

The Theory Of Deep Learning

By

Aadil Hussain

Table of Contents

Chapter 1: Basics Fundamentals of Deep Learning - Page 2

- 1.1: Overview of Deep Learning - Page 3
- 1.2: History of Deep Learning - Page 5
 - 1.2.1: Origin of Neural Networks - The Building Blocks - Page 6
- 1.3: The Perceptron - Page 7
- 1.5: The Neural Networks - Page 12
- 1.6: Training of Neural Networks - Page 14
- 1.7: Backpropagation – Neural Networks Training - Page 15
- 1.8: Advantages of Using Backpropagation - Page 16
- 1.9: Limitation of Backpropagation - Page 17
- 1.10: How to Set Components for Backpropagation in Neural Networks - Page 17
- 1.11: Building a Neural Networks - Page 18
- 1.12: How Forward Propagation Works - Page 19
- 1.13: When to Use Backpropagation - Page 20
- 1.14: Updating Weights in Backpropagation - Page 21
- 1.15: Best Practices to Handle with Backpropagation - Page 22
- 1.16: Activation Functions - Page 23
- 1.17: Types of ANN (Artificial Neural Networks) - Page 25

Chapter 2: The Activation Functions - Page 31

- 2.1: Overview - Page 32
- 2.2: Binary Step Function - Page 33
- 2.3: Exploring the Linear Function - Page 34
- 2.4: Sigmoid Function - Page 35
- 2.5: Tanh Activation Functions - Page 39
- 2.5: ReLU Activation Functions - Page 40
- 2.6: Leaky ReLU - Page 41
- 2.9: Swish - Page 45
- 2.10: Softmax - Page 46

Chapter 3: The Loss Functions - Page 48

- 3.3: MAE (Mean Absolute Error) / L1 Error - Page 52
- 3.4: Huber Loss / Smooth Mean Squared Error - Page 54

Acknowledgements

I would like to express my deepest gratitude to all those who have supported me throughout the journey of writing this book, "The Theory of Deep Learning." First and foremost, I thank my family for their unwavering encouragement and patience during the long hours I spent researching and writing.

Special thanks go to my mentors and colleagues in the field of artificial intelligence, whose insights and discussions have greatly shaped my understanding of deep learning concepts. I am particularly indebted to the pioneers like Warren McCulloch, Walter Pitts, Frank Rosenblatt, Yann LeCun, and many others whose groundbreaking work laid the foundation for this field.

I also appreciate the open-source communities and online resources that provided invaluable data and tools for my research. Without the contributions from platforms like GitHub, arXiv, and educational sites, this book would not have been possible.

Finally, a heartfelt thank you to the readers who engage with this material. Your curiosity and pursuit of knowledge drive the advancement of technology. Any errors or omissions are my own, and I welcome feedback for future improvements.

Aadil Hussain
September 2025

Preface

Deep learning has revolutionized the landscape of artificial intelligence, enabling machines to perform tasks that were once thought to be exclusively human. This book, "The Theory of Deep Learning," aims to provide a comprehensive overview of the fundamental principles, historical evolution, and key components that underpin this transformative technology.

In the following chapters, we delve into the basics of deep learning, explore activation functions in detail, and examine loss functions critical for model optimization. Whether you are a student, researcher, or practitioner, this text is designed to build a solid theoretical foundation while highlighting practical implications.

I hope this book serves as a valuable resource in your journey through the exciting world of deep learning.

Aadil Hussain

Chapter – 1

Basics Fundamental of Deep Learning

1.1 : Overview of Deep Learning

Deep Learning is a subset of AI, fields 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.

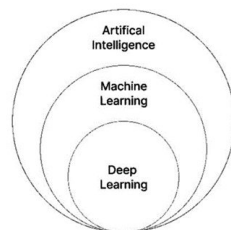
The main difference between deep learning and machine learning is the design of the underlying neural network. Traditional machine learning models use simple neural networks with one or two computational layers. In contrast, deep learning models use three or more layers, often including hundreds or thousands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



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 cumulative 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

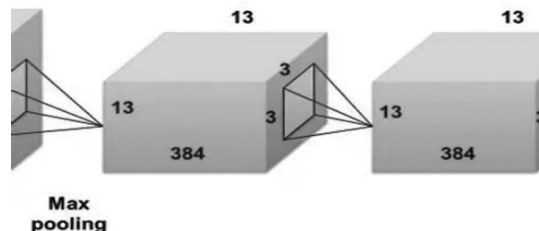
Before 1980, there are two categories of artificial neural networks (ANN): feedforward neural networks (FNN), also known as *multilayer perceptrons* (MLP), and recurrent neural networks (RNN). Unlike FNNs, which lack cycles in their connectivity structure, RNNs incorporate them. In the 1920s, Wilhelm 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.



Basic Architecture of AlexNet

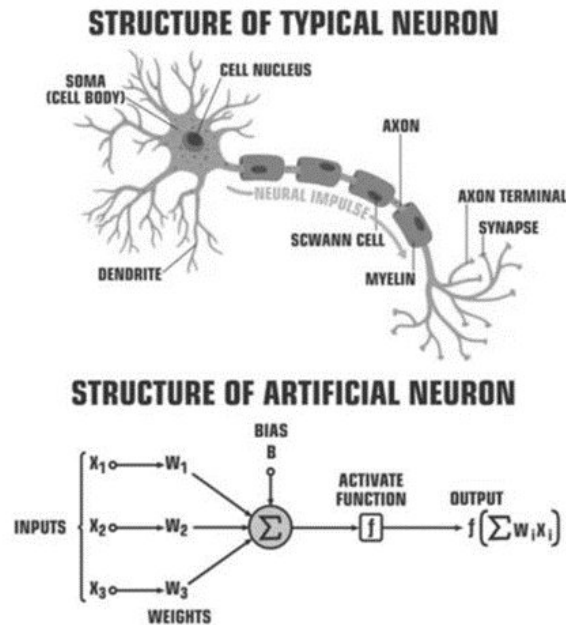
Also in 2012, Google Brain released the results of an unusual project known as The Cat Experiment. The free-spirited project explored the difficulties of “unsupervised learning.” Deep learning uses “supervised learning,” meaning the convolutional neural net is trained using labeled data (think images 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).

1.21 : 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 Mcc-neuron. That laid the foundation for Deep Learning.



Representation of Biological Neuron and Artificial Neuron

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.

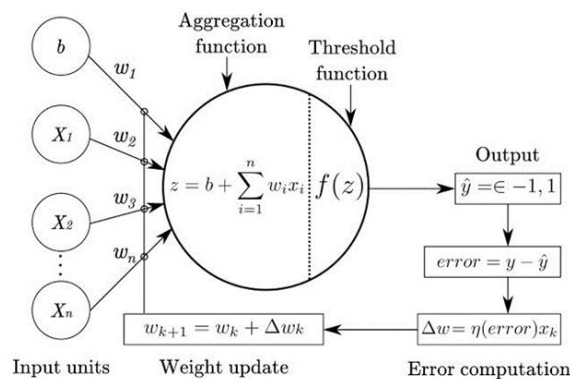
1.3 : The Perceptron

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.

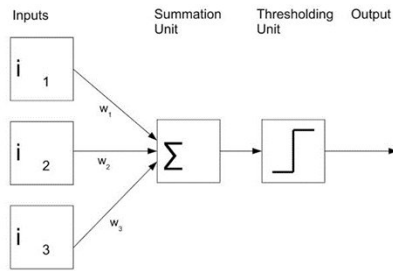
Definition : In the modern sense, the perceptron is an algorithm for learning a binary classifier called a threshold function:



Representation of Perceptron

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 (x_1), the weather (x_2), and perhaps the ticket price (x_3), you would apply their corresponding weights. So, if you were to evaluate x_1 (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.



Representation of a Perceptron

A perceptron, often hailed as an essential artificial 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

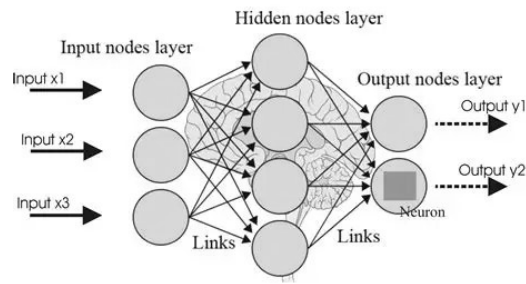
1. Input Layer:

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.

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.



Classification of layers in AN

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.

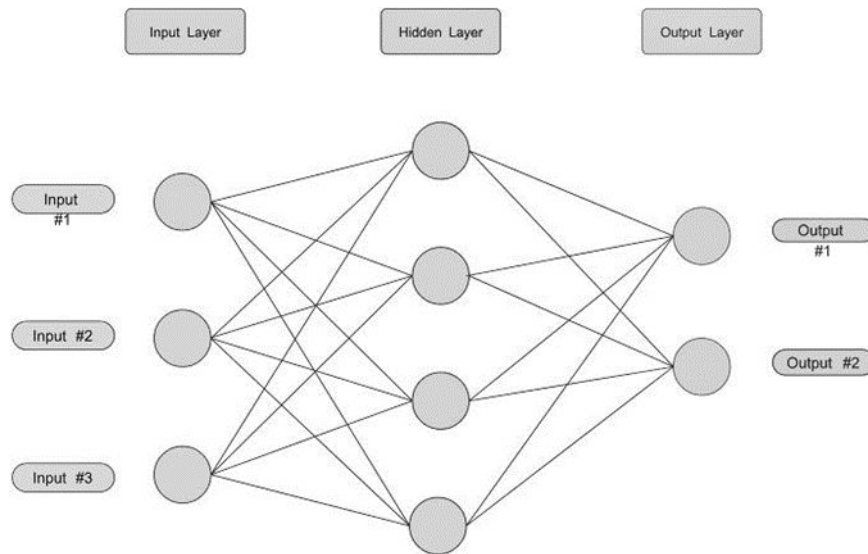
This overview illustrates the essential components of a perceptron, forming the backbone of many foundational machine learning models. Understanding these elements is crucial for anyone looking to delve into artificial neural networks and their applications.

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

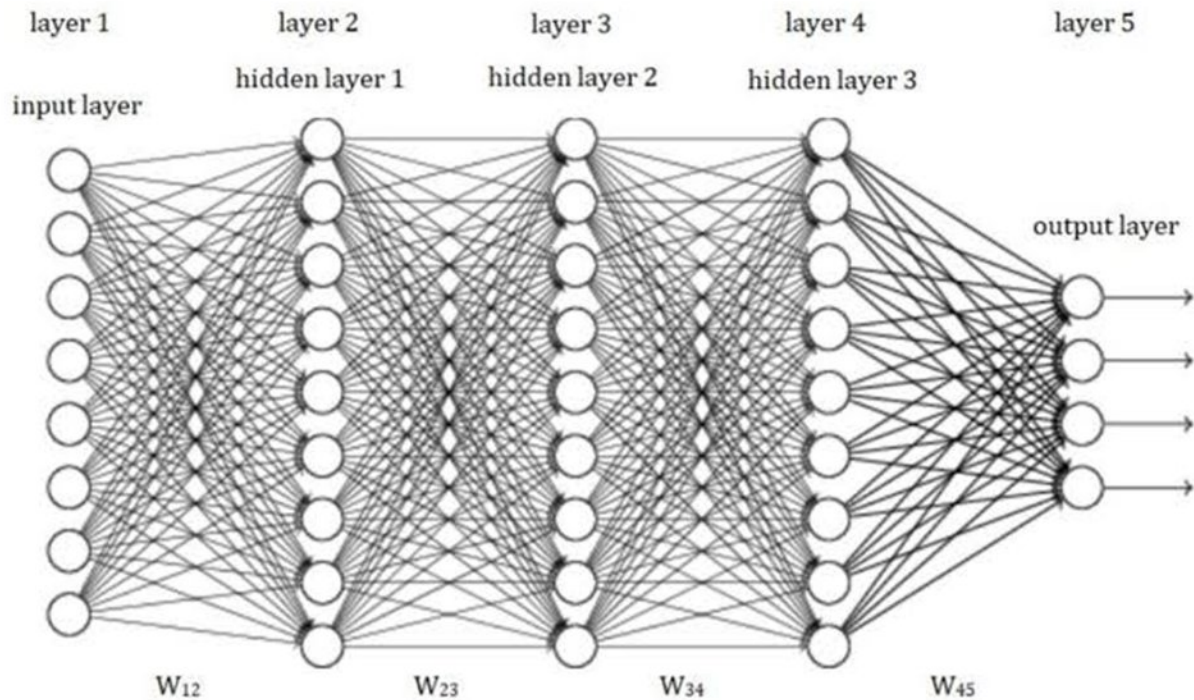
neurons — which represent real numbers. These inputs are then processed through a mathematical construct known as an activation function, which introduces a non-linear transformation. This non-linearity is critical, as it allows the network to capture complex patterns within the data.



Classification ANN layers in three main categories

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.



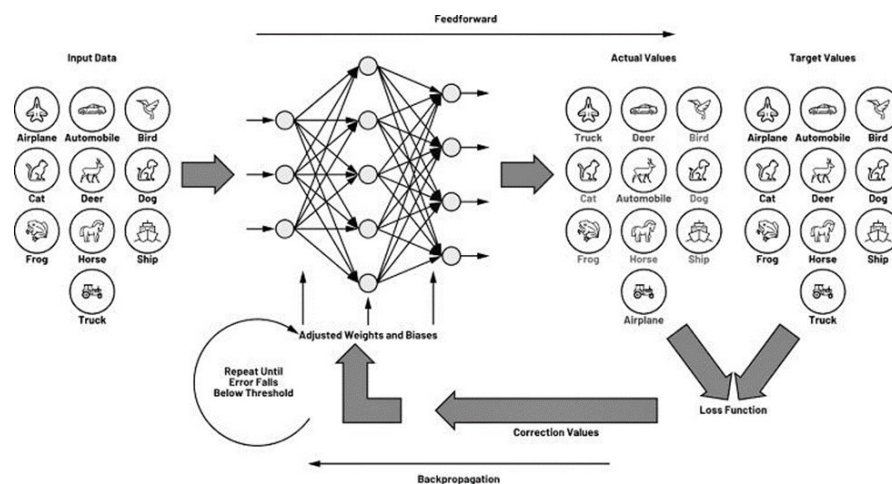
Representation of Deep Neural Networks

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.

