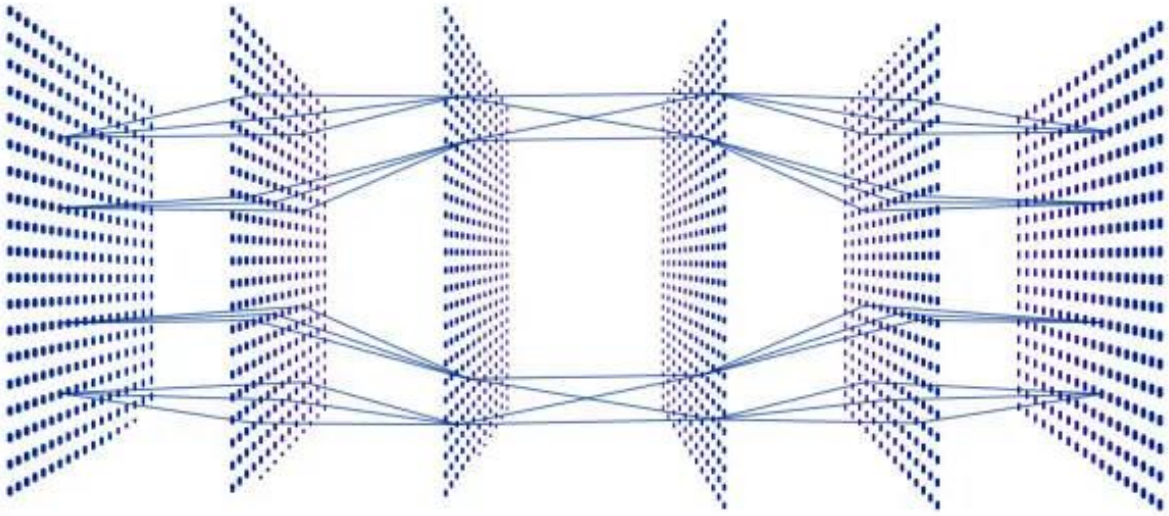
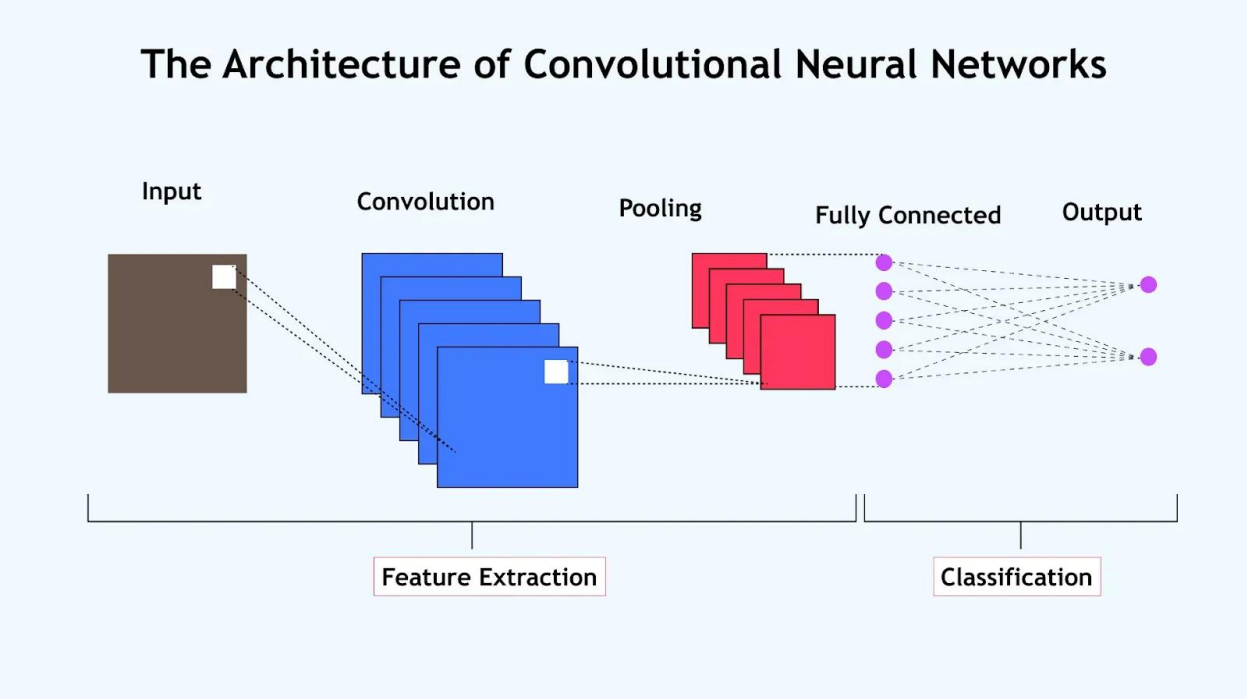
Question 1: Exploring Neural Network Architectures

1. Convolutional Neural Networks (CNN — Architecture



A **convolutional neural network (CNN)**, is a network architecture for deep learning which **learns directly from data**. CNNs are particularly useful for **finding patterns in images** to recognize objects. They can also be quite effective for **classifying non-image data** such as audio, time series, and signal data.

Convolutional Neural Network (CNNs) are a fundamental architecture in machine learning, allowing machines to learn from training data and make accurate predictions. In today's world, machine learning separates efficiency from complacency and success from failure. To enhance system accuracy and performance, understanding the basic CNN architecture is essential.



Reference: Convolutional Neural Networks (CNN) — Architecture Explained

- ## Understanding CNN Architecture

  Convolutional neural networks are distinguished from other neural networks by their superior performance with image, speech, or audio signal inputs. They have three main types of layers, which are:

  - Convolutional layer
  - Pooling layer
  - Fully-connected (FC) layer

  ### 1. Convolution Layer

  The convolution layer is the core **building block of the CNN**. It carries the **main portion** of the network's computational load.

- This layer performs a **dot product between two matrices**, where one matrix is the set of learnable parameters otherwise known as a kernel, and the other matrix is the **restricted portion** of the receptive field. The **kernel** is spatially smaller than an **image but is more in-depth**.

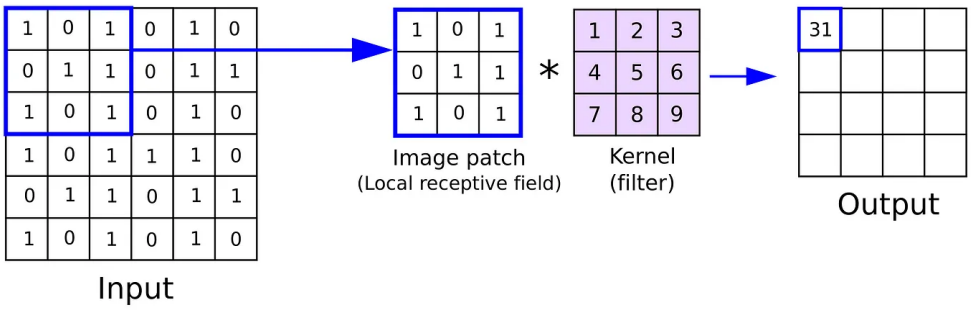
$$W_{out} = \frac{W - F + 2P}{S} + 1$$

i. Kernal and Filter

In a convolutional neural network, the kernel is nothing but **a filter that is used to extract the features from the images**.

Formula = [i-k]+1

i -> Size of input , K-> Size of kernel



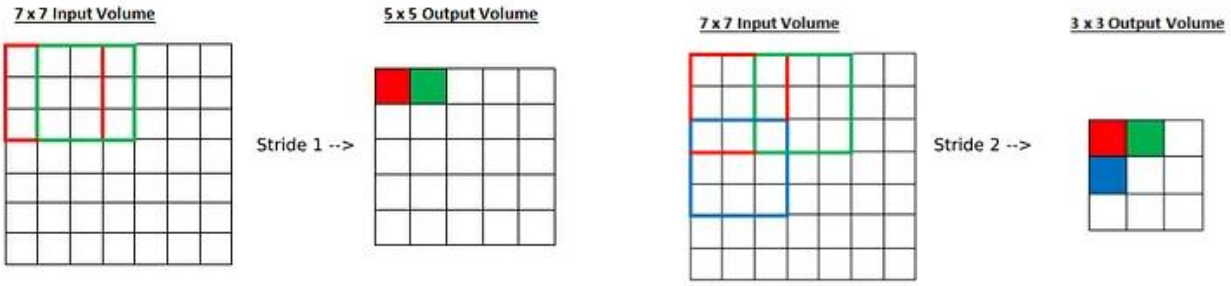
- This means that, if the image is composed of three (RGB) channels, the kernel height and width will be spatially small, but the depth extends up to all three channels.

