Deep Learning

Made by: Reda Mountassir

Chapter 1 : Deep Learning Basics

Table of contents

    - Adam
    - Other Optimizers
- 3.4 Regularization Techniques
    - Regularization
    - Dropout
    - Early Stopping

4. Deep Learning Basics

- 4.1 What Makes a Neural Network "Deep"?
- 4.2 Importance of Non-Linearities
- 4.3 The Universal Approximation Theorem
- 4.4 Challenges in Training Deep Networks
    - Vanishing and Exploding Gradients
    - Overfitting and Underfitting

5. Model Evaluation and Validation

- 5.1 Training, Validation, and Test Sets
- 5.2 Metrics for Classification and Regression
    - Accuracy, Precision, Recall, F1-Score
    - Mean Absolute Error, Mean Squared Error

6. Building and Training Neural Networks

- 6.1 Defining the Architecture
- 6.2 Compiling the Model
- 6.3 Training the Model
- 6.4 Evaluating the Model
- 6.5 Making Predictions

7. Conclusion

1-Introduction to Neural Networks and Deep Learning

The easiest way to think about AI, machine learning, deep learning and neural networks is to think of them as a series of AI systems from largest to smallest, each encompassing the next.

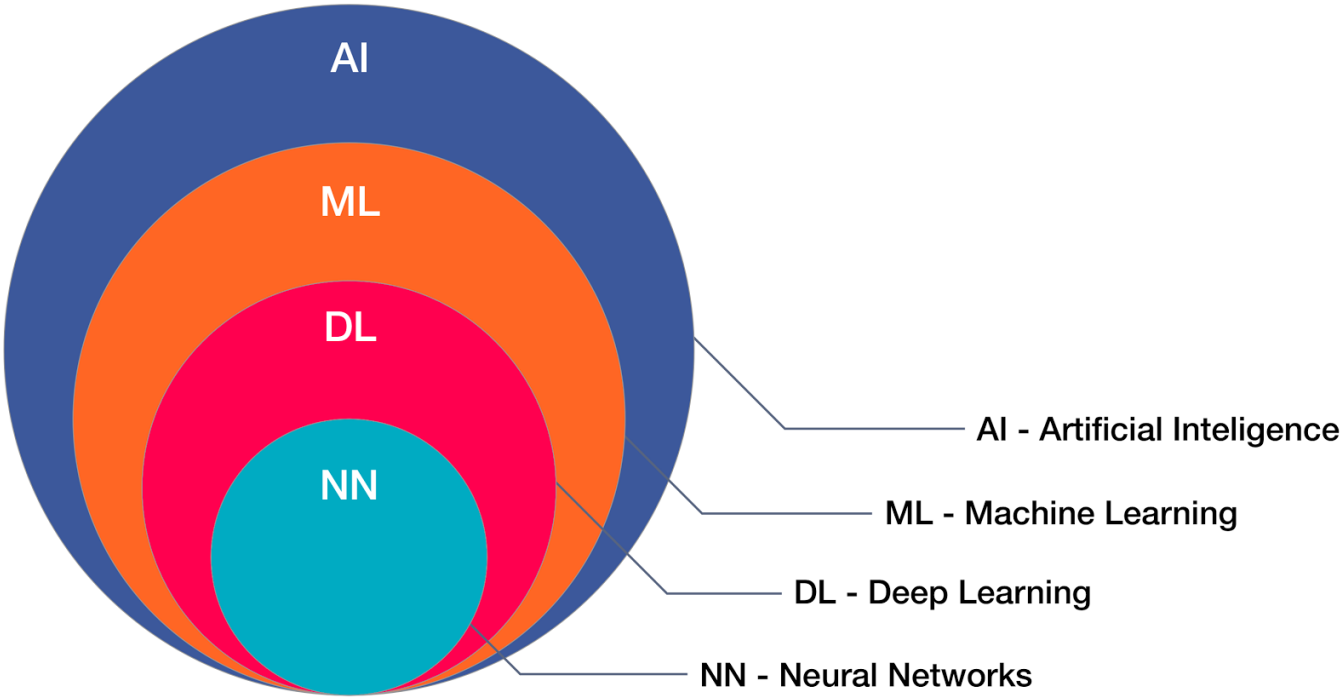


Figure 1 : AI vs Machine Learning vs Deep Learning vs Neural Network

AI is the overarching system. Machine learning is a subset of AI. Deep learning is a subfield of machine learning, and neural networks make up the backbone of deep learning algorithms. It's the number of node layers, or depth, of neural networks that distinguishes a single neural network from a deep learning algorithm, which must have more than three.

1.1-What is Artificial Intelligence?

Artificial intelligence or AI, the broadest term of the three (ML,DL,NN), is used to classify machines that mimic human intelligence and human cognitive functions like problem-solving and learning. AI uses predictions and automation to optimize and solve complex tasks that humans have historically done, such as facial and speech recognition, decision-making and translation.

There's Three main types of AI:

- Artificial Narrow Intelligence (ANI)

> We define weak AI by its ability to complete a specific task, like winning a chess game or identifying a particular individual in a series of photos. Natural language processing and computer vision, which let companies automate tasks and underpin chatbots and virtual assistants such as Siri and Alexa, are examples of ANI.

- Artificial General Intelligence (AGI)

> AGI would perform on par with another human

- Artificial Super Intelligence (ASI)

> *also known as superintelligence* would surpass a human's intelligence and ability. Neither form of Strong AI exists yet, but research in this field is ongoing.

1.2-What is Machine Learning?

Machine learning is a subset of AI that allows for optimization. When set up correctly, it helps you make predictions that minimize the errors that arise from merely guessing.

There's Many types of Machine learning:

- Supervised Learning

> Supervised learning is the most common type of machine learning. It uses labeled data to train algorithms to classify data or predict outcomes accurately. Supervised learning is used in applications where historical data predicts likely future events.

- Unsupervised Learning

> Unsupervised learning is used when the information used to train is neither classified nor labeled. The system tries to learn without a teacher. It uses algorithms such as clustering or dimensionality reduction to identify patterns in data.

- Semi-Supervised Learning

> Semi-supervised learning is a combination of supervised and unsupervised learning. It uses a small amount of labeled data and a large amount of unlabeled data. This method is useful when the cost associated with labeling is too high to allow for a fully supervised model.

- Reinforcement Learning

> Reinforcement learning is often used for robotics, gaming, and navigation. It involves an algorithm that learns to perform an action from experience. The algorithm receives feedback in the form of rewards or penalties as it navigates its problem space.

- Online Learning

> Online learning is a type of ML where a data scientist updates the ML model as new data becomes available.

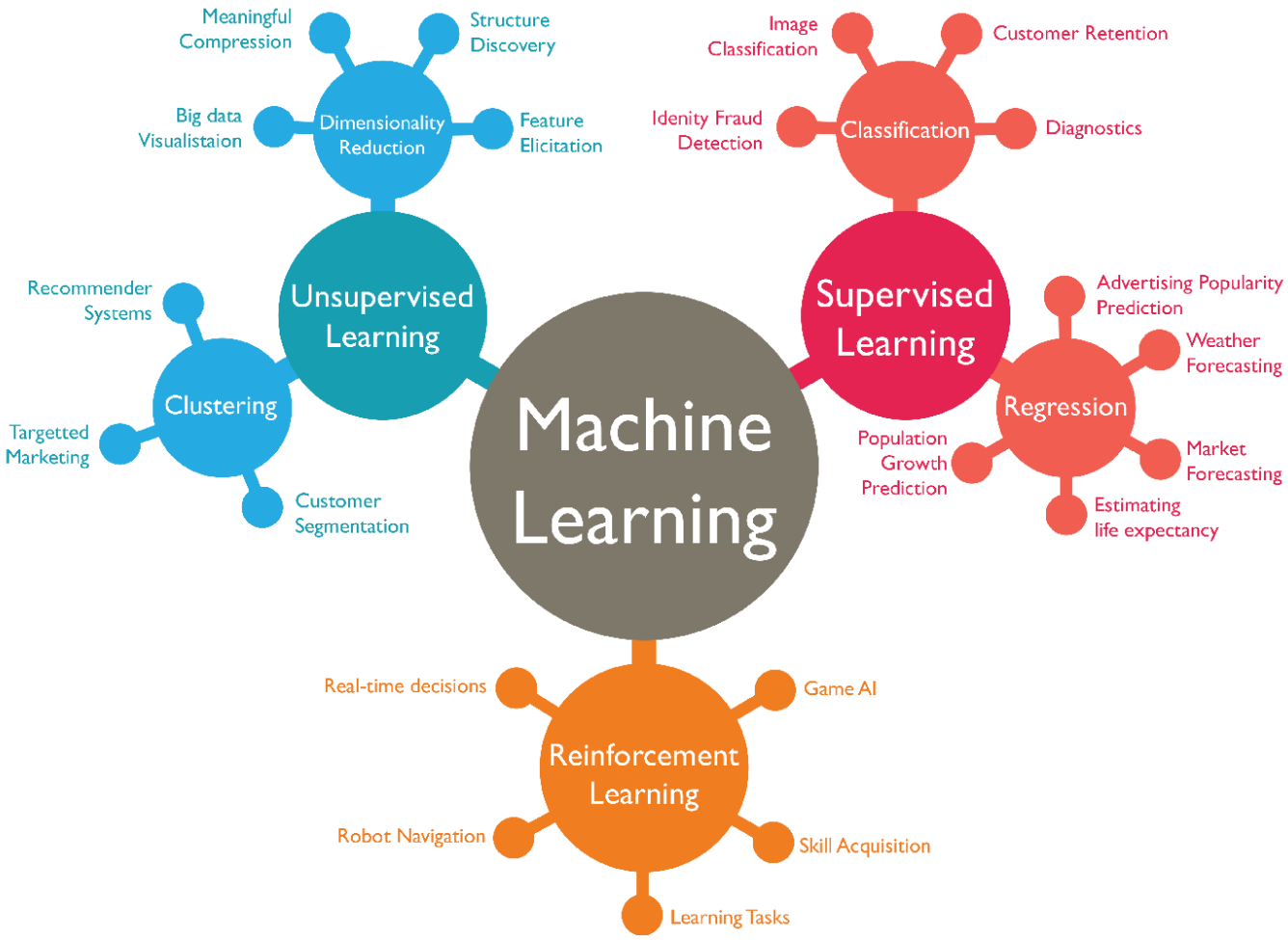


Figure 2 : Types of Machine Learning

1.3-What is Deep Learning?

deep learning is a subset of machine learning. The primary difference between machine learning and deep learning is how each algorithm learns and how much data each type of algorithm uses.

Deep learning automates much of the feature extraction piece of the process, eliminating some of the manual human intervention required. It also enables the use of large data sets, earning the title of scalable machine learning.

Observing patterns in the data allows a deep-learning model to cluster inputs appropriately.

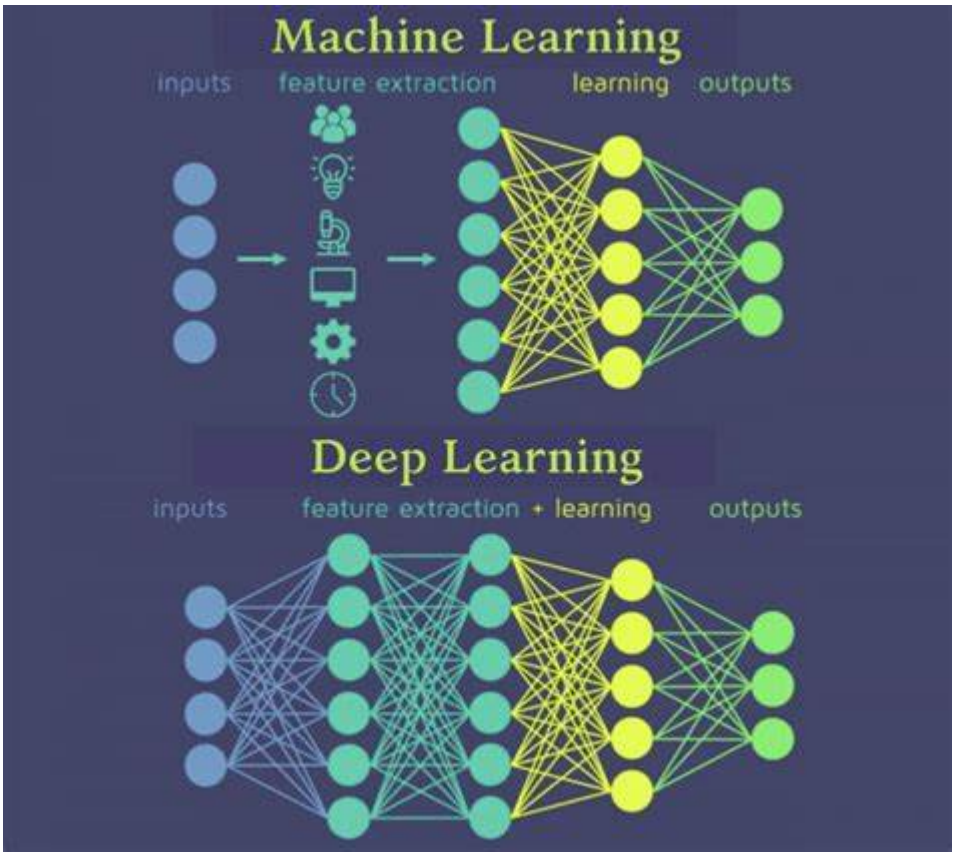


Figure 3 : Deep Learning vs Machine Learning

History and Evolution of Neural Networks

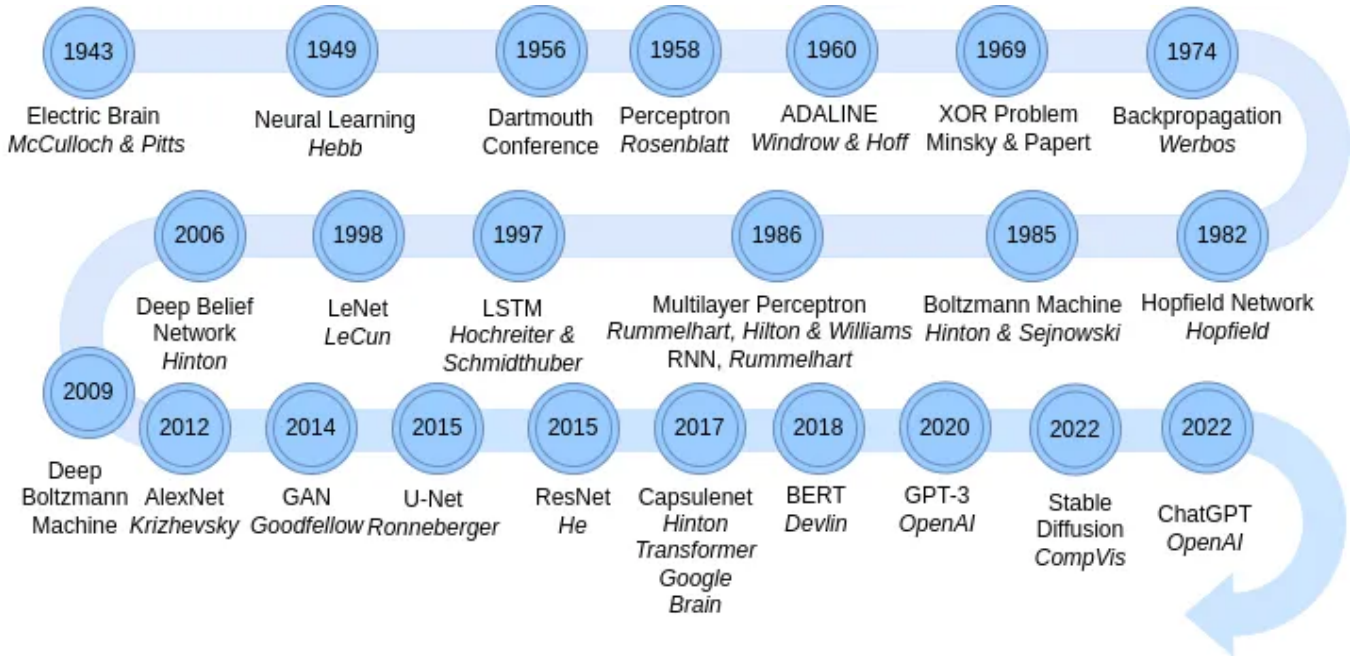


Figure 4 : History of Deep Learning

1. **The perceptron**, developed by Frank Rosenblatt in the 1957, It was a single-layer model designed to mimic the behavior of a biological neuron. The perceptron could process inputs, apply weights to them, and produce an output based on an **activation function**.
2. Despite the initial enthusiasm, the 1970s and 1980s witnessed a period known as the **"AI winter"** ,where progress in neural networks slowed down due to limitations in computing power, data

availability, and theoretical understanding.

3. The introduction of **multi-layer perceptrons (MLPs)** in the 1980s was a breakthrough moment in neural network history. By stacking multiple perceptron layers, researchers could tackle more complex problems. The addition of hidden layers introduced **non-linearity**, enabling neural networks to approximate any function. However, training deeper networks became a challenge due to the **vanishing gradient problem**.
4. The breakthrough that reignited interest in neural networks came with the development of the **backpropagation algorithm**. In the late 1980s, researchers demonstrated how to efficiently update the weights of each neuron in a multi-layer network, significantly improving training speed and accuracy. **Backpropagation** was a game-changer, enabling deeper networks to learn complex representations from data.
5. The 1990s brought about another crucial development with the inception of **Convolutional Neural Networks (CNNs)**. Inspired by the human visual system, CNNs revolutionized image recognition tasks by preserving spatial hierarchies. **LeNet-5**, introduced by Yann LeCun in 1998, laid the groundwork for modern CNNs, becoming a cornerstone in computer vision.
6. Addressing the challenges of sequential data, **Recurrent Neural Networks (RNNs)** emerged in the 1980s. However, it was the introduction of **Long Short-Term Memory (LSTM)** networks by Sepp Hochreiter and Jürgen Schmidhuber in 1997 that mitigated the **vanishing gradient problem**, making RNNs more effective for processing and understanding sequential information.
7. The 2010s marked the era of deep learning, where neural networks with many hidden layers — referred to as deep neural networks — became the driving force behind AI advancements.
8. In recent years, **the transformer architecture** emerged as a disruptive force in natural language processing. The architecture, exemplified by models like **BERT** and **GPT-3**, pushed the boundaries of language understanding and generation.