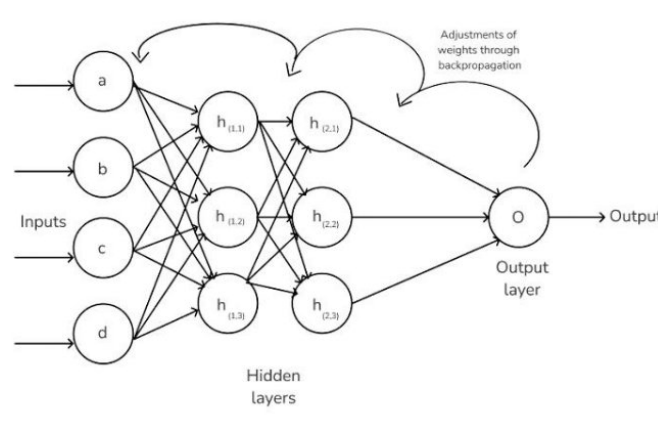


1.7 : Backpropagation – Neural Networks Training

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.



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 Neural 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.

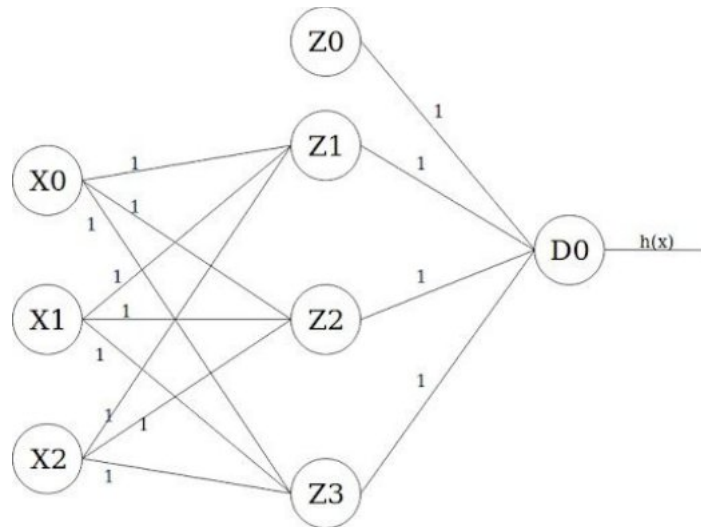
To facilitate the transformation of inputs into outputs within our neural network, we will also adopt a hypothesis function of the form $h(X) = W_0.X_0 + W_1.X_1 + W_2.X_2$. This model outlines how weights and inputs combine to produce the output before applying the activation function. Alternatively, we could express this with a summation notation, $h(X) = \text{sigma}(W.X)$ 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 X_0 as the bias term of value one, and X_1 and X_2 as input features. The layer in the middle is the first hidden layer, which also takes a bias term Z_0 value of one. Finally, the output layer has only one output unit D_0 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 W_0 , W_1 , and W_2 , multiplied by their corresponding inputs, X_0 , X_1 , and X_2 . For instance, in Unit Z_1 , this calculation results in $h(x) = W_0.X_0 + W_1.X_1 + W_2.X_2$, 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 Z_1 , $z = f(a) = f(1) = 1$. 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 Z_2 , the weighted input also remains the same, yielding another output of 1, and similarly for Unit Z_3 , 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 D_0 , which aggregates inputs from the preceding units, Z_0 , Z_1 , Z_2 , and Z_3 . The $h(x)$ for this unit reveals a more complex summation: $h(x) = W_0.Z_0 + W_1.Z_1 + W_2.Z_2 + W_3.Z_3$ translates into $1 * 1 + 1 * 1 + 1 * 1 + 1 * 1$.

$1 + 1 * 1 + 1 * 1$, 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: $\text{Loss} = \text{actual_y} - \text{predicted_y}$. 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 $J'(w) = Z \cdot \text{delta}$, 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.

Back-propagation learning rule

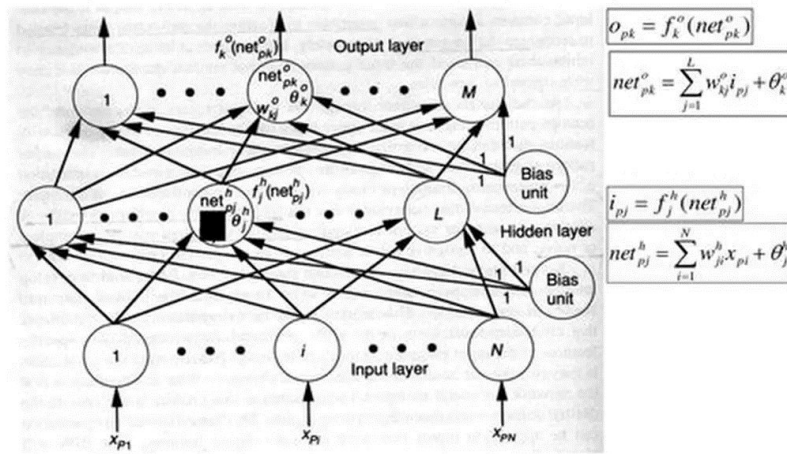
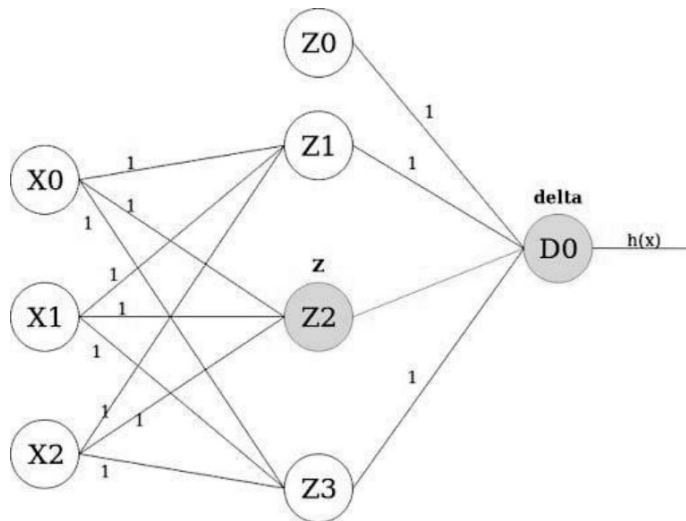


Fig 1.7 : Backpropagation Learning Rule Representation

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, α 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: $J'(W) = Z * \text{delta}$.



Here, Z denotes the Z value derived through forward propagation, while δ 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 $W_{10} := W_{10} - \alpha * Z_{X0} * \delta_{Z1}$, 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.

1.15 : 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 :

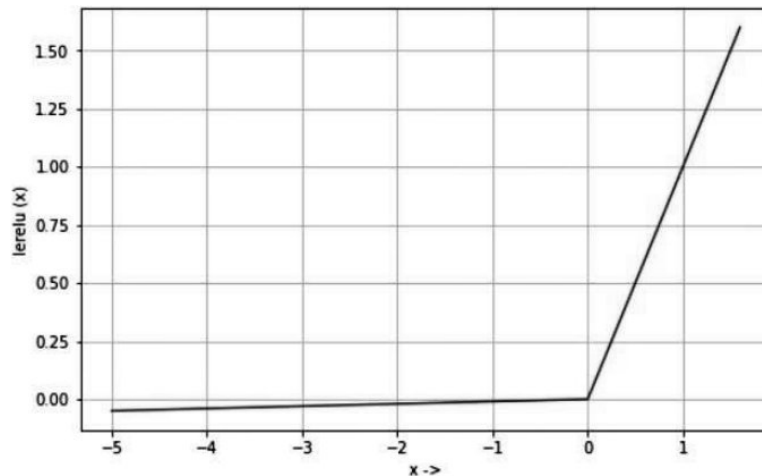
To get a sense of how well a backpropagation model performs, it helps to test the algorithm with data not used during the training period.

1.16 : 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.

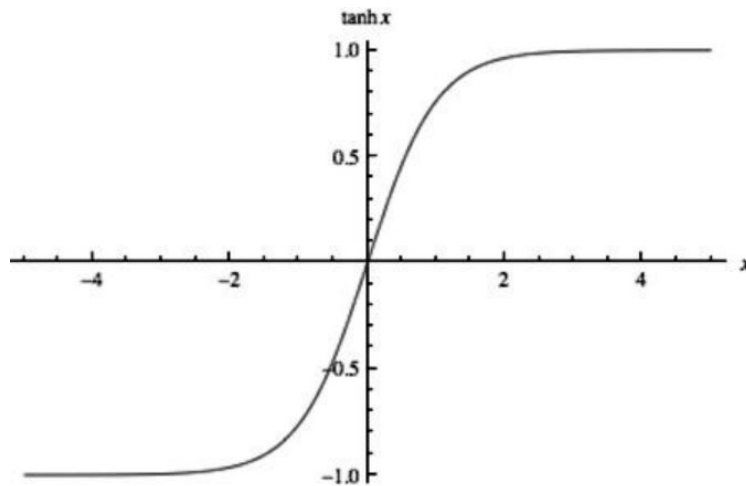
Sigmoid Function: This activation function is particularly useful for binary classification tasks, as it compresses outputs to a range between 0 and 1, making it ideal for determining outcomes such as “cat or not cat.”

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.



Representation of ReLU Activation Functions

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.



Representation of Tanh Hyperbolic AF (Activation Functions)

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.

Softmax Function

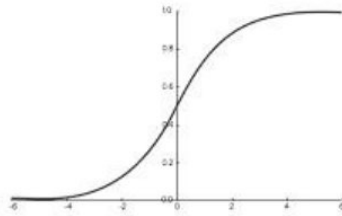


Fig 1.10 : Representation of Sigmoid Curve

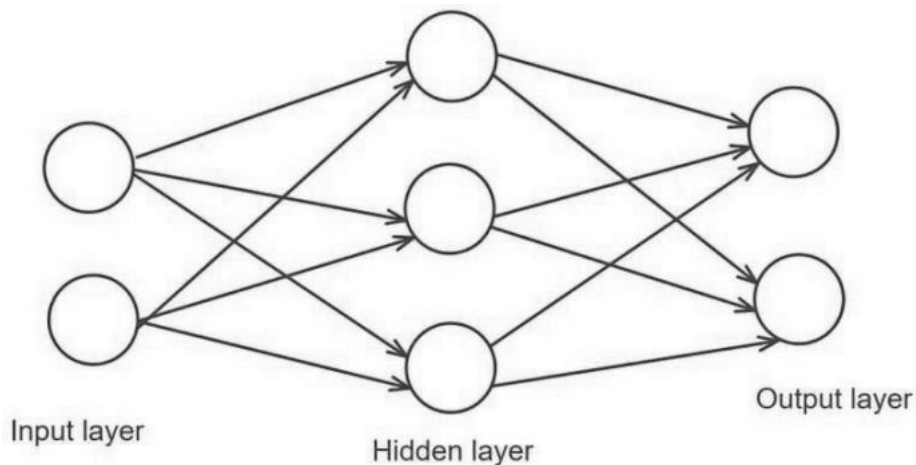
Leaky ReLU: As a modification of the ReLU function, Leaky ReLU addresses the issue of “dead neurons” by allowing a small, non-zero, negative gradient when the input is less than zero. This feature helps maintain the flow of information during training, leading to robust model performance.

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

1.17 : 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.



Representation of FFNN (Feedforward Neural Network)

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.

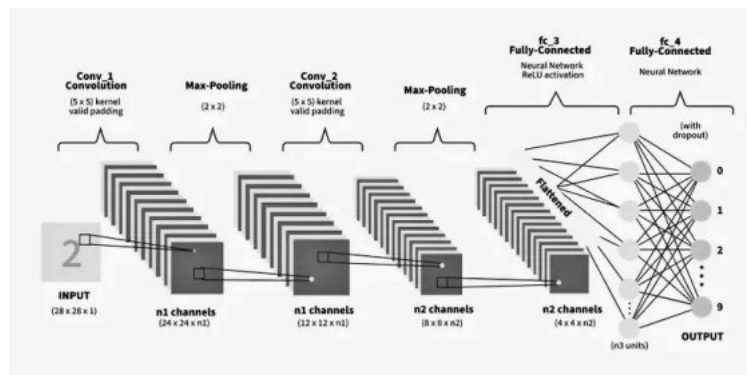
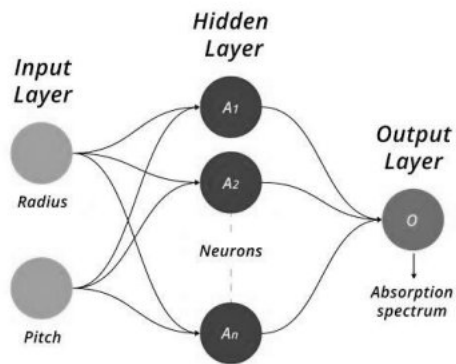


Fig 1.12 : Convolutional Neural Network Architechture

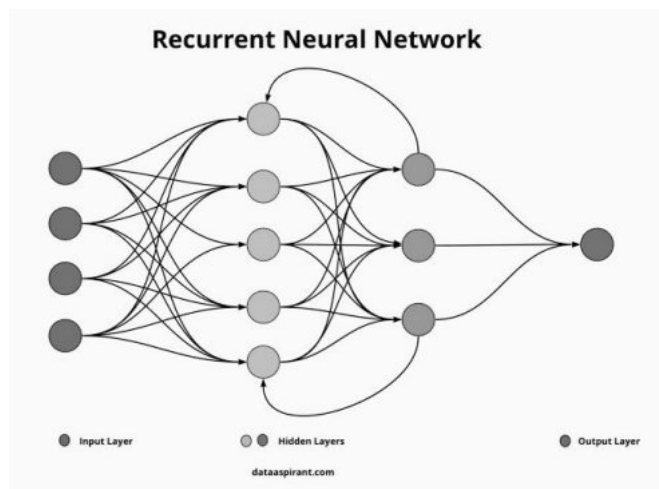
Radial Basis Function Networks (RBFNs) serve a unique purpose by modeling data with radial patterns. Their layered design facilitates classification and regression, especially in scenarios where the underlying data structure is circular or cylindrical.

Radial Basis Function Network



Radial Basis Function Network ARC

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.

