loss = actual_y – predicted_y. Here, our actual_y value is derived from the training set, which indicates the expected outcome for this specific input. Since our model predicted a value of 4, we plug this into our formula, yielding a loss calculation of -4, as the actual_y is 0. This signifies that our model significantly deviated from the expected result during this training session, indicating areas for potential improvement and adjustment in future iterations.

In summary, this detailed walk-through reveals the intricate operations occurring at each unit in the neural network. It highlights not only the mechanical calculations leading to the outputs but also underscores the importance of these processes in understanding how our model learns and makes predictions. As we move forward, we can refine our approach based on the loss calculated, leading us closer to optimizing performance and achieving our predictive goals.

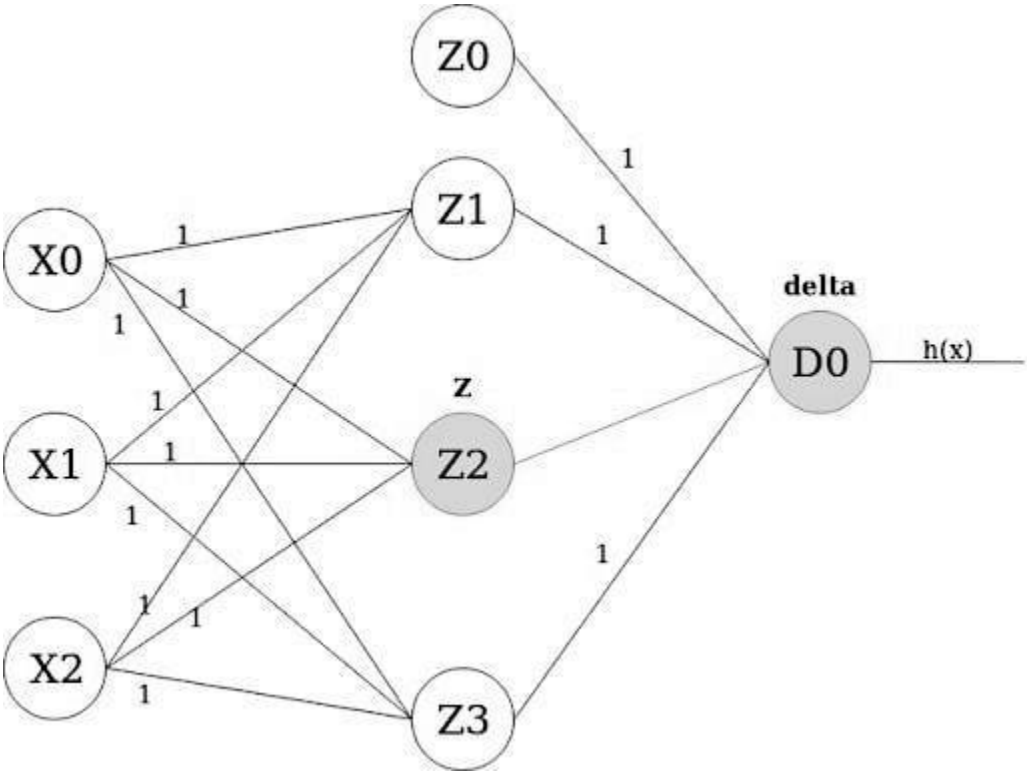
1.13 When to use Backpropagation

In our current project, we find ourselves facing a significant challenge: our predictive model is yielding inaccurate results, specifically giving us an output of four when we expected one. This discrepancy is primarily due to the fact that the model's weights have yet to be properly tuned, as they are all currently set to one. As a result, we are witnessing a loss of -4, indicating that there is

considerable room for improvement. The process of backpropagation will be instrumental in addressing this issue, as it involves propagating the loss backward to adjust the weights accordingly. Using gradient descent as our optimization function, we aim to calibrate these weights to achieve a smaller loss in subsequent iterations. To kickstart this process, we utilize the feeding forward functions described as f(a) = a. For the feeding backward phase, we will employ the partial derivatives of these functions. Thankfully, we do not need to delve into the complexities of the equations to derive these derivatives; the relevant simplifications tell us that f'(a) = 1. Furthermore, we can express the change in our cost function as J'(w) = Z . delta, where Z represents the value obtained from the activation functions during the feed-forward step, and delta corresponds to the loss experienced by the units in the layer. While it may seem overwhelming, I encourage you to take your time to digest this information thoroughly, as a solid understanding of each step is crucial before we progress further. Let's approach this with a mindset of curiosity and perseverance, keeping in mind that the journey of fine-tuning our model is just beginning.