- During the forward pass, the kernel slides across the height and width of the **image-producing** the image representation of that receptive region. This produces a **two-dimensional representation of the image** known as an **activation map** that gives the response of the kernel at each spatial position of the image. The sliding size of the kernel is called a **stride**.

ii. Stride

Stride is a parameter of the neural network's filter that modifies the amount of movement over the image or video. we had stride 1 so it will take one by one. If we give stride 2 then it will take value by skipping the next 2 pixels.

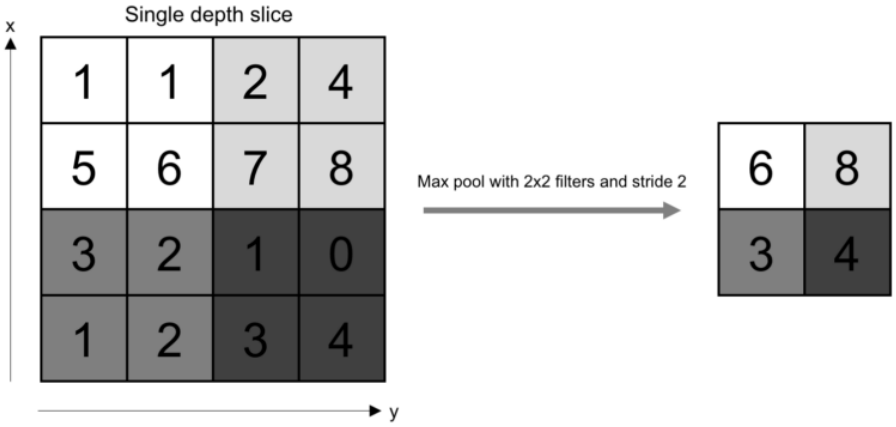
- **Formula =[i-k/s]+1**
- i -> Size of input , K-> Size of kernel, S-> Strid



1. Pooling Layer

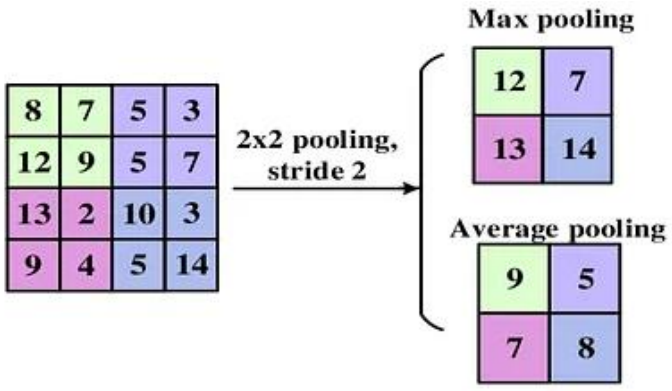
The **pooling layer** replaces the output of the network at certain locations by deriving a **summary statistic of the nearby outputs**. This helps in **reducing the spatial size of the representation**, which decreases the required amount of computation and weights. The pooling operation is processed on every slice of the representation individually.

There are several pooling functions such as the **average of the rectangular neighborhood, L2 norm of the rectangular neighborhood, and a weighted average based on the distance from the central pixel**. However, the most popular process is max pooling, which **reports the maximum output** from the neighborhood.



i. Pooling

Pooling in convolutional neural networks is a technique for **generalizing features** extracted by convolutional filters and helping the network **recognize features independent** of their location in the image.



If we have an activation map of size *W* x *W* x *D*, a pooling kernel of **spatial size *F***, and **stride *S***, then the size of output volume can be determined by the following formula:

$$W_{out} = \frac{W - F}{S} + 1$$

This will yield an output volume of size *Wout* x *Wout* x *D*.

In all cases, pooling provides some translation invariance which means that an object would be recognizable regardless of where it appears on the frame.

2. Fully Connected Layer

The Fully Connected (FC) layer consists of the weights and biases along with the neurons and is used to connect the neurons between two different layers. These layers are usually placed before the output layer and form the last few layers of a CNN Architecture.

Reference: Convolutional Neural Networks, Explained

Key Differences Between CNN and NN

1. **Architecture**:
   — **NN**: Each neuron in one layer is connected to every neuron in the next layer, making it "fully connected."
   — **CNN**: Neurons are only connected to a small region of the previous layer (local connections), focusing on capturing patterns in localized areas.

2. **Input Data**:
   — **NN**: Typically used for tabular data, or when the relationships between features are not spatially dependent (like in stock price prediction or basic classifications).
   — **CNN**: Optimized for grid-like data structures, such as images, where spatial relationships between pixels matter.

3. **Weight Sharing:**
   — NN: Every connection between layers has its own weight, leading to a large number of parameters that can increase with input size.
   — CNN: The same filter (kernel) is applied across different parts of the input, sharing weights. This makes CNNs much more efficient, especially for large inputs like high-resolution images.

4. **Feature Extraction:**
   — **NN**: Relies on the network learning features from scratch,

often requiring a large number of parameters and computational power.
— **CNN**: Automatically extracts features using filters and can capture hierarchical features (e.g., detecting simple shapes like edges first, then more complex patterns).

5. **Performance**:
   — **NN**: Effective for simpler tasks but struggles with high-dimensional data such as images or videos.
   — **CNN**: Specifically designed for tasks involving complex, structured data like images. CNNs have become the go-to architecture for any vision-related task because of their ability to learn spatial hierarchies.

- **Reference:** Key Differences Between CNN and NN

🌍 Real-World Application: Medical Image Diagnosis

CNNs are widely used in **medical imaging**, such as detecting tumors in MRI scans or identifying signs of pneumonia in chest X-rays. Their ability to identify subtle patterns makes them ideal for diagnostic support systems.

---

2. Transformers

A **transformer model** is a neural network that **learns the context** of sequential data and generates **new data** out of it.

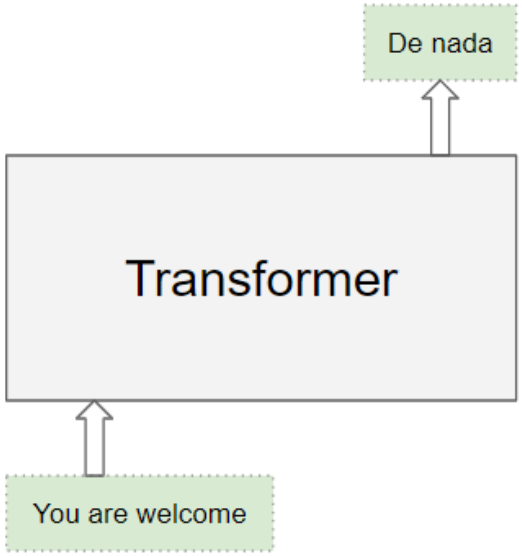
To put it simply:

"**A transformer is a type of artificial intelligence model that learns to understand and generate human-like text by analyzing patterns in large amounts of text data**".

Transformers are a current **state-of-the-art NLP model** and are considered the evolution of the **encoder-decoder architecture**. However, while the encoder-decoder architecture relies mainly on Recurrent Neural Networks (RNNs) to extract sequential information, Transformers completely lack this recurrency.

