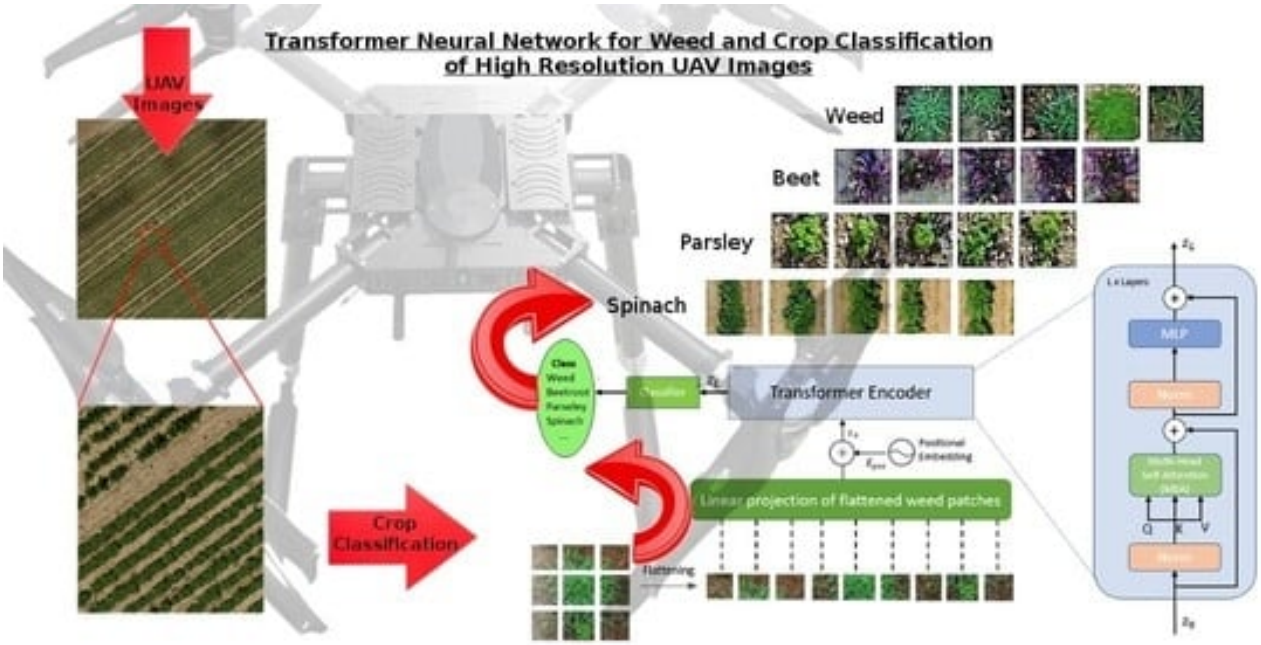


Transformer Neural Network

Dr. Sabzi

Advisor

Leili Motahari

BSc Student

2024

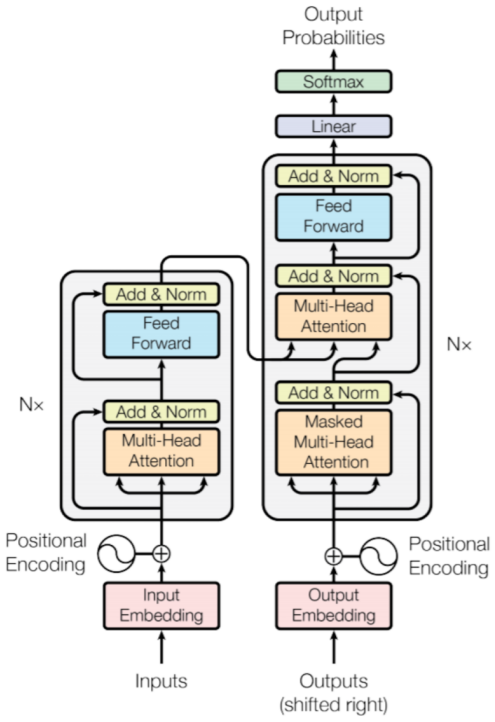
Contents

Chapter 1

Introduction

1.1 What is Transformer Neural Network?

A Transformer is a type of neural network architecture that has become the foundation for many state-of-the-art models in natural language processing (NLP) and beyond. It was introduced in a seminal paper titled "Attention is All You Need" by Vaswani et al. in 2017. The key innovation of the Transformer is its reliance on self-attention mechanisms, which allow it to process input data in parallel and capture complex dependencies in data.[4]



1.2 Key Components of Transformers

- **Attention Mechanism:** The attention mechanism allows the model to weigh the importance of different words in a sentence. This helps the model focus on the most relevant words.Consider the sentence "The quick brown fox jumps over the lazy dog." For the word "fox," the model might focus more on the words "quick" and "jumps." This means the model can determine that these words are more important for understanding "fox."
  How it works: The model assigns weights to each word relative to the target word. For example, the attention weight from "fox" to "jumps" might be 0.8, while from "fox" to "lazy" might be 0.2. These weights represent the relative importance of each word in understanding the target word.[5]

- **Multi-Head Attention:**
  Multi-head attention allows the model to attend to different parts of the input simultaneously, providing

a richer understanding of the text. In the sentence "The quick brown fox jumps over the lazy dog," the model can simultaneously focus on grammatical features (e.g., subject and verb) and semantic features (e.g., adjectives and nouns).
How it works: The model creates multiple attention heads, each focusing on different aspects of the input. The outputs of these heads are then combined. For instance, one head might focus on syntactic relationships, while another head focuses on word meanings.[5]

- **Positional Encoding:**
  Since Transformers do not inherently process sequential data in order (unlike RNNs), they use positional encodings to inject some information about the order of the sequence into the model. This can be done either through sine and cosine functions or learned embeddings.In the sentence "The quick brown fox jumps over the lazy dog," the word "The" is at the beginning and "dog" is at the end. Positional encoding provides this order information to the model.
  How it works: The model uses sine and cosine functions to generate a positional encoding vector for each position in the sentence. This vector is added to the word embeddings, allowing the model to understand the order of words.[5]

- **Layer Normalization** Normalization techniques are used within Transformers to stabilize the training process. Layer normalization is applied to the input of each sub-layer.If different inputs with varying values are fed into the model, layer normalization ensures that these variations are managed, making the training process more stable.
  How it works: Layer normalization computes the mean and variance of the inputs for each layer and normalizes them. This reduces the impact of fluctuations caused by different inputs.[5]

- **Feed-Forward Networks** Each layer of the Transformer contains a fully connected feed-forward network, which is applied to each position separately and identically.In the sentence "The quick brown fox jumps over the lazy dog," each word is processed through a feed-forward network. For instance, the word "fox" first goes through a linear transformation, followed by a ReLU activation, and then another linear transformation.
  How it works: The feed-forward network consists of two linear layers with a ReLU activation in between. This network is applied independently to each word, creating new features for each word.[5]

- **Encoder and Decoder** The original Transformer model is composed of an encoder to process the input and a decoder to produce the output. Each consists of a stack of identical layers that include multi-head attention and feed-forward networks.Suppose we want to translate the English sentence "The quick brown fox jumps over the lazy dog" into French. The encoder processes the input sentence and generates an internal representation. The decoder then uses this representation to generate the French sentence.
  How the Encoder works: The encoder consists of multiple layers of multi-head attention and feed-forward networks. The input is first processed through the attention layers and then through the feed-forward layers.
  How the Decoder works: The decoder also consists of multiple layers of multi-head attention and feed-forward networks, but in addition to attending to the input, it also attends to the previously generated output. This allows the decoder to generate accurate translations by considering both the input and the generated output.[5]

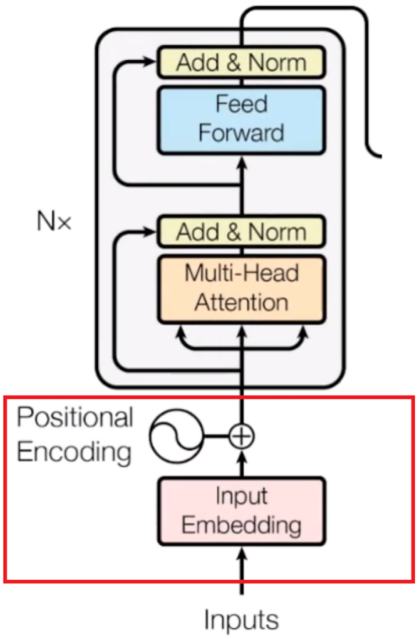
According to the provided details, Transformers work with an encoding and decoding process.[6]

1.3 Encoder Block [1]

1.3.1 Input Embedding

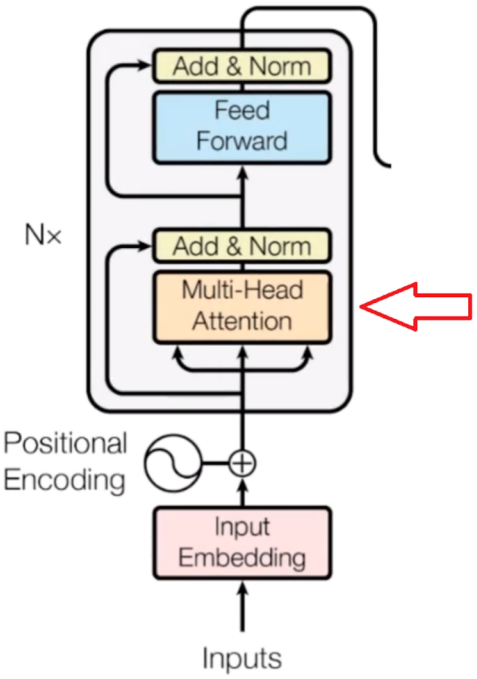
Computers don't understand words. Instead, they work on numbers, vectors or matrices. So, we need to convert our words to a vector. But how is this possible? Here's where the concept of embedding space comes into play. It's like an open space or dictionary where words of similar meanings are grouped together. This is called an embedding space, and here every word, according to its meaning, is mapped and assigned with a particular value. Thus, we convert our words into vectors.
One other issue we will face is that, in different sentences, each word may take on different meanings. So, to solve this issue, we use positional encoders. These are vectors that give context according to the position of the

word in a sentence.
Word → Embedding → Positional Embedding → Final Vector So, now that our input is ready, it goes to the encoder block.