1. 14 : Updating weights in Backpropagation

To effectively update the weights in our neural network, we will apply the batch gradient descent formula, which can be summarized as W := W – alpha * J'(W). In this equation, W represents the weight we're adjusting, alpha is our learning rate (0.1 in our example), and J'(W) is the partial derivative of the cost function relative to W. For our calculations, we'll leverage Andrew Ng's expression for the partial derivative: J'(W) = Z * delta.



Here, Z denotes the Z value derived through forward propagation, while delta corresponds to the loss at the unit on the opposite end of the weighted link. It's essential to ensure that all weights are updated without duplicating any updates within the same iteration. After naming the links in our neural net, we'll perform the weight updates consistently

. For instance, we have W10 := W10 – alpha * Z_X0 * delta_Z1, yielding results that will ultimately advance all weights to approximately 1.4. Although it may seem perplexing that all weights retain the same value post-update, reiterating this process across the entire training set will eventually lead to varied weights that accurately reflect each node's contribution to the overall loss. While the theory of machine learning, especially backpropagation, can be challenging, it's crucial to understand how effectively it operates in real-world scenarios by attributing loss responsibly across nodes and iteratively refining the model towards minimized cost.

1.14 Best Practices to Handle with Backpropagation

Backpropagation in a neural network is inherently designed for efficiency, yet there are several best practices that can enhance the performance of this critical algorithm.

One of the pivotal decisions involves selecting the appropriate training method. Opting for stochastic gradient descent can significantly accelerate the training process; however, it may require meticulous fine-tuning of the backpropagation algorithm, which can be time-consuming. Conversely, batch gradient descent is straightforward to implement, but it often results in a prolonged overall learning curve. Thus, while the stochastic approach is frequently favored for its speed, it is essential to choose a training method that aligns with your specific project needs and circumstances. By doing so, you will ensure that your backpropagation algorithm operates at its peak performance, optimizing the effectiveness of your neural network training process.

- Provide Plenty of data

  Feeding a backpropagation algorithm lots of data is key to reducing the amount and types of errors it produces during each iteration. The size of your data set can vary, depending on the learning rate of your algorithm. In general, though, it's better to include larger data sets since models can gain broader experiences and lessen their mistakes in the future.

- Clean all the data

  Backpropagation training is much smoother when the training data is of the highest quality, so clean your data before feeding it to your algorithm. This means normalizing the input values, which involves checking that the mean of the data is zero and the data set has a standard deviation of one. A backpropagation algorithm can then more easily analyze the data, leading to faster and more accurate results.

- Consider the impact of Learning Rate

  Deciding on the learning rate for training a backpropagation model depends on the size of the data set, the type of problem and other factors. That said, a higher learning rate can lead to faster results, but not the optimal performance. A lower learning rate produces slower results, but can lead to a better outcome in the end. You'll want to consider which learning rate best applies to your situation, so you don't under- or overshoot your desired outcome.

- Test the model with different Examples :

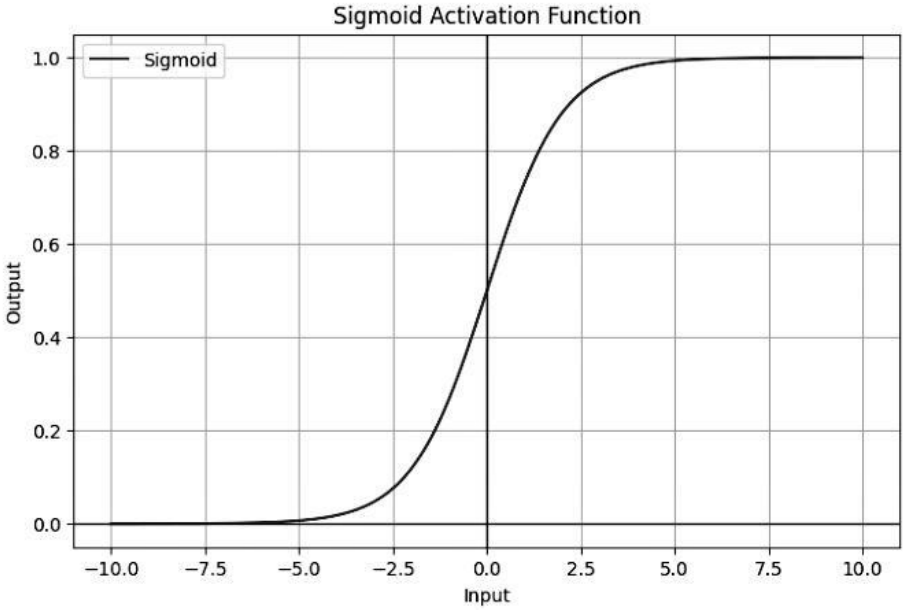
  To get a sense of how well a backpropagation model performs, it helps to test the algorithm with data not used during the training period. Compiling diverse data also exposes the model to different situations and tests how well it can adapt to a range of scenarios. You can then make more informed adjustments to enhance the algorithm's learning process.