Here's bunch of papers about each of the mentioned topics:

- Tranformer
- CNN
- LSTM
- All Papers Roadmap

Applications of Neural Networks and Deep Learning

1. Feedforward Neural Networks (FNN)

- **Description**: The simplest type of artificial neural network, where connections between the nodes do not form a cycle. Each layer is fully connected to the next one.
- **Applications**:
    - Image Classification: Classifying images into different categories.
    - Regression Analysis: Predicting a continuous output variable.
    - Spam Detection: Classifying emails as spam or not spam.

2. Convolutional Neural Networks (CNN)

- **Description**: Designed to process and analyze visual data. They use convolutional layers that apply filters to capture spatial hierarchies in images.

- **Applications**:
    - Image and Video Recognition: Object detection, facial recognition, and video analysis.
    - Medical Image Analysis: Analyzing MRI scans, X-rays, and other medical images.
    - Autonomous Vehicles: Recognizing and interpreting visual data from cameras for navigation.

3. Recurrent Neural Networks (RNN)

- **Description**: Designed for sequential data, where the output from previous steps is fed as input to the current step. They have an internal state that can persist.
- **Applications**:
    - Natural Language Processing (NLP): Language modeling, text generation, and translation.
    - Speech Recognition: Converting spoken language into text.
    - Time Series Prediction: Forecasting stock prices, weather, and other sequential data.

4. Long Short-Term Memory Networks (LSTM)

- **Description:** A type of RNN that can learn long-term dependencies. They are designed to avoid the long-term dependency problem, making them effective for longer sequences.
- **Applications:**
    - NLP: Text generation, language translation, and sentiment analysis.
    - Speech Recognition: More effective in handling longer sequences of speech.
    - Anomaly Detection: Detecting anomalies in time-series data.

5. Gated Recurrent Units (GRU)

- **Description**: Similar to LSTM but with a simpler structure. GRUs combine the forget and input gates into a single update gate.
- **Applications:**
    - NLP: Similar applications as LSTMs but more computationally efficient.
    - Time Series Prediction: Effective in forecasting and anomaly detection.

6. Generative Adversarial Networks (GAN)

- **Description:** Consist of two networks, a generator and a discriminator, that compete against each other. The generator creates data, and the discriminator evaluates it.
- **Applications:**
    - Image Generation: Creating realistic images, such as deepfakes.
    - Data Augmentation: Generating additional training data for machine learning models.
    - Art and Design: Creating artistic images, music, and other creative content.

7. Autoencoders

- **Description:** Unsupervised neural networks used for learning efficient codings of input data. They consist of an encoder and a decoder.
- **Applications:**
    - Dimensionality Reduction: Reducing the number of features in a dataset.
    - Image Denoising: Removing noise from images.
    - Anomaly Detection: Identifying unusual patterns in data.

8. Transformer Networks

- **Description:** Designed to handle sequential data using self-attention mechanisms, allowing them to process entire sequences simultaneously.
- **Applications:**
    - NLP: Machine translation, text summarization, and question answering.
    - Image Processing: Vision transformers for image classification.
    - Time Series Forecasting: Predicting future values in a sequence.

9. Bidirectional LSTM (Bi-LSTM)

- **Description:** A Bidirectional Long Short-Term Memory (Bi-LSTM) network is an extension of the standard LSTM. It processes data in both forward and backward directions, allowing the model to have information from both past and future contexts.

- **Applications:**

    - Natural Language Processing (NLP):
        - Text Classification: Categorizing text into predefined categories.
        - Named Entity Recognition (NER): Identifying entities like names, dates, and locations in text.
        - Machine Translation: Translating text from one language to another.
    - Speech Recognition: Converting spoken language into text more accurately by considering context from both directions.
    - Time Series Analysis: Improving predictions by considering both past and future data points.

10. Multilayer Perceptron (MLP)

- **Description:** A Multilayer Perceptron (MLP) is a type of feedforward neural network composed of an input layer, one or more hidden layers, and an output layer. Each neuron in one layer is connected to every neuron in the next layer, making it a fully connected network. MLPs use activation functions to introduce non-linearity, allowing them to learn complex patterns.

- **Applications:**

    - Classification:
        - Handwritten Digit Recognition: Classifying images of handwritten digits (e.g., MNIST dataset).
        - Spam Detection: Classifying emails as spam or not spam.
    - Regression:
        - Predicting House Prices: Estimating the price of a house based on various features.
        - Pattern Recognition:

        - Image and Signal Processing: Recognizing patterns and features in images and signals.
    - Financial Forecasting:
        - Stock Price Prediction: Predicting future stock prices based on historical data.

2-Fundamentals of Neural Networks

2.1-Biological Neurons vs. Artificial Neurons

Neurons are the basic building blocks for the human brain. they take as input the electrical signals from other neurons and produce an output signal. In the context of deep learning, a neuron is a mathematical function that takes as input a vector of weights and a vector of inputs and produces an output.

the output of a neuron is usually passed through a non-linear function called an activation function. The activation function is used to introduce non-linearity into the output of a neuron. meaning the output of a neuron can be between 0 and 1 (sigmoid function), -1 and 1 (tanh function), or any other non-linear function.

the neural network take as an input a vector of features and produces an output vector of predictions. The neural network is composed of layers of neurons. The first layer is called the input layer, the last layer is called the output layer, and the layers in between are called hidden layers all composed of neurons.

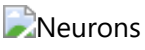


Figure 5 : Biological Neurons vs. Artificial Neurons

2.2-The Perceptron Model

Perceptron Model Paper

Basically, Perceptron are the first version discovered of representing neurons. It was first of all lineaire that's what made it non-relevent at the time, but it was the first step to the neural networks.

After the discovery of the possibility of non-linearity, and the evolution of computer power and data, the perceptron model was evolved to the neural networks.

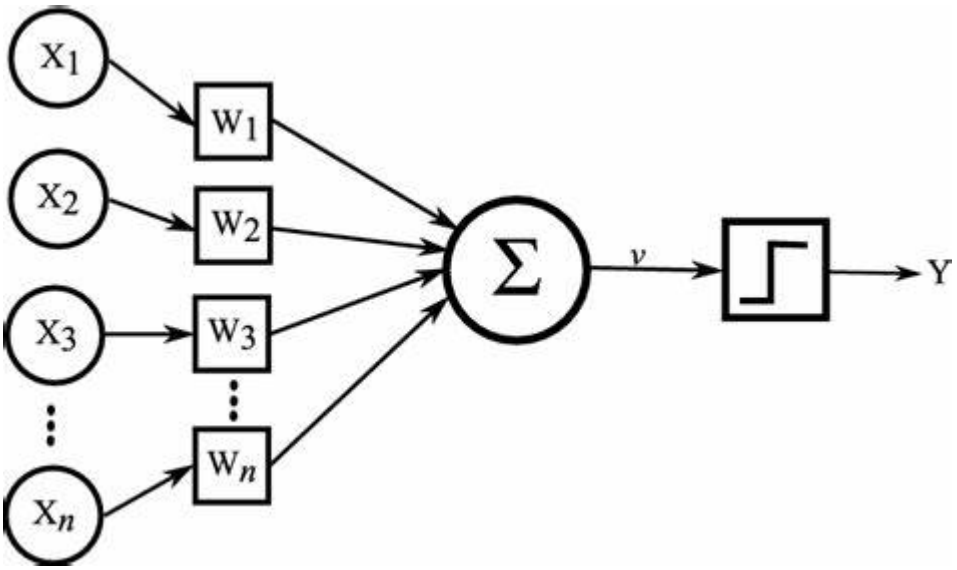


Figure 6 : Perceptron Model

2.3-Activation Functions

the activation function is the function used by the neurons to produce the output, there's various activation functions used in deep learning, the most common ones are:

Step function

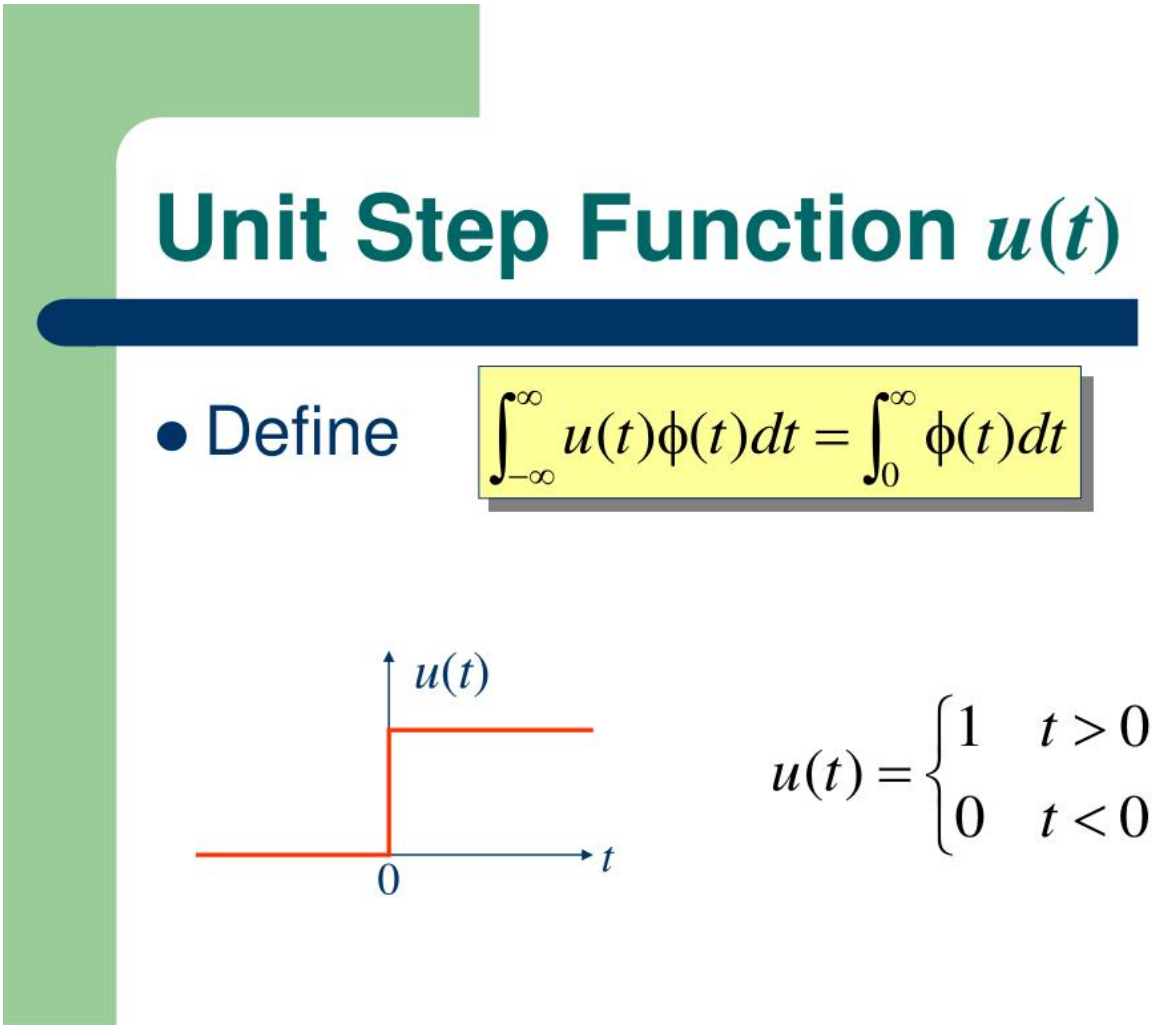


Figure 7 : Step Function

- *Step Function* or *Heaviside Step Function* is the first activation function used in neural networks, it produces a binary output based on the input.
- it was first introduced in the perceptron model, it's not used in deep learning because it's not differentiable, which makes it impossible to use gradient-based optimization algorithms.
- That's why we use other activation functions that are differentiable and can be used with gradient-based optimization algorithms.

Sigmoid function

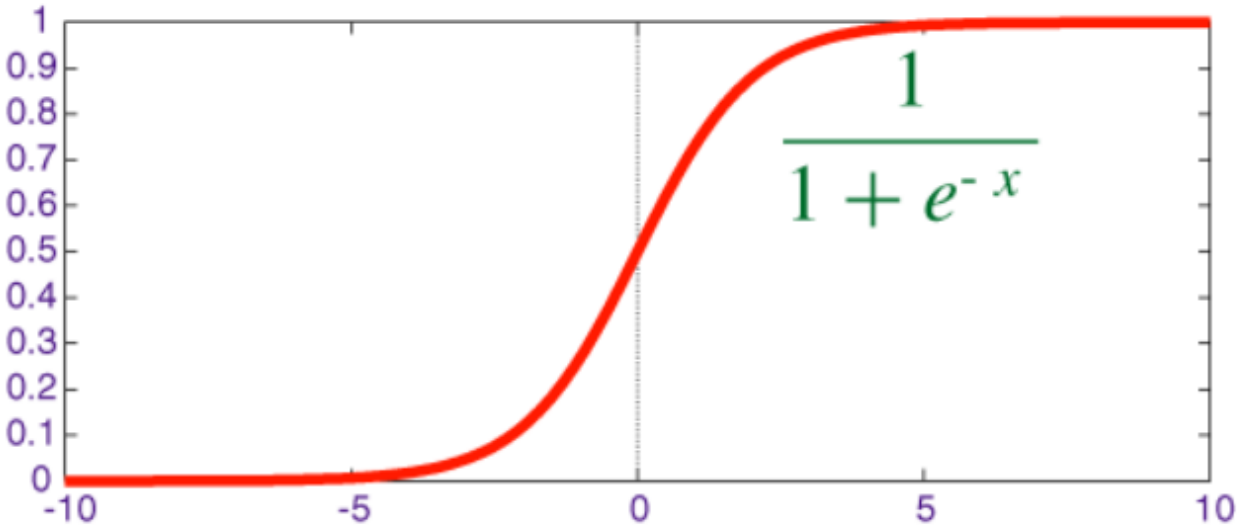


Figure 8 : Sigmoid Function

- *sigmoid function* is great for binary classification problems, it squashes the output of the neuron between 0 and 1.
- you can interpret the output of the sigmoid function as the probability of the input belonging to the positive class. (e.g. if the output is < 0.51 then it's negative, if the output is >= 0.51 then it's positive).
- *Non-Monotonic Derivative* meaning that the learning algorithm can be more difficult to converge and in some cases, it can be stuck in a local minimum.

Tanh function

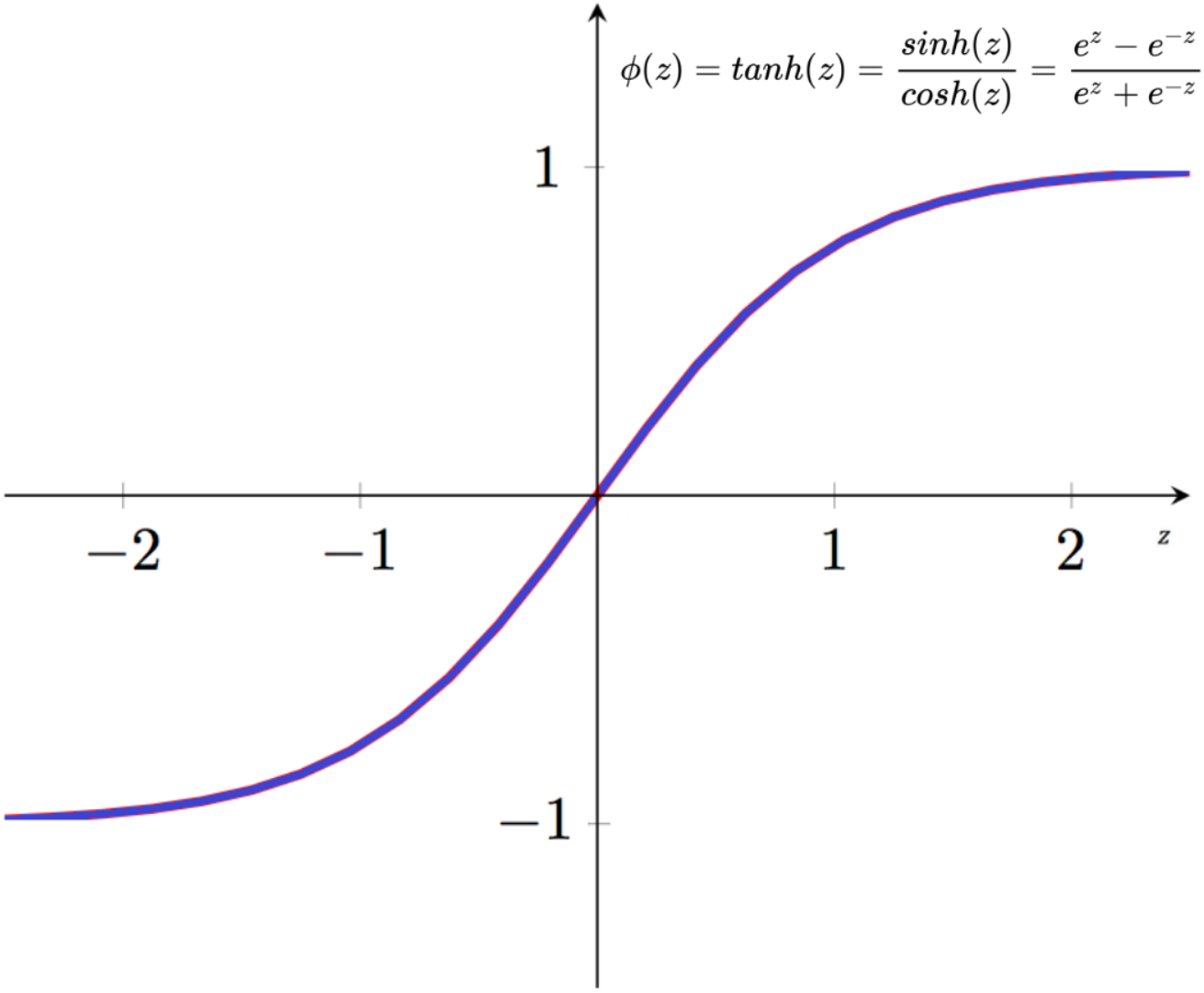


Figure 9 : Tanh Function

- *tanh function* is also a sigmoidal function(shape of S), the main difference is that it squashes the output of the neuron between -1 and 1.
- the advantage of the tanh function is that it's zero-centered, which helps the optimization algorithm converge faster. meaning that if an input is very negative, the output will be close to -1, and if the input is very positive, the output will be close to 1.
- *Non-Monotonic Derivative* meaning that the learning algorithm can be more difficult to converge and in some cases, it can be stuck in a local minimum.