1.3.2 Multi-Head Attention

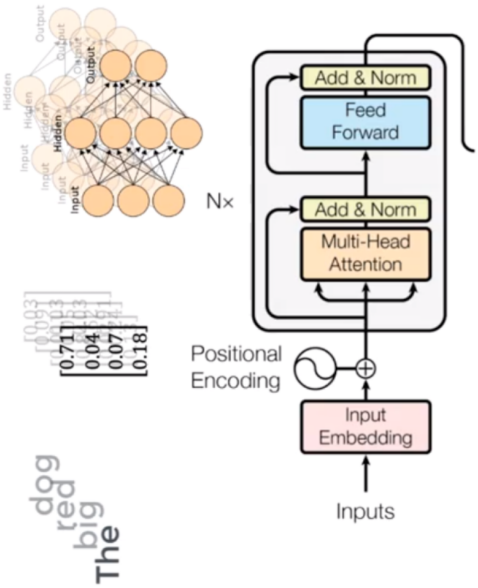
Now comes the main essence of the transformer: self attention.
This focuses on how relevant a particular word is with respect to other words in the sentence. It is represented as an attention vector. For every word, we can generate an attention vector generated that captures the contextual relationship between words in that sentence.
The only problem now is that, for every word, it weighs its value much higher on itself in the sentence, but we want to know its interaction with other words of that sentence. So, we determine multiple attention vectors per word and take a weighted average to compute the final attention vector of every word.
As we are using multiple attention vectors, this process is called the multi-head attention block.



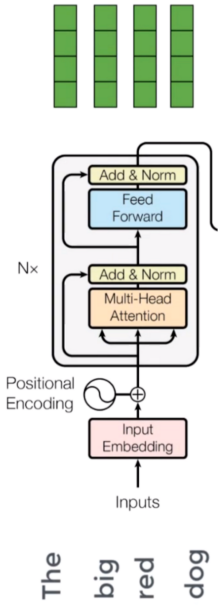
1.3.3 Feed Forward Network [2]

In simple terms, a feed-forward network is a multi-layered structure where information flows in one direction, from input to output. Unlike recurrent neural networks (RNNs) that allow information loops, feed-forward networks process data in a straight line.
In the context of transformers, the feed-forward network is a sub-unit within each encoder and decoder layer. It takes the output from the self-attention layer, which captures relationships between different parts of the input sequence, and transforms it further.
The feed-forward neural network consists of two dense layers:

1. First Dense Layer: This layer acts as the hidden layer and usually has more units than the input layer to extract more complex features.
2. Activation Function (ReLU): After the first layer, a non-linear activation function (usually ReLU) is applied to enable the model to learn non-linear relationships.
3. Second Dense Layer: This layer transforms the output of the hidden layer into the final form that is usable by the next encoder or decoder layer.



A feed-forward network is a multi-layered structure where information flows in one direction, from input to output. Unlike recurrent neural networks (RNNs) that allow information loops, feed-forward networks process data in a straight line.So, we can apply parallelization here, and that makes all the difference.

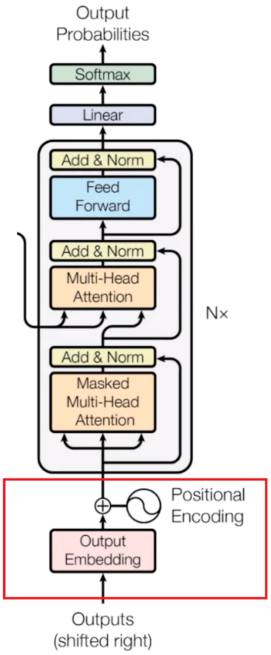


The Green part section represents the parallel processing of attention vectors obtained from the multi-head attention stage. These green vectors represent attention vectors that are processed independently and in parallel. Due to the independence of the attention vectors, each of these vectors can be processed separately in subsequent layers, enabling parallelization and increasing processing speed.

1.4 Decoder Block [1]

1.4.1 Output Embedding

Now, if we're training a translator for English to French, for training, we need to give an English sentence along with its translated French version for the model to learn. So, our English sentences pass through encoder block, and French sentences pass through the decoder block.
At first, we have the embedding layer and positional encoder part, which changes the words into respective vectors. This is similar to what we saw in the encoder part.



1.4.2 Masked Multi-Head Attention

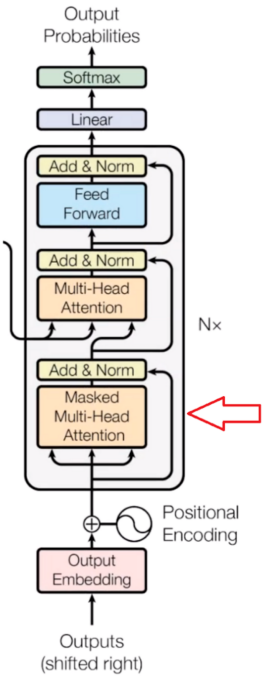
This block is similar to the multi-head attention block in the encoder, but with the added feature of masking. Masking prevents the model from seeing future tokens, helping it learn how to predict the next tokens without having access to future information.
Overall Process

1. English and French Sentences:
   - The English sentence is provided as input to the model and processed through the encoder block.
   - The French sentence is provided as input to the decoder block, but shifted to the right (shifted right) to prevent the model from seeing future tokens.
2. Generating Attention Vectors:
   - The masked multi-head attention block generates attention vectors for each word in the French sentence. These vectors represent how much each word is related to other words in the same sentence.
3. Learning Mechanism:
   - When an English word is given, the model translates it into its French version using previous results. This translation is then compared with the actual French translation fed into the decoder block.

- After comparison, the matrix values are updated. This process helps the model learn after several iterations.

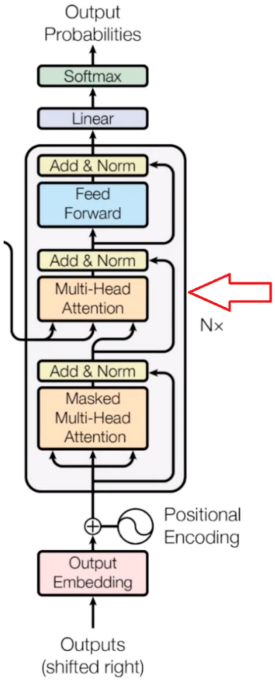
4. Masking:

   - To ensure proper learning, the next French word must be hidden. This prevents the model from using future information and forces it to predict the next word using only previous results.
   - Masking is achieved by transforming future words into zeros so the attention network can't use them.



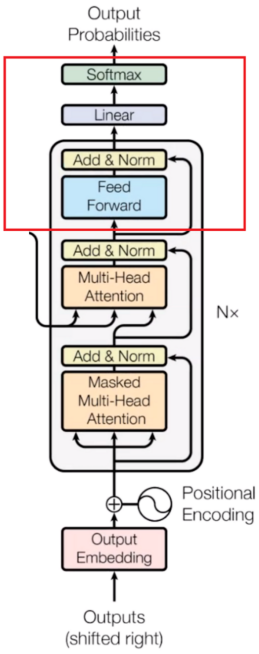
1.4.3 Multi-Head Attention

Now, the resulting attention vectors from the previous layer and the vectors from the encoder block are passed into another multi-head attention block. This is where the results from the encoder block also come into the picture. In the diagram, the results from the encoder block also clearly come here. That's why it is called the encoder-decoder attention block.
Since we have one vector of every word for each English and French sentence, this block actually does the mapping of English and French words and finds out the relation between them. So, this is the part where the main English to French word mapping happens.
The output of this block is attention vectors for every word in the English and French sentences. Each vector represents the relationship with other words in both languages.



1.4.4 Feed-Forward Neural Networks [3]

At this stage, if each attention vector is passed through a feed-forward unit, the output is transformed into a form that can be easily accepted by another decoder block or a linear layer. The linear layer is another feed-forward layer that expands the dimensions of the vectors to the number of words in the French language after translation.
Attention vectors resulting from the encoder-decoder attention block are passed into the feed-forward unit. This feed-forward unit processes the attention vectors and transforms them into a form that can be accepted by subsequent layers.
The output of the feed-forward unit is passed into a linear layer. The linear layer expands the dimensions of the vectors to the number of words in the French language.
The output of the linear layer is passed into a Softmax layer. The Softmax layer transforms the input into a probability distribution that is human-interpretable.