1.15 Activation Functions

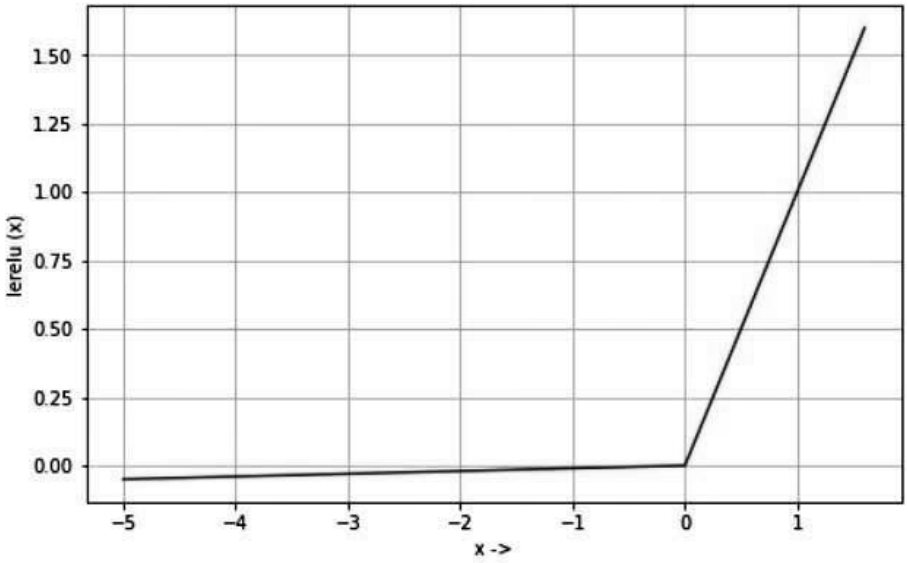
Activation functions play a crucial role in the functionality of artificial neural networks (ANNs) by introducing essential non-linearities that enable the model to learn intricate patterns within the data. Here, we categorize some common activation functions along with their unique benefits.

Sigmoid Function: This activation function is particularly useful for binary classification tasks, as it compresses outputs to a range between 0 and 1, making it ideal for determining outcomes such as “cat or not cat.”

ReLU (Rectified Linear Unit): A go-to activation function for hidden layers, ReLU outputs the

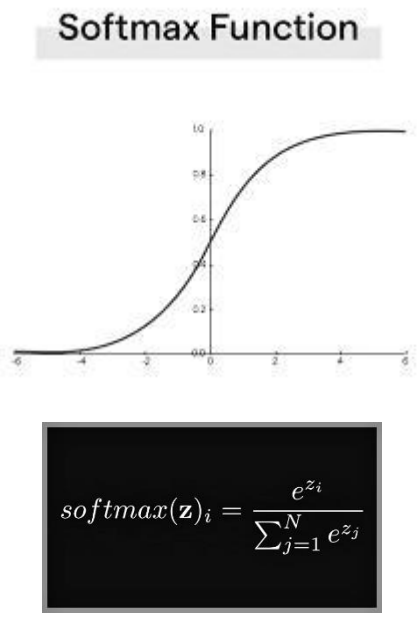


input directly when it is greater than zero, allowing the network to efficiently learn without the vanishing gradient problem that can stifle performance during training.



Tanh (Hyperbolic Tangent): Tanh is similar to the sigmoid function but outputs values within a range of -1 to 1. This broader range can be beneficial in hidden layers where a more diverse set of outputs is needed, helping the network model complex relationships.