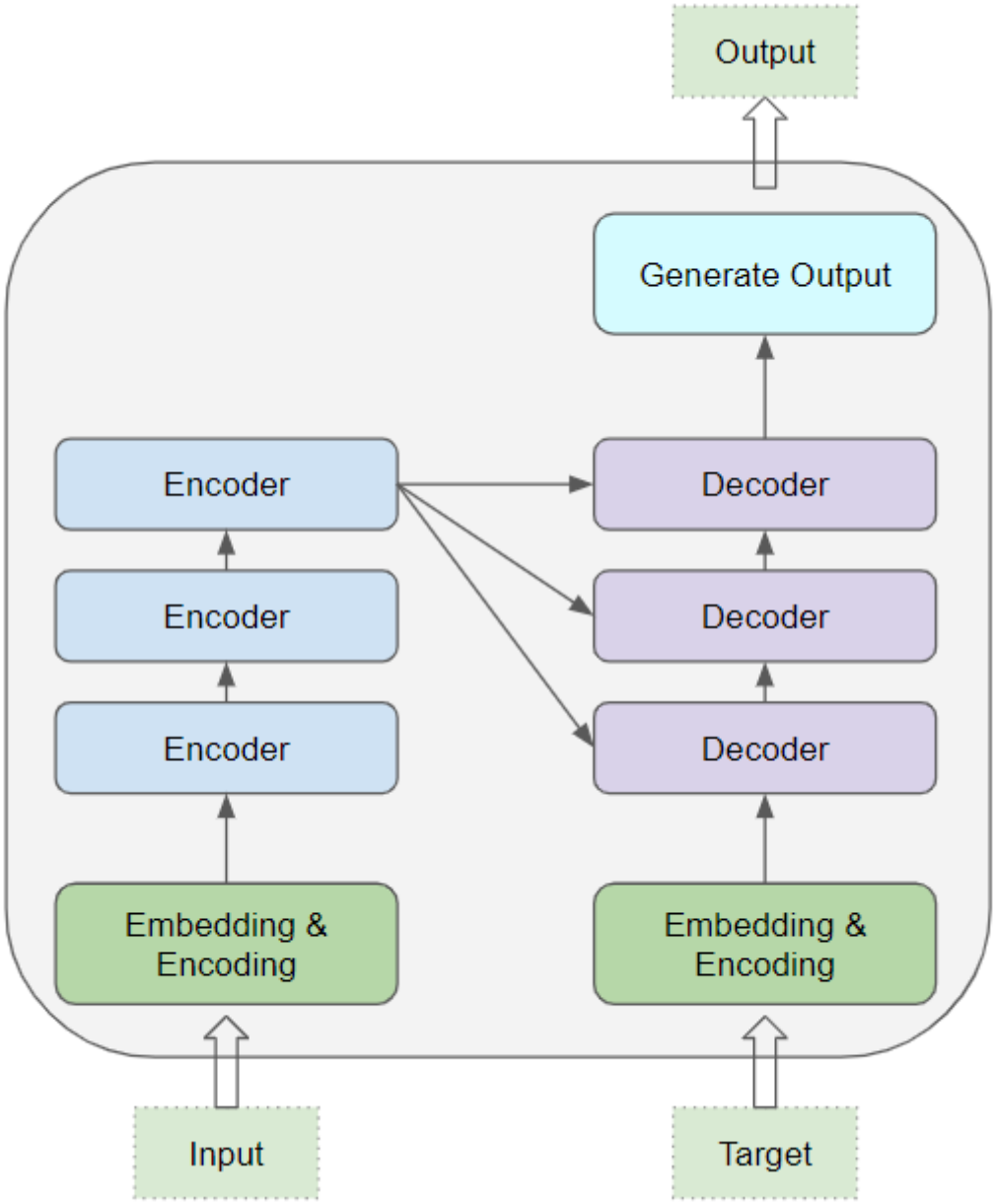
So, how do they do it?

They are specifically designed to comprehend context and meaning by analyzing the relationship between different elements, and they rely almost entirely on a mathematical technique called **attention to do so.**

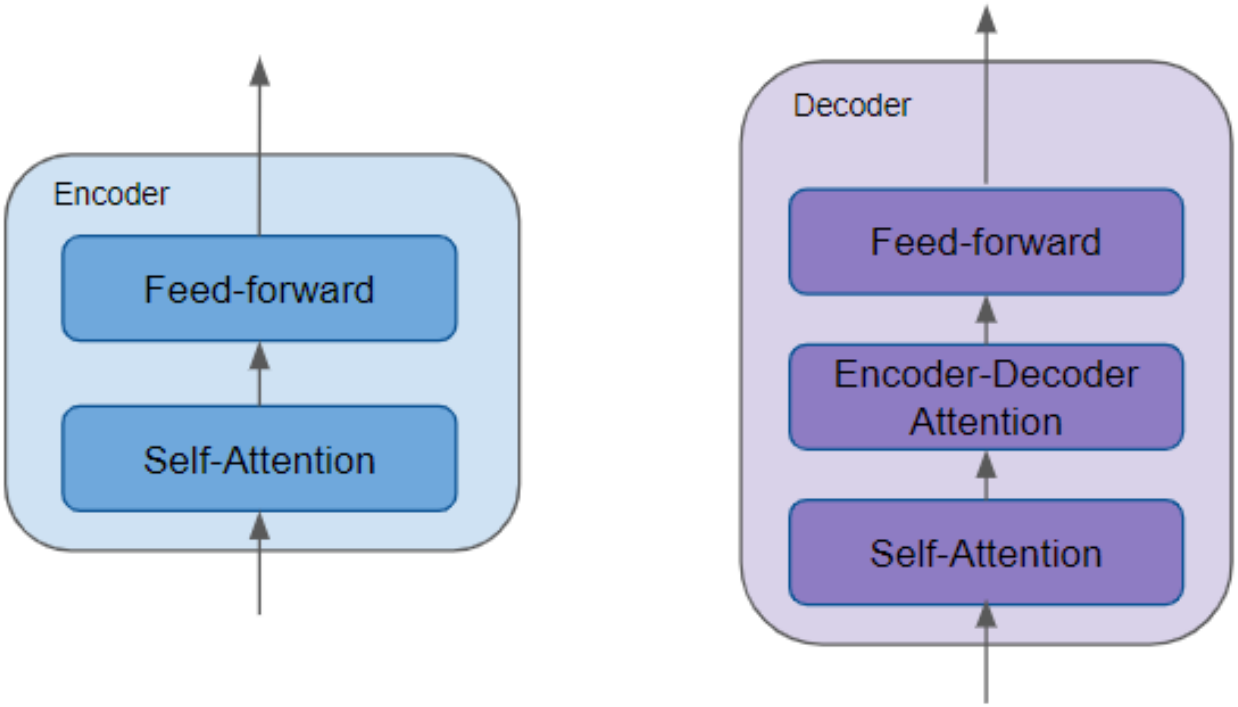


At its core, it contains a stack of Encoder layers and Decoder layers. To avoid confusion we will refer to the individual layer as an Encoder or a Decoder and will use Encoder stack or Decoder stack for a group of Encoder layers.

The Encoder stack and the Decoder stack each have their corresponding Embedding layers for their respective inputs. Finally, there is an Output layer to generate the final output.

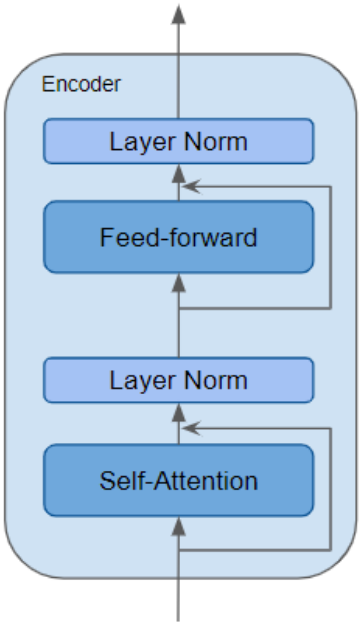


All the Encoders are identical to one another. Similarly, all the Decoders are identical.



- The Encoder contains the all-important Self-attention layer that computes the relationship between different words in the sequence, as well as a Feed-forward layer.
- The Decoder contains the Self-attention layer and the Feed-forward layer, as well as a second Encoder-Decoder attention layer.
- Each Encoder and Decoder has its own set of weights.

The Encoder is a reusable module that is the defining component of all Transformer architectures. In addition to the above two layers, it also has Residual skip connections around both layers along with two LayerNorm layers.



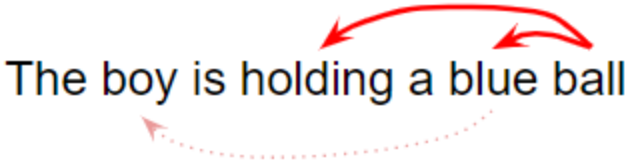
Reference: Transformers Explained Visually (Part 1): Overview of Functionality

What does Attention Do?