1.5 Applications of Transformers

Transformers have been highly successful in a variety of tasks such as:

Natural Language Processing (NLP)

- **Machine Translation**
  One of the primary and successful applications of transformers is in machine translation. Transformer models are capable of translating texts from one language to another with very high accuracy. For example, transformer-based models like Google Translate use this technique to translate sentences and texts.
- **Text Summarization**
  Transformers can convert long texts into short and useful summaries. This application is particularly useful in managing large information sets and scientific articles, where quick and accurate summarization can be highly beneficial.
- **Sentiment Analysis**
  Analyzing user reviews, social media comments, and customer feedback to determine sentiments, which can be positive, negative, or neutral. This application helps businesses understand customer opinions and improve their products or services accordingly.For example Amazon uses sentiment analysis to evaluate customer reviews and provide insights into product performance. By analyzing the sentiment of reviews, Amazon can identify common issues or highly praised features of products.
- **Question Answering**
  Powering conversational AI and chatbots that can understand and respond to human queries effectively. Examples include virtual assistants like Siri and Alexa, as well as customer support bots that can handle inquiries and provide relevant information. Virtual assistants like Siri, Google Assistant, and Alexa use transformers to understand and answer user questions. For instance, when you ask "What's the weather like today?" the assistant processes the query and provides a weather update.
- **Text Generation**
  Used in tools like OpenAI's GPT series, which can generate coherent, contextually relevant text based on a given prompt. This application is useful for content creation, writing assistance, and even generating code.

Speech Processing

- **Speech Recognition**
  Transformers are also used in speech recognition. This technology can convert human speech into text, which is useful in applications like voice assistants (such as Siri and Google Assistant) and speech translation services.
- **Speech Synthesis** Translating spoken language into text. Transformers help in understanding and transcribing spoken content with high accuracy.Google's Text-to-Speech and Amazon's Polly use transformers to convert written text into natural-sounding speech. This technology powers voice assistants and accessibility tools, enabling devices to read aloud text in a human-like voice. For example, you can type a message into a text-to-speech application, and it will be read out loud in a clear, natural-sounding voice.

Computer Vision

- **Image Classification** Vision Transformers (ViTs) apply the Transformer architecture to image patches for classifying images into categories. Google's Vision Transformer (ViT) models divide an image into patches and process them similarly to how transformers process words in a sentence. This approach has achieved state-of-the-art results in image classification tasks, such as identifying whether an image contains a cat, dog, car, etc.
- **Object Detection and Localization** Identifying objects within an image and their boundaries. Transformers help in processing various parts of an image to recognize and localize multiple objects accurately. DEtection TRansformers (DETR) is a model that uses transformers for object detection and localization. DETR can identify and draw bounding boxes around multiple objects in an image, such as recognizing people, cars, and bicycles in a street scene, and indicating their exact locations within the image.

- **Image Captioning**
  With adaptations to handle image inputs, transformers can be used for image captioning. In this application, models can generate textual descriptions for images, which can be useful in areas such as visual aids for the visually impaired and image categorization and search.

Multimodal Tasks

- **Video Processing**
  For tasks like action recognition or generating descriptions of video content, where the model needs to understand both visual elements and temporal dynamics. Transformers are used in models like TimeSformer for video action recognition. This model processes video frames as sequences and applies attention mechanisms to capture both spatial and temporal information, allowing it to recognize actions like running, jumping, or dancing in a video.
- **Visual Question Answering (VQA)** Answering questions based on the content of an image, where the Transformer integrates and processes information from both text and image inputs. The VQA model combines text and visual inputs to answer questions about images. For instance, given an image of a park and the question "How many people are sitting on the bench?" the model uses transformers to analyze the image and the question simultaneously to provide an accurate answer.

Healthcare

- **Drug Discovery**
  Predicting molecular properties and generating novel molecular structures, leveraging Transformers to understand complex patterns in chemical compounds. Pharmaceutical companies use transformers to predict the properties of new molecules and generate potential drug candidates. By analyzing vast datasets of chemical compounds, transformers can identify promising molecules that may lead to effective new medications, speeding up the drug discovery process.
- **Medical Image Analysis**
  Analyzing medical scans to detect abnormalities or assist in diagnostic processes, where adapting Transformers can help in interpreting complex image data. Hospitals use transformer models to analyze medical imaging data such as MRI or CT scans. These models can detect tumors, fractures, and other abnormalities with high accuracy, assisting radiologists in diagnosing conditions and planning treatments.

Finance

- **Fraud Detection**
  Analyzing transaction data to identify potentially fraudulent activities, where Transformers can process sequences of transactions to detect anomalies. Financial institutions use transformer models to analyze sequences of credit card transactions. By detecting patterns that deviate from typical user behavior, transformers can flag transactions that are potentially fraudulent, such as sudden large purchases in a different location or rapid successive transactions.
- **Algorithmic Trading**
  Using historical data to predict market trends and make automated trading decisions. Hedge funds and trading firms use transformers to analyze historical market data, news articles, and social media sentiment to predict future stock movements. Transformers can identify complex patterns in the data, enabling the development of trading algorithms that make real-time buy or sell decisions.

1.6 The applications of Transformer Neural Network in agriculture

Transformers, with their advanced capabilities in handling large and complex datasets, have found applications in the field of agriculture as well. Their ability to process and analyze various types of data can help in enhancing productivity, sustainability, and efficiency. Here are some specific applications of Transformer neural networks in agriculture:

1. **Crop Disease and Pest Detection** Transformers can be utilized to analyze images from drones or satellites to identify signs of crop diseases or pest infestations early. By processing sequences of images over time, they can detect changes in crop health that might indicate the presence of disease or pests, enabling timely intervention to prevent spread.

2. **Precision Agriculture**

   - **Soil and Crop Health Monitoring**
     Analyzing data from various sensors in the field (such as moisture, nutrients, and temperature sensors) to assess soil and crop health. Transformers can process this time-series sensor data to predict soil nutrient deficiencies or crop stress.
   - **Yield Prediction**
     Using historical data on weather conditions, crop performance, and management practices, Transformers can forecast crop yields. This helps farmers make informed decisions about resource allocation and harvest planning.

3. **Automated Weeding and Harvesting**

   - **Weed Detection**
     Identifying weeds among crops using imagery data, which can be used to guide precision herbicide application or robotic weed removal, minimizing herbicide usage and promoting sustainable practices.
   - **Robotic Harvesting** Analyzing real-time visual data to identify ripe fruits and vegetables for harvesting. Transformers can help in distinguishing between ripe and unripe produce, optimizing the timing and efficiency of harvesting operations.

4. **Supply Chain Optimization** Transformers can be used to model and predict various factors in the agricultural supply chain, such as demand forecasting, logistics, and distribution planning. By analyzing historical data and current market trends, Transformers can help in optimizing the supply chain, reducing waste, and improving market responsiveness.

5. **Climate Impact Modeling** By processing large datasets including historical weather patterns, crop performance data, and climate models, Transformers can help in predicting the impact of climate change on agricultural productivity. This information can be crucial for adapting farming practices to changing environmental conditions.

6. **Genetic Improvement and Breeding** Transformers can analyze genomic data alongside phenotypic data of plants to identify traits associated with desirable characteristics such as drought resistance or improved yield. This can accelerate the breeding process by predicting the outcomes of genetic crosses with higher accuracy.

In many of these applications, Transformers are often used in conjunction with other machine learning techniques and technologies such as computer vision, IoT (Internet of Things) devices, and geographic information systems (GIS). This integration allows for a more comprehensive approach to solving complex agricultural challenges. By leveraging their ability to efficiently process and analyze large amounts of diverse data, Transformers contribute significantly to advancing precision agriculture, enhancing sustainable practices, and improving overall farm management and productivity.

1.7 The advantages of Transformer neural network

The Transformer neural network architecture has several distinct advantages that have made it a popular choice across various fields, particularly in natural language processing (NLP) and increasingly in other domains like computer vision and audio processing. Here are some of the key advantages of Transformer models:

1. **Parallelization**
   Unlike recurrent neural networks (RNNs), Transformers do not process data sequentially. This allows for much more efficient training as more computations can be performed in parallel. This parallelization significantly speeds up training times, especially on modern hardware optimized for parallel computations such as GPUs.

2. **Handling Long-Range Dependencies**
   Transformers are particularly adept at handling long-range dependencies in data. The self-attention mechanism allows the model to weigh the importance of each part of the input data, regardless of their positional distances. This capability enables Transformers to capture more context and nuances compared to models that process data sequentially, where distant elements might have less influence.

3. **Scalability**
   The architecture of Transformers allows them to be scaled up (more layers, larger embedding sizes, more heads in multi-head attention) to handle larger datasets and more complex tasks. This scalability has been demonstrated in models like GPT-3 and T5, which show improved performance with an increasing number of parameters.

4. **Versatility**
   Originally designed for NLP tasks, the Transformer architecture has proven to be highly adaptable and effective across different domains, including computer vision (Vision Transformers), audio signal processing, and even tasks that involve multimodal inputs (e.g., image captioning, visual question answering).