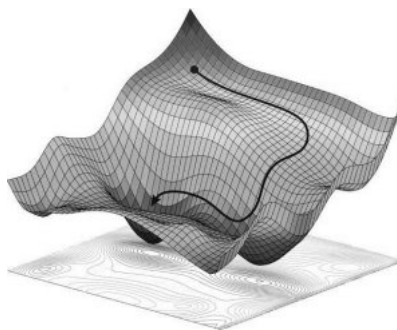


Recurrent Neural Networks Representation

1.18: Optimization Algorithms in Neural Networks Training

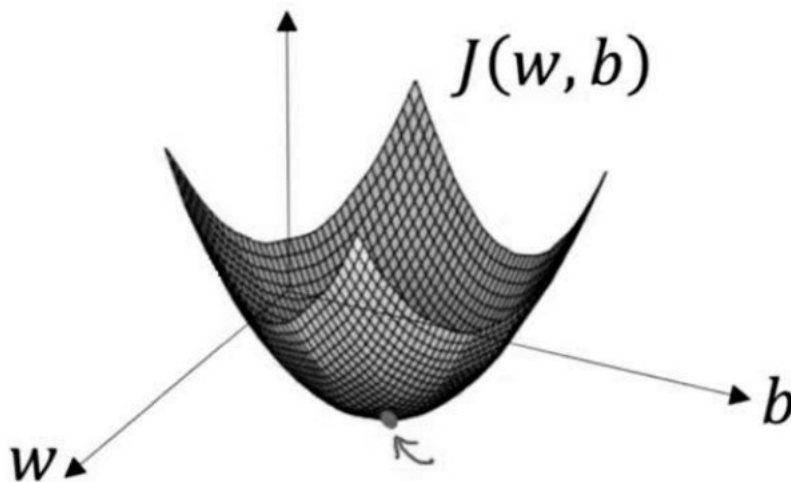
Introduction : In the realm of deep learning, one of the pivotal aspects of model training revolves around optimization algorithms, which play a vital role in refining the performance of neural networks. By utilizing these optimizers, we can adjust key parameters, such as weights and learning rates, 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.



- **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 (\hat{y}) 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 is 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 disproportionately, 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.

Chapter – 2

The Activation Functions

2.1 : Overview

Activation functions play a crucial role in the landscape of artificial neural networks, acting as the gatekeepers for neuron activation. They determine whether the information that flows into a neuron is relevant or if it should be disregarded, significantly influencing the network's performance.

$$Y = \sum (weight * input) + bias$$

So, can we think about skipping activation functions altogether? That's an intriguing question! While it may seem enticing to simplify things by eliminating them, the truth is that without activation functions, the weight and bias of the neurons would merely execute a linear transformation. Linear equations are straightforward and easy to solve, but they fall short in tackling complex problems. Essentially, a neural network without activation functions is nothing more than a linear regression model. The magic of activation functions lies in their ability to perform nonlinear transformations on inputs, empowering the network to learn and handle intricate tasks. When we aspire to have our networks tackle challenging issues like language translation or image classification, relying solely on linear transformations would be woefully inadequate.

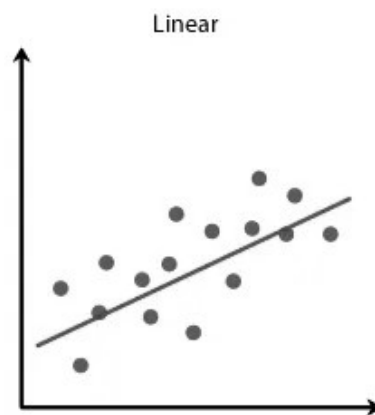
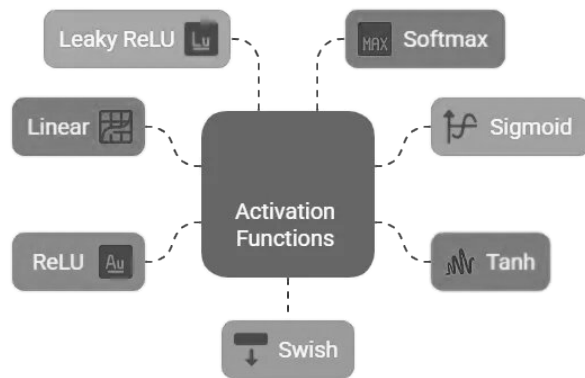


Fig 2.1 - Representation of Linear Regression

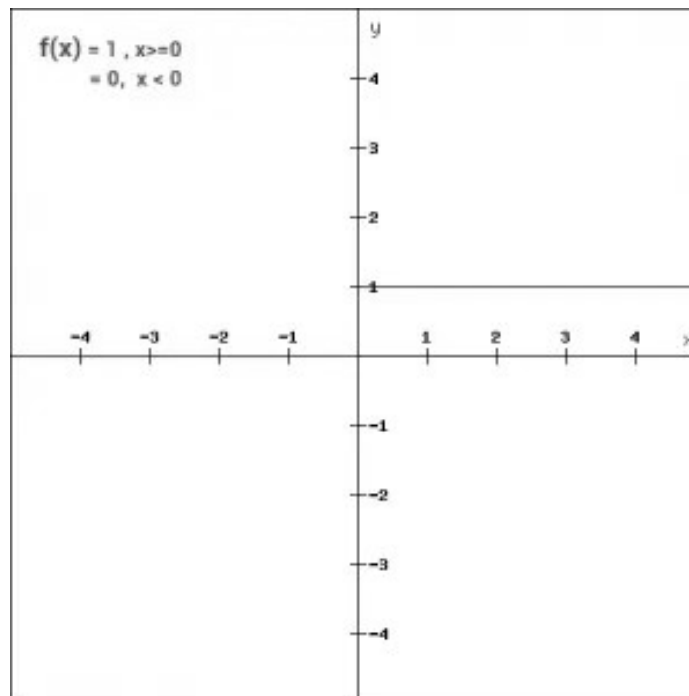
Let's dive into some popular activation functions and their ideal applications!

Activation Functions in Deep Learning



2.2 : Binary Step Function.

When activating on the topic of activation functions, the binary step function often comes to mind first. This threshold-based classifier activates a neuron if the value x exceeds a specified threshold; otherwise, it remains inactive.



Binary Step Function

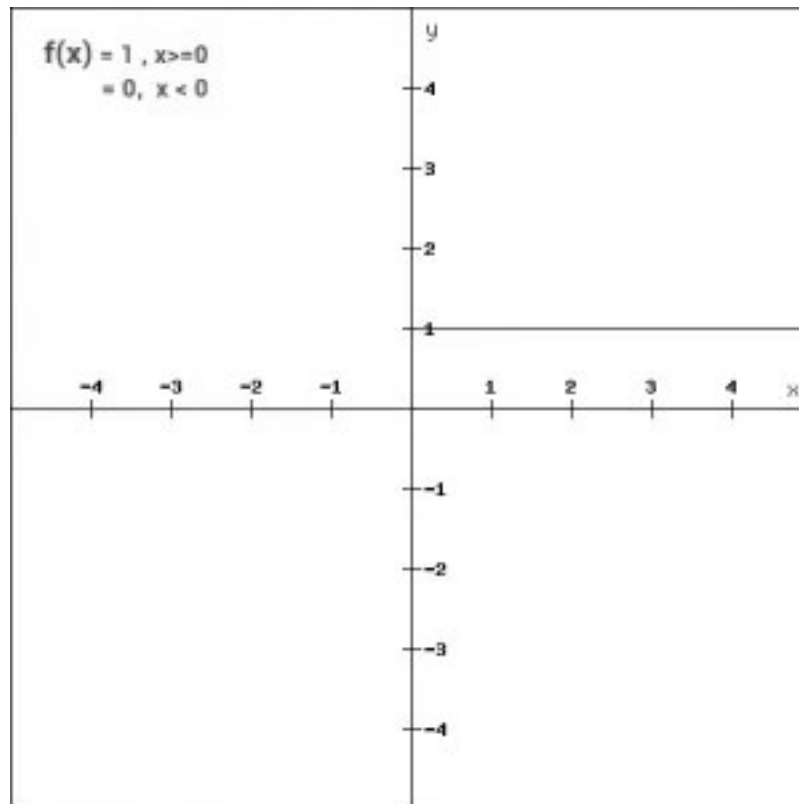
- *Pros* : It's incredibly simple and is perfect for binary classification tasks. When the need arises to simply classify data into two categories—yes or no—this step function shines as an excellent choice.
- *Cons* : However, the binary step function is more theoretical than practical in many real-world scenarios, where we typically deal with multiple classes rather than just one. Its limitations in handling diverse classifications become evident. Additionally, the gradient of the step function is consistently zero, which poses a significant drawback.

2.3: Exploring the Linear Function

In our previous discussions, we encountered a critical limitation with the step function in neural networks, particularly concerning the phenomenon where the gradient becomes zero. This scenario creates a significant obstacle during the back-propagation process since it renders the ability to update the weights nonexistent. When the gradients do not change, the model stumbles, preventing any meaningful learning from taking place.

To address these shortcomings, we can explore the implementation of a linear function as an alternative. One can represent this linear function mathematically as

$$f(x) = a \cdot x, \text{ with 'a' is belong to } \mathbb{R}$$



if $a = 4$, we have...

where "a" is a constant that modifies the output in a straightforward manner. The beauty of this approach lies in its simplicity and effectiveness in certain contexts.

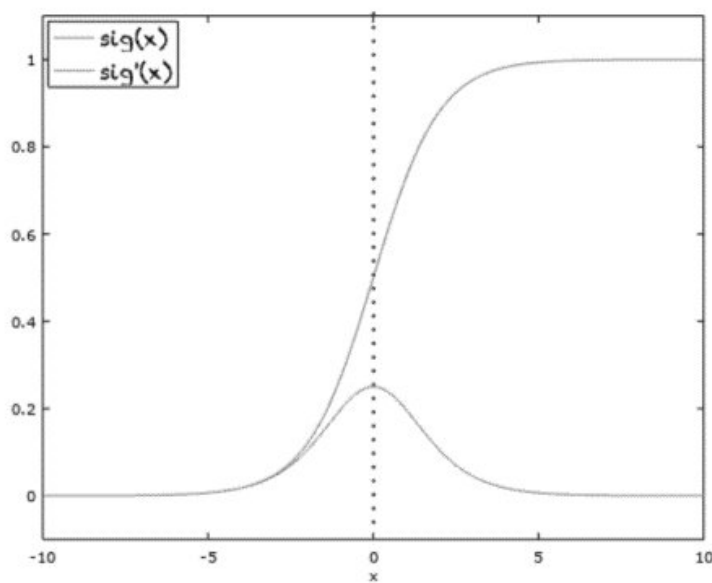
2.4 : Sigmoid Function

The sigmoid function is a special form of the logistic function and is usually denoted by $\sigma(x)$ or $\text{sig}(x)$. It is given by:

$$\sigma(x) = 1/(1+\exp(-x))$$

Properties and Identities Of Sigmoid Function

The graph of sigmoid function is an S-shaped curve as shown by the green line in the graph below. The figure also shows the graph of the derivative in pink color. The expression for the derivative, along with some important properties are shown on the right.



Plot of $\sigma(x)$ and its derivate $\sigma'(x)$

Domain: $(-\infty, +\infty)$

Range: $(0, +1)$

$\sigma(0) = 0.5$

Other properties

$\sigma(x) = 1 - \sigma(-x)$

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

$\sigma'(x) = \sigma(x)(1 - \sigma(x))$

A few other properties include:

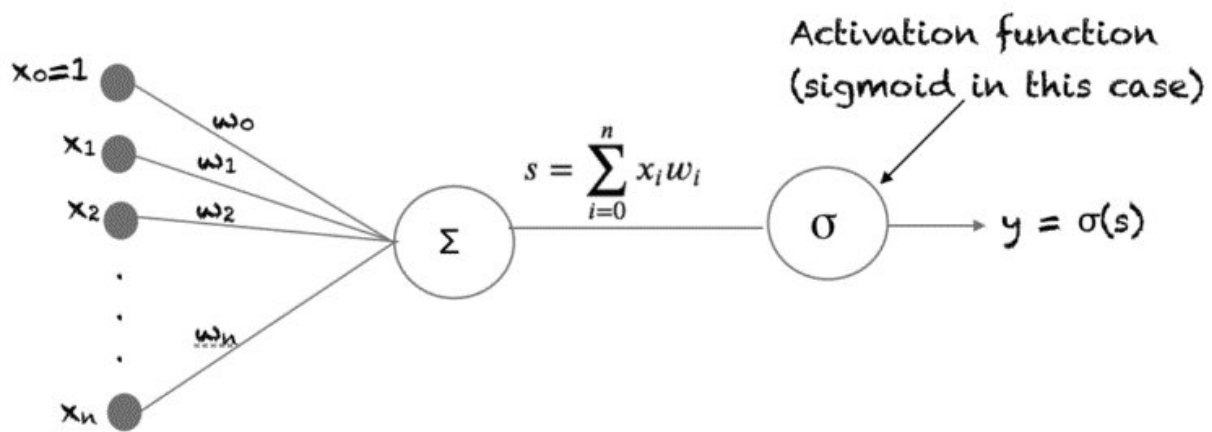
1. Domain: $(-\infty, +\infty)$
2. Range: $(0, +1)$
3. $\sigma(0) = 0.5$
4. The function is monotonically increasing.
5. The function is continuous everywhere.
6. The function is differentiable everywhere in its domain.

The Sigmoid As A Squashing Function

The sigmoid function is also called a squashing function as its domain is the set of all real numbers, and its range is (0, 1). Hence, if the input to the function is either a very large negative number or a very large positive number, the output is always between 0 and 1. Same goes for any number between $-\infty$ and $+\infty$.

Sigmoid As An Activation Function In Neural Networks

The sigmoid function is used as an activation function in neural networks. Just to review what is an activation function, the figure below shows the role of an activation function in one layer of a neural network. A weighted sum of inputs is passed through an activation function and this output serves as an input to the next layer.



A sigmoid unit in a neural network

When the activation function for a neuron is a sigmoid function it is a guarantee that the output of this unit will always be between 0 and 1. Also, as the sigmoid is a non-linear function, the output of this unit would be a non-linear function of the weighted sum of inputs. Such a neuron that employs a sigmoid function as an activation function is termed as a sigmoid unit.

Linearity and Non Linearity

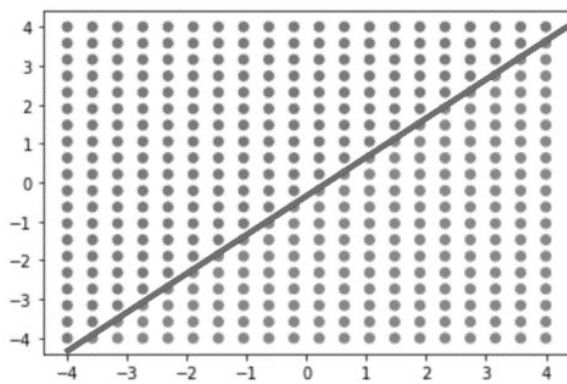
The concepts of linear and non-linear separability are fundamental to understanding classification problems in machine learning and data analysis. Let's delve deeper into these terms and their implications.