ReLU and its Variants

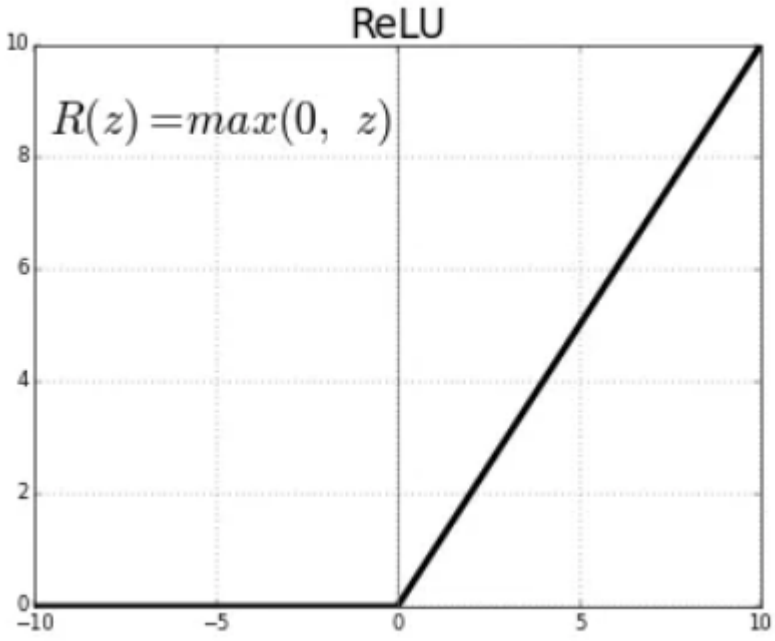


Figure 10 : ReLU Function

- The most popular activation function in the world
- *ReLU function* is a non-linear function that outputs the input directly if it's positive, otherwise, it outputs zero.
- Mostly used for CNN.
- The ReLU function is computationally efficient, it's easy to compute and it's fast. that's why it's used in most deep learning models.
- It adresses the vanishing gradient problem, which is the problem of the gradient becoming very small as the input becomes very negative, which makes the learning algorithm converge very slowly.

Leaky ReLU

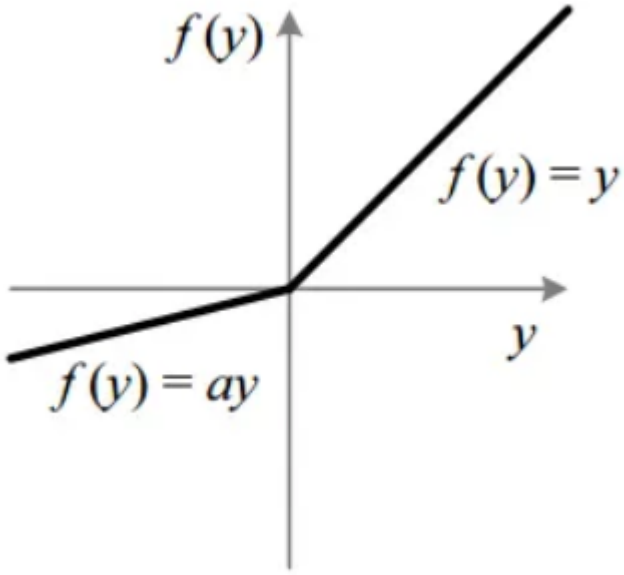


Figure 11 : Leaky ReLU Function

- *Leaky ReLU* can help to prevent the “Dying ReLU” problem, where some neurons may stop activating because they always receive negative input values, which is more likely to occur in deeper networks.

Softmax function

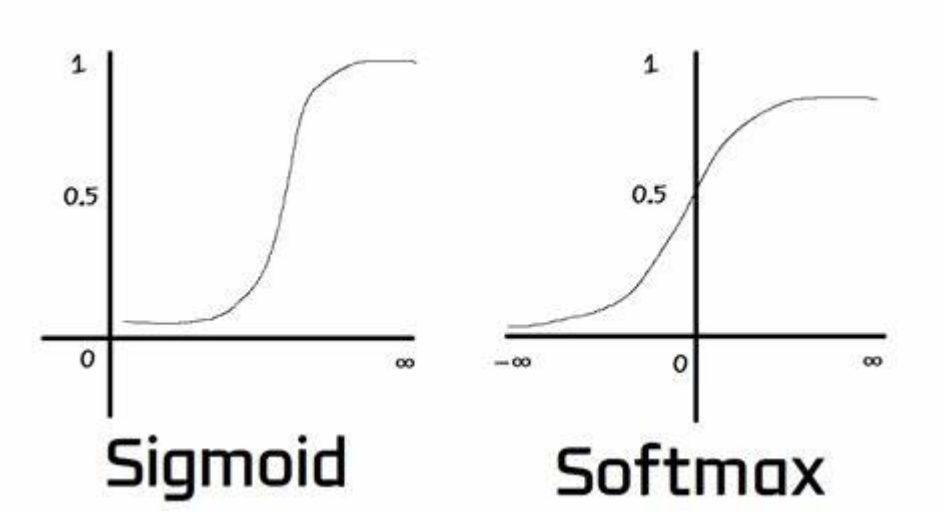


Figure 12 : Softmax vs Sigmoid

$$f_i(x) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Figure 13 : Softmax Function

- *Softmax function* is used for multi-class classification problems, it squashes the output of the neuron into a probability distribution over multiple classes.
- The output of the softmax function can be interpreted as the probability of the input belonging to each class. (e.g. if the output is [0.1, 0.9] then the input belongs to the second class with a probability of 90%).
- The softmax function is used in the output layer of the neural network for multi-class classification problems.

2.4 Feedforward Neural Networks

the feedforward neural network is the simplest type of neural network, it's composed of an input layer, one or more hidden layers, and an output layer. The input data is passed through the input layer, the hidden layers, and the output layer to produce the output of the neural network. it's the model of learning used for supervised learning in neural network. Many of the neural networks we use today are feedforward neural networks, like MLPs, CNNs, and RNNs.

> But, Makes a neural network feedforward?

There's many specific caracteristic that makes a neural network feedforward:

1. Feedforward Architecture:
    - Unidirectional Flow
    - No Cycles or Loops
2. Layers:
    - Input Layer
    - Hidden Layers
    - Output Layer
3. Neurons and Connections:
    - **Neurons**: Each layer consists of interconnected neurons (nodes) that perform computations using weighted inputs and activation functions.
    - **Connections**: Neurons are connected to neurons in the next layer, and weights associated with these connections determine the strength and influence of inputs.
4. Activation Functions:
    - **Non-linear Activation**: Neurons typically apply non-linear activation functions (e.g., sigmoid, tanh, ReLU) to introduce non-linearity into the network, enabling it to learn complex relationships in data.
5. Training with Backpropagation:
    - **Gradient Descent**: Trained using backpropagation with gradient descent or its variants, where gradients of a loss function with respect to weights are computed and used to update weights iteratively.
    - **Supervised Learning**: Primarily used in supervised learning tasks where labeled data pairs (input-output) are available for training.
6. Universal Approximator:
    - FNNs are theoretically capable of approximating any continuous function given a sufficiently large number of neurons and layers, subject to the constraints of computational resources and training data.

2.5 Forward and Backward Propagation

The forward propagation is the flow of the input data through the neural network to produce the output. The input data is passed through the input layer, the hidden layers, and the output layer to produce the output of the neural network.

Here's how it works for a FNN:

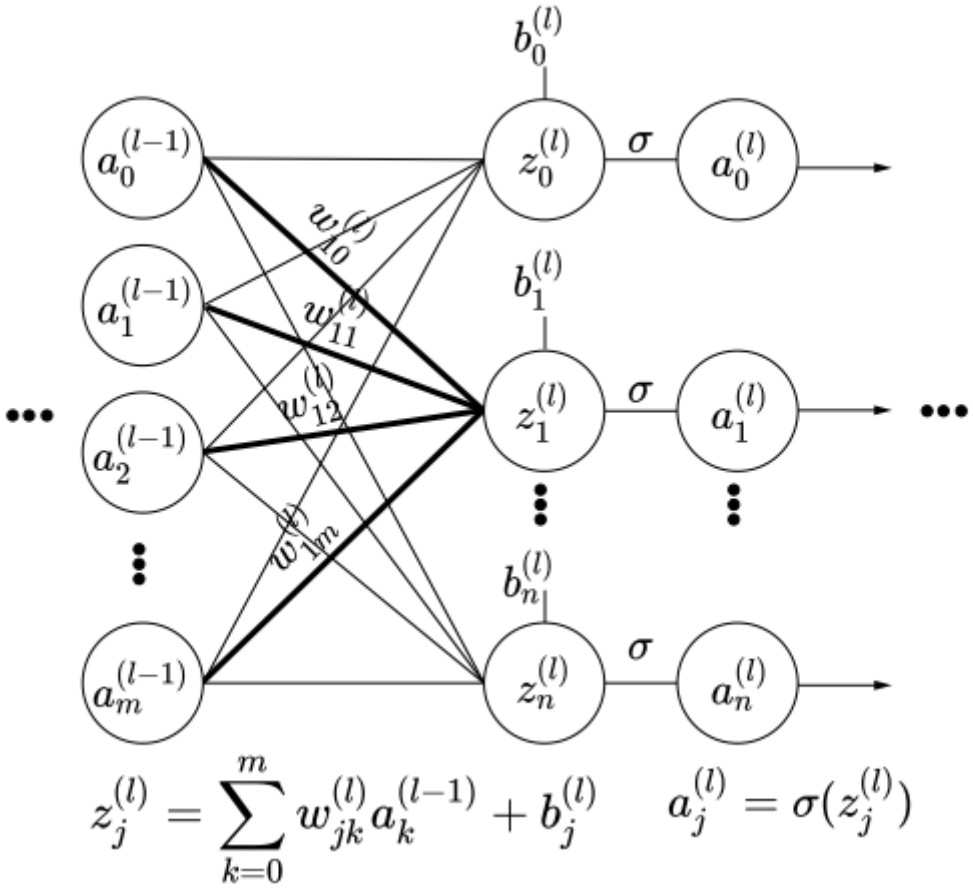


Figure 14 : Fully Connected Layer

The forward propagation is done by multiplying the input data by the weights of the neurons and adding the biases, then passing the result through the activation function to produce the output of the neuron.

$$\left.\begin{aligned} z_j^{(l)} &= \begin{bmatrix} w_{j0}^{(l)} & w_{j1}^{(l)} & \cdots & w_{jm}^{(l)} \end{bmatrix} \begin{bmatrix} a_0^{(l-1)} \\ a_1^{(l-1)} \\ \cdots \\ a_m^{(l-1)} \end{bmatrix} + b_j^{(l)} \\ a_j^{(l)} &= \sigma\left(z_j^{(l)}\right) \end{aligned}\right\} \text{for } j = 0 \cdots n$$

Figure 15 : Forward Propagation Equation

The whole set of equations can be combined into a single equation with matrix multiplication:

$$\vec{z}^{(l)} = W^{(l)}\vec{a}^{(l-1)} + \vec{b}^{(l)}$$
$$\vec{a}^{(l)} = \sigma\left(\vec{z}^{(l)}\right) \tag{8.7}$$

where

- $W^{(l)}$ is an $n \times m$ matrix representing the weights of *all connections from layer* $l-1$ *to layer* l:

$$W^{(l)} = \begin{bmatrix} w_{00}^{(l)} & w_{01}^{(l)} & \cdots & w_{0m}^{(l)} \\ w_{10}^{(l)} & w_{11}^{(l)} & \cdots & w_{1m}^{(l)} \\ \vdots & & & \\ w_{j0}^{(l)} & w_{j1}^{(l)} & \cdots & w_{jm}^{(l)} \\ \vdots & & & \\ w_{n0}^{(l)} & w_{n1}^{(l)} & \cdots & w_{nm}^{(l)} \end{bmatrix} \tag{8.8}$$

Figure 16 : Set of Forward Propagation Equations combined

$$MLP\left(\vec{x}\right) = \vec{a}^{(L)} = \vec{y} = \sigma\left(W^{(L)} \ldots \sigma\left(W^{(1)}\sigma\left(W^{(0)}\vec{x} + \vec{b}^{(0)}\right) + \vec{b}^{(1)}\right) \cdots + \vec{b}^{(L)}\right) \tag{8.9}$$

In a computer implementation, this expression is evaluated step by step by repeated application of the linear layer:

$$\vec{a}^{0} = \sigma\left(W^{(0)}\vec{x} + \vec{b}^{(0)}\right)$$
$$\vec{a}^{1} = \sigma\left(W^{(1)}\vec{a}^{0} + \vec{b}^{(1)}\right)$$
$$\cdots$$
$$\vec{a}^{L} = \sigma\left(W^{(L)}\vec{a}^{L-1} + \vec{b}^{(L)}\right) \tag{8.10}$$

Figure 17 : The MLP function

Note: The MLP is a type of FNN, it's composed of an input layer, one or more hidden layers, and an output layer. A rule of thumb is that all MLPs are FNNs, but not all FNNs are MLPs.

Now that we did a forward propagation, we need to do a backpropagation to adjust the weights and biases of the neural network to minimize the loss function. in other words, minimize error.

The backpropagation is the flow of the error through the neural network to update the weights and biases of the neurons. The error is calculated by comparing the predicted output of the neural network with the actual output.

Here's how it updates the weights and biases of the neurons:

$$w_{jk}^{(l)} = w_{jk}^{(l)} - r\frac{\partial \mathbb{L}}{\partial w_{jk}^{(l)}} \quad \text{for all } l, j, k$$

$$b_{j}^{(l)} = b_{j}^{(l)} - r\frac{\partial \mathbb{L}}{\partial b_{j}^{(l)}} \quad \text{for all } l, j$$

Figure 18 : Weight and biases updates equations

so here we can understand that each weight is updated depending on the loss function which is `L`, the learning rate `r`.

the meaning of these equations is that (for weights as example) a weight after each propagation is corrected depending on the change of the loss function with respect to the change of the specific weight.

`r` is a hyperparameter that we choose by ourselves, it's the step size of the optimization algorithm. if the learning rate is too small, the learning algorithm will converge very slowly, and if the learning rate is too large, the learning algorithm will diverge.

Here's How the process of backpropagation works, let's take a simple example of a neural network with many layers and 1 neuron in each layer:

we need to determine first of all the change of the loss function with respect to the change of the output of the last neuron, then we need to determine the change of the loss function with respect to the change of the weights and biases of the last neuron, then we need to determine the change of the loss function with respect to the change of the output of the second last neuron, then we need to determine the change of the loss function with respect to the change of the weights and biases of the second last neuron, and so on until we reach the first neuron.

here's a demonstration of the equations we use to determine each change:

We first define an auxiliary variable:

$$\delta^{(l)} = \frac{\partial \mathbb{L}}{\partial z^{(l)}} \quad \text{for } l \in \{0, L\}$$

The physical significance of $\delta^{(l)}$ is that it is the rate of change of the loss with the (pre-activation) output of layer l (remember, in this network, layer l has a single neuron).