Softmax: Primarily employed in the final layer of a multi-class classification network, the Softmax function transforms raw scores into interpretable probabilities, allowing the model to predict class membership more effectively.



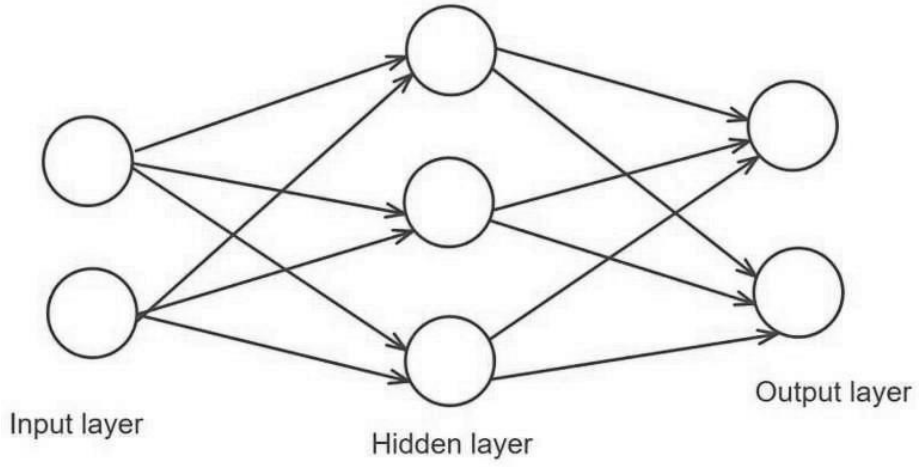
Leaky ReLU: As a modification of the ReLU function, Leaky ReLU addresses the issue of "dead neurons" by allowing a small, non-zero, negative gradient when the input is less than zero. This feature helps maintain the flow of information during training, leading to robust model performance.

Each activation function has its strengths and applicable scenarios, making them fundamental components in building effective neural network architectures.

1.16 Types of ANN (Artificial Neural Networks)

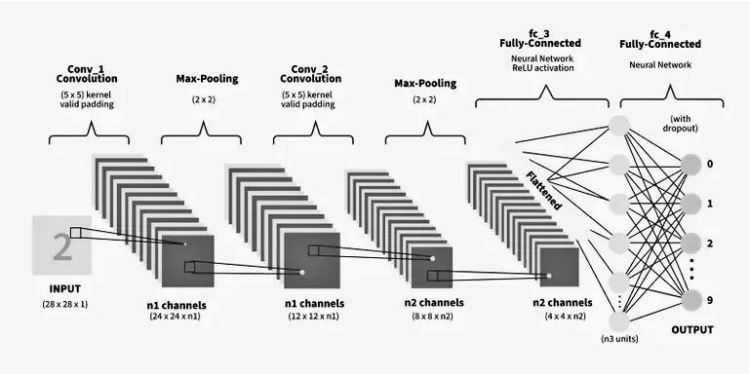
In the realm of artificial neural networks, we can categorize the various types based on their architecture and the types of data they effectively handle.

Feedforward Neural Networks (FNNs) : are foundational models used primarily for straightforward classification and regression tasks. Their structure is linear, with data flowing unidirectionally from input to output, which simplifies their implementation but limits their complexity.

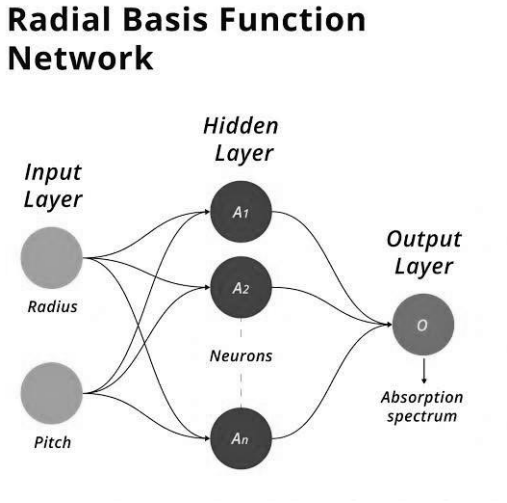


Convolutional Neural Networks (CNNs) : are tailored for grid-like data such as images and are particularly effective in image and speech recognition. The application of filters in convolutional layers allows these networks to capture significant features, making them powerful tools in computer vision.