5. **Improved Performance on Benchmark Tasks**
   Since their introduction, Transformers have consistently achieved state-of-the-art results on a variety of benchmark datasets across multiple tasks, including machine translation, text summarization, and sentiment analysis. Their ability to model complex patterns and relationships in data has set new standards in AI performance.

6. **Less Reliance on Domain-Specific Engineering**
   Transformers require less feature engineering and domain-specific tweaks compared to earlier models. Their ability to learn from data what is important reduces the need for manual feature selection, which is a significant advantage in rapidly evolving fields or in applications where expert knowledge is scarce.

7. **Dynamic Attention**
   The attention mechanism can dynamically focus on different parts of the input data depending on the task at hand. This allows Transformers to be more context-aware and responsive to the specifics of the input, enhancing their accuracy and effectiveness.

8. **Transfer Learning Capabilities**
   Transformers can be pre-trained on large datasets and then fine-tuned for specific tasks with smaller amounts of data. This transfer learning approach, popularized by models like BERT and GPT, enables effective model deployment even when task-specific data is limited.

1.8 What are the limitations of Transformer Neural Network?

While Transformer neural networks have many strengths, they also come with several limitations and challenges. Understanding these limitations is crucial for effectively applying and improving Transformer-based models. Here are some notable limitations:

1. **High Computational Costs**
   Transformers require significant computational resources, particularly for training. Large models like GPT-3 involve millions or even billions of parameters, necessitating powerful GPUs or TPUs and substantial energy consumption.

2. **Memory Requirements**
   In the self-attention mechanism, each token (a text unit like a word) in a sequence attends to all other tokens in the same sequence. This process allows the model to capture long-term and complex dependencies between tokens. It is achieved by computing an attention matrix, the size of which depends on the number of tokens in the sequence.
   Attention Matrix: Suppose we have a sequence with n tokens. In the self-attention mechanism, an attention matrix of size $n \times n$ is computed. This matrix indicates how much each token should attend to every other token.
   Time and Space Complexity: The computation and storage of this matrix have a time and space complexity of $O(n^2)$. In other words, if the number of tokens doubles, the amount of computation and memory required increases fourfold.

3. **Data Hungray**
   Dependence on Large Datasets: Transformers generally perform better when trained on vast amounts of data. Their performance can be suboptimal on smaller datasets, where traditional models or those with more inductive biases might outperform them.

4. **Overfitting**
   Prone to Overfitting in Small Data Scenarios: Due to their large number of parameters and capacity, Transformers are susceptible to overfitting when the available training data is not sufficiently large or diverse.

5. **Interpretability**
Black Box Nature: like many deep learning models, are considered "black boxes." This means that the internal decision-making process of these models is difficult to observe and understand. Unlike simpler models such as linear regression or decision trees, which can provide clear and understandable explanations for their predictions, transformers are composed of multiple complex layers of nonlinear computations.
Bias and Fairness
Propagation of Bias: Transformers can perpetuate and even amplify biases present in their training data. This can lead to fairness issues, such as biased language generation, reinforcing stereotypes, or unfair decision-making processes.

6. **Efficiency at Scale**
Diminishing Returns: As Transformers are scaled up, the incremental improvements in performance can diminish relative to the additional computational costs. Balancing this scale-efficiency trade-off is a key challenge in model design and deployment.

7. **Handling of Sequential Data**
While transformers excel at handling dependencies in long sequences, they may struggle with capturing temporal dynamics as effectively as some recurrent models, particularly in tasks where precise timing and order are crucial.For example,In stock price prediction, accurately capturing the temporal dynamics and sequential order of price changes is essential. Let's compare how a transformer model and a recurrent neural network (RNN) might perform in this task.
Transformer Model Strengths:

- Can process long sequences of stock prices and capture long-term dependencies.
- Uses self-attention to consider all past prices simultaneously, which helps in understanding overall trends.

Transformer Model Weaknesses:

- Might miss subtle, time-specific patterns such as short-term fluctuations or immediate reactions to market events.
- The self-attention mechanism does not inherently account for the temporal order of events, which can be critical in stock price prediction.

RNN Strengths:

- Processes data sequentially, maintaining the order of price changes.
- Better at capturing time-specific patterns and short-term dependencies due to its sequential nature.

RNN Weaknesses:

- Struggles with long-term dependencies and can suffer from issues like vanishing gradients in very long sequences.
- Generally slower to train compared to transformers due to its sequential processing nature.

8. **Generalization Across Tasks**
One of the significant challenges in using transformer neural networks is the need for task-specific tuning to achieve optimal performance. Although these models can be pre-trained on general tasks and then fine-tuned for specific applications, each specific task requires special adjustments. These adjustments may include extensive hyperparameter optimization and adaptations that consume considerable resources.
Pre-training In the pre-training stage, transformer models are trained on large and general datasets such as Wikipedia or a collection of books. This stage allows the model to learn general language features. Pre-training on large datasets enables the model to capture broad and general patterns that can be beneficial across various tasks.
Fine-tuning After pre-training, the model needs to be fine-tuned for specific tasks. This process involves training the model on datasets related to the particular task, such as text classification, machine translation, or sentiment analysis. Fine-tuning requires precise adjustments and optimizations for each specific task to ensure the model achieves the best performance.

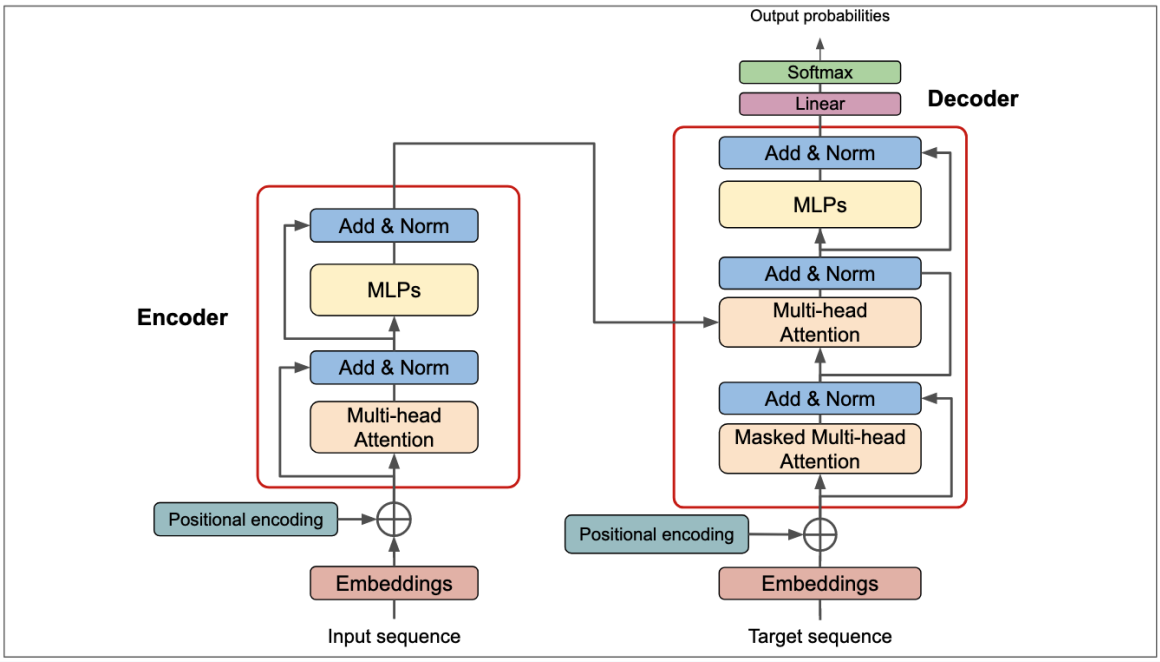
9. **Local vs. Global Attention**
Trade-offs in Attention Mechanisms: The original Transformer design uses global self-attention, where each token attends to every other token. This might not always be optimal, especially for tasks where local context is more relevant. Modifying attention mechanisms to better suit specific requirements can add complexity to model architecture.

10. **Environmental Impact**
    Carbon Footprint: The environmental impact of training and deploying large-scale Transformer models, given their energy demands, is a growing concern. This includes the carbon footprint associated with their use, which is critical in the context of global efforts to reduce carbon emissions.

These limitations highlight the trade-offs involved in using Transformer models and underscore the importance of ongoing research into more efficient, transparent, and fair AI systems. Efforts such as developing more computationally efficient architectures, enhancing model interpretability, and addressing biases in training data are essential steps forward.