Let's establish a few important equations for the MLP in figure 8.10:

- *Forward propagation for an arbitrary layer* $l \in \{0, L\}$

$$z^{(l)} = w^{(l)} a^{(l-1)} + b^{(l)} \tag{8.16}$$

$$a^{(l)} = \sigma\left(z^{(l)}\right) \tag{8.17}$$

- *Loss*—Here we are working with a single training data instance, x_i, whose GT output is $\bar{y}_i$:

$$\mathbb{L} = \frac{1}{2}\left(a^{(L)} - \bar{y}_i\right)^2$$

- *Partial derivative of loss with respect to the weight and bias in terms of an auxiliary variable for the last layer,* L—Using the chain rule for partial derivatives,

$$\frac{\partial \mathbb{L}}{\partial w^{(L)}} = \frac{\partial \mathbb{L}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial w^{(L)}}$$

Examining the terms on the right, we see

$$\frac{\partial \mathbb{L}}{\partial z^{(L)}} = \delta^{(L)}$$

(auxiliary variable for layer L). And using the forward propagation equations,

$$\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)}$$

Together, they lead to

$$\frac{\partial \mathbb{L}}{\partial w^{(L)}} = \delta^{(L)} \cdot a^{(L-1)}$$

Similarly,

$$\frac{\partial \mathbb{L}}{\partial b^{(L)}} = \frac{\partial \mathbb{L}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial b^{(L)}} = \delta^{(L)} \cdot 1$$

Consequently, we have the following pair of equations expressing the partial derivative of loss with respect to weight and bias, respectively, in terms of the auxiliary variable for the last layer:

$$\frac{\partial \mathbb{L}}{\partial w^{(L)}} = \delta^{(L)} \cdot a^{(L-1)} \tag{8.18}$$

$$\frac{\partial \mathbb{L}}{\partial b^{(L)}} = \delta^{(L)} \tag{8.19}$$

- *Auxiliary variable for the last layer, L*—Using the chain rule for partial derivatives,

$$\delta^{(L)} = \frac{\partial \mathbb{L}}{\partial z^{(L)}} = \frac{\partial \mathbb{L}}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} = \left(a^{(L)} - \bar{y}_i\right) \frac{d\sigma\left(z^{(L)}\right)}{dz^{(L)}}$$

Using equation 8.5 for the derivative of a sigmoid, we get

$$\delta^{(L)} = \left(a^{(L)} - \bar{y}_i\right) \sigma\left(z^{(L)}\right)\left(1 - \sigma\left(z^{(L)}\right)\right)$$

which, using equation 8.17, leads to

$$\delta^{(L)} = \left(a^{(L)} - \bar{y}_i\right) a^{(L)}\left(1 - a^{(L)}\right) \tag{8.20}$$

- *Partial derivative of the loss with respect to the weight and bias in terms of an auxiliary variable for an arbitrary layer l*—Using the chain rule for partial derivatives,

$$\frac{\partial \mathbb{L}}{\partial w^{(l)}} = \frac{\partial \mathbb{L}}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial w^{(l)}}$$

Using the definition of the auxiliary variable and the forward propagation equation 8.16, this leads to

$$\frac{\partial \mathbb{L}}{\partial w^{(l)}} = \delta^{(l)} a^{(l-1)} \tag{8.21}$$

Similarly,

$$\frac{\partial \mathbb{L}}{\partial b^{(l)}} = \frac{\partial \mathbb{L}}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial b^{(l)}}$$

Using the definition of the auxiliary variable and the forward propagation equation 8.16, this leads to

$$\frac{\partial \mathbb{L}}{\partial b^{(l)}} = \delta^{(l)} \tag{8.22}$$

$\partial b^{(l)}$ $\partial z^{(l)}$ $\partial b^{(l)}$

Using the definition of the auxiliary variable and the forward propagation equation 8.16, this leads to

$$\frac{\partial \mathbb{L}}{\partial b^{(l)}} = \delta^{(l)} \tag{8.22}$$

- *Auxiliary variable for an arbitrary layer, l*—Using the chain rule for partial derivatives,

$$\delta^{(l)} = \frac{\partial \mathbb{L}}{\partial z^{(l)}} = \frac{\partial \mathbb{L}}{\partial z^{(l+1)}} \frac{\partial z^{(l+1)}}{\partial a^{(l)}} \frac{\partial a^{(l)}}{\partial z^{(l)}}$$

Using the definition of the auxiliary variable and the forward propagation equation 8.16, this leads to

$$\delta^{(l)} = \delta^{(l+1)} w^{(l+1)} \frac{d\sigma\left(z^{(l)}\right)}{dz^{(l)}} = \delta^{(l+1)} w^{(l+1)} \sigma\left(z^{(l)}\right)\left(1 - \sigma\left(z^{(l)}\right)\right)$$

which yields

$$\delta^{(l)} = \delta^{(l+1)} w^{(l+1)} a^{(l)}\left(1 - a^{(l)}\right) \tag{8.23}$$

Figure 22 : Démonstration of backpropagation for single neuron

In Backpropagation, we use for the last layer the equations (8.20) which gives us the auxiliary variable, this auxiliary variable helps use determine the change of the loss function with respect to the change of the weights and biases of the last neuron. this helps us update the weights and biases of the last neuron.

Then we use the equation (8.23) to determine the auxiliary variable of the second last neuron, this helps us determine the change of the loss function with respect to the change of the weights and biases of the second last neuron, and so on until we reach the first neuron. and this process helps us update the weights and biases of all the neurons in the neural network with the optimal values.

Now, What about layers with multiple neurons?

The main differences in this case is that the weights will have 2 indices j,k for example, which will represent the weight between the k-th neuron in the previous layer and the j-th neuron in the current layer.

$$z_j^{(l)} = \sum_{k=0}^{m} w_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)} \qquad \vec{z}^{(l)} = W^{(l)} \vec{a}^{(l-1)} + \vec{b}^{(l)}$$

$$a_j^{(l)} = \sigma\left(z_j^{(l)}\right) \qquad \vec{a}^{(l)} = \sigma\left(\vec{z}^{(l)}\right) \qquad (8.24)$$

Figure 23 : Forward pass equations for multi-neurons layers in a NN

The Loss function also becomes a sum of the loss functions of each neuron in the output layer. as we can have more than 1 neuron in the output layer.

$$\mathbb{L} = \frac{1}{2}\|\vec{a}^{(L)} - \bar{y}\|^2 = \frac{1}{2}\sum_j \left(a_j^{(L)} - \bar{y}_j\right)^2 \qquad (8.25)$$

Figure 24 : Loss Function for multi-neurons layers in a NN

Now we need to get Auxiliary variables for each neuron in the output layer, then we need to get the auxiliary variables for each neuron in the hidden layers, and so on until we reach the first layer. but with multiple neurons it's becoming different:

Auxiliary variables—Now that a layer has multiple neurons, we have one auxiliary variable per neuron. Thus the auxiliary variable has a subscript identifying the specific neuron in that layer. It continues to have a superscript indicating its layer. We define

$$\delta_j^{(l)} = \frac{\partial \mathbb{L}}{\partial z_j^{(l)}} \quad \forall j \in \{0 \cdots \text{ number of neurons in layer } l\}, \forall l \in \{0 \cdots L\}$$

– *Auxiliary variable for the last layer*

$$\delta_j^{(L)} = \frac{\partial \mathbb{L}}{\partial z_j^{(L)}} = \frac{\partial \mathbb{L}}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}}$$

Using equation 8.25 and observing that only one of the terms in the summation—the jth term—will survive the differentiation with respect to $a_j^{(L)}$ (since the a_js

are independent of each other), we get

$$\frac{\partial \mathbb{L}}{\partial a_j^{(L)}} = \left(a_j^{(L)} - \bar{y}_j\right)$$

Also, using the lower-left equation from 8.24 and equation 8.5, we get

$$\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} = \frac{d\sigma\left(z_j^{(L)}\right)}{dz_j^{(L)}} = a_j^{(L)}\left(1 - a_j^{(L)}\right)$$

Combining these, we get

$$\delta_j^{(L)} = \left(a_j^{(L)} - \bar{y}_j\right) a_j^{(L)}\left(1 - a_j^{(L)}\right) \tag{8.26}$$

$$\vec{\delta}^{(L)} = \left(\vec{a}^{(L)} - \bar{y}\right) \circ \vec{a}^{(L)} \circ \left(\vec{1} - \vec{a}^{(L)}\right) \tag{8.27}$$

Here, $\circ$ denotes the Hadamard product between two vectors. It is basically a vector of elementwise products of corresponding vector elements. Thus,

$$\vec{a} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} \qquad \vec{b} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix} \tag{8.28}$$

$$\vec{a} \circ \vec{b} = \begin{bmatrix} a_0 b_0 \\ a_1 b_1 \\ \vdots \\ a_n b_n \end{bmatrix} \tag{8.29}$$

Figure 26 : Démonstration for Auxiliary Variable for multi-neurons layers in a NN

we have one last piece to complete the puzzle, it's the auxiliary variable for the hidden layers, which is calculated as follows:

$$\delta_j^{(l)} = \frac{\partial \mathbb{L}\left(z_0^{(l+1)}, z_1^{(l+1)}, z_2^{(l+1)}, \cdots\right)}{\partial z_j^{(l)}} = \sum_k \frac{\partial \mathbb{L}}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}}$$

$$= \sum_k \frac{\partial \mathbb{L}}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}}$$

Now, by definition,

$$\frac{\partial \mathbb{L}}{\partial z_k^{(l+1)}} = \delta_k^{(l+1)}$$

And using equation 8.24,

$$\frac{\partial z_k^{(l+1)}}{\partial a_j^{(l)}} = w_{kj}^{(l+1)}$$

while

$$\frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} = \frac{d\sigma\left(z_j^{(l)}\right)}{dz_j^{(l)}} = a_j^{(l)}\left(1 - a_j^{(l)}\right)$$

Combining all these, we get the scalar expression for a single auxiliary variable. It is presented here along with its equivalent vector equation for the entire layer:

$$\delta_j^{(l)} = \sum_k \delta_k^{(l+1)} w_{kj}^{(l+1)} a_j^{(l)}\left(1 - a_j^{(l)}\right) \tag{8.30}$$

$$\vec{\delta}^{(l)} = \left(\left(W^{(l+1)}\right)^T \vec{\delta}^{(l+1)}\right) \circ \vec{a}^{(l)} \circ \left(\vec{1} - \vec{a}^{(l)}\right) \tag{8.31}$$

Figure 27 : Démonstration for Auxiliary Variable for hidden layers in a NN

Now, we calculate the weights and biases as follows:

- *Derivatives of loss with respect to weights and biases in terms of auxiliary variables*—We have already seen how to compute auxiliary variables. Now we will express the partial derivatives of loss with respect to weights and biases in terms of those. This will provide us with the gradients we need to update the weights and biases along the negative gradient, which is the optimal move to minimize loss:

$$\frac{\partial \mathbb{L}}{\partial w_{jk}^{(l)}} = \frac{\partial \mathbb{L}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} a_k^{(l-1)} \tag{8.32}$$

$$\nabla_{w^{(l)}} \mathbb{L} = \vec{\delta}^{(l)} \left(\vec{a}^{(l-1)}\right)^T \tag{8.33}$$

Equations 8.32 and 8.33 are equivalent. The first is scalar and pertains to individual weights in layer l, and the second describes the entire layer. Similarly, equations 8.34 and 8.35 are equivalent:

$$\frac{\partial \mathbb{L}}{\partial b_j^{(l)}} = \frac{\partial \mathbb{L}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = \delta_j^{(l)} \tag{8.34}$$

$$\nabla_{b^{(l)}} \mathbb{L} = \vec{\delta}^{(l)} \tag{8.35}$$

The first is scalar and pertains to individual biases in layer l, and the second describes the entire layer.

Figure 28 : Démonstration for weights and biases for hidden layers in a NN

the main difference between single neuron layers and multi-neurons layers is that we use vectors and matrices to represent the weights and biases of the neurons, and we use vectors to represent the output of the neurons, the auxiliary variables, and the gradients of the weights and biases.

2.6-Loss Functions

The loss function is a function that measures how well the model is performing. it measures the difference between the predicted output and the actual output. The goal of the learning algorithm is to minimize the loss function.

Mean Squared Error

The Interpreting MSE involves understanding the magnitude of the error and its implications for model's performance.

A lower MSE indicates that the model's predictions are closer to the actual values signifying better accuracy. Conversely, a higher MSE suggests that the model's predictions deviate further from true values indicating the poorer performance.

The Mean Squared Error is widely used in the various fields including the statistics, machine learning and econometrics due to its several important properties:

It provides the quantitative measure of the accuracy of the predictive models. It penalizes large errors more heavily than small errors making it sensitive to the outliers. It is mathematically convenient and easy to the interpret making it a preferred choice for the evaluating model performance.

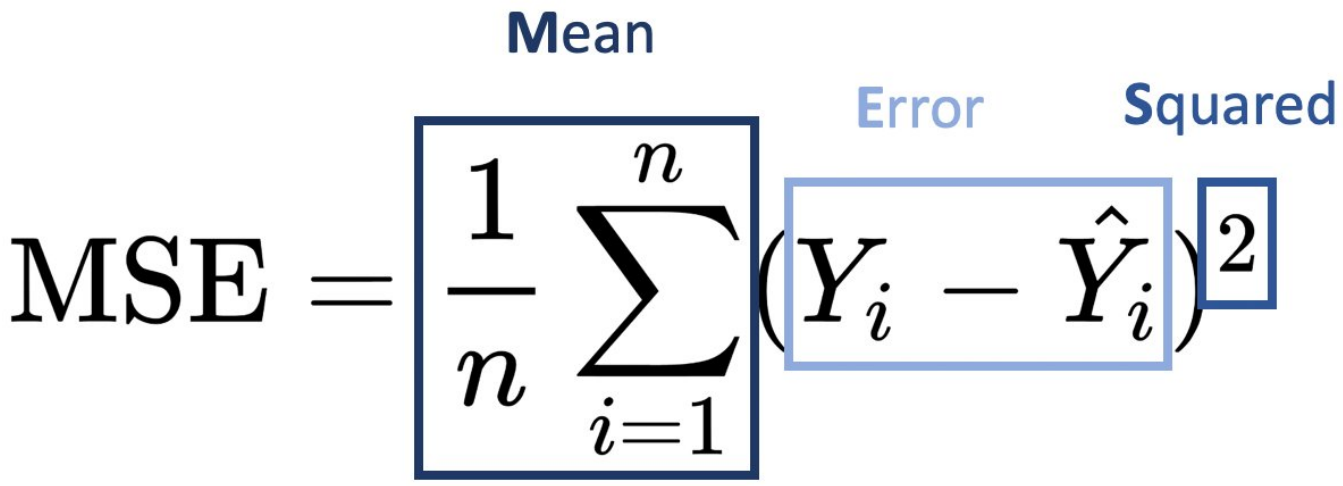


Figure 29 : Mean Squared Error

Cross-Entropy Loss

the cross-entropy loss is used for classification problems, it measures the difference between the predicted probability distribution and the actual probability distribution.

Binary Cross-Entropy Loss

For binary Cross-Entropy Loss, the formular has a sum of the loss for each class divided by minus the number of classes.

so it's like an average of the loss.

$$L_{BCE} = -\frac{1}{n}\sum_{i=1}^{n}(Y_i \cdot \log \hat{Y}_i + (1 - Y_i) \cdot \log (1 - \hat{Y}_i))$$

Figure 30 : Binary Cross-Entropy Loss

in the image, Yi is the actual output of the i-th class, and Ŷi is the predicted output of the i-th class, n is the number of classes.

Categorical Cross-Entropy Loss

the difference is that we take into consideration that we have more than 2 classes, so we have to sum the loss for each class and divide by the number of classes.

But They differ in the formula?

- it's because binary crossentropy loss is made for an output from a single neuron (sigmoid activation function) and categorical crossentropy loss is made for an output from multiple neurons (softmax activation function).

$$L = -\frac{1}{m}\sum_{i=1}^{m} y_i \cdot \log(\hat{y}_i)$$

Figure 31 : Categorical Cross-Entropy Loss

3. Training Neural Networks

3.1 Data Preprocessing

Normalization and Standardization

Data Normalization

Imagine tou have ages between 0 and 100 and salaries between 0 and 100000, the neural network will give more importance to the salaries because they have higher values than the ages, so we need to normalize the data to have the same scale.

the formula for normalization is:

$$x_{\text{norm}} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Figure 32 : Normalization Formula

- X_norm is the normalized value.
- X is the original feature value.
- X_min is the minimum feature value.
- X_max is the maximum feature value.

Data Standardization

Data standardization is the process of rescaling the features so that they have the properties of a standard normal distribution with a mean of 0 and a standard deviation of 1.

This is particularly useful when data follows a normal (or approximately normal) distribution. The formula for standardization is given by:

$$z = \frac{x_i - \mu}{\sigma}$$

Figure 33 : Standardization Formula

Where: -z is the standardized value.

- X is the original feature value.
- mu is the mean of the feature values.
- sigma is the standard deviation of the feature values.

One-Hot Encoding

One hot encoding is a process used to convert categorical data variables into a form that could be provided to machine learning algorithms to do a better job in prediction. Categorical data are variables that contain label values rather than numeric values. The number of possible values is often limited to a fixed set, for example, users by country, where the users can be from one of the countries in the dataset.

Many machine learning algorithms cannot work with categorical data directly. They require all input variables and output variables to be numeric. This is where one hot encoding comes into play, where each unique category value is assigned a binary vector that has all zero values except the index of the category, which is marked with a 1.