In many situations, we face classification problems where we have a collection of points located within a specific space. Each of these points is assigned a label that categorizes it into a particular class. This leads us to the concept of linear separability. A classification problem is considered linearly separable if we can effectively distinguish

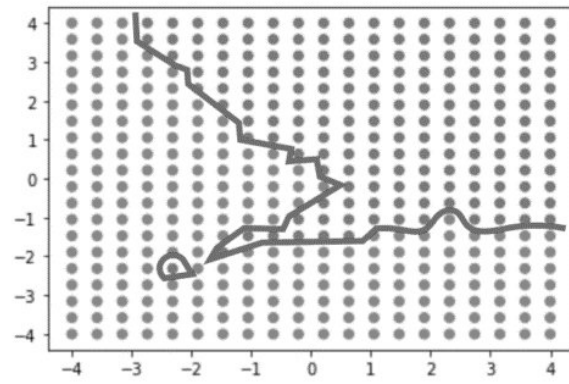
between the two classes using a straight line, or, in higher dimensions, a hyperplane. Such linear boundaries serve as clear dividers, allowing us to neatly categorize the data points into their respective classes without any overlap.

To visualize this, imagine a two-dimensional graph where we plot points representing different classes. In instances of linear separability, we can draw a straight line that cleanly separates the red points from the blue points, indicating that all points of one class reside on one side of the line, while all points of the other class lie on the opposite side. This scenario is often straightforward and lends itself well to various linear classification algorithms.

Conversely, we encounter non-linear separability in situations where a straight line fails to adequately divide the two classes. In these cases, the data points are intermingled in such a way that no linear boundary can effectively separate the red points from the blue ones. Here, more complex decision boundaries become necessary. These boundaries can take various forms, such as curves or other non-linear shapes, to effectively distinguish between the classes.



Linear boundary



Non-Linear boundary

Linear Vs. Non-Linearly separable problems

Referencing our previous visual example, in the second figure, representing the non-linearly separable problem, it becomes evident that a straight line is insufficient. Instead, a non-linear boundary is required to navigate the intricate arrangement of the points, ensuring that we accurately classify them while respecting their true relationships in space.

Understanding the distinction between linear and non-linear separability is crucial as it influences the choice of algorithms and approaches we employ in machine learning tasks. Selecting the correct model that aligns with the nature of the data is fundamental to achieving accurate classifications and, ultimately, successful outcomes in analytical projects.

The Significance of the Sigmoid Function in Neural Networks

In the realm of neural networks, the choice of activation function plays a crucial role in determining the model's capacity to learn and solve problems. If we rely solely on a linear activation function for our network, we would severely limit its ability to address a diverse set of issues since such a model would only be capable of understanding linearly separable problems. However, incorporating a single hidden layer along with a sigmoid activation function into the model significantly expands its capabilities. This powerful combination enables the neural network to adeptly tackle non-linearly separable problems, drawing non-linear boundaries that allow for a richer representation of complex decision functions.

It is important to note that the only non-linear functions suitable for activation in a neural network must possess specific characteristics; they must be monotonically increasing. This eliminates options such as $\sin(x)$ or $\cos(x)$, as their oscillatory nature disqualifies them from being effective activation functions. Additionally, an ideal activation function must be defined for all real numbers, maintaining continuity throughout the entire range. When we consider the functionality further, it's also essential that the activation function is differentiable across the entire spectrum of real numbers. This characteristic is particularly important as it relates to the process of learning within the neural network.

In the typical training process for a neural network, the backpropagation algorithm, which employs gradient descent, is utilized to optimize the weights associated with the model. For this optimization to occur effectively, it is necessary to compute the derivative of the activation function. Here, the virtues of the sigmoid function truly stand out. The sigmoid function is not only monotonic, continuous, and differentiable everywhere, but its derivative possesses a remarkable property: it can be expressed in terms of the function itself. This inherent efficiency simplifies the derivation of the update equations necessary for learning the weights in a neural network utilizing the backpropagation algorithm.

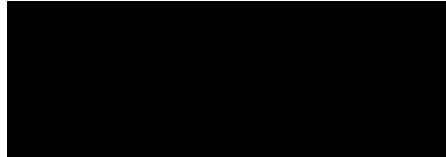
In conclusion, the significance of the sigmoid function in neural networks cannot be overstated. Its unique qualities enable the model to go beyond linear limitations, facilitating the learning of intricate patterns and aiding in the development of robust decision-making frameworks. Through the combination of hidden layers and appropriate activation functions like the sigmoid, we position ourselves to tackle complex real-world problems with greater efficacy.

The vanishing Gradient Problem

- **Vanishing gradient problem** is encountered when training artificial neural networks with gradient-based learning methods and backpropagation. In such methods, each of the neural network's weights receives an update proportional to the partial derivative of the error function with respect to the current weight in each iteration of training. The problem is that in some cases, the gradient will be vanishingly small, effectively preventing the weight from changing its value. In the worst case, this may completely stop the neural network from further training.

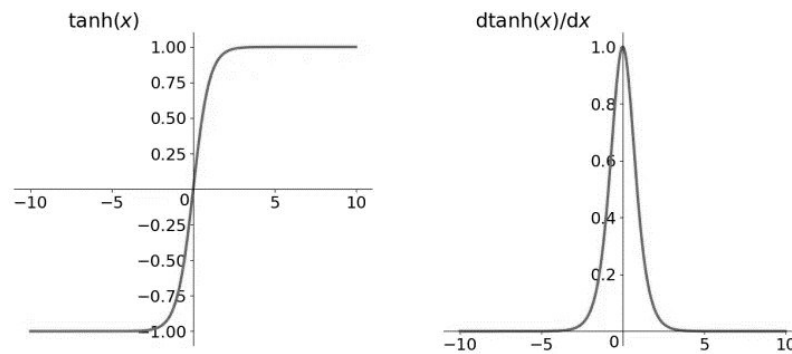
2.5 : Tanh Activation Functions

The Hyperbolic Tangent function, often abbreviated as Tanh, is a mathematical function that bears a resemblance to the sigmoid function, albeit with some notable variations that make it particularly advantageous in certain contexts. Specifically, Tanh effectively addresses and mitigates some of the issues that are inherently present in the sigmoid function. For instance, while both functions are utilized in various applications such as neural networks and logistic regression models, Tanh allows for a broader output range, which stretches from -1 to 1. This characteristic makes Tanh especially useful in scenarios where dealing with values that can be both negative and positive is crucial.



Tanh Function

In addition, the fact that Tanh has a steeper gradient than the sigmoid function at the origin enables it to converge more rapidly during the training process of computational models. This leads to a more efficient learning pattern, as the Tanh function reduces the likelihood of encountering saturation problems that can potentially slow down learning or lead to vanishing gradients, a challenge often associated with sigmoid. Overall, by choosing to implement the Hyperbolic Tangent function in place of the traditional sigmoid, one can significantly enhance the performance and reliability of the models that utilize it, offering a superior alternative that effectively counteracts the limitations found in sigmoid functions.



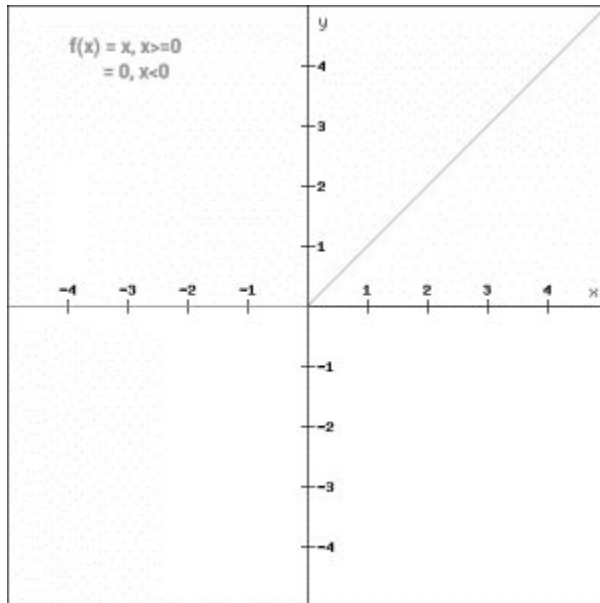
Plot of Tanh and Derivative

Hence, in conclusion, incorporating Tanh instead of sigmoid is not merely a minor variation; it represents a thoughtful and strategic improvement designed to optimize the outputs and functionality of complex mathematical models in various real-world applications, making it a preferred choice for many practitioners in the field.

2.5 : RELU Activation Functions

The ReLU Activation function, which stands for Rectified Linear Unit, is a type of non-linear activation function that has seen a significant rise in its application and popularity within the realm of deep learning. One of the key advantages of utilizing the ReLU function compared to other activation functions is its ability to manage neuron activation more effectively. Specifically, it ensures that not all neurons are activated simultaneously. Instead, neurons will only become deactivated when the result of the linear transformation falls below zero.

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



This behavior enhances computational efficiency and helps in mitigating issues such as vanishing gradients, which can plague other activation functions. Understanding this concept is crucial, and visualizing it through a plot can provide further clarity on how the ReLU function operates in practice, illustrating the relationship between the input values, the output, and the thresholds that determine the activation of the neurons.

python function for ReLU:

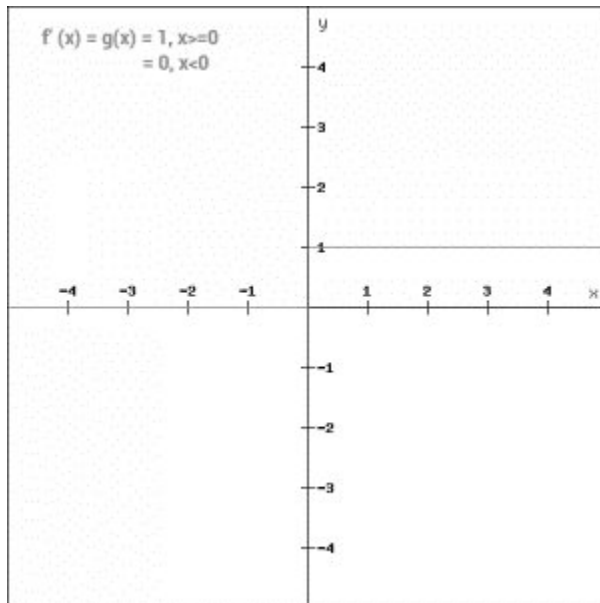
```
def relu_function(x):  
    if x < 0:  
        return 0  
    else:  
        return x  
relu_function(7), relu_function(-7)
```

Output:

(7, 0)

Let's look at the gradient of the ReLU function.

$$\begin{aligned} f'(x) &= 1, x \geq 0 \\ &= 0, x < 0 \end{aligned}$$



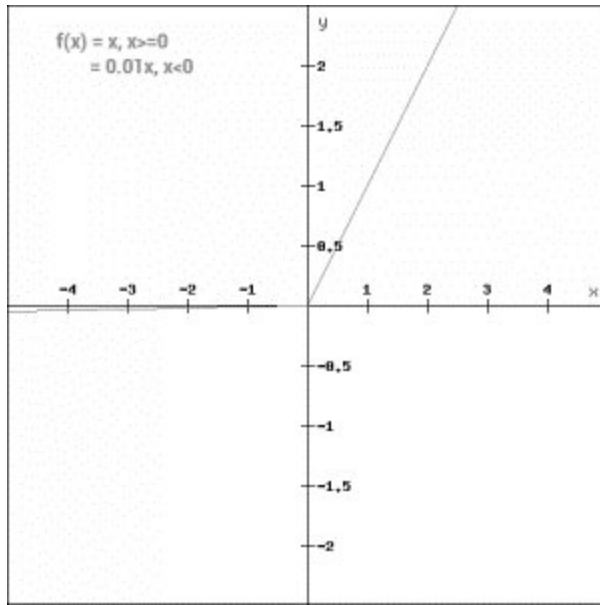
In deep learning models, especially those that involve neural networks, the choice of activation function can greatly influence the training dynamics and overall performance of the model. With this in mind, the ReLU function stands out for its simplicity, effectiveness, and the practical advantages it brings to the training of complex models, making it a vital tool in the arsenal of deep learning practitioners.

2.6 : Leaky ReLU

Leaky ReLU function is nothing but an improved version of the ReLU function. As we saw that for the ReLU function, the gradient is 0 for $x < 0$, which would deactivate the neurons in that region.

Leaky ReLU is defined to address this problem. Instead of defining the Relu function as 0 for negative values of x , we define it as an extremely small linear component of x . Here is the mathematical expression-

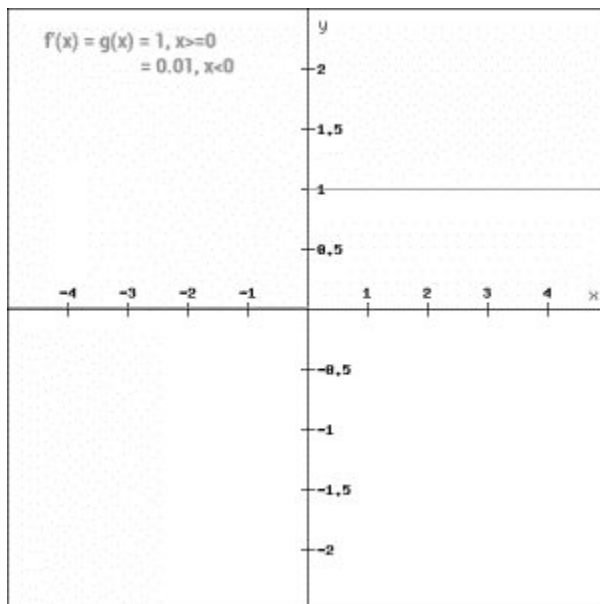
$$\begin{aligned} f(x) &= 0.01x, x < 0 \\ &= x, x \geq 0 \end{aligned}$$



By making this small modification, the gradient of the left side of the graph comes out to be a non zero value. Hence we would no longer encounter dead neurons in that region. Here is the derivative of the Leaky ReLU function

$$f'(x) = 1, x \geq 0$$

$$= 0.01, x < 0$$



Since Leaky ReLU is a variant of ReLU, the python code can be implemented with a small modification-

```
def leaky_relu_function(x):
    if x<0:
        return 0.01*x
    else:
        return x
leaky_relu_function(7), leaky_relu_function(-7)
```

Output:

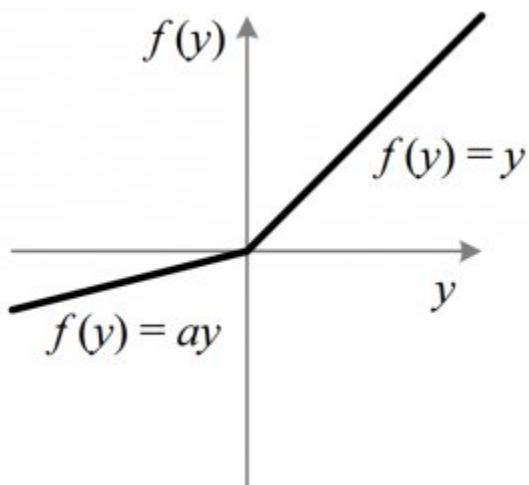
(7, -0.07)

Apart from Leaky ReLU, there are a few other variants of ReLU, the two most popular are – Parameterised ReLU function and Exponential ReLU.