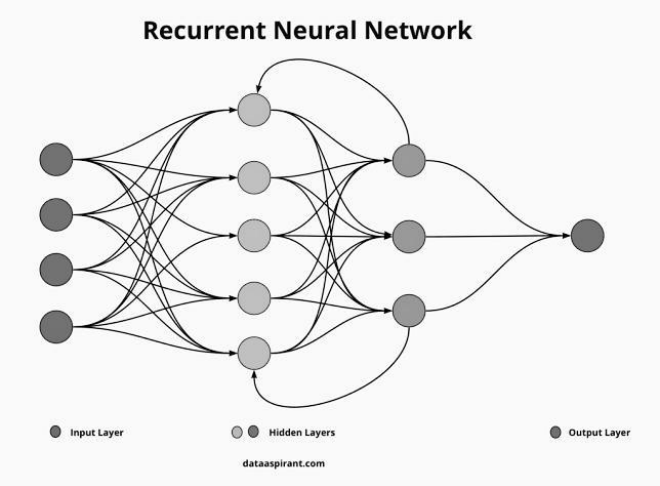
DL using Tensorflow



Radial Basis Function Networks (RBFNs) serve a unique purpose by modeling data with radial patterns. Their layered design facilitates classification and regression, especially in scenarios where the underlying data structure is circular or cylindrical.



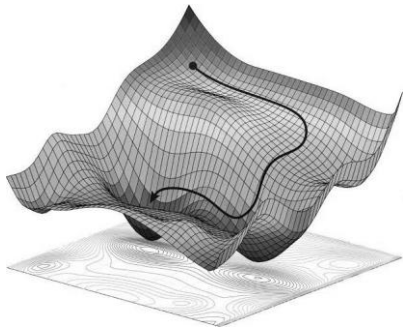
Recurrent Neural Networks (RNNs) introduce the ability to manage sequential data, thanks to their feedback loops. This capacity to retain information from previous time steps equips RNNs with a form of memory, enabling them to excel in applications involving time-series analysis and natural language processing. Each type of neural network offers distinct advantages tailored to specific tasks, making them invaluable in the arsenal of machine learning techniques.



1.17 Optimization Algorithms in Neural Networks Training

Introduction : In the realm of deep learning, one of the pivotal aspects of model training revolves around optimization algorithms, which play a vital role in refining the performance of neural networks. By utilizing these optimizers, we can adjust key parameters, such as weights and learning rates, to effectively minimize the loss function—a critical measure of how well the model is performing. The overarching goal is to navigate the complexities of loss minimization, thereby enhancing the model's predictive capabilities. This article delves into the various optimizers utilized in training, shedding light on how they contribute to achieving optimal performance. As we explore their mechanisms and applications, we uncover the importance of selecting the right optimization algorithm to tackle specific challenges, ensuring our neural networks evolve efficiently and effectively.

How does Optimizers work : In the complex world of neural networks, envision a hiker navigating down a treacherous mountain path—blindfolded. The journey may seem daunting, as it's nearly impossible to determine the best route initially. All she can truly sense is whether she's making progress by moving downward or losing ground as she ascends. In the realm of machine learning, this metaphor holds true; the precise weights of your model are not immediately clear. However, by leveraging the loss function as a guide, akin to the hiker discerning her downward progress, incremental improvements can be realized through careful adjustments. The pivotal role of optimizers becomes evident here—they serve as the navigational tools that adjust your model's weights and learning rates, propelling you toward reduced losses and enhanced accuracy. In recent years, a wealth of research has expanded our understanding of various optimization algorithms, each presenting its own set of pros and cons. By delving into this comprehensive article, you will gain valuable insights into the workings, advantages, and drawbacks of these algorithms, empowering you on your journey toward mastering neural network optimization. Don't miss the opportunity to navigate this crucial aspect of machine learning and elevate your projects to new heights.