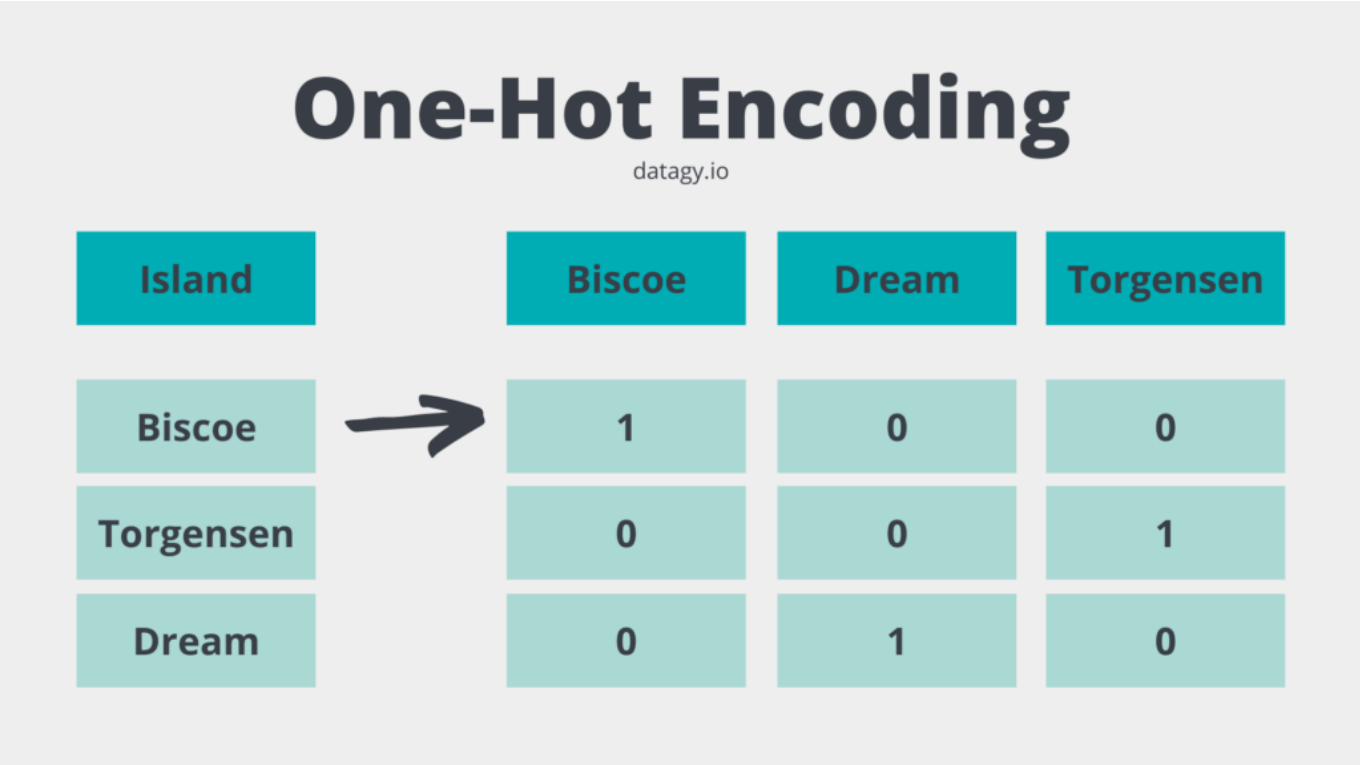


Figure 34 : One-Hot Encoding

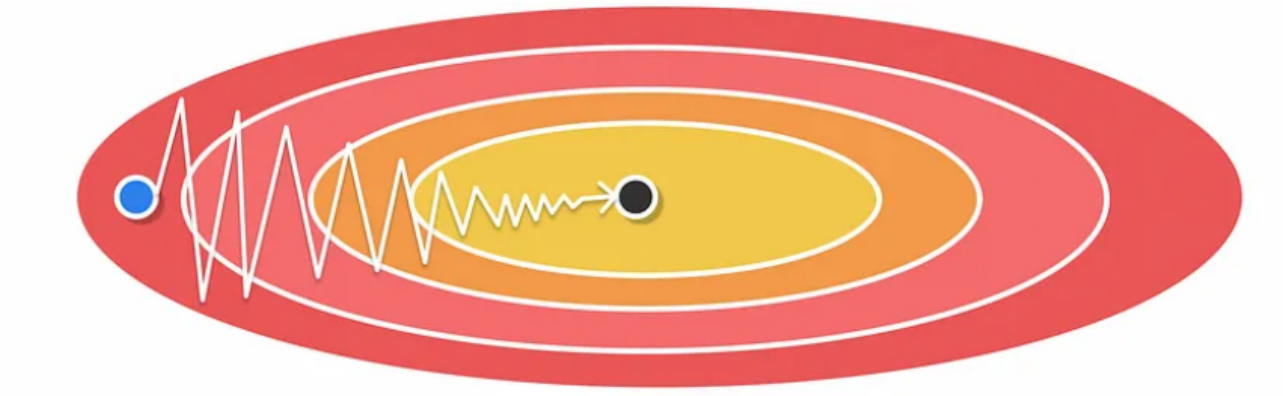
3.2 Gradient Descent

The backpropagation algorithm is an iterative optimization algorithm that tries to minimize the loss function by updating the weights of the neural network. There are many optimization algorithms that can be used to update the weights of the neural network, such as:

- Simple Gradient Descent

$$w_t = w_{t-1} - \alpha \cdot dw_t$$

Gradient descent equation. w is the weight vector, dw is the gradient of w, α is the learning rate, t is the iteration number



Example of an optimization problem with gradient descent in a ravine area. The starting point is depicted in blue and the local minimum is shown in black.

From the image, we can see that the starting point and the local minima have different horizontal coordinates and are almost equal vertical coordinates. Using gradient descent to find the local minima will likely make the loss function slowly oscillate towards vertical axes. These bounces occur because gradient descent does not store any history about its previous gradients making gradient steps more undeterministic on each iteration. This example can be generalized to a higher number of dimensions.

As a consequence, it would be risky to use a large learning rate as it could lead to disconvergence.

Batch Gradient Descent

In Batch Gradient Descent, all the training data is taken into consideration to take a single step. We take the average of the gradients of all the training examples and then use that mean gradient to update our parameters. So that's just one step of gradient descent in one epoch.

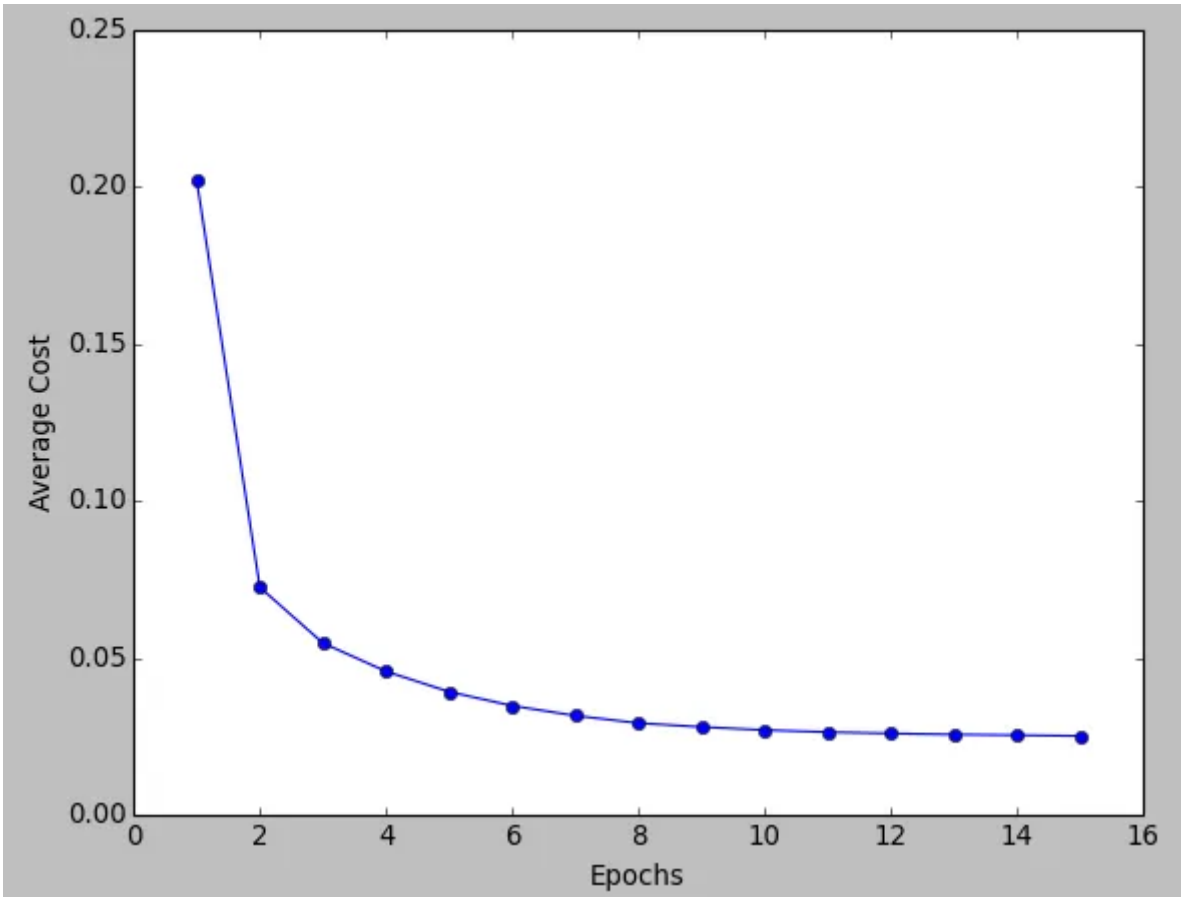


Figure 35 : Batch Gradient Descent

The graph of cost vs epochs is also quite smooth because we are averaging over all the gradients of training data for a single step. The cost keeps on decreasing over the epochs.

Stochastic Gradient Descent

n Stochastic Gradient Descent (SGD), we consider just one example at a time to take a single step. We do the following steps in one epoch for SGD:

1. Take an example
2. Feed it to Neural Network
3. Calculate it's gradient
4. Use the gradient we calculated in step 3 to update the weights
5. Repeat steps 1–4 for all the examples in training dataset

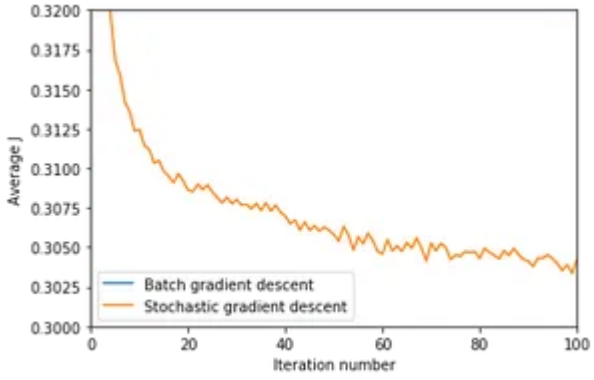


Figure 36 : Stochastic Gradient Descent

Since we are considering just one example at a time the cost will fluctuate over the training examples and it will not necessarily decrease. But in the long run, you will see the cost decreasing with fluctuations.

Also because the cost is so fluctuating, it will never reach the minima but it will keep dancing around it.

Mini-batch Gradient Descent

Neither we use all the dataset all at once nor we use the single example at a time. We use a batch of a fixed number of training examples which is less than the actual dataset and call it a mini-batch. Doing this helps us achieve the advantages of both the former variants we saw. So, after creating the mini-batches of fixed size, we do the following steps in one epoch:

1. Pick a mini-batch
2. Feed it to Neural Network
3. Calculate the mean gradient of the mini-batch
4. Use the mean gradient we calculated in step 3 to update the weights
5. Repeat steps 1–4 for the mini-batches we created

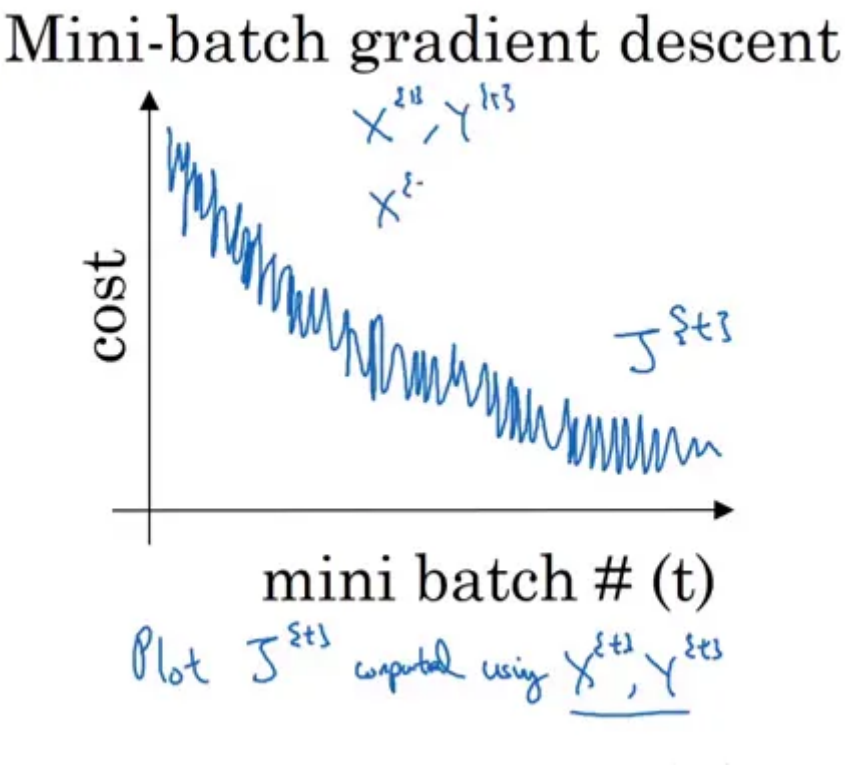


Figure 37 : Mini-batch Gradient Descent

Just like SGD, the average cost over the epochs in mini-batch gradient descent fluctuates because we are averaging a small number of examples at a time.

3.3 Learning Rate and Optimization

Learning Rate Schedules

The learning rate is a hyperparameter that controls how much we are adjusting the weights of our network with respect to the loss gradient. The lower the value, the slower we travel along the downward slope. While this might be a good idea (using a low learning rate) in terms of making sure that we do not miss any local

minima, it could also mean that we'll be taking a long time to converge — especially if we get stuck on a plateau region.

There's 2 ways to adjust the learning rate:

1. Schedule-Based Learning Rate Adjustment
2. Adaptive Learning Rate Adjustment

Now we will talk about the first one:

Constant Learning Rate

The constant learning rate is the simplest learning rate schedule. It keeps the learning rate constant throughout the training process. While this method is simple and easy to implement, it may not be the most efficient way to train a model. A constant learning rate may lead to slow convergence or oscillations around the minima.

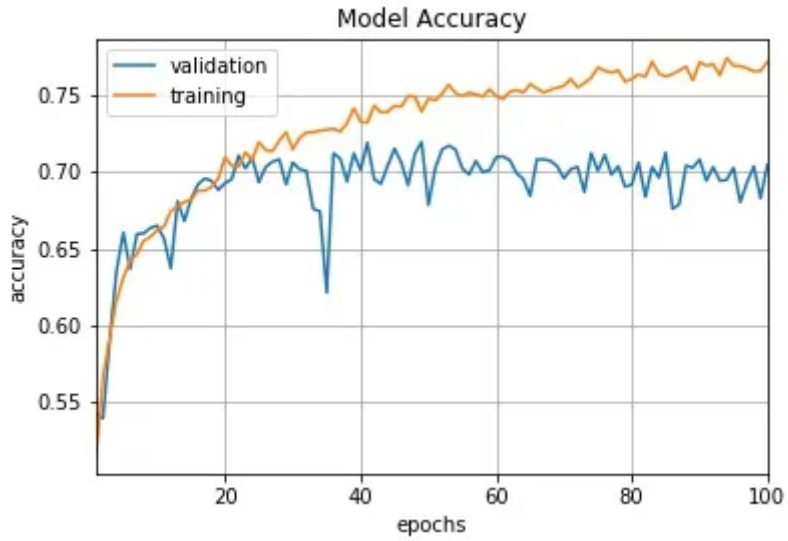


Figure 38 : Constant Learning Rate Results

Time-Based Learning Rate Decay

Time-based learning rate decay reduces the learning rate by a fixed amount after a certain number of epochs. The learning rate is updated according to the following formula:

```
lr = lr0/(1+kt)
```

where lr0 is the initial learning rate, k is a hyperparameter, and t is the number of epochs. The learning rate decreases as the number of epochs increases. This method is simple and easy to implement, but it may not be the most efficient way to train a model.

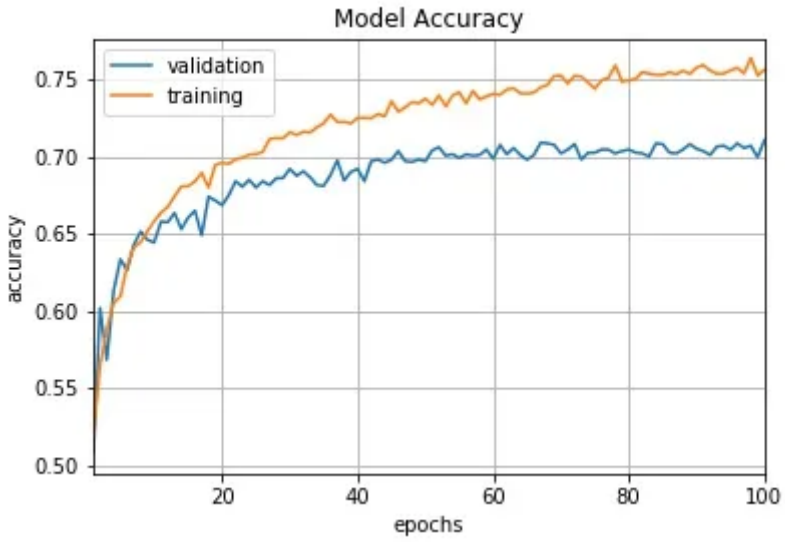


Figure 39 : Time-Based Learning Rate Decay Results

Step-Based Learning Rate Decay

Step-based learning rate decay reduces the learning rate by a fixed amount after a certain number of epochs. The learning rate is updated according to the following formula:

```
lr = lr0 * drop^floor((1+epoch)/epochs_drop)
```

where lr0 is the initial learning rate, drop is the factor by which the learning rate is reduced, epochs_drop is the number of epochs after which the learning rate is reduced, and epoch is the current epoch. This method is simple and easy to implement, but it may not be the most efficient way to train a model.