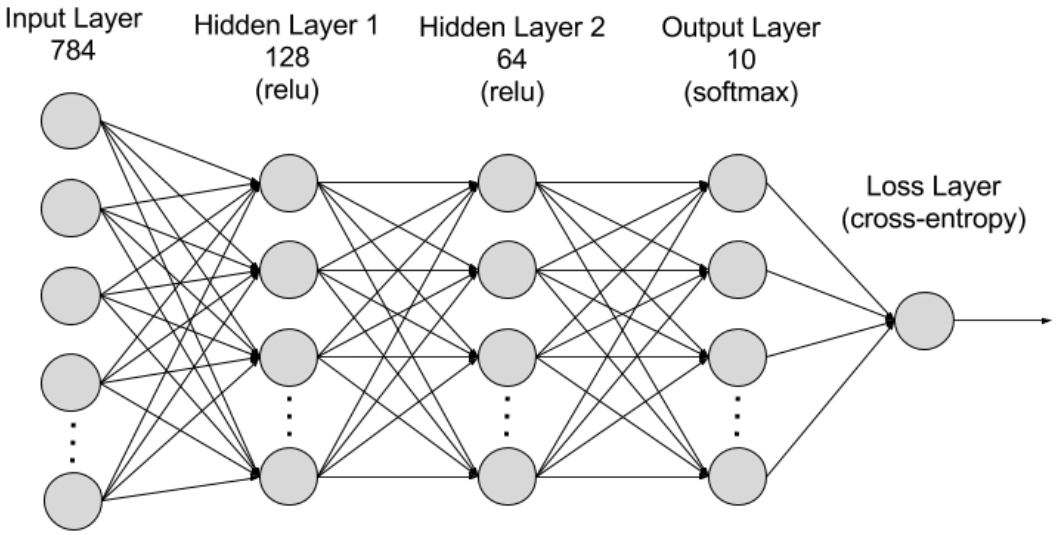
1.9 Differences Between Transformer, Recurrent, Convolutional, Spiking, and Siamese Neural Networks

- **Transformer Neural Networks [7]**



  - **Primary Feature:** Utilizes self-attention mechanisms to process sequences of data. Transformers weigh the importance of different parts of the input data irrespective of the sequence order, allowing parallel processing of the data.
  - **Advantages:**Highly effective at handling long-range dependencies and capable of parallel computation, which significantly speeds up training.
  - **Applications:**Primarily used in natural language processing (NLP) for tasks like translation, text summarization, and sentiment analysis. Also adapted for use in computer vision and other data-rich fields.

- **Recurrent Neural Networks (RNNs) [8]**

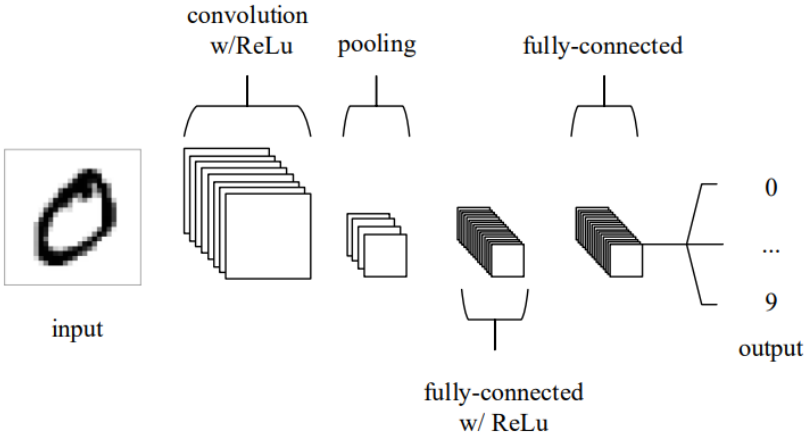


  - **Primary Feature:**Recurrent Neural Networks (RNNs) are a type of neural network architecture primarily used to detect patterns in a sequence of data. Such data can include handwriting, genomes, text, or numerical time series, which are often produced in industrial settings (e.g., stock markets or sensors). However, RNNs can also be applied to images if these are decomposed into a series of patches and treated as a sequence.RNNs are made of neurons: data-processing nodes that work together to perform complex tasks. The neurons are organized as input, output, and hidden layers. The input layer receives the information to process, and the output layer provides the result. Data processing, analysis, and prediction take place in the hidden layer.
    Hidden Layers: RNNs work by passing the sequential data that they receive to the hidden layers one step at a time. However, they also have a self-looping or recurrent workflow: the hidden layer can remember and use previous inputs for future predictions in a short-term memory component. It uses the current input and the stored memory to predict the next sequence.For example, consider the sequence: Apple is red. You want the RNN to predict red when it receives the input sequence Apple is. When the hidden layer processes the word Apple, it stores a copy in its memory. Next, when it sees the word is, it recalls Apple from its memory and understands the full sequence: Apple is for context. It can then predict red for improved accuracy. This makes RNNs useful in speech recognition, machine translation, and other language modeling tasks.
  - **Advantages:**Naturally suited for time-series data or any sequence where the order of data points is crucial.
  - **Applications:**Widely used for language modeling, speech recognition, and other tasks where the temporal sequence of data inputs is important.

- **Convolutional Neural Networks (CNNs)[9]**

  - **Primary Feature:**CNNs primarily focus on the basis that the input will be comprised of images. This focuses the architecture to be set up in way to best suit the need for dealing with the specific type of data. CNNs are comprised of three types of layers. These are convolutional layers, pooling layers and fully-connected layers.
    * **Convolutional Layers:** These layers are responsible for extracting various features from the input image.
    * **Pooling Layers:** These layers reduce the size of the extracted features and help to lower computational requirements and prevent overfitting.
    * **Fully-Connected Layers:** These layers are used for final decision-making and classification.

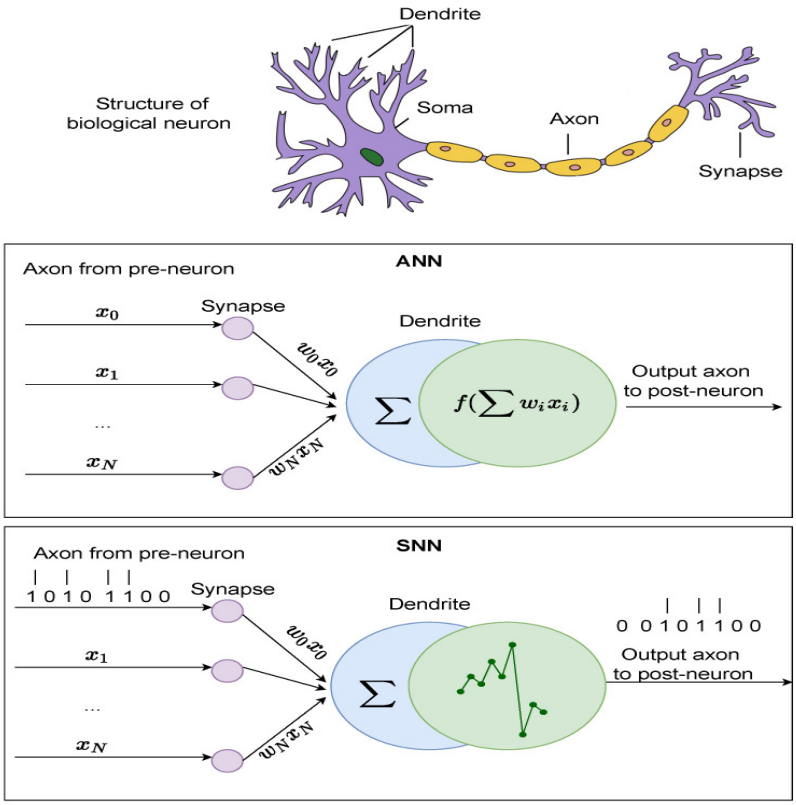
    When these layers are stacked, a CNN architecture has been formed. A simplified CNN architecture for MNIST classification is illustrated in the picture below.



- **Advantages:**Efficient in parameter use, highly effective at capturing spatial patterns, and relatively invariant to the position of features in the input space.
- **Applications:**Dominant in image and video recognition, also used in any applications involving spatial data, such as medical image analysis and autonomous vehicle navigation.

• Spiking Neural Networks (SNNs)[10]

- **Primary Feature:**In the field of neuroscience and neural networks, a spike refers to an action potential, which is a brief and intense electrical signal produced by neurons that is used to transmit information within the nervous system. The ability to simultaneously record the activity of multiple cells has led to the idea that the time difference between spikes in different neurons and the spike timing itself can have functional significance. Since the firing rate model cannot handle this perspective, a model describing the timing of spikes and the variation of the sub-threshold membrane potential has been investigated.A model that handles the generation of such spikes is distinguished from the firing rate model and is called the spiking model. Such neuron models are generally expressed in the form of ordinary differential equations. The figure below depicts the differences between the biological neuron, artificial neuron, and spiking neuron.

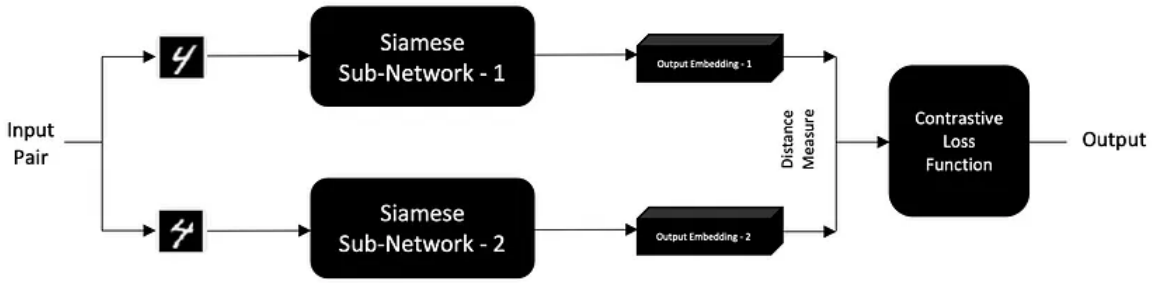


In spiking models, each neuron has a membrane potential that changes with incoming inputs. When the membrane potential reaches a certain value (threshold), the neuron generates a spike (action

potential). After generating a spike, the membrane potential returns to its resting state. Neurons produce spikes as brief and rapid signals, which are transmitted to other neurons. The connections between neurons have weights that determine the impact of incoming spikes on the neuron's membrane potential. Synaptic weights change based on learning algorithms to enable the network to perform better over time.
Spiking Neural Networks (SNNs) can process information temporally, meaning that the precise timing of spikes carries important information. The network continuously updates synaptic weights using various learning algorithms, such as Spike-Timing-Dependent Plasticity (STDP).