2.7: Paramrterized ReLU

This is another variant of ReLU that aims to solve the problem of gradient's becoming zero for the left half of the axis. The parameterised ReLU, as the name suggests, introduces a new parameter as a slope of the negative part of the function. Here's how the ReLU function is modified to incorporate the slope parameter-

$$f(x) = x, x \geq 0 \\ = ax, x < 0$$



When the value of a is fixed to 0.01, the function acts as a Leaky ReLU function. However, in case of a parameterised ReLU function, ' a ' is also a trainable parameter. The network also learns the value of ' a ' for faster and more optimum convergence.

The derivative of the function would be same as the Leaky ReLU function, except the value 0.01 will be replaced with the value of a .

$$f'(x) = 1, x \geq 0$$

$$= a, x < 0$$

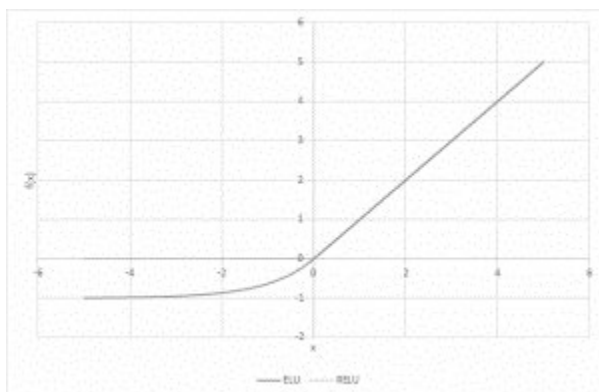
The parameterized ReLU function is used when the leaky ReLU function still fails to solve the problem of dead neurons and the relevant information is not successfully passed to the next layer.

2.8: Exponential ReLU

Exponential Linear Unit or ELU for short is also a variant of Rectified Linear Unit (ReLU) that modifies the slope of the negative part of the function. Unlike the leaky relu and parametric ReLU functions, instead of a straight line, ELU uses a log curve for defining the negative values. It is defined as

$$f(x) = x, x \geq 0$$

$$= a(e^x - 1), x < 0$$



Let's define this function in python

```
def elu_function(x, a):  
    if x<0:  
        return a*(np.exp(x)-1)  
    else:  
        return x  
elu_function(5, 0.1),elu_function(-5, 0.1)
```

Output:

(5, -0.09932620530009145)

The derivative of the elu function for values of x greater than 0 is 1, like all the relu variants. But for values of $x < 0$, the derivative would be $a \cdot e^x$.

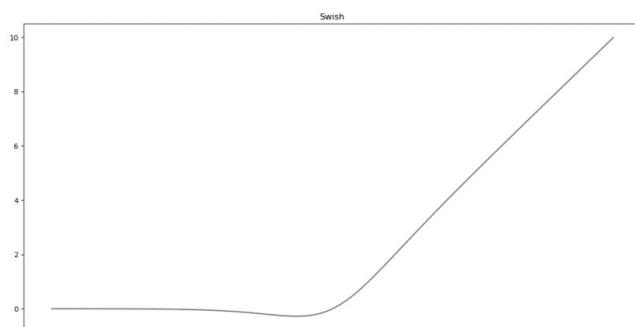
$$f'(x) = x, \quad x \geq 0$$
$$= a(e^x), \quad x < 0$$

2.9 : Swish

Swish is a lesser known activation function which was discovered by researchers at Google. Swish is as computationally efficient as ReLU and shows better performance than ReLU on deeper models. The values for swish ranges from negative infinity to infinity. The function is defined as -

$$f(x) = x * \text{sigmoid}(x)$$

$$f(x) = x / (1 - e^{-x})$$



As you can see, the curve of the function is smooth and the function is differentiable at all points. This is helpful during the model optimization process and is considered to be one of the reasons that swish outperforms ReLU.

A unique fact about this function is that swish function is not monotonic. This means that the value of the function may decrease even when the input values are increasing. Let's look at the python code for the swish function

```
def swish_function(x):  
    return x/(1+np.exp(-x))  
swish_function(-67), swish_function(4)
```

Output:

(5.349885844610276e-28, 4.074629441455096)

2.10 : Softmax

Softmax function is often described as a combination of multiple sigmoids. We know that sigmoid returns values between 0 and 1, which can be treated as probabilities of a data point belonging to a particular class. Thus sigmoid is widely used for binary classification problems.

The softmax function can be used for multiclass classification problems. This function returns the probability for a datapoint belonging to each individual class. Here is the mathematical expression of the same-

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}} \quad \text{for } j = 1, \dots, K.$$

While building a network for a multiclass problem, the output layer would have as many neurons as the number of classes in the target. For instance if you have three classes, there would be three neurons in the output layer. Suppose you got the output from the neurons as [1.2 , 0.9 , 0.75].

Applying the softmax function over these values, you will get the following result – [0.42 , 0.31, 0.27]. These represent the probability for the data point belonging to each class. Note that the sum of all the values is 1.

Let us code this in python

```
def softmax_function(x):  
    z = np.exp(x)  
    z_ = z/z.sum()  
    return z_  
softmax_function([0.8, 1.2, 3.1])
```

Output:

```
array([0.08021815, 0.11967141, 0.80011044])
```

When working with various activation functions in neural networks, it's crucial to have some heuristics or guiding principles to determine which function is most appropriate for a given situation. Although there isn't a definitive rule of thumb to follow, understanding the properties of your specific problem can lead to a more informed decision, ultimately facilitating easier and faster convergence of your network model. In scenarios involving classifiers, sigmoid functions and their combinations tend to perform quite effectively. However, practitioners sometimes steer clear of using sigmoid and tanh functions due to potential issues related to the vanishing gradient problem, which can hinder the learning process. In contemporary applications, the ReLU (Rectified Linear Unit) function has emerged as a versatile general-purpose activation function that is widely adopted in most neural network architectures.

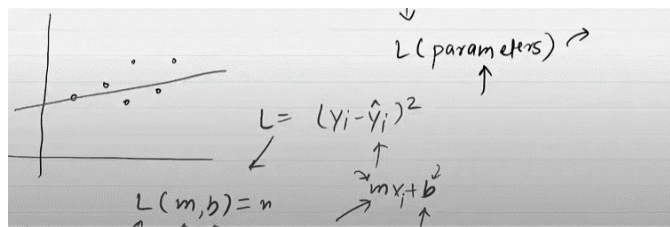
It's vital to note that if your network encounters the phenomenon known as dead neurons—where certain neurons become inactive and stop learning—the leaky ReLU function is usually the best alternative to consider. Always remember that the ReLU function is primarily suited for use within the hidden layers of the network. A practical piece of advice would be to start with the ReLU function as your initial choice for activation and then explore other activation functions if you find that ReLU does not yield the desired performance outcomes for your specific case.

CHAPTER – 3

The Loss Functions

3.1: What is Loss Function

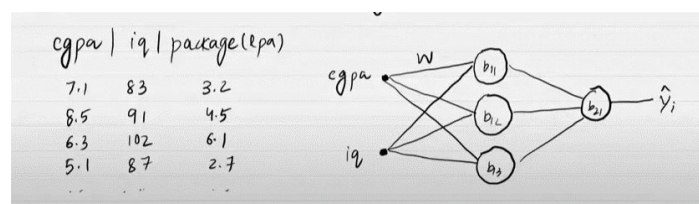
The loss function, commonly known as the error function, plays an essential and multifaceted role in the realm of machine learning, serving as a foundational element that helps gauge the performance of predictive models by quantifying discrepancies between the model's predicted outputs and the actual values that we aim for. This calculation is especially significant when addressing regression tasks, much like the problem of estimating car prices based on a dataset comprised of historical pricing information, where the loss function meticulously assesses the predictions generated by a neural network against a particular training sample pulled from the overarching training dataset. In this scenario, the loss function acts as a crucial metric that calculates the gap—or the margin of error—between the car price that the neural network predicts and the true market value of the car.



Definition:

"A loss function is a mathematical function used to measure the difference between a model's predictions and the true values. Optimization algorithms, such as gradient descent, utilize this loss function to iteratively adjust the model's parameters, such as weights and biases, to minimize the loss."

The output produced by the loss function, known simply as the loss, serves as an indicator of how accurately the model is performing in its predictions. A lower loss value generally signifies that the model predictions are closer to the actual target values, thereby reflecting a higher degree of accuracy. As the model undergoes training, it employs a learning algorithm, such as the backpropagation algorithm, which is rooted in calculus and optimization principles.



"After initializing the weights (w) and biases (b) with random values, the neural network takes inputs such as CGPA and IQ and produces an output (\hat{y}). Let's say the initial output is 3.6. The loss function then calculates the squared difference between this predicted value and the actual value. This loss value quantifies how far off the prediction is from the truth."

This algorithm utilizes the gradient—which is the rate of change—of the loss function in relation to the parameters of the model. By computing this gradient, the learning algorithm makes adjustments to the model's parameters through iterative processes, aiming to minimize the loss. This minimization process is crucial because it directly contributes to enhancing the predictive performance of the model on the provided dataset, allowing it to make more accurate predictions that align closely with the actual outcomes observed in the training data.

In sum, understanding the role of the loss function in machine learning is vital, as it not only provides a numerical value that represents the accuracy of model predictions but also drives the optimization processes that lead to more robust and effective machine learning solutions. Through careful measurement and adjustment facilitated by the loss function, machine learning models can continually evolve and improve, resulting in a significant enhancement in their ability to forecast accurately, whether they are being applied to car price predictions or any other complex predictive tasks across various domains.

1	Loss functions in Regression based problem	a. Mean Square Error Loss
		b. Mean Absolute Error Loss
		c. Huber Loss
2	Loss functions in Binary classification-based problem	a. Binary Cross Entropy Loss
		b. Hinge Loss
3	Loss functions in Multiclass classification-based problem	a. Multiclass Cross Entropy Loss
		b. Sparse Multiclass Cross Entropy Loss
		c. Kullback Leibler Divergence Loss

In the process of training a neural network, the next critical step involves the implementation of backpropagation, a method through which the network effectively recalibrates its weights and biases. This adjustment is achieved by meticulously calculating the gradients of the loss function in relation to the aforementioned parameters. Following this calculation, gradient descent is utilized as a powerful algorithm to update the weights and biases, ensuring that these adjustments are made in a direction that serves to minimize the overall loss incurred during the training phase. This intricate, iterative process persists, operating in multiple cycles, until the loss reaches a level that can be deemed sufficiently minimized. As a result of this continual optimization, the neural network is able to produce predictions that are markedly improved, showcasing the effectiveness of the backpropagation and gradient descent techniques in refining model accuracy over successive training iterations.

Although there are different types of loss functions, fundamentally, they all operate by quantifying the difference between a mode’s predictions and the actual target value in the dataset. The official term for this numerical quantification is the prediction error. The learning algorithm and mechanisms in a machine learning model are optimized to minimize the prediction error, so this means that after a calculation of the value for the loss function, which is determined by the prediction error, the learning algorithm leverages this information to conduct weight and parameter updates which in effect during the next training pass leads to a lower prediction error.

When exploring the topic of loss function, machine learning algorithms, and the learning process within neural networks, the topic of **Empirical Risk Minimization(ERM)** comes up. ERM is an approach to selecting the optimal parameters of a machine learning algorithm that minimizes the empirical risk. The empirical risk, in this case, is the training dataset.


The risk minimization component of ERM is the process by which the internal learning algorithm minimizes the error of prediction of machine learning algorithm to a known dataset in the outcome that the model has an expected performance and accuracy in a scenario where an unseen dataset or data sample which could have similar statical data distribution as the dataset the model's has been initially trained on.

3.2: The Mean Squared Error (Loss Function for Regression)

Mean Square Error (MSE) / L2 Loss

The Mean Square Error(MSE) or L2 loss is a loss function that quantifies the magnitude of the error between a machine learning algorithm prediction and an actual output by taking the average of the squared difference between the predictions and the target values. Squaring the difference between the predictions and actual target values results in a higher penalty assigned to more significant deviations from the target value. A mean of the errors normalizes the total errors against the number of samples in a dataset or observation.

The mathematical equation for Mean Square Error (MSE) or L2 Loss is:



Where:

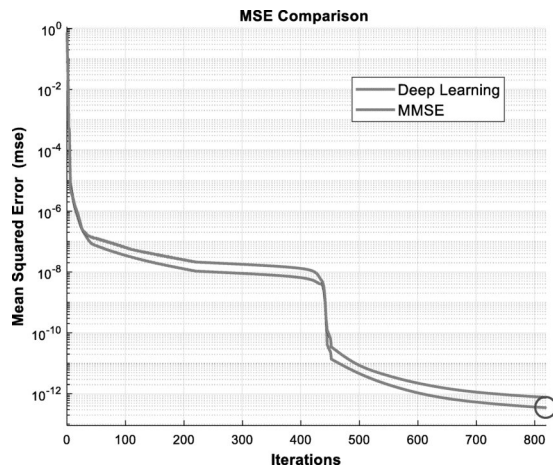
- n is the number of samples in the dataset
- y_i is the predicted value for the i -th sample
- \bar{y} is the target value for the i -th sample

When to use MSE

Understanding the appropriate contexts and scenarios for utilizing Mean Squared Error (MSE) is fundamentally important in the intricate process of developing effective machine learning models. MSE serves as a widely accepted and standard loss function that is predominantly employed in a majority of regression tasks within the field of machine learning. Its primary objective is to guide the model towards optimizing performance by systematically minimizing the squared differences between the values predicted by the model and the actual target values that we aim to estimate.

One key attribute of MSE that makes it particularly advantageous in certain machine learning applications is its ability to impose a significant penalty on the model when it encounters outliers within the dataset. This can be quite beneficial in scenarios where it is advantageous for the learning algorithm to treat those outlying data points with heightened scrutiny, thereby allowing the model to adjust and learn from the significant discrepancies that those outliers present. However, it is essential to recognize that the very properties that make MSE favorable in some instances can also render it less suitable in other circumstances—especially when outliers arise from random noise in the data rather than reflecting true positive signals that are crucial to the predictive modeling process.

As a practical illustration of when the MSE loss function is effectively put to use, one can consider the case of predicting real estate prices. This particular application falls within the broader domain of predictive modeling, where various features such as the number of rooms in a dwelling, the geographical location of a property, the total area encompassed by the land and structure, proximity to essential amenities, as well as other relevant numerical features are meticulously analyzed to derive a plausible estimate of house prices. In most cases, house prices within a localized market tend to follow a normal distribution, which highlights the importance of penalizing outliers effectively. This focus on penalization is vital in ensuring that the model develops the capacity to generate reliable predictions for house prices that are both accurate and realistic.