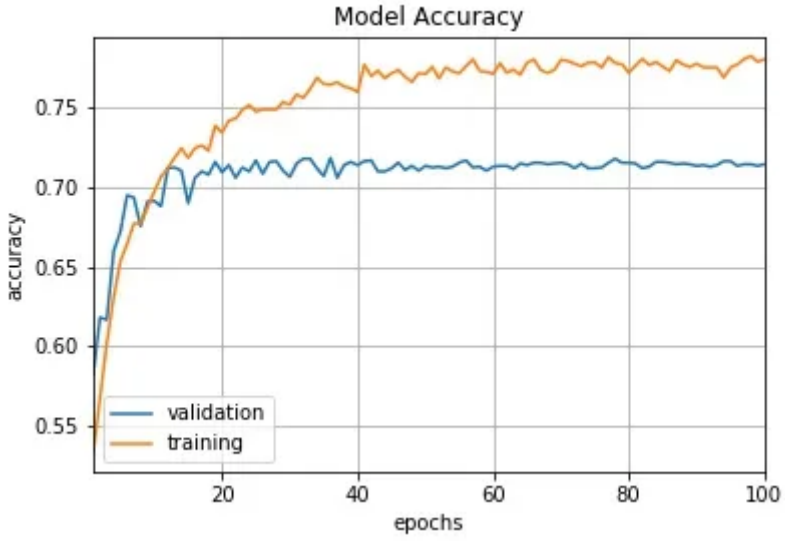


Figure 40 : Step-Based Learning Rate Decay Results

Exponential Learning Rate Decay

Exponential learning rate decay reduces the learning rate exponentially after each epoch. The learning rate is updated according to the following formula:

```
lr = lr0 * e^(-kt)
```

where lr0 is the initial learning rate, k is a hyperparameter, t is the number of epochs, and e is the base of the natural logarithm. This method is simple and easy to implement, but it may not be the most efficient way to train a model.

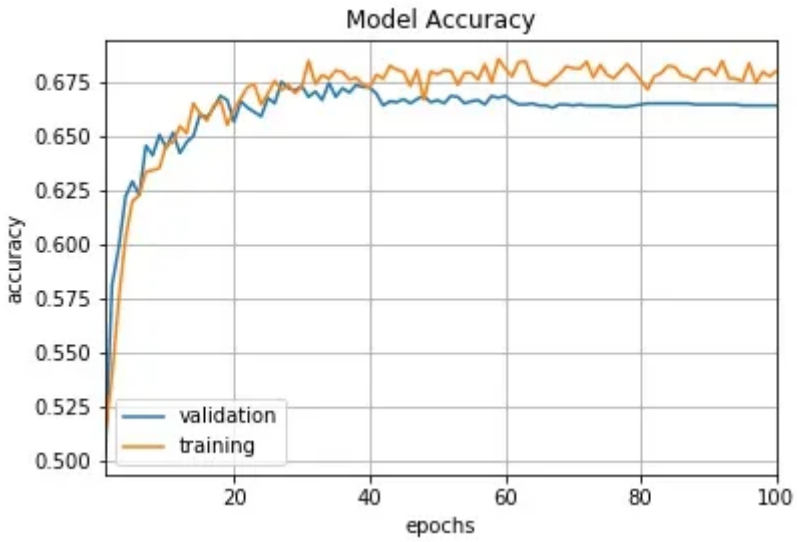


Figure 41 : Exponential Learning Rate Decay Results

After knowing about different types of manual learning rate schedules, let’s move on to adaptive learning rate adjustment.

Momentum

- Momentum

$$v_t = \beta v_{t-1} + (1 - \beta) dw_t$$

$$w_t = w_{t-1} - \alpha v_t$$

Momentum equations

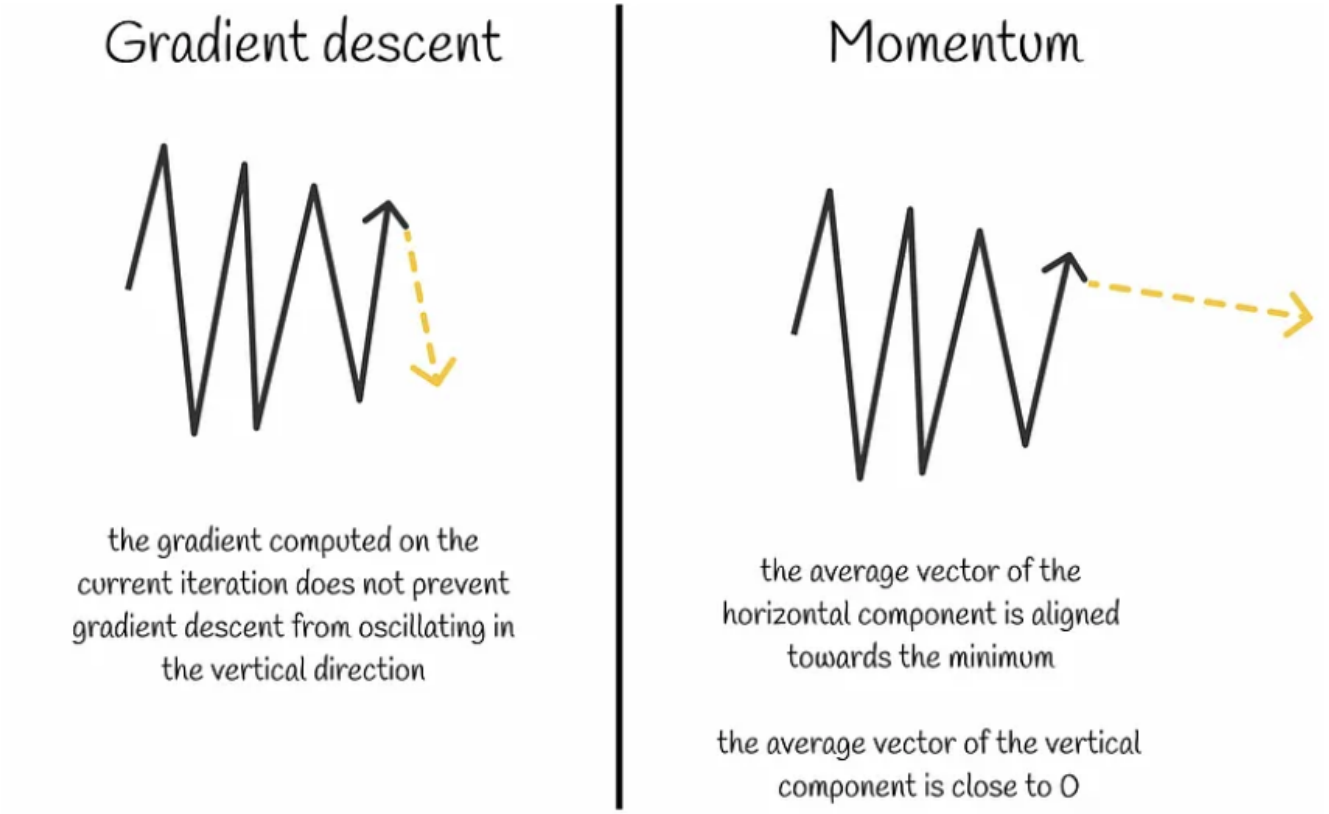


Figure 42 : Momentum

Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations. It does this by adding a fraction of the update vector of the past time step to the current update vector.

> Sebastian Ruder concisely describes the effect of Momentum in his paper: “The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation”.

Adagrad

- Adaptive Gradient Algorithm (Adagrad)

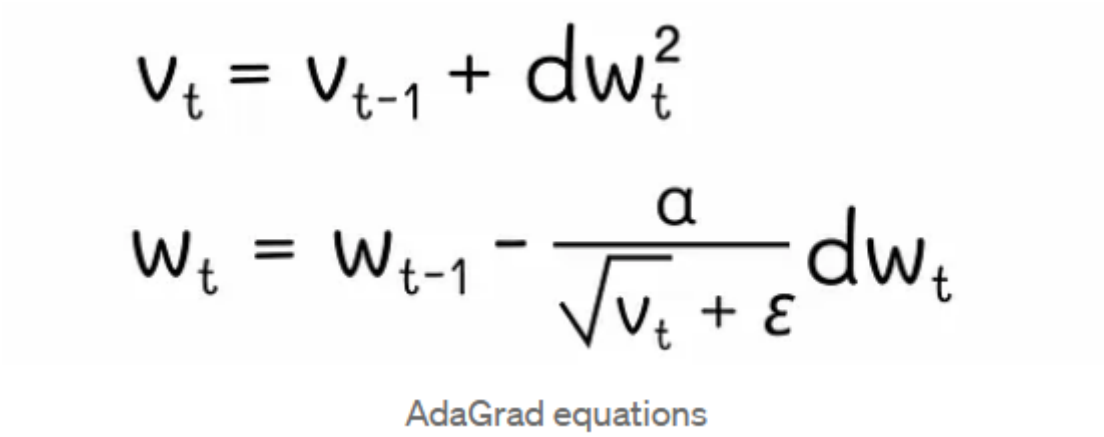


Figure 43 : Adagrad formula

There might occur situations when during training, one component of the weight vector has very large gradient values while another one has extremely small. This happens especially in cases when an infrequent model parameter appears to have a low influence on predictions.

AdaGrad deals with the aforementioned problem by independently adapting the learning rate for each weight component. If gradients corresponding to a certain weight vector component are large, then the respective learning rate will be small. Inversely, for smaller gradients, the learning rate will be bigger. This way, Adagrad deals with vanishing and exploding gradient problems.

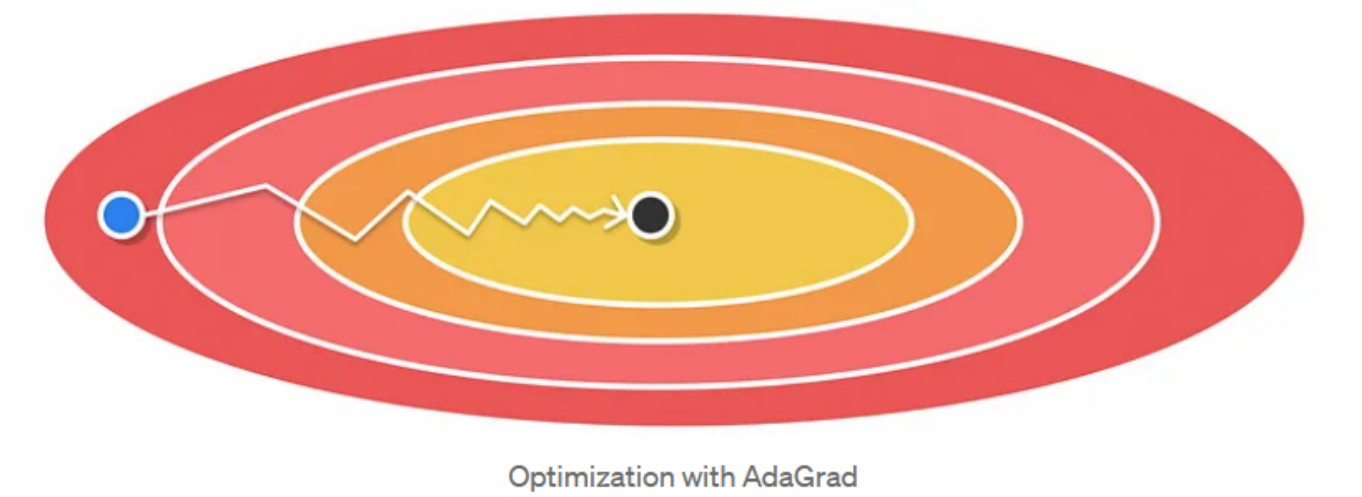


Figure 44 : Adagrad

there is a negative side of AdaGrad: the learning rate constantly decays with the increase of iterations (the learning rate is always divided by a positive cumulative number). Therefore, the algorithm tends to converge slowly during the last iterations where it becomes very low.

RMSprop

- Root Mean Square Propagation (RMSprop)

$$v_t = \beta v_{t-1} + (1 - \beta) dw_t^2$$

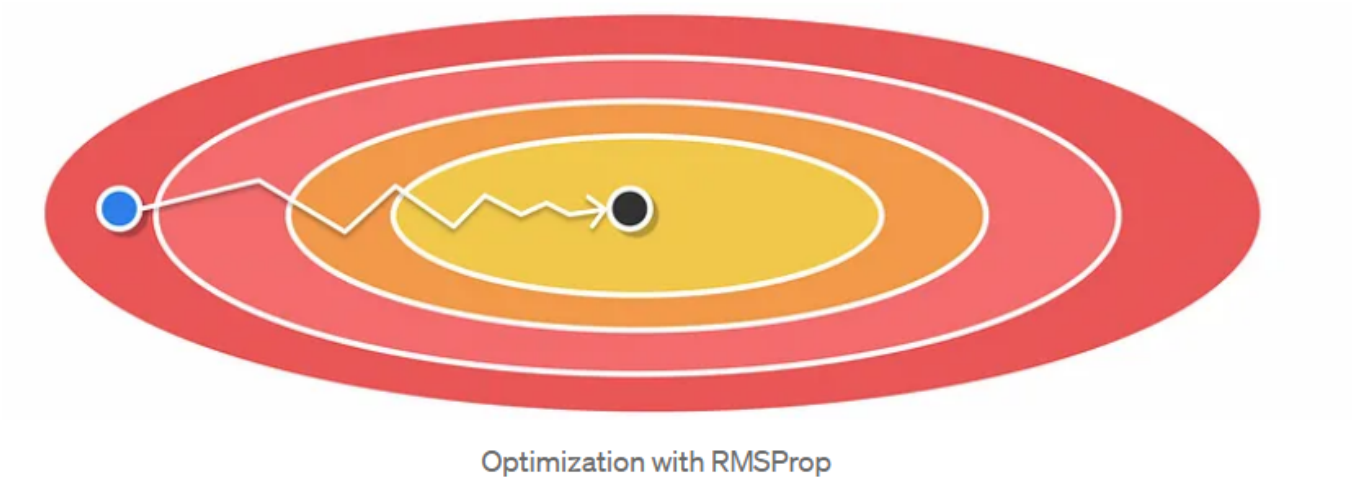
$$w_t = w_{t-1} - \frac{\alpha}{\sqrt{v_t} + \varepsilon} dw_t$$

RMSProp equations

Figure 45 : RMSprop Formula

RMSProp was elaborated as an improvement over AdaGrad which tackles the issue of learning rate decay. Similarly to AdaGrad, RMSProp uses a pair of equations for which the weight update is absolutely the same.

However, instead of storing a cumulated sum of squared gradients dw^2 for v_t, the exponentially moving average is calculated for squared gradients dw^2. Experiments show that RMSProp generally converges faster than AdaGrad because, with the exponentially moving average, it puts more emphasis on recent gradient values rather than equally distributing importance between all gradients by simply accumulating them from the first iteration. Furthermore, compared to AdaGrad, the learning rate in RMSProp does not always decay with the increase of iterations making it possible to adapt better in particular situations.



Optimization with RMSProp

Figure 46 : RMSprop

> In RMSProp, it is recommended to choose β close to 1.

Adam

- Adaptive Moment Estimation (Adam)

Adam is the most famous optimization algorithm in deep learning. At a high level, Adam combines Momentum and RMSProp algorithms. To achieve it, it simply keeps track of the exponentially moving averages for computed gradients and squared gradients respectively.

$$v_t = \beta_1 v_{t-1} + (1 - \beta_1) dw_t \xrightarrow[\text{correction}]{\text{bias}} \hat{v}_t = \frac{v_t}{1 - \beta_1^t}$$

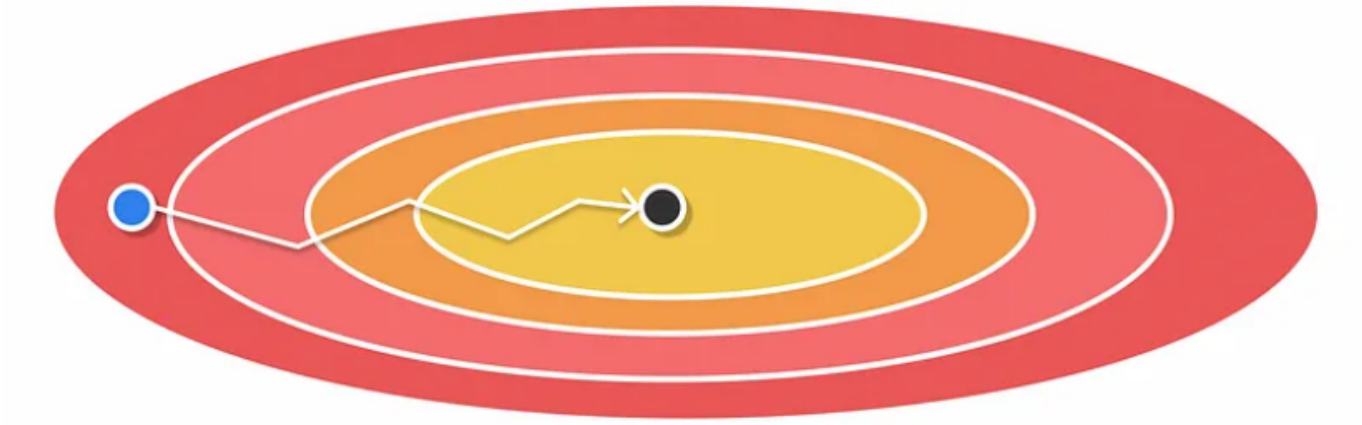
$$s_t = \beta_2 s_{t-1} + (1 - \beta_2) dw_t^2 \xrightarrow[\text{correction}]{\text{bias}} \hat{s}_t = \frac{s_t}{1 - \beta_2^t}$$

$$w_t = w_{t-1} - \frac{\alpha \hat{v}_t}{\sqrt{\hat{s}_t} + \varepsilon} dw_t$$

Adam equations

Figure 47 : Adam Formula

Furthermore, it is possible to use bias correction for moving averages for a more precise approximation of gradient trend during the first several iterations. The experiments show that Adam adapts well to almost any type of neural network architecture taking the advantages of both Momentum and RMSProp.



Optimization with Adam

Figure 48 : Adam

> According to the Adam paper, good default values for hyperparameters are $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 1e\text{-}8$.