- **Advantages:**Potentially more power-efficient, especially when implemented on neuromorphic hardware. They offer a more biologically plausible model of neural computation.
- **Applications:** Emerging applications in robotics, edge computing, and any area where low power consumption and real-time processing are critical.

- **Siamese Neural Networks (SNNs)[11]**
  - **Primary Feature:** Neural Networks have taken strides in recent years. From ANN, RNN, CNN to the state of the art Transformers, we all know how data-hungry all these networks are. As we know, sometimes it is not the amount of data needed to learn a task but the number of features and variance involved in the feature embeddings.Face Recognition, finding out about fingerprints or handwriting recognition and even trying to detect a forgery through fake signatures. These tasks do not require a large dataset. Instead, they need is to learn the specific features from the high-dimensional inputs. One such network which lets you accomplish all this is Siamese Neural Network or SNN.
  SNN is a novel artificial neural network that consists of two identical sub-networks. These two sub-networks join together at their output. The initialisation for both the sub-networks involves the same weights and parameters. Parameter updating while training also gets shared across these networks. This is done to avoid any differences in feature learning between the two networks.
  The two sub-networks (Fig-1 provided below) extract features from the two inputs. The first input is the main anchor image (Top-left image in Fig-1). The second image is either a negative pair or a positive pair (Bottom-left image in Fig-1 is the augmented image of the anchor image. Hence, a positive pair). These images are then passed onto the respective sub-networks having the same weight and the parameters which result in the output feature vector for both the inputs respectively.These feature vectors are then further combined to compute the distance measure (or neighbourhood relationship) between the two outputs. This distance measure forms the input for the loss function. The loss function which we will talk about in the later section computes the loss w.r.t. the target value to back-propagate the error and hence continue learning of the network.



  - **Advantages:** Effective in tasks where labeled data is limited, especially useful for learning embeddings that map input data to vectors in a way that distances between vectors represent the similarity of inputs.
  - **Applications:** Commonly used in face verification, signature verification, and other applications where the similarity between two inputs needs to be assessed.

Each of these architectures is tailored to specific types of data and problems, reflecting the diversity and adaptability of neural networks in tackling various challenges across computational fields.

Chapter 2

A Step by step Guide to Transformers

2.1 Training

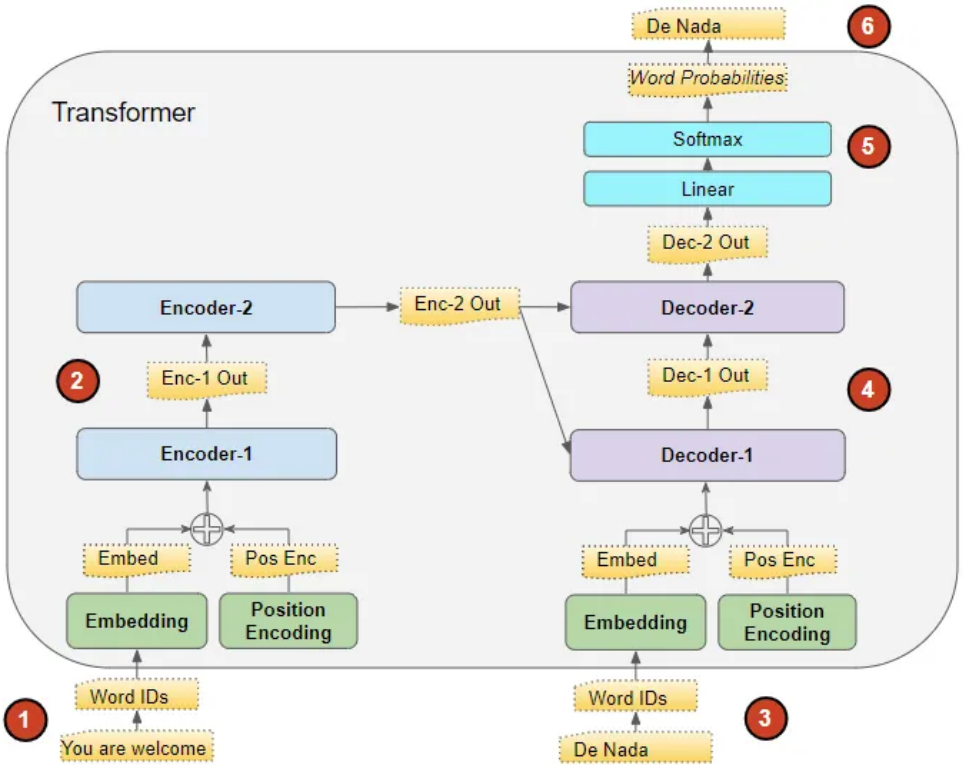


Figure 2.1: Trainig Flow [12]

Step 1 : Data Preparation

1. **Collect Data:** Gather a dataset suitable for the task, such as pairs of sentences in different languages for translation.
2. **Preprocess Data:**
   - **Tokenization:** Convert text into tokens (words, subwords, or characters).

Figure 2.2: Tokenization

   - **Encoding:** Convert tokens into numerical IDs using a vocabulary.

- **Add special tokens:** For example, start-of-sequence (SOS) and end-of-sequence (EOS) tokens.
- **Padding:** Pad sequences to a uniform length for batch processing.
- **Embedding** Once the input has been tokenized, it's time to turn words into numbers. For this, we use an embedding. In a previous chapter you learned about how text embeddings send every piece of text to a vector (a list) of numbers. If two pieces of text are similar, then the numbers in their corresponding vectors are similar to each other (componentwise, meaning each pair of numbers in the same position are similar). Otherwise, if two pieces of text are different, then the numbers in their corresponding vectors are different.[13]

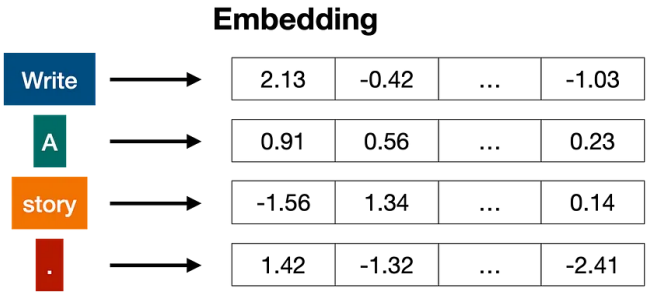


Figure 2.3: Embedding

3. **Positional encoding:** Once we have the vectors corresponding to each of the tokens in the sentence, the next step is to turn all these into one vector to process. The most common way to turn a bunch of vectors into one vector is to add them, componentwise. That means, we add each coordinate separately.Positional encoding consists of adding a sequence of predefined vectors to the embedding vectors of the words. This ensures we get a unique vector for every sentence, and sentences with the same words in different order will be assigned different vectors.[13]

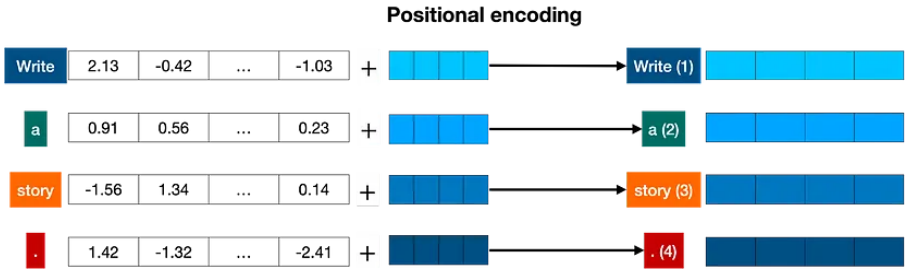


Figure 2.4: Positional Encoding

4. **Split Data:** Divide the dataset into training, validation, and test sets.

Step 2: Model Configuration

1. **Define Hyperparameters:**
   - Model size (dimensions of embeddings and hidden layers)
   - Number of layers in the encoder and decoder
   - Number of heads in the multi-head attention mechanism
   - Vocabulary size
   - Sequence length
   - Learning rate, batch size, etc.
2. **Build Model Architecture:**
   - **Encoder and Decoder Layers:** Each consists of multi-head attention mechanisms, feed-forward neural networks, and normalization layers.

- **Embedding Layers:** For converting token IDs into vectors. Positional encodings are added to embeddings to incorporate sequence order information.
- **Output Layer:** Typically a linear layer followed by a softmax to generate probabilities over the vocabulary.

Step 3: Training Setup

1. **Initialize Parameters:** Randomly initialize the model's weights or use pre-trained embeddings.
2. **Optimization Algorithm:** Choose an optimizer (commonly Adam).
3. **Loss Function:**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.