The key to the Transformer's ground-breaking performance is its use of Attention.

While processing a word, Attention enables the model to focus on other words in the input that are closely related to that word.

eg. 'Ball' is closely related to 'blue' and 'holding'. On the other hand, 'blue' is not related to 'boy'.



The Transformer architecture uses self-attention by relating every word in the input sequence to every other word.

eg. Consider two sentences:

- The *cat* drank the milk because **it** was hungry.
- The cat drank the *milk* because **it** was sweet.

In the first sentence, the word ‘it’ refers to ‘cat’, while in the second it refers to ‘milk. When the model processes the word ‘it’, self-attention gives the model more information about its meaning so that it can associate ‘it’ with the correct word.

To enable it to handle more nuances about the intent and semantics of the sentence, Transformers include multiple attention scores for each word.

eg. While processing the word ‘it’, the first score highlights ‘cat’, while the second score highlights ‘hungry’. So when it decodes the word ‘it’, by translating it into a different language, for instance, it will incorporate some aspect of both ‘cat’ and ‘hungry’ into the translated word.

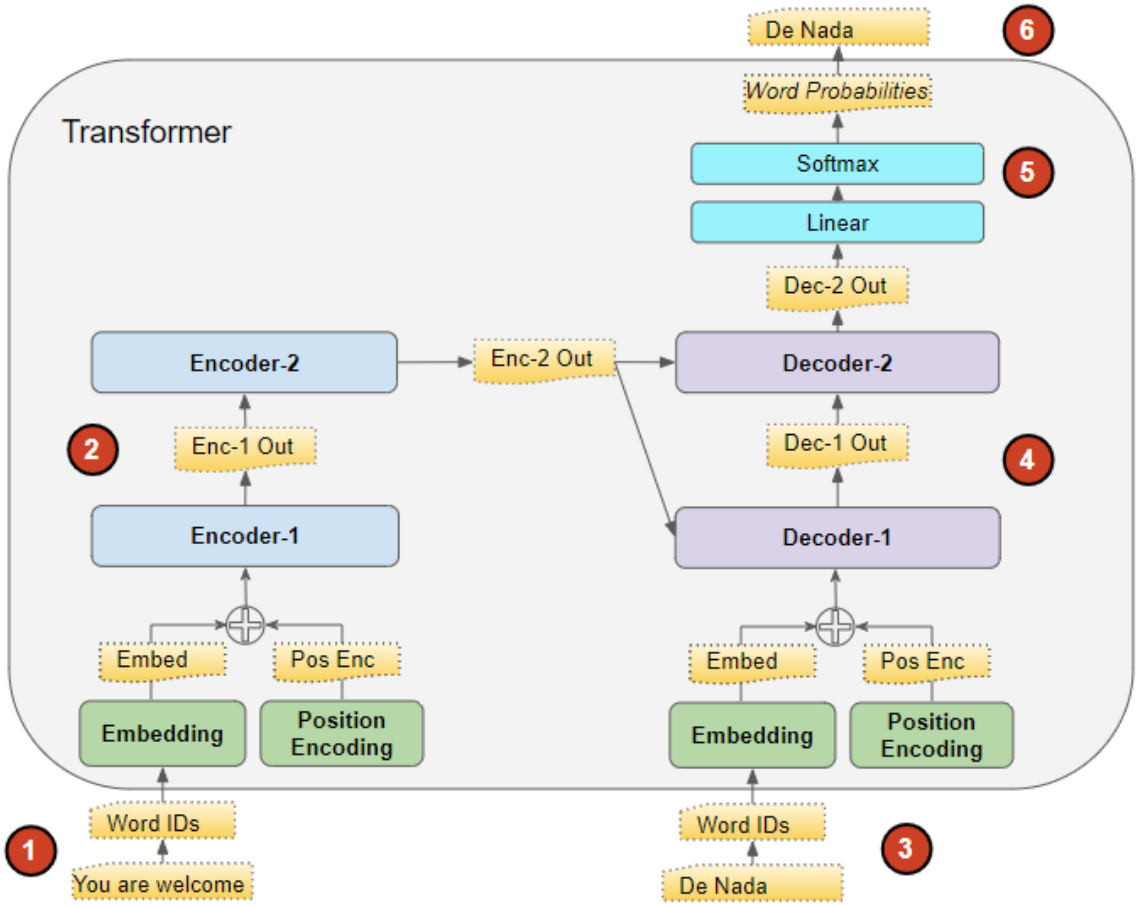
Training the Transformer

The Transformer works slightly differently during Training and while doing Inference.

Let's first look at the flow of data during Training. Training data consists of two parts:

1. The source or input sequence (eg. "You are welcome" in English, for a translation problem)
2. The destination or target sequence (eg. "De nada" in Spanish)

The Transformer's goal is to learn how to output the target sequence, by using both the input and target sequence.

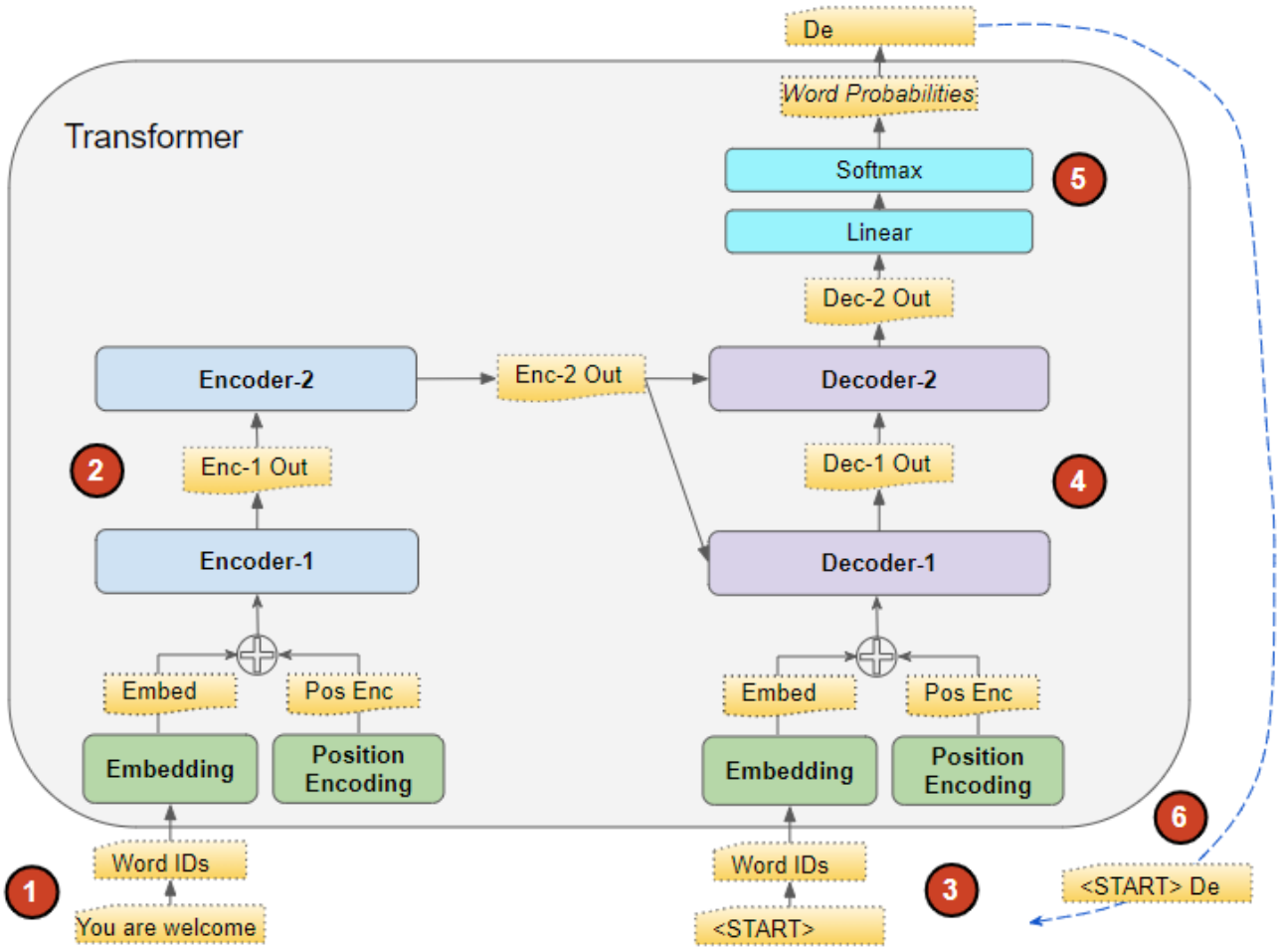


The Transformer processes the data like this:

1. The input sequence is converted into Embeddings (with Position Encoding) and fed to the Encoder.
2. The stack of Encoders processes this and produces an encoded representation of the input sequence.
3. The target sequence is prepended with a start-of-sentence token, converted into Embeddings (with Position Encoding), and fed to the Decoder.
4. The stack of Decoders processes this along with the Encoder stack's encoded representation to produce an encoded representation of the target sequence.
5. The Output layer converts it into word probabilities and the final output sequence.
6. The Transformer's Loss function compares this output sequence with the target sequence from the training data. This loss is used to generate gradients to train the Transformer during back-propagation.