Finally, in most cases in deep learning the Adam optimizer is used as it is the most efficient and effective optimization algorithm.

Other Optimizers

1. AdaDelta:
    - *Description:* AdaDelta adapts learning rates over time without needing a global learning rate.
    - *Use Case*: It is useful in scenarios where setting a global learning rate is challenging or inefficient.
2. AdaMax:
    - *Description:* A variant of Adam based on the infinity norm.

- *Use Case:* AdaMax is particularly effective when dealing with very sparse gradients, such as in natural language processing tasks with large embeddings.

3. Nadam (Nesterov-accelerated Adaptive Moment Estimation):

   - *Description:* Nadam combines Nesterov's Accelerated Gradient (NAG) method with Adam.
   - *Use Case:* It aims to improve convergence speed and stability compared to standard Adam, often used in tasks where fast convergence is desired.

4. AMSGrad:

   - *Description:* AMSGrad modifies Adam to ensure the optimizer doesn't "forget" previously seen minima.
   - *Use Case:* It addresses issues in Adam where the exponential moving average of past gradients might lead to convergence issues in some cases.

5. RAdam (Rectified Adam):

   - *Description:* RAdam rectifies the variance of the adaptive learning rate using a warm-up phase during training.
   - *Use Case:* It aims to provide more stable performance across different architectures and datasets compared to standard Adam.

3.4 Regularization Techniques

Regularization

Regularization is a technique used to prevent overfitting in neural networks. Overfitting occurs when the model learns the noise in the training data instead of the underlying patterns. Regularization helps to reduce the complexity of the model and prevent it from learning noise in the training data.

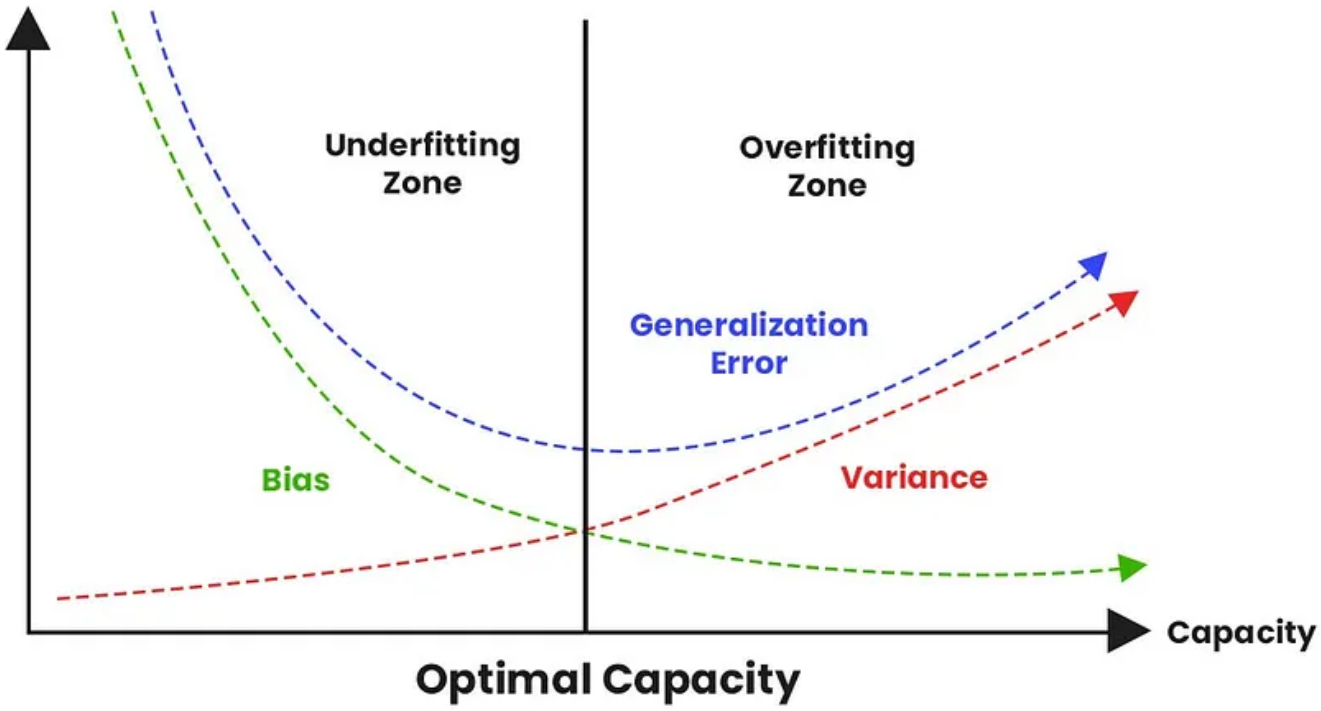


Figure 49 : Bias vs Variance tradeoff graph

> An effective regularize will be the one that makes the best trade between bias and variance, and the end-product of the tradeoff should be a significant reduction in variance at minimum expense to bias.

Parameter Norm Penalties

Parameter norm penalties are regularization techniques used in machine learning, particularly in neural networks, to prevent overfitting and improve model generalization. Overfitting occurs when a model learns the training data too well, including the noise and outliers, leading to poor performance on new, unseen data. Parameter norm penalties help mitigate this by discouraging the model from learning overly complex patterns that don't generalize well.

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\Omega(\theta)$$

Figure 50 : Parameter Norm Penalties

where α lies within [0, ∞) is a hyperparameter that weights the relative contribution of a norm penalty term, Ω, pertinent to the standard objective function J.

The meaning is : To prevent overfitting, we add a penalty for departing from “simplicity” to the original loss term.

There's various types of parameter norm penalties, including L1 and L2 regularization, which are commonly used in neural networks.

L2 Regularization (Ridge)

in the L2 Regularization, we add the euclidien norm to penalize the bigger weights more than the smaller weights. preventing the model from learning overly complex patterns that don't generalize well.

$$\mathbb{L}\left(\vec{w},\vec{b}\right)=\sum_{i=0}^{n-1}\mathbb{L}^{(i)}\left(\vec{y}^{(i)},\bar{y}^{(i)}\right)+\lambda\ \left(\|\vec{w}\|^2+\|\vec{b}\|^2\right)$$

Figure 51 : L2 Regularization

L1 Regularization (Lasso)

the main difference between L1 and L2 is that L1 tend to make a lot of weights equal to zero, which can be useful for feature selection. this is called sparsity.

$$\mathbb{L}\left(\vec{w},\vec{b}\right)=\sum_{i=0}^{n-1}\mathbb{L}^{(i)}\left(\vec{y}^{(i)},\bar{y}^{(i)}\right)+\lambda\ \left(|\vec{w}|+|\vec{b}|\right)$$

Figure 52 : L1 Regularization

Dropout

Randomly set some of the neurons in the hidden layers to zero during training.

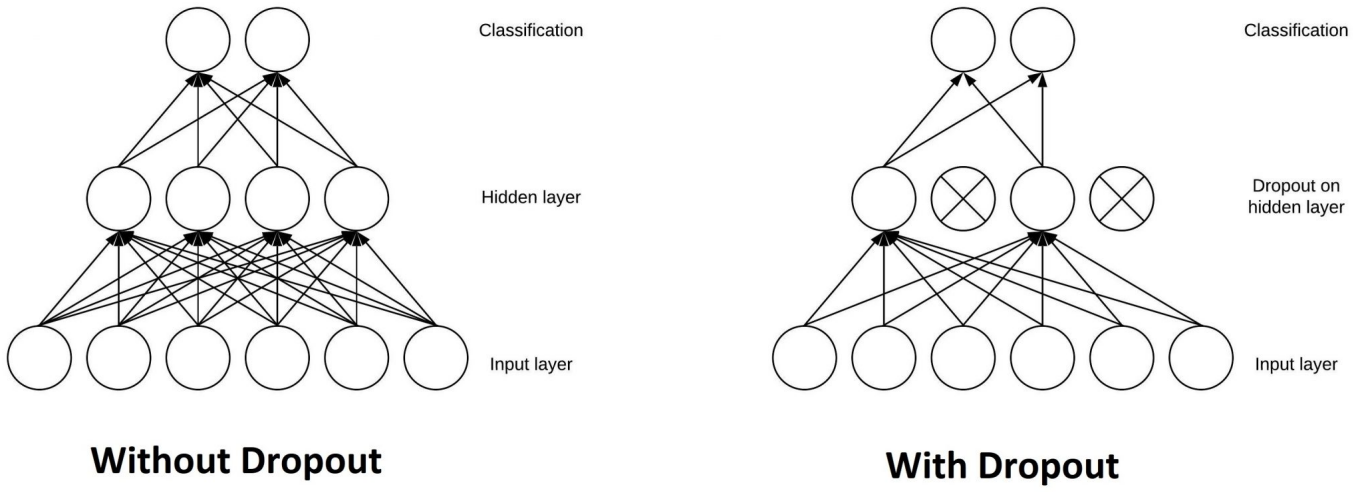


Figure 53 : Without Dropout vs With Dropout

Early Stopping

Stop training the model when the validation loss starts to increase.

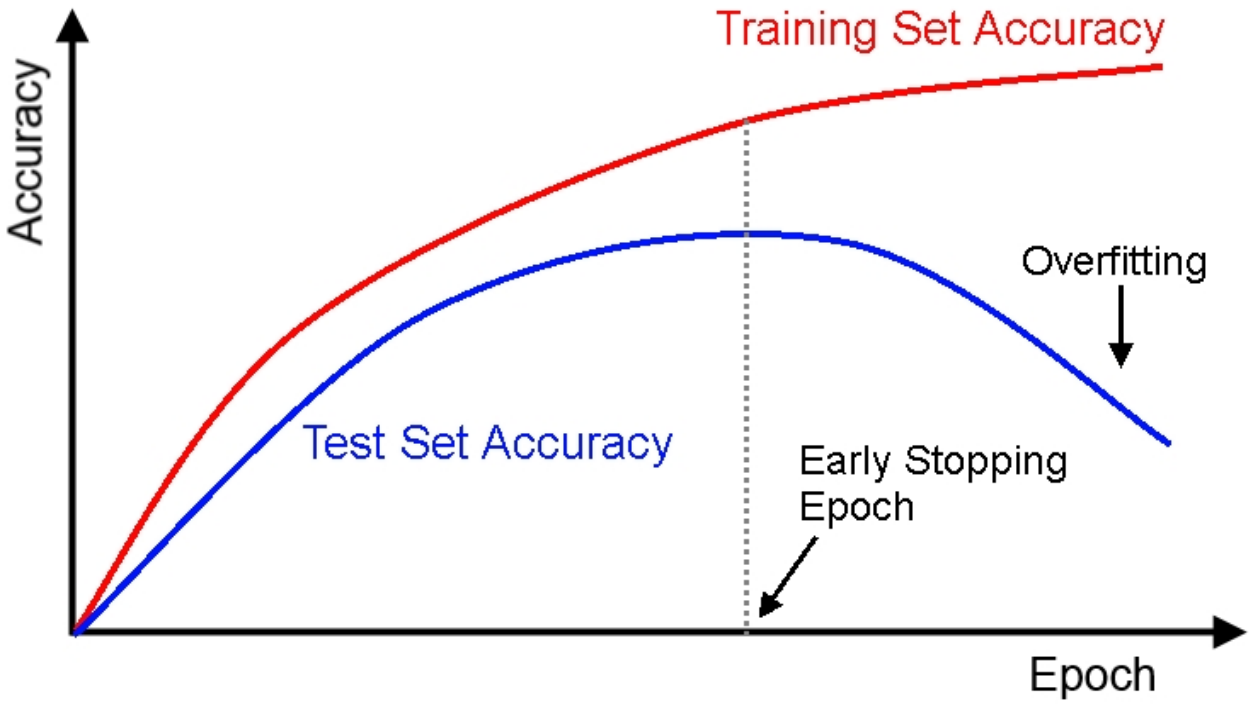


Figure 54 : Early Stopping

4. Deep Learning Basics

4.1 What Makes a Neural Network "Deep"?

A neural network is considered deep if it has more than one hidden layer. The depth of a neural network refers to the number of hidden layers it has. The more hidden layers a neural network has, the deeper it is. otherwise if it has only 1 hidden layer, it's called a shallow neural network.

"Deep" means the neural network can learn more complex patterns in the data. Deep neural networks are able to learn hierarchical representations of the data, where each layer learns a different level of abstraction. This allows deep neural networks to learn more complex patterns in the data compared to shallow neural networks.

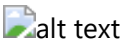


Figure 55 : Shallow vs Deep Neural Network

4.2 Importance of Non-Linearities

the main purpose of non-linearities is to maintain the complexety of the neural network, a neural network without non-linearities is equivalent to a linear regression model, which can't learn complex patterns in the data. as linear layers would be equivalent to a single linear layer each time.

4.3 The Universal Approximation Theorem

The Universal Approximation Theorem tells us that Neural Networks has a kind of universality i.e. no matter what f(x) is, there is a network that can approximately approach the result and do the job! This result holds for any number of inputs and outputs.

The Universal Approximation Theorem states that a feedforward neural network with a single hidden layer containing a finite number of neurons can approximate any continuous function on a compact subset of the input space to any desired degree of accuracy, given enough neurons in the hidden layer.

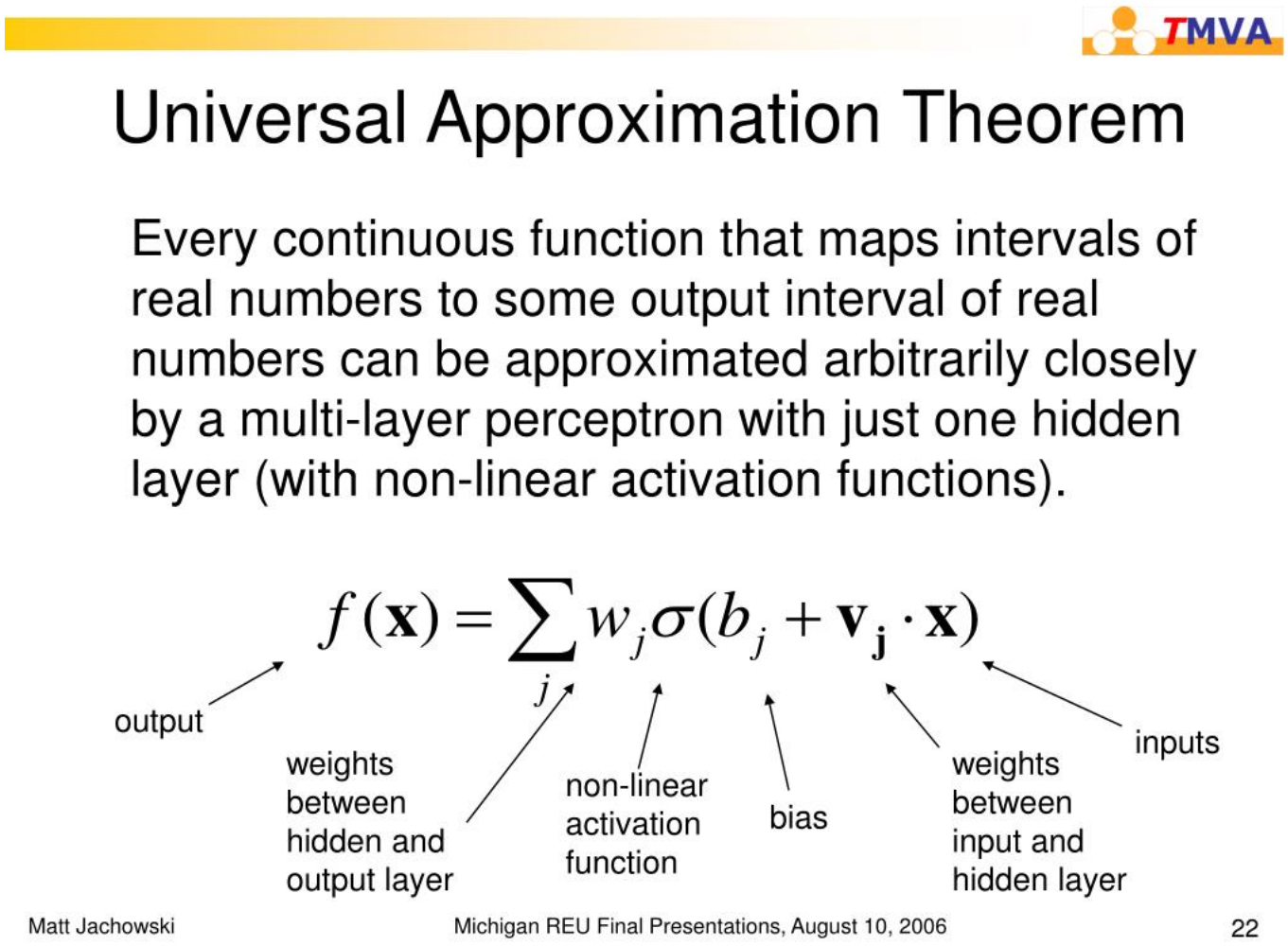


Figure 56 : Universal Approximation Theorem

4.4 Challenges in Training Deep Networks

Vanishing and Exploding Gradients

Vanishing Gradients

Vanishing gradients occur when the gradients of the loss function with respect to the weights become very small, causing the weights to stop updating. This can happen when the activation functions have very small gradients, such as the sigmoid or tanh functions. When the gradients are small, the weights don't update much, and the model learns slowly.