The financial implications of even a small percentage of error in predicting real estate values can be quite significant, thereby underscoring the critical nature of employing robust predictive techniques. For example, if one were to consider a situation where there is a 5% discrepancy in the valuation of a property priced at \$200,000, that tiny percentage translates to a monetary difference of \$10,000—a substantial sum that can have serious consequences for consumers and investors alike. Thus, by squaring the errors, as mandated by the computational approach of MSE, the method inherently attributes a higher weight to larger errors. This approach compels the model to strive for precision, particularly in relation to the higher-valued properties that are common in the real estate market, ultimately fostering an environment where the predictions made by the machine learning model are as accurate and reliable as possible over time.

3.3 : MAE (Mean Absolute Error) / L1 Error

Mean Absolute Error, commonly referred to as MAE and sometimes identified by the term L1 Loss, serves as a pivotal loss function within the realm of regression tasks in machine learning. It plays a crucial role by meticulously calculating the average of the absolute differences that arise between the values predicted by a machine learning model and the corresponding actual target values that we seek to predict. This aspect of MAE is significant because it provides a clear and understandable measure of error. Unlike its counterpart, Mean Squared Error (MSE), which elevates the differences to a square, MAE adopts a more direct approach. It treats each discrepancy with equal importance, without amplifying the influence of larger errors by squaring them. This characteristic allows it to handle outliers in the data more effectively, making it a preferred choice in scenarios where a balanced evaluation of prediction accuracy is paramount. In summary, MAE offers an insightful way to gauge model performance by focusing purely on the average size of the prediction errors, ensuring that every miscalculation, regardless of its scale, is treated fairly within the assessment process.

The Mathematical Equation for MAE is :

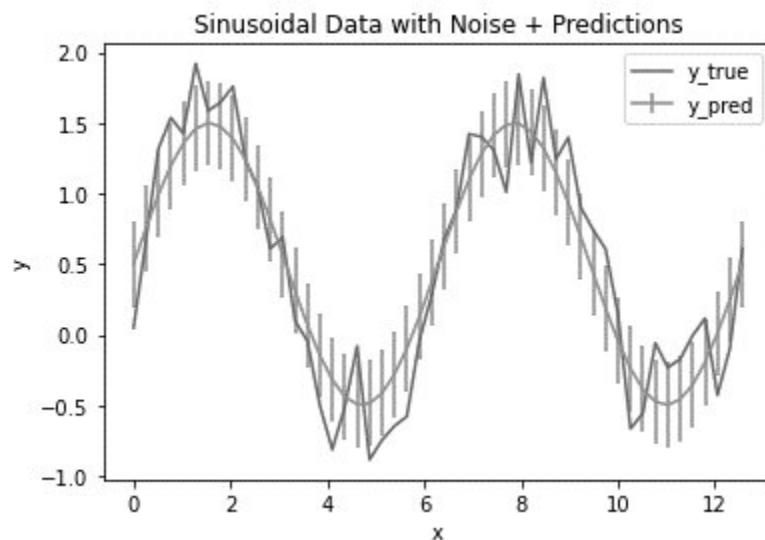


Where:

- n is the number of samples in the dataset
- y_i is the predicted value for the i -th sample
- \bar{y} is the target value for the i -th sample

When to use MAE

MAE measures the average absolute difference between the predicted and actual values. Unlike MSE, MAE does not square the differences, which makes it less sensitive to outliers. Compared to Mean Squared Error (MSE), Mean Absolute Error (MAE) is inherently less sensitive to outliers because it assigns an equal weight to all errors, regardless of their magnitude.



This means that while an outlier can significantly skew the MSE by contributing a disproportionately large error when squared, its impact on MAE is much more contained. An outlier's influence on the overall error metric is minimal when using MAE as a loss function. In contrast, MSE amplifies the effect of outliers due to the squaring of error terms, affecting the model's error estimation more substantially.

3.4 : Huber Loss / Smooth Mean Squared Error

Huber Loss or Smooth Mean Absolute Error is a loss function that takes the advantageous characteristics of the Mean Absolute Error and Mean Squared Error loss functions and combines them into a single loss function. The hybrid nature of Huber Loss makes it less sensitive to outliers, just like MAE, but also penalizes minor errors within the data sample, similar to MSE. The Huber Loss function is also utilized in regression machine learning tasks.

$$\text{Huber Loss} = \begin{cases} \frac{1}{2}(y - y_p)^2, & |y - y_p| \leq \delta \\ \delta|y - y_p| - \frac{1}{2}\delta^2, & |y - y_p| > \delta \end{cases},$$

When to use Huber Loss / Smooth Mean Absolute Error

The Huber Loss function effectively combines two components for handling errors differently, with the transition point between these components determined by the threshold δ :

- Quadratic Component for Small Errors: For errors smaller than δ , it uses the quadratic component $(1/2) * (f(x) - y)^2$
- Linear Component for Large Errors: For errors larger than δ , it applies the linear component $\delta * |f(x) - y| - (1/2) * \delta^2$

Huber loss operates in two modes that are switched based on the size of the calculated difference between the actual target value and the prediction of the machine learning algorithm. The key term within Huber Loss is delta (δ). Delta is a threshold that determines the numerical boundary at which the Huber Loss utilizes the quadratic application of loss or linear calculation.

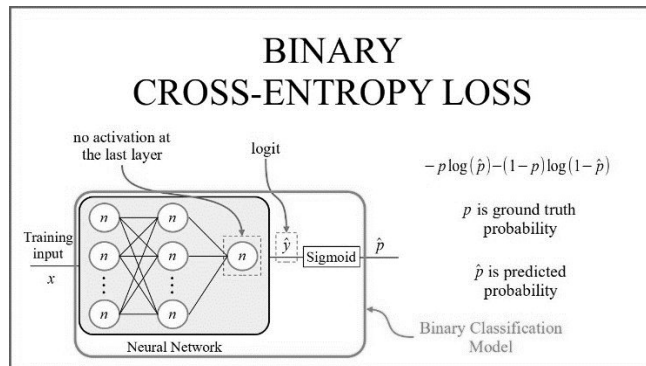
The quadratic component of Huber Loss characterizes the advantages of MSE that penalize outliers; within Huber Loss, this is applied to errors smaller than delta, which ensures a more accurate prediction from the model.

Suppose the calculated error, which is the difference between the actual and predicted values, is larger than the delta. In that case, Huber Loss utilizes the linear calculation of loss similar to MAE, where there is less sensitivity to the error size to ensure the trained model isn't over-penalizing large errors, especially if the dataset contains outliers or unlikely-to-occur data samples.

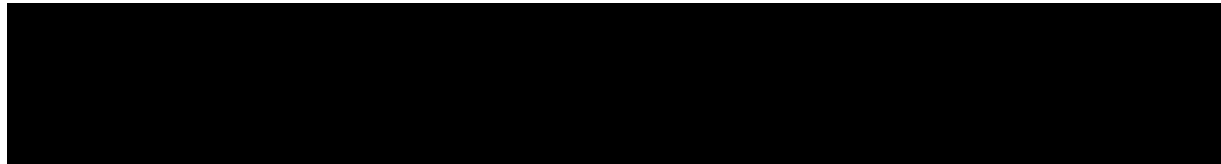
3.5: Binary Cross-Entropy Loss / Log Loss (Loss Functions for Classification)

Binary Cross-Entropy Loss, commonly referred to as BCE, serves as a crucial metric for evaluating the performance of classification models, particularly those that generate predictions represented as probabilities. These probability values typically range from 0 to 1, with each value indicating the likelihood that a given data sample belongs to a specific class or category. In the context of Binary Cross-Entropy Loss, the model is primarily concerned with distinguishing between two distinct classes. It is important to note, however, that there exists a related variant known as Categorical Cross-Entropy, which is specifically designed for scenarios involving multiple classes within a single problem.

To fully grasp the concept of Binary Cross-Entropy Loss, which is often interchangeably referred to as Log Loss in various discussions, it can be quite beneficial to delve deeper into the individual components encompassed within this term.



Firstly, the term "Loss" denotes a mathematical representation of the degree of difference or margin that exists between the predictions made by a machine learning algorithm and the actual target values that those predictions aim to approximate. This quantification is essential for evaluating how well the model is performing in terms of its predictive capabilities. Next, we have "Entropy," which can be understood in its simplest terms as a measure of the level of randomness or disorder prevalent within a particular system. This concept is foundational as it relates to the uncertainty associated with predictions made by the model.



The term "Cross Entropy" is rooted in information theory, and it serves a vital function by evaluating the differences that exist between two probability distributions. This measurement can be instrumental in identifying how closely one distribution—typically that generated by the model—aligns with the target distribution associated with the observation. Lastly, the prefix "Binary" relates to the representation of numerical values using only two distinct states, which are commonly represented as 0 and 1. This notion extends to the definition of Binary Classification, where the goal is to differentiate between two distinct classes, referred to here as class A and class B. In this framework, class A is conventionally assigned the numerical representation of 0, whereas class B is designated with the numerical value of 1, creating a clear demarcation between the two classes based on their binary representations.

In summary, Binary Cross-Entropy Loss plays an integral role in the assessment of classification models that yield probabilistic predictions within binary classifications.

3.6: Categorical Cross-Entropy Loss

Cross-entropy loss is a measure of the difference between two probability distributions, the ground truth distribution P and the predicted distribution Q of the model.

Under classification, P is usually a one-hot encoded vector (all mass on the correct class), and Q is the softmax or sigmoid output of the model.

The cross-entropy loss formula is:

$$H(P, Q) = - \sum_x P(x) \log Q(x)$$

This formulation is based on information theory, where cross-entropy measures the average number of bits required to code samples of P with a code optimized to Q . When Q equals P , the value represents the entropy of P . If Q differs from P , the cost increases.

Entropy vs. Cross Entropy

While entropy measures the randomness of a true distribution, cross-entropy evaluates the discrepancy between the predicted and true distributions. Their connection is fundamental to the comprehension of model performance in [classification](#).

Cross-entropy can be viewed mathematically as the sum of the entropy of the true distribution and the KL divergence between the true and predicted distributions. That is, it not only reflects the uncertainty of the data but also discourages deviations from correct predictions.

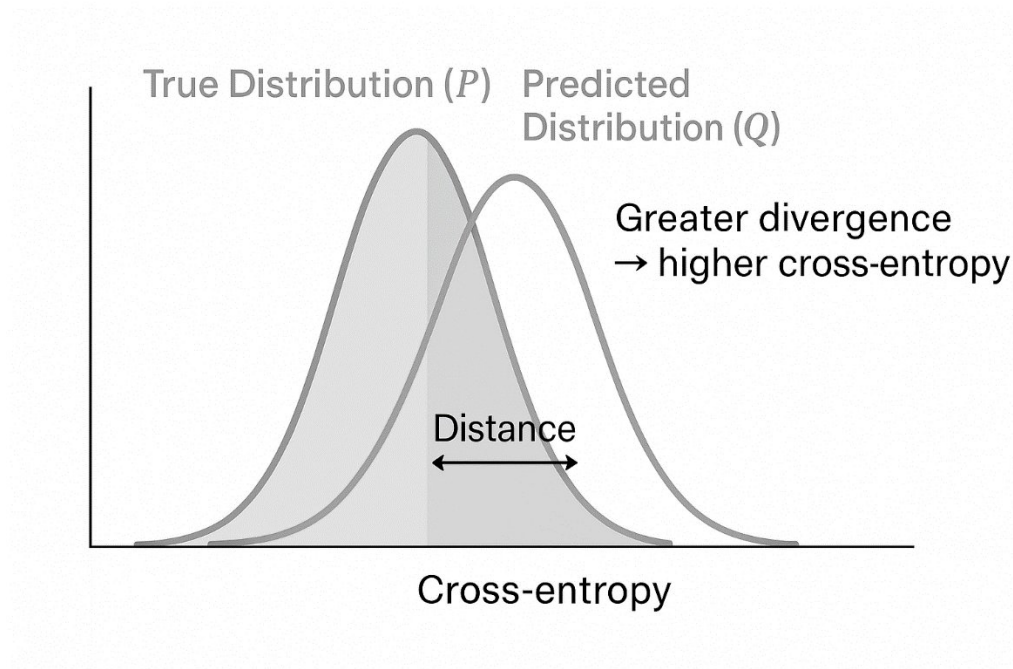


Illustration of cross-entropy as the difference between predicted and actual distributions; a greater mismatch implies a higher loss.

Table 1: Entropy vs. Cross-Entropy.

Aspect	Entropy	Cross-Entropy
Focus	True distribution only (p)	Comparison between true (p) and predicted (q) distributions
Interpretation	Intrinsic unpredictability of outcomes	Loss due to the discrepancy between p and q
Relationship	Base measure of uncertainty	$H(p,q)=H(p)+DKL(p q)$
Relationship	Base measure of uncertainty	$H(p,q)=H(p)+DKL(p q)$
Value Bounds	≥ 0	\geq Entropy; equals Entropy only if $p = q$
Use Case	Measure randomness in the data source	Supervised learning loss function (e.g., classification)
Units	Bits, nats (depends on log base)	Same as entropy (if log base matches)

Intuition (Surprise and “Bits”)

In information theory, rare events convey more information than common ones. For example, a coin landing on its edge, which is rare, tells you more than if it lands on heads, which is common.

Entropy captures this idea by quantifying the average uncertainty in a distribution. Cross-entropy loss extends this further by calculating the number of bits required to represent true outcomes given the distribution predicted by the model.

Training a classifier with cross-entropy loss involves adjusting the distribution Q predicted by the model to represent the true labels with fewer bits. This means the model needs to make correct predictions and assign probabilities that match the true distribution.

Cross-Entropy as a Distance

Although cross-entropy is not a valid distance measure (it is not symmetric), it indicates the direction of distribution mismatch.

Specifically, it measures the inadequacy of a predicted distribution Q to the true one P . The cross-entropy loss increases as Q shifts probability mass toward areas where P is mainly concentrated.

Example

Suppose the true distribution is:

$$P = [0.7, 0.3]$$

Now compare three different model predictions Q:

Table 2: Predictions Q Comparison.

Prediction Q	Cross-Entropy $H(P,Q)$	Interpretation
[0.5, 0.5]	= 0.985	Poor calibration
[0.7, 0.3]	= 0.610	Perfect match (equals entropy)
[0.9, 0.1]	= 1.100	Overconfident in the wrong class

In this case, the second prediction is identical to P, and the resulting loss is the entropy of the real distribution. The remaining branches expand, resulting in an increased average cost of encoding the actual outcome with a mismatched prediction.

Pro Tip: For better feature separation in low-label scenarios, try [contrastive learning](#). It trains the model to group similar inputs closer and separate different ones, improving representation quality even with limited labeled data.