Inference

7. During Inference, we have only the input sequence and don't have the target sequence to pass as input to the Decoder. The goal of the Transformer is to produce the target sequence from the input sequence alone.
8. So, like in a Seq2Seq model, we generate the output in a loop and feed the output sequence from the previous timestep to the Decoder in the next timestep until we come across an end-of-sentence token.
9. The difference from the Seq2Seq model is that, at each timestep, we re-feed the entire output sequence generated thus far, rather than just the last word.



The flow of data during Inference is:

1. The input sequence is converted into Embeddings (with Position Encoding) and fed to the Encoder.
2. The stack of Encoders processes this and produces an encoded representation of the input sequence.
3. Instead of the target sequence, we use an empty sequence with only a start-of-sentence token. This is converted into Embeddings (with Position Encoding) and fed to the Decoder.
4. The stack of Decoders processes this along with the Encoder stack's encoded representation to produce an encoded representation of the target sequence.
5. The Output layer converts it into word probabilities and produces an output sequence.

6. We take the last word of the output sequence as the predicted word. That word is now filled into the second position of our Decoder input sequence, which now contains a start-of-sentence token and the first word.
7. Go back to step #3. As before, feed the new Decoder sequence into the model. Then take the second word of the output and append it to the Decoder sequence. Repeat this until it predicts an end-of-sentence token. Note that since the Encoder sequence does not change for each iteration, we do not have to repeat steps #1 and #2 each time.

What are Transformers used for?

Transformers are very versatile and are used for **most NLP tasks** such as **language models and text classification**. They are frequently used in **sequence-to-sequence models** for applications such as Machine Translation, Text Summarization, Question-Answering, Named Entity Recognition, and Speech Recognition.

There are different flavors of the **Transformer architecture for different problems.** The basic Encoder Layer is used as a common building block for these architectures, with different application-specific 'heads' depending on the problem being solved.

Reference: What are Transformers used for?

🌍 Real-World Application: Language Translation

Transformers are the backbone of modern language models like **Google Translate** and **ChatGPT**. They enable accurate translation by

understanding the context and meaning of entire sentences rather than translating word by word.

How Transformers Differ from Fully Connected Networks

- **Sequence Awareness**: FCNs do not consider the order or relationships between input elements, making them unsuitable for sequence tasks. Transformers excel in this area.
- **Contextual Understanding**: Using attention, Transformers can understand the context of each input relative to others, which FCNs cannot do.
- **Architecture**: Transformers are modular, consisting of attention layers, feed-forward layers, and positional encodings—very different from the simple layer-by-layer structure of FCNs.

📚 Conclusion and Personal Reflection

Learning about CNNs and Transformers opened my eyes to how tailored architectures can solve very different types of problems. CNNs show how **visual features** can be captured with efficiency, while Transformers demonstrate how **context and relationships in sequences** can be modeled effectively. Unlike the simpler FCNs we studied in class, these **advanced networks highlight the creativity** and power of deep learning in tackling real-world challenges. This exploration has deepened my appreciation for how AI is applied across industries—from healthcare to language technology—and has motivated me to explore practical applications in **future projects.**

Reference: Vaswani, A., et al. (2017). "Attention is All You Need." Advances in Neural Information Processing Systems.

Question 2: Beyond Sigmoid: Activation Functions in Neural Networks

We'll discuss two activation functions

- **ReLU**
- **Tanh**

1. ReLU and Its Significance in Deep Learning

The **Rectified Linear Unit (ReLU)** is one of the most widely used **activation functions** in **deep learning models** today. Its popularity stems from its simplicity and remarkable performance in a variety of **neural network architecture**. Unlike traditional **activation functions** such as **sigmoid** and **tanh**, which can saturate and lead to slower training times, **ReLU** introduces a straightforward mechanism that accelerates the convergence of **neural networks**.

ReLU functions by outputting the input directly if it is positive; otherwise, it returns zero. This property not only maintains the **non-linearity** essential for learning complex patterns but also helps to mitigate issues such as the **vanishing gradient problem** that can plague deeper networks. As a result, **ReLU** allows models to learn more effectively, making it a preferred choice among practitioners and researchers alike.

- **Reference:** Introduction to ReLU and Its Significance in Deep Learning

It is defined mathematically as:

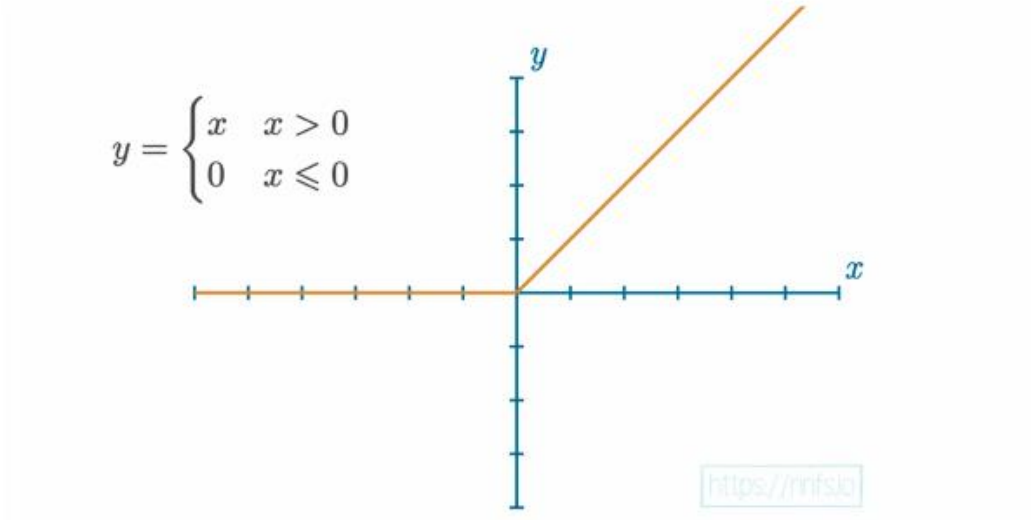
$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

This simple yet effective function outputs the input directly if it is positive; otherwise, it outputs zero. The function can also be expressed in a more compact form:

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

Graphical Representation of ReLU

The graph of the ReLU function clearly illustrates its behavior:



In this graph:

- The x-axis represents the input x.
- The y-axis represents the output f(x).
- For any positive input, the output is equal to the input (the line y=x).
- For any non-positive input, the output is zero.

Significance of ReLU in Deep Learning:

1. **Non-Linearity**: Despite being a simple linear function for positive values, **ReLU** introduces non-linearity to the model. This is crucial for neural networks because it allows them to learn complex patterns in the data. Without **non-linear activation functions**, **neural networks** would behave like **linear regression models**, limiting their capacity to fit complex datasets.
2. **Sparsity**: **ReLU** activation leads to sparsity in the activation of neurons. When the input is negative, the output is zero, effectively deactivating those neurons. This sparsity is beneficial as it reduces the number of active neurons, leading to more efficient computations and reduced risk of overfitting.
3. **Computational Efficiency**: **ReLU** is computationally efficient compared to traditional activation functions like **sigmoid** and **tanh**. This efficiency stems from its simple mathematical operation

(comparing and selecting), allowing faster training times, especially in deep networks.