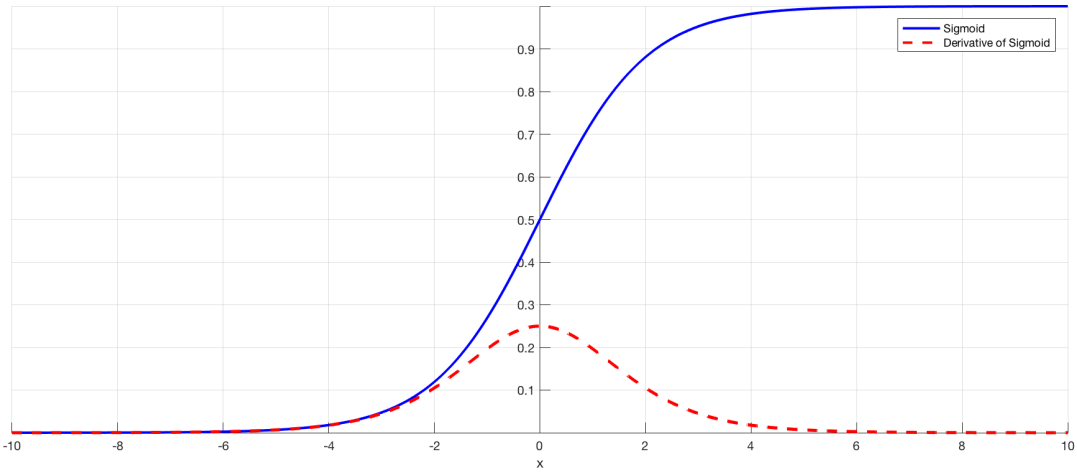


Figure 57 : Vanishing Gradients Problem

to solve this problem, we can use activation functions with larger gradients, such as the ReLU function, or use normalization techniques like batch normalization.

Exploding Gradients

Exploding gradients occur when the gradients of the loss function with respect to the weights become very large, causing the weights to update too much. This can happen when the gradients are multiplied by each other and become very large. When the gradients are large, the weights update too much, and the model learns too quickly.

to solve this problem, we can use gradient clipping, which limits the size of the gradients to a certain threshold.

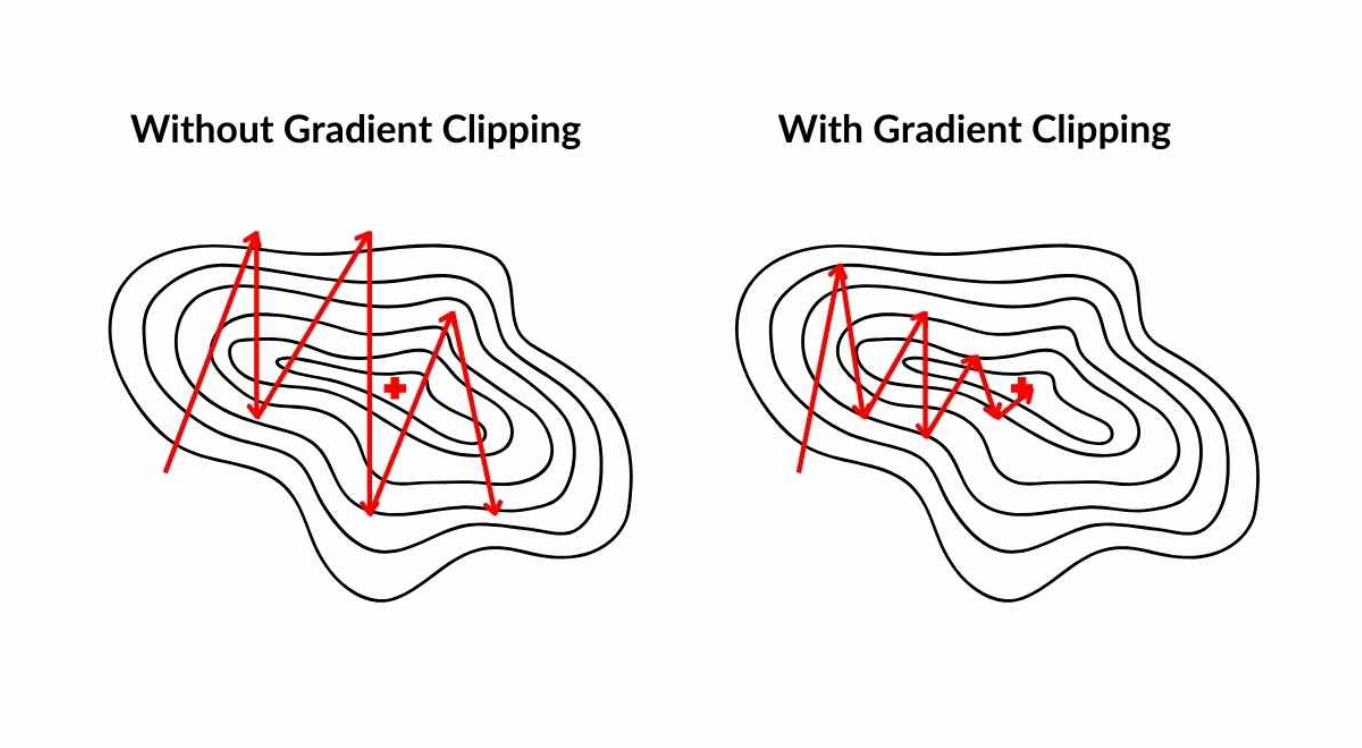


Figure 58 : Without Gradient Clipping vs With Gradient Clipping

Overfitting and Underfitting

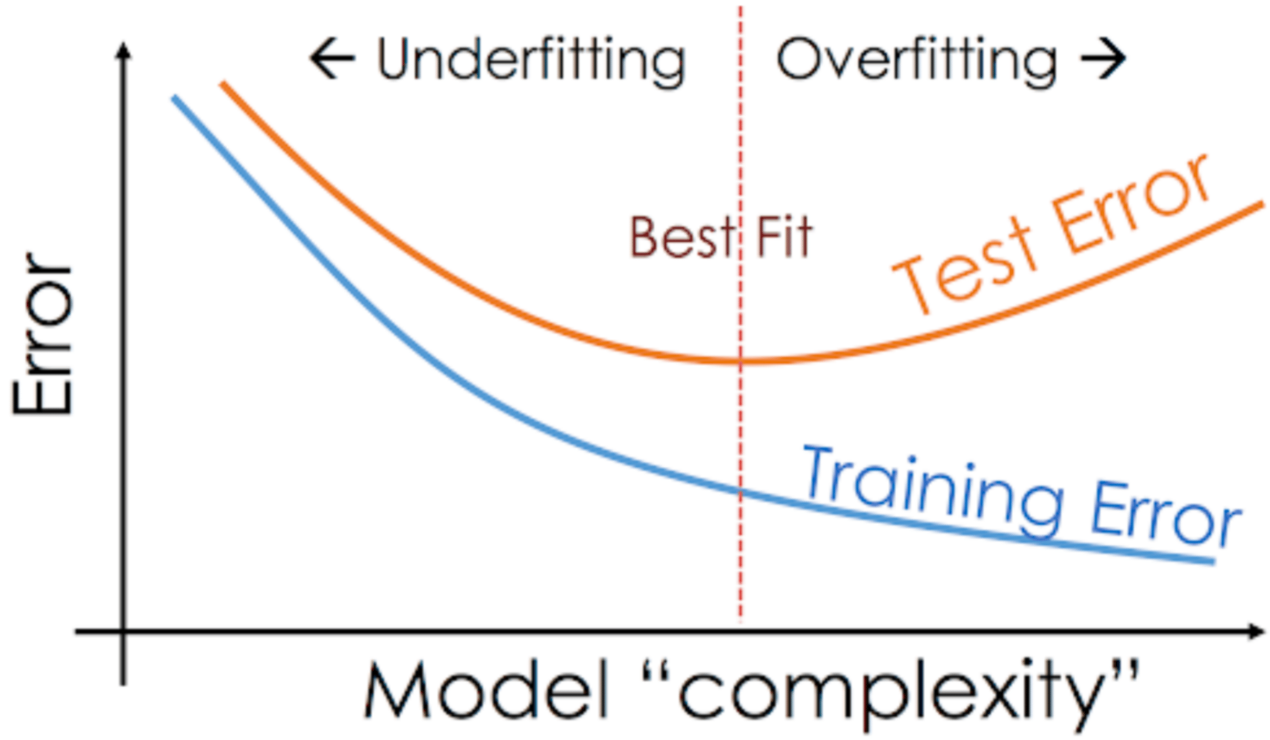


Figure 59 : Overfitting vs Underfitting

Overfitting

Overfitting is a common problem in machine learning where the model performs well on the training data but poorly on the test data. Overfitting occurs when the model learns the noise in the training data instead of the underlying patterns.

Overfitting occurs when you achieve a good fit of your model on the training data, while it does not generalize well on new, unseen data. In other words, the model learned patterns specific to the training data, which are irrelevant in other data.

We can identify overfitting by looking at validation metrics, like loss or accuracy. Usually, the validation metric stops improving after a certain number of epochs and begins to decrease afterward. The training metric continues to improve because the model seeks to find the best fit for the training data.

Underfitting

Underfitting is the opposite of overfitting. It occurs when the model is too simple to capture the underlying patterns in the data. The model performs poorly on both the training and test data because it is not complex enough to learn the patterns in the data.

5. Model Evaluation and Validation

5.1 Training, Validation, and Test Sets

In deep learning, we typically split the data into three sets: training, validation, and test sets. The training set is used to train the model, the validation set is used to tune hyperparameters and evaluate the model during training, and the test set is used to evaluate the final performance of the model.

To do the this split, we can use the train_test_split function from the scikit-learn library.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.2,
random_state=42)
```

to ensure the same proportion of classes in the training, validation, and test sets, we can use the stratify parameter.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42, stratify=y)
```

the random_state parameter is used to ensure reproducibility of the results. it's used to set the seed of the random number generator.

5.2 Metrics for Classification and Regression

Accuracy, Precision, Recall, F1-Score

Accuracy

Accuracy is the ratio of correctly predicted observations to the total observations. It is used to measure the performance of a classification model. It is calculated as follows:

```
accuracy = (TP + TN) / (TP + TN + FP + FN)
```

where:

- TP is the number of true positives
- TN is the number of true negatives
- FP is the number of false positives
- FN is the number of false negatives

the accuracy could be used as a metric for classification problems, but it's not always the best metric, especially when the classes are imbalanced.

Precision

Precision is the ratio of correctly predicted positive observations to the total predicted positive observations. It is used to measure the performance of a classification model. It is calculated as follows:

```
precision = TP / (TP + FP)
```

the precision is used when the cost of false positives is high.

Recall

Recall is the ratio of correctly predicted positive observations to the total actual positive observations. It is used to measure the performance of a classification model. It is calculated as follows:

```
recall = TP / (TP + FN)
```

the recall is used when the cost of false negatives is high.

F1-Score

The F1-Score is the harmonic mean of precision and recall. It is used to measure the performance of a classification model. It is calculated as follows:

```
f1_score = 2 * (precision * recall) / (precision + recall)
```

the F1-Score is used when we want to balance between precision and recall.

Mean Absolute Error, Mean Squared Error

Mean Absolute Error (MAE)

Mean Absolute Error (MAE) is the average of the absolute differences between the predicted and actual values. It is used to measure the performance of a regression model. It is calculated as follows:

```
MAE = 1/n * Σ|y - ŷ|
```

where:

- y is the actual value
- ŷ is the predicted value
- n is the number of observations

Mean Squared Error (MSE)

Mean Squared Error (MSE) is the average of the squared differences between the predicted and actual values. It is used to measure the performance of a regression model. It is calculated as follows:

```
MSE = 1/n * Σ(y - ŷ)²
```

where:

- y is the actual value
- ŷ is the predicted value
- n is the number of observations

6. Building and Training Neural Networks

6.1 Defining the Architecture

There's 2 ways to define the architecture of a neural network, the Sequential API and the Functional API.

Sequential API

The Sequential API is the simplest way to build a neural network in Keras. It allows you to create a neural network by stacking layers on top of each other. You can add layers to the model using the add method.

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(784,)))
model.add(Dense(64, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

Functional API

The Functional API is a more flexible way to build a neural network in Keras. It allows you to create complex neural network architectures with multiple inputs and outputs. You can define the architecture of the model by connecting layers using the functional API.

```
from keras.models import Model
from keras.layers import Input, Dense

inputs = Input(shape=(784,))
x = Dense(64, activation='relu')(inputs)
x = Dense(64, activation='relu')(x)
outputs = Dense(10, activation='softmax')(x)

model = Model(inputs=inputs, outputs=outputs)
```

There's also various types of layers that we can use in a neural network, including Dense layers, Convolutional layers, Recurrent layers, and Pooling layers.

Input Layer

The input layer is the first layer in a neural network. It takes the input data and passes it to the next layer. The input layer has a fixed size that matches the size of the input data.

```
from keras.layers import Input

inputs = Input(shape=(784,))
```

Dense Layer

The Dense layer is the most common type of layer in a neural network. It is a fully connected layer where each neuron is connected to every neuron in the previous layer.

```
from keras.layers import Dense

x = Dense(64, activation='relu')(inputs)
```

Convolutional Layer

The Convolutional layer is used in convolutional neural networks (CNNs) to extract features from images. It applies a filter to the input data to detect patterns in the data.

```
from keras.layers import Conv2D

x = Conv2D(32, kernel_size=(3, 3), activation='relu')(inputs)
#For 1D Convolutional Layer
x = Conv1D(32, kernel_size=3, activation='relu')(inputs)
```

Long-short Term Memory Layer

The Recurrent layer is used in recurrent neural networks (RNNs) to process sequences of data. It applies the same operation to each element in the sequence and passes the output to the next element.

```
from keras.layers import LSTM

x = LSTM(64, activation='relu')(inputs)
```

Pooling Layer

The Pooling layer is used in convolutional neural networks (CNNs) to reduce the size of the feature maps. It applies a pooling operation to the input data to extract the most important features.

```
from keras.layers import MaxPooling2D, GlobalAveragePooling2D, AveragePooling2D,
GlobalMaxPooling2D
```

```
#Max Pooling
x = MaxPooling2D(pool_size=(2, 2))(inputs)
#Global Average Pooling
x = GlobalAveragePooling2D()(inputs)
#Average Pooling
x = AveragePooling2D(pool_size=(2, 2))(inputs)
#Global Max Pooling
x = GlobalMaxPooling2D()(inputs)
# There's also 1D Pooling layers
```

Dropout Layer

The Dropout layer is used to prevent overfitting in neural networks. It randomly sets some of the neurons in the hidden layers to zero during training.

```
from keras.layers import Dropout

x = Dropout(0.5)(inputs)
```

Batch Normalization Layer

The Batch Normalization layer is used to normalize the input data to the neural network. It helps to stabilize and speed up the training process.

```
from keras.layers import BatchNormalization

x = BatchNormalization()(inputs)
```

Activation Function

The Activation function is used to introduce non-linearity into the neural network. It is applied to the output of each neuron in the hidden layers.

```
from keras.layers import Activation

x = Activation('relu')(inputs)
```

Attention Layer

The Attention layer is used in neural networks to focus on the most important parts of the input data. It assigns weights to the input data based on their importance.

```
from keras.layers import Attention, AdditiveAttention, MultiHeadAttention

# Normal Attention
```

```
x = Attention()(inputs)
# Additive Attention
x = AdditiveAttention()(inputs)
# Multi-Head Attention
x = MultiHeadAttention()(inputs)
```

Embedding Layer

The Embedding layer is used in neural networks to convert categorical data into a continuous representation. It is commonly used in natural language processing tasks.

```
from keras.layers import Embedding

x = Embedding(input_dim=1000, output_dim=64)(inputs)
```

Flatten Layer

The Flatten layer is used to flatten the input data into a one-dimensional array. It is commonly used to convert the output of convolutional layers into a format that can be passed to fully connected layers.

```
from keras.layers import Flatten

x = Flatten()(inputs)
```

ConvLSTM Layer

The ConvLSTM2D layer is used in neural networks to process spatiotemporal data. It combines the convolutional and LSTM layers to process sequences of images.

```
from keras.layers import ConvLSTM2D

x = ConvLSTM2D(32, kernel_size=(3, 3), activation='relu')(inputs)
```

6.2 Compiling the Model

It's important to compile the model before training it. The compile method is used to configure the model for training. You can specify the loss function, optimization algorithm, and evaluation metrics using the compile method.

```
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=
['accuracy'])
```

for loss functions, there's various types of loss functions that we can use:

- categorical_crossentropy
- binary_crossentropy
- mean_squared_error
- mean_absolute_error

for optimization algorithms, there's various types of optimization algorithms that we can use:

- adam
- sgd
- rmsprop
- adagrad
- adadelta
- adamax
- nadam

for evaluation metrics, there's various types of evaluation metrics that we can use:

- accuracy
- precision
- recall
- f1-score

6.3 Training the Model

After defining the architecture and compiling the model, we can train the model using the fit method. The fit method is used to train the model on the training data. You can specify the number of epochs, batch size, and validation data using the fit method.

```
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_val,
y_val))
```

6.4 Evaluating the Model

After training the model, we can evaluate the model using the evaluate method. The evaluate method is used to evaluate the model on the test data. You can specify the test data using the evaluate method.

```
loss, accuracy = model.evaluate(X_test, y_test)
```

6.5 Making Predictions

After training the model, we can make predictions using the predict method. The predict method is used to make predictions on new data. You can specify the new data using the predict method.

```
y_pred = model.predict(X_new)
```

7. Conclusion

In this article, we have covered the basics of deep learning, including neural networks, activation functions, loss functions, optimization algorithms, regularization techniques, and model evaluation and validation. We have also discussed the challenges in training deep networks, such as vanishing and exploding gradients, overfitting, and underfitting. We have also covered the importance of non-linearities and the universal approximation theorem. Finally, we have discussed the training, validation, and test sets, as well as metrics for classification and regression problems. We have also covered the steps involved in building and training neural networks, including defining the architecture, compiling the model, training the model, evaluating the model, and making predictions.