Types of Cross-Entropy Loss Functions

Various classification tasks require customized cross-entropy loss formulations that are more representative of label structure and computing requirements.

The most prominent types address binary classification, multi-class classification, and multi-label classification scenarios.

Choosing the Right Loss Function for Classification

Cross-entropy is a core component of classification models. Its performance depends on the loss function, output layer, [labeling scheme](#), and model architecture. Let's explore its two variants:

Binary Cross-Entropy (BCE): Binary cross-entropy loss is applied in binary classification or multi-label classification, where each output is a separate binary decision. It uses a sigmoid activation function on the output layer, which converts logits to independent predicted probabilities in the range of [0, 1]. Unlike softmax, sigmoid doesn't assume classes are exclusive.

Categorical Cross-Entropy (CCE): Categorical cross-entropy is used in multi-class classification, where the input belongs to exactly one class. It applies a [softmax activation](#) to transform logits into a normalized probability distribution over all classes. This makes the model produce only one correct instance of a class.

Pro Tip: Strong [embeddings](#) boost classification model robustness. Mapping similar items close in embedding space makes your model more efficient and robust, and reduces confusion in class indices.

Binary Cross-Entropy (Log Loss)

Also known as log loss, binary cross-entropy calculates the penalty for predicting a probability that diverges from the true distribution in binary outcomes.

Use Case

It's used in binary classification (two classes) and multi-label classification (independent binary decisions per label). Common applications include:

Recognition of spam (spam or not spam)

Medical diagnosis (whether a disease is present, or not)

Multi-label tagging (e.g., multiple labels on an image)

Output Activation

Requires a sigmoid activation function on the output unit. This converts logits to independent probabilities between the [0,1] range.

Loss Computation

For a single binary output:

$$\text{BCE} = - [y \cdot \log(p) + (1 - y) \cdot \log(1 - p)]$$

Where:

$y \in \{0,1\}$ is the true label

$p \in [0,1]$ is the predicted probability after sigmoid

In multi-label cases, it is calculated separately for each label and then averaged across outputs.

Behavior

Non-Exclusive Predictions: The classes are treated separately (e.g., with sigmoid activation), so the predicted probabilities do not need to add up to 1. This makes BCE suitable for multi-label classification.

Class Imbalance Weighted: Loss for each class can be weighted, often by the inverse of class frequency.

Multi-Label Support: The loss sums independent binary cross-entropy losses per class, with inputs labeled with any set of active labels, not just one winner.

Error Sensitivity in Prediction: When the predicted probability is significantly different from the actual label (e.g., predicting 0.1 when the true label is 1), the loss is high. When the prediction is similar to the actual value (e.g., 0.9 vs 1), the loss is small. This steep gradient assists in correcting significant errors within a short time.

Example

[CheXNet](#), a neural network (121-layer DenseNet), was trained on more than 100,000 chest X-rays to detect 14 diseases. It used weighted binary cross-entropy loss to address the issue of class imbalance, with rare diseases getting larger weights.

The performance of the resulting model had an F1 score of 0.435, which was better than the F1 of the average radiologist of 0.387.

Categorical Cross-entropy

Use Case

Categorical cross-entropy is the standard loss function for the multi-class classification tasks in which each input has a single, pre-defined label among N mutually exclusive classes. For example, in digit recognition (e.g., MNIST), each image is labeled as a single digit from 0 to 9.

It is the default option when you have a single correct class and want to fit a probability distribution over all possible classes.

The typical applications are:

Image classification (e.g., CIFAR-10, ImageNet)

Sentiment analysis (positive/neutral/negative)

Language modeling (predicting the next word using a vocabulary)

Output Activation

The CE model requires a softmax activation function for its output layer. Softmax is a function that transforms raw logits into a probability distribution with a sum of 1. This makes the model outputs mutually exclusive and appropriate for single-label classification activities.

Loss Computation

The categorical cross-entropy loss is computed as:

$$\text{CE} = - \sum_{j=1}^C y_j \cdot \log(p_j)$$

Where:

C - total number of classes

y_j - one-hot encoded true label vector

p_j - predicted probability for class j

Behavior

Inter-Class Competition: Drives competition between classes. The higher the probability for one class, the lower the probabilities for others.

Confidence Maximization: The model maximizes confidence in the correct class while minimizing it for others.

High-Confidence Error Sensitivity: Sensitive to inaccurate high-confidence predictions, which is useful for training well-calibrated classifiers.

Class Imbalance Mitigation: In imbalanced classes, weighting or sampling may be required to prevent bias toward frequent classes.

Example

Sentiment classification of movie reviews is a complicated process because people express their views in subtle and varied ways. A recent study used a BERT-based [neural network](#) with a bi-directional LSTM to incorporate forward and backward context.

The training of this classification model involves categorical cross-entropy to facilitate confident prediction of the true label among various sentiment classes. The parameters of the model are optimized to minimize the entropy loss in all the predictions.

In binary sentiment classification (positive vs. negative), the model achieved 97.7% accuracy. While in fine-grained multi-class classification (e.g., 5 classes of sentiment), it achieves 59.5% accuracy.

Binary vs Categorical Cross-Entropy

The following table highlights key differences between binary cross-entropy (BCE) and categorical cross-entropy (CCE).

Table 3: Binary vs Categorical Cross-Entropy.

Aspect	Binary Cross-Entropy (BCE)	Categorical Cross-Entropy (CCE)
Use Case	For binary classification and multi-label classification tasks	For multi-class classification tasks where each input belongs to one class
Output Layer Activation	Sigmoid activation function, independent probabilities	Softmax activation generates a probability distribution that sums to 1
Predicted Probabilities	Each class prediction is independent	Classes compete for probability mass
True Label Format	Single binary value (0 or 1), or multiple binary labels	One-hot encoded vector
Example Task	Spam detection, disease presence, and multi-label image tagging	Digit recognition, image classification, sentiment analysis

Table 3: Binary vs Categorical Cross-Entropy.

Aspect	Binary Cross-Entropy (BCE)	Categorical Cross-Entropy (CCE)
Sensitivity to Error	Sensitive to both underconfident and overconfident predictions	Highly sensitive to high-confidence errors
Loss Function Type	Cross-entropy loss variant for binary cross-entropy (or log loss)	Cross-entropy loss for multi-class classification

Cross-Entropy vs. Mean Squared Error (MSE)

A common point of confusion in classification tasks is whether to use cross-entropy loss or mean squared error. Here's how they compare across key dimensions.

Table 4: Cross-Entropy vs. Mean Squared Error.

Aspect	Cross-Entropy Loss	Mean Squared Error (MSE)
MSE Recap	Not applicable, cross-entropy loss is designed for classification tasks involving probability distributions.	Calculates the mean of squared differences between predicted and actual values.
Classification Problem	Excels in classification problems, especially with softmax or sigmoid activation functions that output predicted probabilities.	Performs poorly in classification, doesn't penalize confident incorrect classes, leading to vague decision boundaries.
Gradient Dynamics	Delivers strong, directional gradients for incorrect predictions, avoids gradient saturation in deep neural networks.	Gradients diminish when predicted and actual probabilities are close, leading to learning stagnation and ineffective updates.
Performance	Faster convergence, better accuracy, and calibration in classification models across benchmark datasets.	Weak gradients misalign optimization, leading to poor classification performance.
Calibration	Produces probability distributions interpretable as confidence levels; suitable for probabilistic classification.	Outputs are unbounded and not normalized, and lack a probabilistic foundation.

Cross-Entropy Loss vs. Hinge Loss

To understand when to use cross-entropy loss or hinge loss, especially in classification tasks, here's a technical comparison of the key differences:

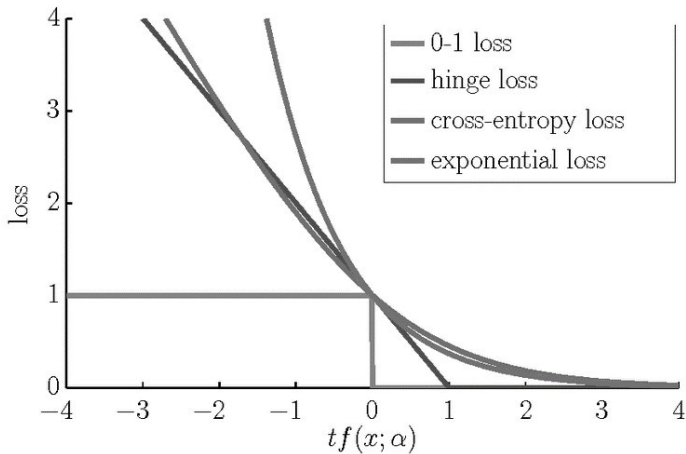


Figure 2: [Loss function comparison: Hinge loss penalizes low-margin predictions, while cross-entropy and exponential losses heavily penalize confident mistakes; 0-1 loss offers no gradient for learning.](#)

Table 5: Cross-entropy Loss vs. Hinge Loss.

Aspect	Cross-Entropy Loss	Hinge Loss
Hinge Loss Recap	Not applicable	A margin-based loss function originally designed for support vector machines (SVMs).
Probability vs Margin	Focuses on predicted probability distribution using softmax or sigmoid activation.	Maximizes the margin between the correct and incorrect classes; doesn't output probabilities.
Differentiability	Fully differentiable, ideal for gradient descent optimization in neural networks.	Not fully differentiable at the margin; may require sub-gradient methods.
Multiclass and Implementation	Easily supports multi-class classification tasks with categorical cross-entropy.	Needs extensions like multiclass hinge loss or one-vs-rest strategies for multiclass support.
Performance	Delivers better accuracy and calibration on many classification models, especially deep nets.	May struggle with probability calibration and is less effective in deep learning scenarios.

Table 5: Cross-entropy Loss vs. Hinge Loss.

Aspect	Cross-Entropy Loss	Hinge Loss
Use Cases	Used in binary classification, multi-class classification, NLP, and image classification.	Primarily used in SVMs and simpler margin-based classifiers; less common in modern deep networks.

Cross-Entropy Loss vs. Negative Log-Likelihood (NLL)

Cross-entropy and negative log likelihood loss are usually the same in supervised classification tasks. NLL is the negative log-probability of the correct class of the model. Cross-entropy is a generalization of this concept, the expected log-loss with respect to a true probability distribution (typically one-hot).

The implementation of cross-entropy loss is usually numerically safer due to the fusion of log-softmax and NLL steps. The separate computation of softmax and log can be less stable when using NLL.

Both maximize the log-likelihood of the correct label, which strengthens the same statistical principle. But in [end-to-end pipelines](#), cross-entropy is more commonly used because it has implicit preprocessing.

When Might You Not Use Cross-Entropy?

Cross-entropy loss isn't suitable for regression with continuous targets. Instead, use MSE or MAE. It is sensitive to noisy labels or outliers, as incorrect confident predictions are heavily penalized, risking unstable training.

Additionally, in these situations, more robust regression techniques or other loss functions can be appropriate. These reduce the impact of overconfident mistakes and prevent training collapse.

In the case of noisy target labels or [significant class imbalance](#), focal loss is more suitable. Focal loss alters the cross-entropy by reducing the weight of easy-to-classify examples and increasing the loss of hard-to-classify examples.

This prompts the model to prioritize complex samples, enhancing performance in skewed or noisy data without compromising the essence of cross-entropy.

Loss Functions Comparison Table

The following table provides a technical comparison of common loss functions used in classification tasks, highlighting their strengths, weaknesses, and ideal use cases:

Table 6: Loss Functions Comparison.

Loss Function	Used For	Pros	Cons
Cross Entropy (Log Loss)	Binary or multi-class classification tasks with a normalized output layer	Smooth optimization; probabilistic; strong gradients; based on information theory	Sensitive to incorrect classes; may require label smoothing for noisy data

Table 6: Loss Functions Comparison.

Loss Function	Used For	Pros	Cons
Mean Squared Error (MSE)	Regression tasks and some scoring in classification	Simple, convex, widely understood, suitable for continuous predicted values	Poor fit for classification tasks; lacks probability distribution; weak gradients
Hinge Loss	Margin-based classification models (e.g., SVM)	Encourages clear class separation in two classes; convex	Non-probabilistic; hard to optimize in neural network training
0-1 Loss	Benchmark for classification accuracy	Directly reflects correct vs. incorrect predictions	Not usable with gradient descent; non-convex; only for evaluation
Exponential Loss	Used in boosting (e.g., AdaBoost)	Heavily penalizes incorrect classes; good for difficult examples	Sensitive to outliers; rarely used in modern deep learning workflows
Negative Log-Likelihood (NLL)	Probabilistic classification models using true label	Equivalent to cross-entropy loss; interpretable outputs	Less numerically stable unless paired with softmax; sensitive to mislabeling

Cross-Entropy Loss in Machine Learning and Deep Learning

The classification problem in machine learning is generally framed as the task of aligning probability distributions through cross-entropy loss. The model is designed to predict a distribution over classes, with training focused on minimizing the distance to the true distribution.

In [logistic regression](#), the cross-entropy loss arises directly from maximizing the Bernoulli likelihood. In logistic regression, we model the probability of the positive class:

$$P(y = 1|x; \theta) = \hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}}, \quad \text{where } z = x^\top \theta$$

For binary labels $y \in \{0,1\}$, the Bernoulli likelihood of a single prediction is:

$$L(\theta) = \hat{y}^y (1 - \hat{y})^{1-y}$$

Taking the negative log-likelihood to get the loss:

$$-\log L(\theta) = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$$

So, the binary cross-entropy loss is:

$$\text{BCE} = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$$

This principle extends to deep learning, where neural networks parameterize the likelihood function, and optimization is performed via stochastic gradient descent or its variants.

Compared to losses like MSE, cross-entropy provides more precise gradients and faster convergence. It also offers better-calibrated probability estimates, which improve training dynamics.

Pro Tip: You don't always need labels to pretrain powerful models. [Self-supervised learning](#) lets the model learn useful features on its own, which can later be fine-tuned with cross-entropy loss.

Cross Entropy in Pytorch

In PyTorch, cross-entropy loss works directly on raw logits to measure the prediction error at the class level and does not require explicit probability normalization.

It combines `log_softmax` and negative log-likelihood internally, offering both computationally efficient and numerically stable. These features make it a commonly used loss function in classification tasks.

Implementation and Use Cases in Neural Networks

Cross-entropy loss is common in training neural networks to classify. Now, let us see how it is applied in common cases and use examples.

Table 7: Cross-entropy Implementations.