4. **Mitigating the Vanishing Gradient Problem**: Unlike **sigmoid** and **tanh** functions, which can saturate (output values close to 0 or 1), **ReLU** does not suffer from the **vanishing gradient problem** for positive inputs. This characteristic allows for better gradient flow during **backpropagation**, enhancing the learning capability of deep networks.

Overall, **ReLU's** mathematical properties and behavior make it a fundamental component in modern deep learning frameworks, contributing significantly to the performance and efficiency of various neural network architectures.

Limitations of ReLU:

While the **Rectified Linear Unit (ReLU)** has become a staple in deep learning due to its advantages, it also has notable limitations that can impact the performance of neural networks. Understanding these limitations is essential for effectively utilizing **ReLU** and considering alternative activation functions when necessary.

1. **Dying ReLU Problem**:
   -> One of the most significant drawbacks of **ReLU** is the **"dying ReLU"** problem. This occurs when neurons become inactive and consistently output zero. If a large number of neurons in a network

output zero, they do not contribute to the learning process, effectively rendering them useless. This issue can arise when a large number of inputs to a neuron are negative, leading to persistent outputs of zero.
-> **Example**: During training, if the weights of a neuron are adjusted in such a way that it receives negative inputs most of the time, it will output zero for all subsequent inputs, and the gradient during backpropagation will also be zero. Consequently, the weights of this neuron will not be updated, stunting its learning potential.

2. **Unbounded Output**:
   -> While the linear output for positive values can be an advantage, it can also be a drawback. The output of **ReLU** is unbounded, which means it can take any positive value. In practice, this can lead to numerical instability, especially in deep networks, where the values can grow excessively large, complicating the training process.
   -> **Example**: If a layer produces very large outputs, it can cause gradient explosion during backpropagation, leading to erratic weight updates and making the training process unstable.

3. **Lack of Negative Outputs**:
   -> Since **ReLU** outputs zero for all negative inputs, it can lead to a loss of valuable information. In certain contexts, negative activations may carry significant information that could aid in learning more complex patterns. This limitation may hinder the model's ability to learn effectively from the data.
   -> **Example**: In cases where features have negative correlations with

the target variable, a model relying solely on **ReLU** may struggle to capture these relationships.

4. **Sensitivity to Initialization**:
   -> **ReLU** is sensitive to **weight initialization**. If the weights are initialized poorly, it can exacerbate the **dying ReLU problem**. For instance, if weights are set to very high values, it can lead to outputs that saturate at zero for many neurons, particularly in the early stages of training.
   -> **Example**: Using common initialization techniques (like **Xavier** or **He initialization**) can help, but even with these methods, poor initialization can still cause issues in deep networks.

5. **Comparison to Other Activation Functions**: Although **ReLU** is often preferred, it is not the only option. Alternative **activation functions** like **Leaky ReLU**, **Parametric ReLU (PReLU)**, and **Exponential Linear Unit (ELU)** were developed to address some of the limitations of **ReLU**, particularly the **dying ReLU problem**.

In summary, while **ReLU** is a powerful activation function that drives many successful deep learning models, it's essential to be aware of its limitations. Being mindful of these challenges allows practitioners to make informed decisions about when to use **ReLU** and when to consider alternatives that might better suit their specific application.

Reference: Applications of ReLU in Neural Networks

Advantages of ReLU Over Sigmoid

Feature	ReLU	Sigmoid
Range	[0, ∞)	(0, 1)
Gradient	Constant for x>0x > 0x>0	Vanishes as x→∞x \to \inftyx→∞
Speed	Faster to compute	More expensive (involves exp func)
Sparsity	Produces sparse activations	Dense activations
Avoids Vanishing Gradients	Yes (mostly)	Prone to vanishing gradients

👉 ReLU doesn't saturate for positive values, so gradients don't vanish as quickly — which is crucial in deep neural networks!

Common Use Cases

- Used in **hidden layers** of deep learning models, especially **Convolutional Neural Networks (CNNs)** for tasks like image classification (e.g., CIFAR-10, ImageNet).
- Also applied in **fully connected layers** of various deep architectures.

- **Reference:** ChatGPT

2. Tanh (Hyperbolic Tangent)

The hyperbolic tangent function, or tanh, is another non-linear activation function used in neural networks to learn more complex relationships in the data. It is based on the following mathematical formula:

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

The tanh function transforms the input value into the range between -1 and 1. In contrast to Sigmoid, the values are therefore distributed around zero. This results in some advantages compared to the previously presented activation functions, as the centering around zero helps to improve the training effect and the weight adjustments move faster in the right direction.

It is also advantageous that the tanh function also scales smaller input values more strongly in the output range so that the values can be better separated from each other, especially when the input range is close together.

Due to these properties, the hyperbolic tangent is often used in recurrent neural networks where temporal sequences and dependencies play an important role. By using positive and negative values, the state changes in an RNN can be represented much more precisely.

Compared to the sigmoid function, the hyperbolic tangent also struggles with the same problems. The vanishing gradient problem can also occur

with this activation function, especially with extremely large or small values. With very deep neural networks, it then becomes difficult to keep the gradients in the front part of the network strong enough to make sufficient weight adjustments. In addition, saturation effects can also occur in these value ranges, so that the gradient decreases sharply near 1 or -1.

Here are **some key properties of the tanh function**:

Range: The tanh function outputs values between -1 and 1. As the input approaches positive infinity, the output approaches 1, and as the input approaches negative infinity, the output approaches -1. The function is symmetric around the origin (0, 0).

Smoothness: The tanh function is smooth and differentiable over its entire range, making it suitable for optimization algorithms and gradient-based learning.

Non-linear: Like the sigmoid function, the tanh function is non-linear. It is capable of capturing non-linear relationships between inputs and outputs, enabling neural networks to learn complex patterns and make non-linear transformations.

Zero-Centered: One advantage of the tanh function compared to the sigmoid function is that it is zero-centered. This means that the average of the function's outputs for inputs around zero is close to zero. This property can help with training neural networks as it balances positive and negative activations.

Vanishing Gradient: Similar to the sigmoid function, the tanh function can suffer from the vanishing gradient problem. In deep neural networks, gradients can become very small during backpropagation, leading to slow learning or convergence issues. This limitation is mitigated by using alternative activation functions like ReLU or variants of the tanh function.

The tanh function is commonly used as an activation function in neural network architectures, especially in recurrent neural networks (RNNs) and certain layers of feedforward neural networks. It is particularly useful when the output needs to be in the range [-1, 1] or when zero-centered activations are desired.

In practical applications, the tanh function is used for tasks such as sentiment analysis, speech recognition, language modeling, and image classification, among others. However, it is important to note that other activation functions such as ReLU and its variants have gained popularity in recent years due to their computational efficiency and ability to mitigate the vanishing gradient problem.

Reference: Hyperbolic tangent (Tanh)

Advantages Over Sigmoid

- **Zero-centered output**: Unlike sigmoid, which outputs values between 0 and 1, tanh is centered at zero. This helps the optimization process, especially in deeper networks.
- **Stronger gradients**: In the range around 0, tanh's gradient is steeper than sigmoid's, which helps during training.

Common Use Cases

- Frequently used in **Recurrent Neural Networks (RNNs)** where modeling sequential data requires balanced, zero-centered activations.
- Often seen in **hidden layers** when data is normalized between -1 and 1.

Reflection and Personal Insight

During my exploration of these activation functions, I realized how subtle changes in mathematical behavior can lead to significant differences in model performance. While sigmoid served as a useful introduction to activation functions, ReLU and Tanh offer more practical benefits in deep learning applications. This deeper understanding helps me appreciate the design of modern neural networks and prepares me to make more informed choices in future projects.

Question 3: Exploring Loss Functions

In deep learning (DL), a **loss function** measures a model's performance by quantifying the deviation of its **predictions from the correct,** ground truth values. **Model optimization** involves adjusting parameters to **minimize the output** of a chosen loss function, **leading to improved accuracy and performance.**

Loss Functions in Deep Learning

In mathematical optimization and decision theory, a loss function (also referred to as a cost function or error function) assigns a numerical value to an event or prediction, indicating the associated "cost" or error.
In simple terms, a loss function evaluates how well an algorithm models a given dataset. It is a mathematical function dependent on the parameters of the machine learning model.