Step 4: Model Training

1. **Forward Pass:**
   - Input data through the encoder to generate context.
   - Pass encoder outputs and decoder inputs through the decoder to generate predictions
2. **Calculate Loss:** Compare the predictions against the true outputs using the loss function.
3. **Backward Pass:**
   - Compute gradients with respect to the loss.
   - Update model parameters using the optimizer.
4. **Repeat:** Iterate over batches of data, performing forward and backward passes, and updating weights.

Step 5: Evaluation and Validation

1. **Validate Model:** Periodically evaluate the model on the validation set to monitor performance and adjust hyperparameters or stop training to prevent overfitting.
2. **Adjust Learning Rate:** Optionally reduce the learning rate based on validation performance (learning rate schedules or decay).

Step 6: Testing and Deployment

1. **Test Model:** After training is complete, assess model performance on the unseen test set to estimate real-world performance.
2. **Fine-Tuning:** Optionally fine-tune the model on more specific tasks or datasets.
3. **Deployment:** Integrate the model into the target environment or application.

Step 7: Monitoring and Maintenance

1. **Monitor Performance:** Continuously monitor the model's performance in production, ensuring it handles new data effectively.
2. **Update Model:** Re-train or fine-tune the model periodically with new data or to correct drifts in performance.

These steps outline a generic process for training a Transformer model, primarily focused on language-related tasks. Variations might exist depending on specific applications, such as text summarization, question answering, or even non-language tasks.

2.2 Inference

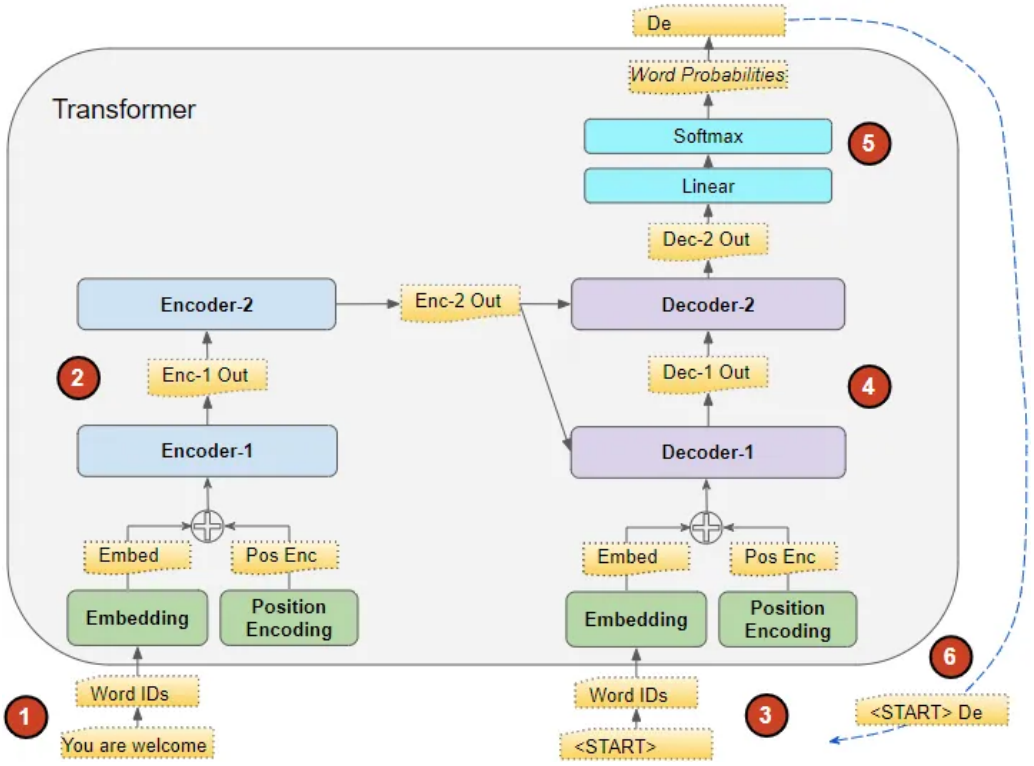


Figure 2.5: Inference Flow [12]

Step 1: Input Data Preparation

- **Tokenization:** Convert the input text into tokens using the same tokenization method as used during training. This usually involves breaking the text into subwords or characters.To exemplify for the sentence "Hello, world!", tokenization might result in ["Hello", ",", "world", "!"].
- **Encoding:** Map these tokens to numerical IDs based on the vocabulary used during training.Suppose the the result of the tokenization part : ["Hello", ",", "world", "!"] might map to [1543, 12, 672, 23].
- **Adding Special Tokens:** Add necessary special tokens for sequence processing, like [SOS] (start of sequence) and [EOS] (end of sequence).For instance adding [SOS] and [EOS] tokens results in [101, 1543, 12, 672, 23, 102].
- **Padding:** If necessary, pad the sequence to match the model's expected input length.If the model expects input lengths of 10, the sequence might be padded to [101, 1543, 12, 672, 23, 102, 0, 0, 0, 0].

Step 2: Loading the Trained Model

- **Load Model Parameters:** Load the trained Transformer model, including all its weights and configuration settings.
- **Configuration Check:** Ensure that the inference configuration matches the training settings (e.g., number of layers, hidden units, attention heads).

Step 3: Model Forward Pass

- **Input Embedding:** Convert the encoded inputs into embeddings and add positional encodings to include information about the order of tokens in the sequence.
- Encoder Processing:

  The embeddings pass through the encoder stack.

  The encoder processes the input sequence through multiple layers of self-attention and feed-forward networks, generating a set of encoder outputs.

- **Decoder Processing:**
  - For tasks involving an output sequence (like translation), start with the initialized token (e.g., [SOS]).
  - For each output step :

    Apply embeddings and add positional encodings.

    Pass the current output tokens (initially just the start token) and the encoder outputs to the decoder.

    The decoder uses both self-attention over its inputs and encoder-decoder attention to predict the next token.
- **Output Generation:**
  - The final layer of the decoder outputs a vector of logits or scores for each token in the vocabulary.
  - Apply a softmax function to these scores to get a probability distribution over all possible next tokens.

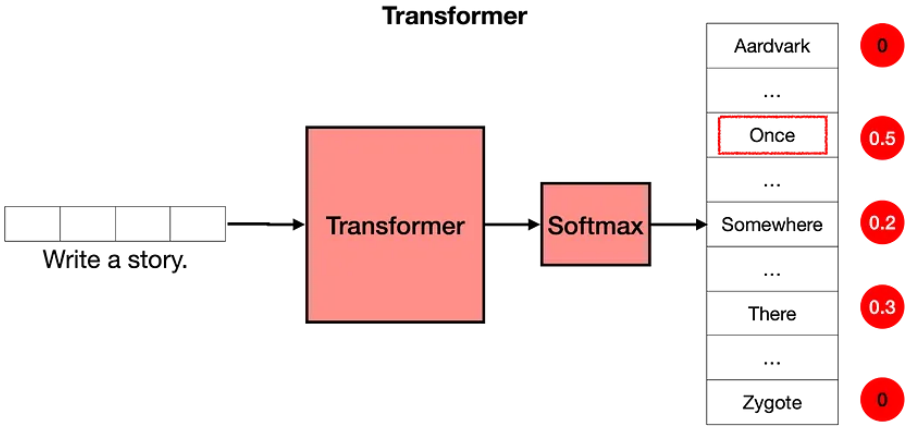


Figure 2.6: Softmax

Step 4: Token Prediction and Sequence Generation

- **Token Selection:** Select the token with the highest probability or sample from the probability distribution, depending on the application (deterministic vs. probabilistic approaches).
- **Repeat:** Use the updated sequence of output tokens as input to the decoder for the next step. Continue until an end-of-sequence token is generated or a maximum length is reached.

Step 5: Post-processing

- **Detokenization:** Convert the sequence of output tokens back into human-readable text.
- **Cleanup:** Remove any special formatting or tokens used for processing but not part of the final output.

Step 6: Output the Result

Deliver the Final Text: The resulting output text is now ready for presentation or further processing, depending on the application.

Chapter 3

3.1 BLEU Score

BLEU score measures the quality of predicted text, referred to as the candidate, compared to a set of references. There can be more than one correct/reference for one candidate in Sequence to Sequence tasks. Hence, it is important that the references are chosen carefully and all the possible references are included. BLEU score is a precision based measure and it ranges from 0 to 1. The closer the value is to 1, the better the prediction. It is not possible to achieve a value of 1 and usually a value higher than 0.3 is considered a good score.

Precision

If we recall the definition of precision from our previous post, it is True Positives/(True Positives + False Positives). In this context, the True Positives are all the matching n-grams[1] between the candidate and reference. The False Positives here can be seen as the n-grams that appear in the candidate but are not present in the reference. This means that the precision can be obtained by dividing the number of matching n-grams by the total number of n-grams in the candidate.
Precision works well for many cases but consider the example set above. What is the precision in this case if we consider unigrams(separate words)? It is $4/4 = 1$, which is the highest value precision can take. However, we can see that this is not a good translation. To address this issue, a measure called Modified Precision is used for calculating BLEU.