Task Type	Output Layer	Loss Function
Binary Classification	Single output + sigmoid	<code>nn.BCEWithLogitsLoss()</code>
Multi-Class	<code>num_classes</code> outputs (raw logits)	<code>nn.CrossEntropyLoss()</code>
Multi-Label	<code>N</code> outputs + sigmoid per label	<code>nn.BCEWithLogitsLoss()</code>

Framework Implementations

PyTorch loss functions used in classification tasks are closely tied to the design of the output layer. Let's look at the implementation of multi-class classification:

This is used for single-label, multi-class problems, where each input belongs to exactly one of CCC classes.

Input (logits): shape [batch_size,num_classes][\text{batch_size}, \text{num_classes}][batch_size,num_classes], raw scores (no softmax applied)

Target: shape [batch_size][\text{batch_size}][batch_size], class indices (integers)

Internally: combines LogSoftmax + NLLLoss

Now, let's look at the implementation using PyTorch:

```
import torch
import torch.nn as nn

logits = torch.tensor([[2.1, 0.1, -1.0], [1.0, 3.0, 0.2]]) # shape [2, 3]
targets = torch.tensor([0, 1]) # class indices

loss_fn = nn.CrossEntropyLoss()
loss = loss_fn(logits, targets)

print(f"Loss: {loss.item():.4f}")
```

Loss: 0.1725

Figure 3:

Cross-entropy loss calculation.

This result (Loss: 0.1725) indicates that the model is making correct predictions with reasonably high confidence, though it is not perfectly accurate.

Example Use Cases

With the fundamentals in place, let's see how it can be applied across a variety of real-world use cases.

Table 9: Cross-entropy Use-Cases.

Task / Domain	Description	Loss Function	Output Layer	Example Dataset
Image Classification	Single-label image classification	nn.CrossEntropyLoss()	Softmax	CIFAR-10, ImageNet
Language Modeling	Next-token prediction in	nn.CrossEntropyLoss()	Linear + Softmax	WikiText,

Table 9: Cross-entropy Use-Cases.

Task / Domain	Description	Loss Function	Output Layer	Example Dataset
	sequences			OpenWebText
Semantic Segmentation	Per-pixel class prediction	nn.CrossEntropyLoss()	Softmax per pixel	Cityscapes, PASCAL VOC
Object Detection	Classify objects in bounding boxes	CrossEntropy + RegLoss	Softmax/Sigmoid	COCO, Pascal VOC
Generative Models	Discriminator classification	nn.BCEWithLogitsLoss()	Sigmoid	CelebA, MNIST (GANs)
Ensemble Models	Averaging diverse classifiers	nn.CrossEntropyLoss()	Varies	CIFAR-100, TinyImageNet

Behavior During Training

Cross-entropy loss generates high gradients on confidently wrong predictions, which allows the model to quickly rectify its errors. Gradients decrease as the model's predictions approach the true labels, which results in stable convergence.

This process helps the model concentrate its probabilities on the correct class, which improves both training efficiency and generalization.

Integration with Gradient Descent

Cross-entropy loss generates steep, non-zero gradients even in the case of incorrect classes. This makes it highly suitable to train classification models using gradient descent. As a result, the model converges faster and learns more effectively, especially in deep learning systems.

Cross-entropy, unlike MSE, does not suffer from the problem of gradient saturation, and the signal propagates strongly through backpropagation. This behavior allows neural networks to better match predicted probabilities to true labels, which speeds up learning and boosts accuracy.

The diagram below highlights the loss and gradient behavior during binary classification:

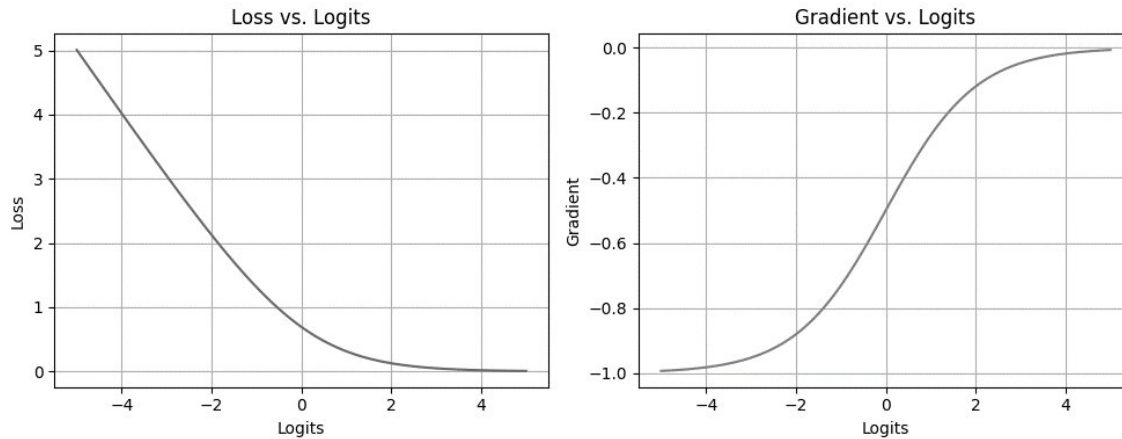


Figure 4: Loss vs gradients representation.

This diagram shows the behaviour of cross-entropy loss and its derivative during binary classification training. The loss punishes incorrect high-confidence predictions (left plot), encouraging the model to be more confident in correct predictions.

Meanwhile, the gradient (right plot) has the largest absolute value when the model is most incorrect, which guarantees successful updates at the initial stages of training. Both loss and gradient fade out as predictions get better, stabilizing learning.

Real-World Example

In this example, we train a simple convolutional neural network on the CIFAR-10 dataset using PyTorch with cross-entropy loss.

Code

```

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Load CIFAR-10 dataset
transform = transforms.Compose([transforms.ToTensor()])
train_dataset = torchvision.datasets.CIFAR10(root='./data', train=True, transform=transform, download=True)
train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=32, shuffle=True)

# Define a simple CNN model
class SimpleCNN(nn.Module):
    def __init__(self, num_classes=10):
        super(SimpleCNN, self).__init__()
        self.net = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(16 * 32 * 32, num_classes)
        )

    def forward(self, x):
        return self.net(x)

# Initialize model, loss function, and optimizer
model = SimpleCNN(num_classes=10).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

```

Figure 5: Implementation of a convolutional neural network on the CIFAR-10 dataset.

Result

```

100%|██████████| 170M/170M [00:03<00:00, 48.2MB/s]
Epoch [1/5], Loss: 1.5404
Epoch [2/5], Loss: 1.2538
Epoch [3/5], Loss: 1.1573
Epoch [4/5], Loss: 1.0894
Epoch [5/5], Loss: 1.0331

```

During early training, model predictions are often inaccurate. As training progresses and weights adjust, the loss steadily decreases as the model learns to align its predictions with the correct classes. Here is the loss curve during learning:

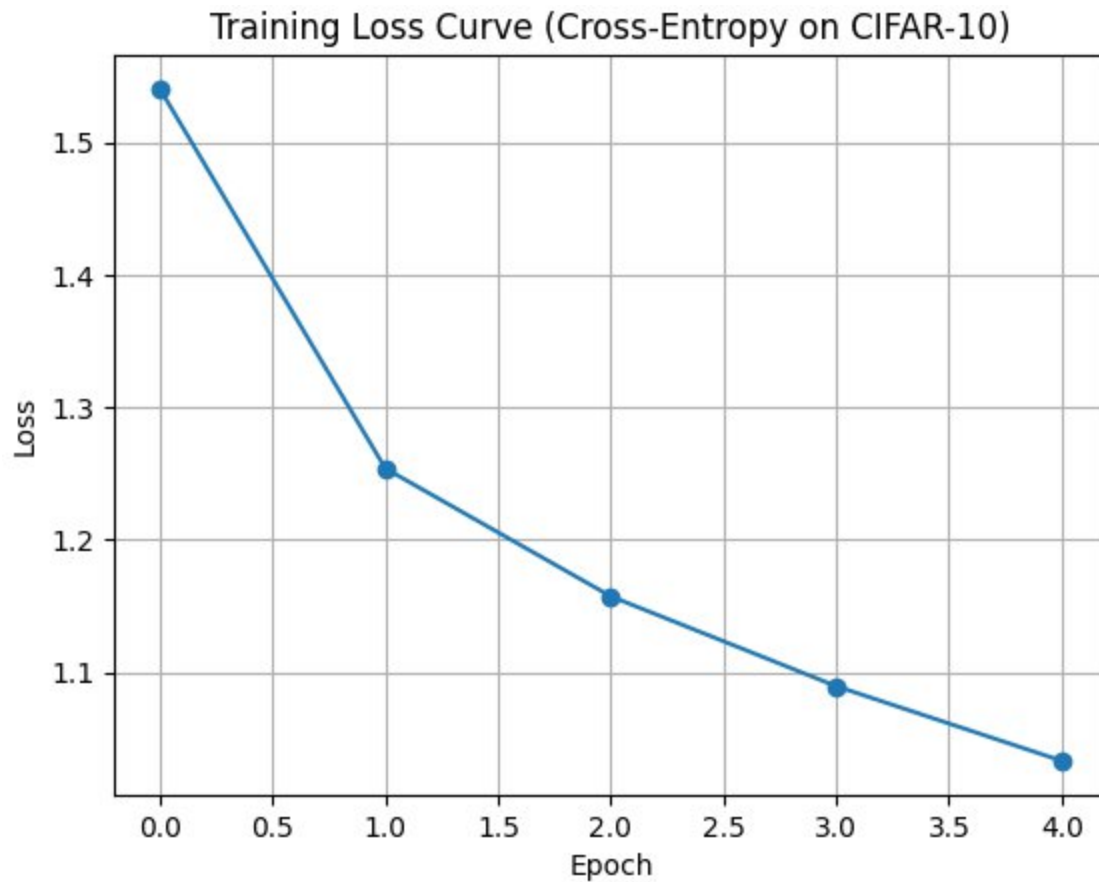


Figure 6: Training loss curve

The steadily decreasing loss across epochs indicates effective learning and model convergence.

Cross-Entropy Loss Behavior with Predicted Probabilities

Cross-entropy loss offers a smooth, differentiable signal indicating how well predicted probabilities match true labels. It rapidly decreases as the correct class probability increases, rewarding confident, accurate predictions.

Binary Case Behavior

In binary classification, where the model's probability is close to the true label (0 or 1), the loss will be close to zero. But if the model assigns a high estimated probability to the wrong class, the penalty grows steeply.

This asymmetry helps the model overcome overconfidence during the initial stages of training data.

Multi-Class Behavior

When a high probability is incorrectly assigned to the wrong class in multi-class scenarios, the loss increases. Cross-entropy encourages the model to adjust probabilities, ensuring that the final layer's confidence is focused on the correct target.

Correct class vs Incorrect classes: The loss function is minimized when the model parameters drive the network to assign complete confidence to the correct class. On the other hand, when the probability is placed on the wrong classes (i.e., over-belief), the loss is higher.

Gradient Perspective: Due to its sharp slope around high-confidence errors, cross-entropy ensures that any incorrect prediction will induce significant updates through gradient descent. This property avoids vanishing gradients and supports solid early optimization.

Confidence Penalty and Saturation: False positives are costly because training with cross-entropy penalizes overconfidence in wrong categories more than underconfidence in correct ones. Plateaus and saturation occur when predicted values near certainty (0 or 1). This results in smaller updates that prevent late-stage training instability.

Edge Cases: Cross-entropy exhibits extreme values on the edges of the predicted probabilities. Minimum loss occurs when the label and the predicted class are equal with a probability of 1. Maximum loss is unbounded when the estimated probability of the correct class is near zero, due to the logarithmic penalty on confident mistakes.

Training Interpretation

Cross-entropy is used as a training signal, and points out where the model is uncertain or confidently wrong. This feedback process refines the model parameters, which allows for performance optimization even with noisy or complicated data.

Cross-Entropy and Model Uncertainty

Softmax produces uncertain, nearly uniform probabilities, which increases loss even if the top guess is correct. This indicates the model's reluctance and highlights the need to define decision boundaries better.

Cross-Entropy and KL Divergence

The minimization of cross-entropy with one-hot labels is mathematically identical to minimizing KL divergence. From an information theory perspective, this involves reducing the number of additional bits of information needed to define the actual labels using the estimated distribution.

How Lightly AI Improves Cross-Entropy Loss Training

Cross-entropy performance is closely related to data quality, representation learning, and training efficiency.

This is where [Lightly AI](#) comes in, not to replace cross-entropy, but to improve the conditions in which it is optimized. Let's have a look:

LightlyTrain

[LightlyTrain](#) is a self-supervised pretraining method that does not require labeled data to train CNNs and Vision Transformers. It uses contrastive learning techniques like SimCLR and [DINOv2](#) to create effective feature extractors from raw images.

When these pretrained backbones are later fine-tuned with cross-entropy loss on downstream classification tasks, the models tend to optimize better. This approach becomes especially useful when you're dealing with small or imbalanced labeled datasets, because it helps the training process stay more consistent.

LightlyEdge

[LightlyEdge](#) is a real-time edge SDK to capture high-value training data. It assists in implementing smarter data collection strategies in distributed or production environments.

For cross-entropy trained models:

Training data is aligned with real-world distributions

Duplicate low-value samples are removed, reducing the risk of overfitting

Loss values are prioritized for rare or edge-case inputs that drive high loss values

LightlyEdge ensures that your classification model is being trained on the right data, where cross-entropy loss is most important.

Conclusion

Cross-entropy loss optimizes classification tasks by minimizing the difference between predicted and true label distributions. It works well with softmax activations, preserving distributional semantics, unlike regression losses.

This results in more stable convergence, better label sparsity, and refined confidence calibration. As a result, cross-entropy remains the default loss function for deep learning models.

Sparse Categorical Cross-Entropy

Sparse Categorical Cross-Entropy Loss is a variant of the Categorical Cross-Entropy Loss, specifically designed for cases where the true labels are integers rather than one-hot encoded vectors. It's commonly used in multi-class classification tasks where each example belongs to only one class.

For a single training example, the Sparse Categorical Cross-Entropy Loss is defined as:

$$L(y, \hat{y}) = -\log(\hat{y}_y)$$

Mathematically

Where:

- y is the true class label (an integer).
- \hat{y} is a vector representing the predicted probability distribution over all classes.

- \hat{y} denotes the predicted probability assigned to the true class label.

Points to be Remember

1. When Mean Absolute Error (MAE) is used as the loss function, the output node of the neural network should have a linear activation function.
2. When Binary Cross Entropy is used as the loss function, the output node of the neural network should have a Sigmoid activation function.
3. When Categorical Cross Entropy is used as the loss function, the output node of the neural network should have a softmax activation function. And the number of output nodes depends on the categories in the output column.

Conclusion:

Regression loss Functions are: MSE, MAE, Huber Loss.

Classification loss Functions are: Binary Cross Entropy, Categorical Cross Entropy, Sparse Cross Entropy (when there are too many categories on the output column)