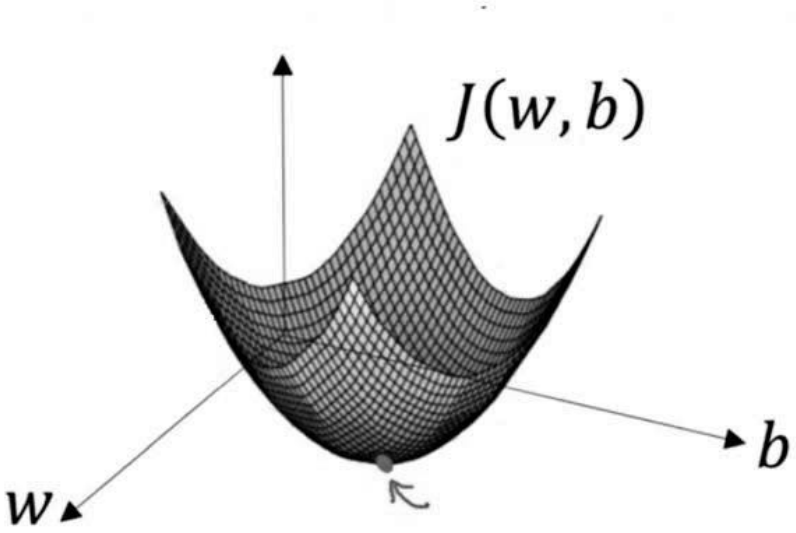


.

Fig 1.9 : Example of Optimizer (MATLAB plot)

- **Gradient Descent :** Gradient descent is an essential optimization algorithm widely employed in the training of machine learning models and neural networks, forming a backbone of modern AI. At its core, this iterative process seeks to minimize the discrepancies between predicted outcomes and actual results, transforming raw data into valuable insights. The training data is pivotal, allowing models to learn and evolve over time, while the cost function serves as a crucial indicator of model accuracy. Each iteration adjusts parameters based on the cost function's evaluations—continuing this cycle until the function approaches zero, signifying a minimal error rate. This relentless pursuit of optimization enables machine learning models to harness their full potential, turning them into formidable assets in fields as diverse as artificial intelligence, data science, and beyond. With the right fine-tuning and dedication, these models can unlock innovative solutions and drive substantial advancements in technology, paving the way for a future shaped by intelligent systems. As we explore uncharted territories in

computer science, the significance of gradient descent and its applications cannot be overstated, making it a vital concept for anyone looking to delve into the world of machine learning and AI.



How it Works : Before we dive into gradient descent, it may help to review some concepts from linear regression. You may recall the following formula for the slope of a line, which is y = mx + b, where m represents the slope and b is the intercept on the y-axis.

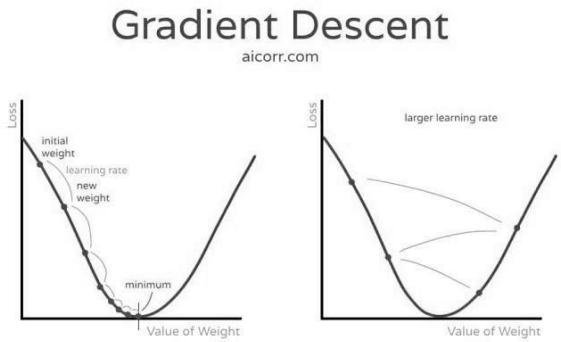
You may also recall plotting a scatterplot in statistics and finding the line of best fit, which required calculating the error between the actual output and the predicted output (y-hat) using the mean squared error formula. The gradient descent algorithm behaves similarly, but it is based on a convex function.

The starting point is just an arbitrary point for us to evaluate the performance. From that starting point, we will find the derivative (or slope), and from there, we can use a tangent line to observe the steepness of the slope. The slope will inform the updates to the parameters—i.e. the weights and bias. The slope at the starting point will be steeper, but as new parameters are generated, the steepness should gradually reduce until it reaches the lowest point on the curve, known as the point of convergence.

Similar to finding the line of best fit in linear regression, the goal of gradient descent is to minimize the cost function, or the error between predicted and actual y. In order to do this, it requires two data points—a direction and a learning rate. These factors determine

the partial derivative calculations of future iterations, allowing it to gradually arrive at the local or global minimum (i.e. point of convergence).