> **Reference:** Loss Functions in Deep Learning

Why Are Loss Functions Important?

Loss functions play a crucial role in machine learning because they:

1. **Guide Model Training:** The loss function is the foundation of the optimization process. Algorithms like Gradient Descent use it to adjust model parameters, reducing errors and improving predictive performance.
2. **Measure Performance:** By quantifying the difference between predicted and actual values, loss functions serve as benchmarks for

evaluating model performance. A lower loss value generally indicates a better model.

3. **Influence Learning Dynamics:** The choice of a loss function impacts the speed of learning and determines which types of errors are penalized more heavily. Different loss functions can lead to distinct learning behaviors and final outcomes.

Regression vs. Classification Loss Functions

Loss functions are categorized based on the type of machine learning task:

1. Regression Loss Functions

Regression models predict continuous numerical values, such as age, house prices, or stock values. Regression loss functions quantify the mismatch between predicted and target values.

Common regression loss functions include:

- **Mean Squared Error (MSE):** Computes the average squared difference between predicted and actual values, penalizing larger errors more heavily.
- **Mean Absolute Error (MAE):** Calculates the average absolute difference between predicted and actual values, treating all errors equally.

2. Classification Loss Functions

Classification models assign input data to discrete categories, such as "spam" vs. "not spam" in email filtering or digit recognition (0–9) in handwriting analysis. Classification loss functions measure the discrepancy between predicted class probabilities or labels and actual class labels.

Common classification loss functions include:

- **Binary Cross-Entropy (Log Loss):** Used for binary classification (e.g., spam vs. not spam), this function penalizes incorrect predictions by computing the negative log-likelihood of the true label.
- **Categorical Cross-Entropy:** Used for multi-class classification tasks, this function extends binary cross-entropy to multiple classes by summing the negative log-probabilities of the correct class labels.

Two common loss functions used in deep learning are:

1. Mean Squared Error (MSE)
2. Cross-Entropy Loss (CEL)

1. Mean Square Error (MSE)

In the fields of **regression** analysis and machine learning, the **Mean Square Error (MSE)** is a crucial metric for evaluating the performance of predictive models. It measures the average squared difference between the p**redicted** and the **actual target values** within a dataset. The primary objective of the MSE is to assess the **quality of a model's predictions** by measuring how closely they align with the ground truth.

Mathematical Formula

The MSE is calculated using the following formula

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

- MSE = Mean Square error

- n = Number of Data points
- Yi = Observed Values
- Y = Predicted Values

Reference: Mean Square Error (MSE)

Calculate Mean Square Error Using Excel

Now, you will learn how you can calculate the MSE using Excel. Suppose you have the sales data of a product of all the months.

1. **Step 1**: Enter the actual and forecasted data into two separate columns.

Month	Actual	Forecasted
January	67	70
February	50	44
March	36	38
April	74	44
May	84	64
June	84	80
July	64	54
August	34	44
September	23	43
October	72	90
November	62	56
December	42	38

2. **Step 2**: Calculate the squared error of each data

Month	Actual	Forecasted	Squared Error
January	67	70	9
February	50	49	1
March	36	38	4
April	74	76	4
May	84	83	1
June	84	80	16
July	64	67	9
August	34	30	16
September	23	20	9
October	72	75	9
November	62	60	4
December	42	38	16

3. **Step 3**: Calculate the Mean Squared Error

Month	Actual	Forecasted	Squared Error	
January	67	70	9	
February	50	49	1	
March	36	38	4	
April	74	76	4	
May	84	83	1	
June	84	80	16	
July	64	67	9	
August	34	30	16	
September	23	20	9	
October	72	75	9	
November	62	60	4	
December	42	38	16	
			8.166666667	**MSE**

MSE = (1/12) * (98) = 8.166

The MSE for this model is 8.17.

Reference: Simplilearn tutorial - Mean-Squared-Error

In statistics, the **mean squared error (MSE)** is a risk function that measures the **square of errors**. When performing regression, use MSE if you believe your target is normally distributed and you want large errors to be penalized more than small ones.

Mean Squared Error Applications

In addition to the mentioned central applications, the Mean Squared Error has several worthwhile applications in various contexts:

- **Model Selection:** MSE is commonly used to evaluate the performance of regression models. Therefore, the model that minimizes this measure will usually be chosen because it best fits data from training samples and thus is likely to make the most accurate predictions for new ones also.

- **Hyperparameter Tuning:** Most models can adjust certain parameters by hand, not determined through backpropagation. It is important to explore how different values for these hyperparameters affect the model's performance. It is often able to give the first sign so that when judging the effects of various hyperparameter values MSE can provide this.

- **Regression Diagnostics:** In regression analysis, a high mean squared error could signal something wrong with one's model. Examining the joint distribution of squared errors in machine learning can reveal patterns like heteroscedasticity (varying error spread) or nonlinearity (deviation from linear relationships) that the current model fails to capture.

- **Signal Processing:** In signal filtering and denoising, the square of the error is used as an objective criterion to minimize. How this technique can purify signals is as follows: by one means or another, these techniques eliminate noise in a signal and try to keep as much as possible from its original form.

- **Image Reconstruction:** Compression trades off between quality and achievability. The MSE assesses how well image reconstructions that have been compressed can provide the original. A reduced MSE value represents images that more closely match other sourced media files formed using certain techniques.

- **Financial Modeling**: Financial predictions from stock prices, futures market trends, and portfolio performance have models. The MSE tracks the accuracy of the input models. The smaller the error, the more accurate the models are updated for reliability, as reflected in investment decisions.

Common Mistakes and Pitfalls While Calculating Mean Squared Error

While Mean Squared Error is a valuable performance metric, **several reasons can lead to deceptive conclusions.** Here are the **most common mistakes and pitfalls that should be avoided**:

I. **Scale-dependent**: As we have already mentioned, MSE is correlated with the scale of the data. In this case, let us look at the example of the data that is used for predicting house prices. If the data is in dollars, a low MSE indicates that your model performs well. However, if the price data are in cents and thus are 100 times smaller than dollars, the MSE for the same prediction's correctness will be 10,000 times greater, which is rather misleading, especially for a comparison of different models of the same problem, but with different scales of input data. The possible solution is the following:

- **Standardization or normalization**: These approaches imply that the data for the variables would be in a similar range and scale. This way, the MSE comparison between two models can be viewed as fair.
- **Root Mean Squared Error (RMSE)**: RMSE allows representing MSE in the original data units. While it is not a complete solution, RMSE will provide more understandable information about the error size than its squared number.

II. **Doesn't account for outliers**: When it comes to data, outliers are the numbers that significantly differentiate from most of the data that my model uses for making predictions. Since MSE uses squared mistakes, it prioritizes the outliers. Hence, if there are just a few extra points, the modeling outcomes can be distorted. Even though the MSE value can be low, it does not mean that the data trend is discovered since it's "distracted" by the work with outliers. Here's the solutions for the same:

- **Robust Loss Functions**: You can consider other loss functions that are not outlier-sensitive. An example would be Mean Absolute Error: the average difference between the prediction and observation absolute values.
- **Outliers Detection**: You can also use the Interquartile Range method to detect the outliers. Depending on the possible correlations and explanations, you can decide whether to simply remove them or work with them using winsorization.

III. **Directionally-unaware**: The third most common mistake is being indifferent to the direction of the error. Low MSE does not imply that the model is unbiased. Analyzing the residuals' direction, its positivity/negativity can identify the signs. Here's the solution to this:

- **Residual Analysis**: You can use the plot with residuals to understand the biases.

- **X-Y Scatter**: The plot will show the difference between the target data points and their predicted values.