Learning rate (also referred to as step size or the alpha) is the size of the steps that are taken to reach the minimum. This is typically a small value, and it is evaluated and updated based on the behavior of the cost function. High learning rates result in larger steps but risks overshooting the minimum. Conversely, a low learning rate has small step sizes. While it has the advantage of more precision, the number of iterations compromises overall efficiency as this takes more time and computations to reach the minimum.The cost (or loss) function measures the difference, or error, between actual y and predicted y at its current position. This improves the machine learning model's efficacy by providing feedback to the model so that it can adjust the parameters to minimize the error and find the local or global minimum. It continuously iterates, moving along the direction of steepest descent (or the negative gradient) until the cost function is close to or at zero. At this point, the model will stop learning. Additionally, while the terms, cost function and loss function, are considered synonymous, there is a slight difference between them. It's worth noting that a loss function refers to the error of one training example, while a cost function calculates the average error across an entire training set.



- **Types of gradient descent**

There are three types of gradient descent learning algorithms: batch gradient descent, stochastic gradient descent and mini-batch gradient descent.

Batch gradient descent

Batch gradient descent sums the error for each point in a training set, updating the model only after all training examples have been evaluated. This process referred to as a training epoch.While this batching provides computation efficiency, it can still have a long processing time for large training datasets as it still needs to store all of the data into memory. Batch

gradient descent also usually produces a stable error gradient and convergence, but sometimes that convergence point isn't the most ideal, finding the local minimum versus the global one.

Stochastic gradient descent

Stochastic gradient descent (SGD) runs a training epoch for each example within the dataset and it updates each training example's parameters one at a time. Since you only need to hold one training example, they are easier to store in memory. While these frequent updates can offer more detail and speed, it can result in losses in computational efficiency when compared to batch gradient descent. Its frequent updates can result in noisy gradients, but this can also be helpful in escaping the local minimum and finding the global one.

Mini-batch gradient descent

Mini-batch gradient descent combines concepts from both batch gradient descent and stochastic gradient descent. It splits the training dataset into small batch sizes and performs updates on each of those batches. This approach strikes a balance between the computational efficiency of batch gradient descent and the speed of stochastic gradient descent.

Challenges with gradient descent can indeed present a significant hurdle when tackling optimization problems, particularly those that are not convex. While gradient descent remains a go-to method due to its straightforwardness and ease of implementation, the complexities of nonconvex landscapes can impede its effectiveness in finding the global minimum—the point where the model achieves optimal performance. In such scenarios, the risk of the algorithm getting stuck in local minima or sliding down saddle points looms large. Local minima can often masquerade as global minima since they create a situation where the slope of the cost function is minimal, causing the algorithm to erroneously halt its search for improvement. This results in a model that does not reach its full potential. Saddle points add another layer of difficulty; while the slope may be flat at one point, the surrounding terrain can lead to vastly different outcomes on opposing sides of the saddle—one direction could lead to local maximums while the other descends into a local minimum. This geometric nuance can deceive standard gradient descent practices. Notably, introducing the concept of noisy gradients can provide a path forward, as these unpredictable fluctuations have the potential to destabilize the learning trajectory enough to allow the algorithm to leap out of the confines of these local traps. The ability of these noisy gradients to assist gradient descent in navigating through challenging landscapes highlights a key innovation in optimization techniques, allowing for greater flexibility and potentially leading to more successful outcomes in complex problem spaces. By acknowledging and addressing these

challenges, practitioners can refine their approaches to optimization, fostering improved model performance and more reliable results across diverse applications.

In the realm of deep learning, particularly with recurrent neural networks (RNNs), practitioners often encounter two significant issues that can profoundly affect model performance during training: vanishing and exploding gradients. The phenomenon of vanishing gradients is particularly troublesome, as it leads to gradients diminishing to insignificant values—essentially approaching zero—as they are propagated backward through the layers of the network during backpropagation. This slow learning process affects the earlier layers disproportionally, rendering them almost stagnant while later layers continue to adjust their weights, causing a detrimental imbalance in the learning dynamics of the model. On the flip side, exploding gradients present their own set of challenges—here, the gradients can grow excessively large, resulting in weight parameters that diverge to infinity and often trigger numerical instability, with values represented as NaN. Such instability compromises the reliability of the learning process and can bring training to a halt. To mitigate the risks associated with exploding gradients, employing dimensionality reduction techniques can be an effective strategy; by simplifying the model's complexity, these techniques help create a more stable training environment, ensuring that the learning process remains robust and conducive to effective neural network training. Addressing both of these gradient-related issues is crucial for developing deep learning models that perform well across various tasks, highlighting the importance of understanding and implementing appropriate solutions in neural network training.

DL using Tensorflow