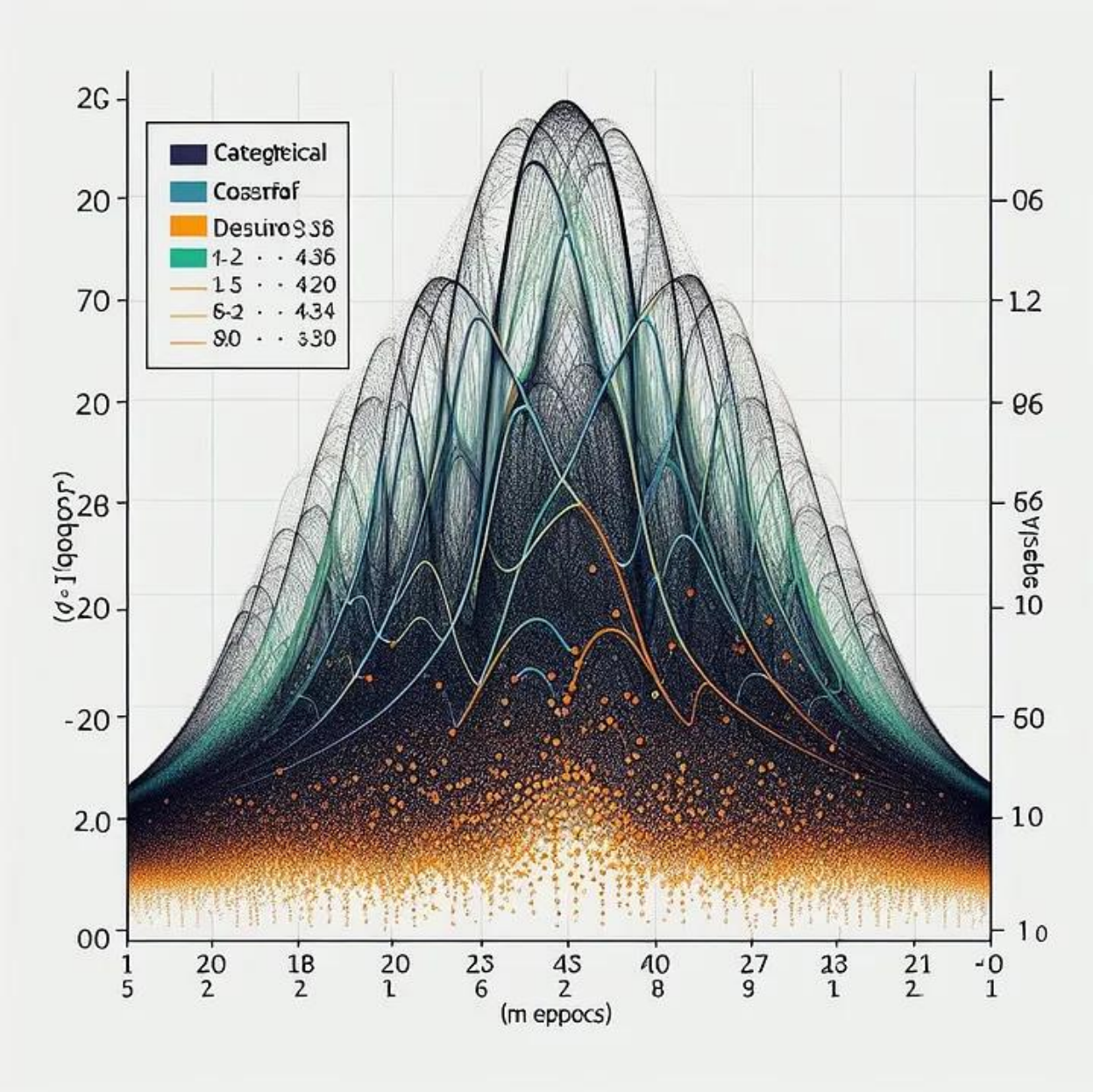
Reference: Interviewkickstart.com-blogs- Mean Squared Error- Examples and Application

A Real-World Example (Personal Reflection):

While working on a classification problem "Find Picture Category", I used categorical cross-entropy loss to train a deep learning model. Initially, I experimented with Mean Squared Error (MSE), but I realized it did not perform well because it treats class labels as continuous values rather than categorical labels. Switching to categorical cross-entropy significantly improved the model's convergence and accuracy.

2. Cross-Entropy Loss

Categorical Cross-Entropy is a **loss function** that is used in **multi-class classification tasks**. These are tasks where an example can belong to **one of many possible categories**, and the model must decide which one.



Reference: Categorical Cross-Entropy: Unraveling its Potentials in Multi-Class Classification

Categorical Cross-Entropy is widely used in many applications of machine learning, including:

- Image classification tasks, where an image must be classified into one of many categories.

- **Natural language processing tasks,** such as language translation and sentiment analysis, where a word or phrase must be classified into one of many categories.

Any other type of machine learning task that involves classifying an example into one of many possible categories.

$$L = -\frac{1}{N}\left[\sum_{j=1}^{N}[t_j \log(p_j) + (1 - t_j)\log(1 - p_j)]\right]$$

for N data points where t_i is the truth value taking a value 0 or 1 and p_i is the Softmax probability for the i^{th} data point.

If we were to calculate the loss of a single data point where the correct value is y=1, here's how our equation would look:

$$-(y \log(p) + (1 - y)\log(1 - p))$$

The predicted probability, p, determines the value of loss, l. If the value of p is high, the model will be rewarded for making a correct prediction - this will be illustrated with a low value of loss, l.

However, a low predicted probability, p, would infer the model was incorrect, and the binary cross-entropy loss function will reflect this by making the value of l higher.
For a multi-class classification task, cross-entropy (or categorical cross-entropy as it is often referred to) can be simply extended as follows:

$$-\sum_{c=1}^{N} y_c log(p_c)$$

Reference: Cross-Entropy Loss Function in Machine Learning: Enhancing Model Accuracy

Why Categorical Cross-Entropy is Suitable for These Tasks?

Handles Multi-Class Outputs Efficiently:

- **CCE** forces the model to output a probability **distribution over all possible classes**.
- This ensures that the **model assigns higher confidence** to the correct class while **reducing confidence in incorrect ones**.

Uses Softmax Activation for Probability Distribution:

- In multi-class problems, the **softmax function** is applied to the output layer, converting raw model scores (logits) into probabilities that sum to 1.
- This is essential for classification, as the model must predict **one class per instance**, not multiple.

Penalizes Incorrect Predictions More Significantly:

- The negative log-likelihood ensures that **incorrect predictions with high confidence** receive a higher penalty than low-confidence mistakes.
- This encourages the model to output a sharp probability distribution that strongly favors the correct class.

Interpretable Loss Values:

- Since CCE is based on probability distributions, it provides a well-defined measure of how well the model is performing in assigning the correct class labels.

Reference: ChatGPT says Why Categorical Cross-Entropy is Suitable for These Tasks?

Practical Applications of CEL

a) Image Classification (CNNs) Used in deep learning models like:

- **ResNet, VGG, EfficientNet** for object recognition.
- **Medical imaging** (tumor detection, pneumonia classification).

Reference: ResNet: Deep Residual Learning for Image Recognition (CVPR 2016)

b) NLP (Natural Language Processing)

- **Sentiment Analysis** (e.g., classifying reviews as positive/negative).
- **Spam Detection** (filtering spam vs. non-spam emails).

Reference: BERT: Pre-training of Deep Bidirectional Transformers (Google Research)

c) Autonomous Systems

- **Self-driving cars** classify objects like pedestrians, traffic lights, vehicles.
- **Facial recognition** in security and authentication.

Reference: YOLO: You Only Look Once - Object Detection (CVPR 2016)

Limitations of Cross-Entropy Loss

a) Overconfidence Issue

- **Deep networks** can become too confident, making small mistakes costly.

Solution: Add Label Smoothing to **prevent the model from assigning 100% probability** to one class.

Reference: Label Smoothing Regularization (NeurIPS 2019)

b) Sensitivity to Noisy Labels

- If the dataset contains incorrect labels, Cross Entropy Loss still forces the model to fit them, leading to overfitting.

Solution: Use Generalized Cross-Entropy Loss (GCE).

Reference: Generalized Cross-Entropy Loss (ICLR 2019)

c) Class Imbalance Problem

- If a dataset is highly imbalanced (e.g., 95% Class A, 5% Class B), CEL can **fail because it optimizes for the majority class**.

Solution: Use Weighted Cross-Entropy or Focal Loss.

Reference: Focal Loss for Dense Object Detection (ICLR 2017)

Comparison with Other Loss Functions (Critical Thinking):

Categorical cross-entropy is particularly useful in multi-class classification, but if I were dealing with highly imbalanced datasets, I would consider a weighted version or focal loss to address class imbalance.