Modified Precision(MP)

For calculating Modified Precision, the count of an n-gram is clipped based on the maximum number of times it appears in the references. It is denoted by the Max-Ref-Count in the formula shown below.

$$\text{Count}_{\text{clip}} = \min(\text{Count}, \text{Max\_Ref\_Count})$$

In our example in the previous section, Transformers appears only once in the references and 4 times in the candidate. min(4, 1) is considered and the modified precision now is $1/4 = 0.25$. This value is lower and hence gives us a better estimate of the quality of the prediction.
To generalize, the formula for MP is given below. For all the candidates in the corpus, the number of matched n-grams with the references are counted and summed up. This is divided by the sum of counts of all n-grams in the references.

$$p_n = \frac{\sum_{C \in \{\text{Candidates}\}} \sum_{\text{n-gram} \in C} \text{Count}_{\text{clip}}(\text{n-gram})}{\sum_{C' \in \{\text{Candidates}\}} \sum_{\text{n-gram}' \in C'} \text{Count}(\text{n-gram}')}$$

Sometimes, for longer sentences the candidates might be very small and missing important information relative to the reference. Consider the example below:
Reference: "Transformers make everything quick and efficient through parallel computation of self-attention heads"
Candidate: "Transformers make everything quick and efficient"
Here, we are missing information because of the short prediction. But, the MP is high(1.0) as the additional words that are in the reference but not in the candidate are not being considered.

Enter, Brevity Penalty(BP). This term is used to penalize the predictions that are too short when compared to the references. This is 1 if the candidate length is greater than the reference length. If the candidate length

[1] N-gram can be defined as the contiguous sequence of n items from a given sample of text or speech. The items can be letters, words, or base pairs according to the application. The N-grams typically are collected from a text or speech corpus (A long text dataset).

is less than the reference implies r/c ¿ 1, (1 — (r/c)) is lower and there is an exponential decay.

The geometric mean of all the modified precisions up until N is calculated and multiplied with BP to get the final BLEU score. Here N is the n-gram order that is to be used for calculation. In general, it is 4 i.e., uni, bi, tri and tetra grams are all considered for calculation. They are weighted by the weights denoted by $w_1, w_2, \ldots which add up to 1 based on N. For N = 4, they are w_1 = w_2 = w_3 = w_4 = 1/4$.

$$\text{BP} = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases}$$

Then,

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right).$$

For the example sentence above, we can see that the BP is calculated as exp(1 — (reference-length/translation-length)) = exp(1 — (6/12)) = 0.37. This value multiplied by the MP(1, since we are using only unigrams) gives much lower BLEU score of 0.37 instead of 1, that accounts for the missing n-grams.[14]

Let's take a look to another example :
Suppose the reference translation is "the cat is on the mat", and the candidate translation is "the cat on mat":

- Unigram precision $p_1 = \frac{4}{5}$ (since "the", "cat", "on", "mat" are correct, total 5 words in candidate)
- Bigram precision $p_2 = \frac{2}{4}$ ("the cat", "on mat" are correct, total 4 bigrams in candidate)

Assume equal weights $w_1 = w_2 = 0.5$ and $c = 5, r = 6$ (shorter candidate):

$$\text{BP} = e^{(1-6/5)} = 0.8187$$

$$\text{BLEU} = 0.8187 \cdot \exp\left(0.5 \cdot \log(4/5) + 0.5 \cdot \log(2/4)\right) \approx 0.5730$$

3.2 ROUGE Score

ROUGE is an evaluation metric used to assess the quality of NLP tasks such as text summarization and machine translation. Unlike BLEU, the ROUGE uses both recall and precision to compare model generated summaries known as candidates against a set of human generated summaries known as references. It measures how many of the n-grams in the references are in the predicted candidate.
First, let us calculate recall = True Positives/(True Positives + False Negatives). Here, True Positives are the matching n-grams between References and Candidates. The False Negatives can be thought of as the n-grams that are in the actual sentences(references) but not in the candidate. The False Positives are then the n-grams that are present in the Candidate but not in the References. Check out this blog post Recall, Precision, F1-Score if you need a refresher on concepts such as True Positives, False Positives, Recall and Precision etc.,
Hence, the recall and precision can be obtained as shown below:

$$\text{Recall} = \frac{\text{Number of matching n-grams}}{\text{Number of n-grams in the Reference}}$$

$$\text{Precision} = \frac{\text{Number of matching n-grams}}{\text{Number of n-grams in the Candidate}}$$

Recall and Precision can be complementary sometimes and hence it is necessary to combine both of them to achieve a balance. For this reason, the ROUGE is calculated as F1-score in Python. The value ranges from 0 to 1 and the closer the value is to 1, the better the quality of the prediction.[15][16]
Consider the provided example below : **Total Unigrams in Reference:** This is simply the count of all individual words in the reference text. For the sentence "the cat sat on the mat", there are 6 words.

Unigram Matches: This counts how many times the unigrams in the candidate text appear in the reference text. In the example:

- "on" appears once in both the candidate and the reference.
- "the" appears twice in both texts.
- "mat" appears once in both texts.
- "sat" appears once in both texts.
- "cat" appears once in both texts.

Hence, there are 6 matching unigrams ("the", "cat", "sat", "on", "the", "mat").

The ROUGE-1 score can be calculated as the ratio of the number of unigrams in the candidate that match the reference to the total number of unigrams in the reference:

$$\text{ROUGE-1} = \frac{\text{Number of Matching Unigrams}}{\text{Total Unigrams in Reference}} = \frac{6}{6} = 1.0$$

The ROUGE score of 1.0 indicates perfect unigram overlap between the candidate and the reference texts.

This suggests that all the words in the candidate text appear in the reference text, indicating a high level of similarity, at least at the word level.

3.3 Perplexity

A number from 1 to infinity that represents how "surprised" a language model generally is to see the actual continuations of fragments of text. The lower the perplexity, the better the language model can predict the actual continuations of those text fragments in the evaluation data. Perplexity is an important intrinsic evaluation for language models. [17] [18] Perplexity is defined as the exponentiated average negative log-likelihood of a sequence. If we have a tokenized sequence $X = (x_0, x_1, \ldots, x_t)$, then the perplexity of X is given by:

$$\text{PPL}(X) = \exp\left(-\frac{1}{t}\sum_{i=0}^{t}\log p_\theta(x_i \mid x_{<i})\right)$$

where $\log p_\theta(x_i \mid x_{<i})$ is the log-likelihood of the i-th token conditioned on the preceding tokens $x_{<i}$ according to our model. Intuitively, it can be thought of as an evaluation of the model's ability to predict uniformly among the set of specified tokens in a corpus. Importantly, this means that the tokenization procedure has a direct impact on a model's perplexity which should always be taken into consideration when comparing different models.

This is also equivalent to the exponentiation of the cross-entropy between the data and model predictions. For more detailed information, one may refer to advanced topics on model evaluation.[19]

Suppose a model gives probabilities of 0.2, 0.4, 0.1 for three words in a sequence. The average log probability is calculated as follows:

$$\text{Average log probability} = \frac{\log(0.2) + \log(0.4) + \log(0.1)}{3}$$

Using the logarithm values:

$$\frac{-1.609 - 0.916 - 2.303}{3} \approx -1.6093$$

The average log probability is approximately -1.6093. This represents the geometric mean of the probabilities, where a higher (less negative) value indicates better model performance.

The perplexity of the model is given by:

$$\text{Perplexity} = \exp(-(-1.6093)) \approx 2.5$$

This indicates that on average, the model is as confused as if it were to choose uniformly and independently among 2.5 choices. The lower the perplexity, the better the model is at predicting the test data. A perplexity of 2.5 suggests a fairly good model performance, especially in contexts where many possible continuations are plausible.

3.4

The application of the CIDEr metric to text evaluation in Transformer models is an idea adapted from image description evaluation. Although originally designed for assessing outputs of image description models, CIDEr can be adapted for Transformers engaged in tasks such as summarization, machine translation, or other Natural Language Processing (NLP) tasks.
Setting up the Metric for Text:
Defining TF-IDF for Text:
While in image description, keywords are weighted based on their frequency across different descriptions, a similar approach can be employed for text to determine the relative importance of words within texts. For this purpose, a corpus of reference texts can be used as a database for calculating TF-IDF. **Calculating Cosine Similarity:**
To compare the model's output with reference texts, cosine similarity can be calculated between the TF-IDF vectors of the model's output and those of the reference texts. This similarity indicates how well the content produced by the model matches with the reference texts.

Implementing CIDEr in NLP Tasks

This adaptation allows Transformer models to be evaluated on how well their generated text aligns with human-like reference texts. This can be particularly useful in tasks where the quality of text generation is crucial, such as in generating reports, summaries, or translating between languages. Using TF-IDF helps in quantifying the significance of each word in the generated content, ensuring that the evaluation metric captures the essence of the text's relevance and informativeness compared to a standard set by reference texts.

Practical Application

In practice, implementing CIDEr in Transformers involves several steps:

- Preprocessing Text: Both the generated and reference texts need to be preprocessed and normalized to ensure consistency in evaluation. This includes tokenization, stemming, and removal of stopwords.
- TF-IDF Calculation: Implement TF-IDF vectorization for both the generated and reference texts. This step transforms the texts into a format where each word's importance is statistically measured against its occurrence in the reference corpus.
- Cosine Similarity Measure: With TF-IDF vectors prepared, the next step is to compute the cosine similarity between each generated text and the reference texts. The average of these similarities can form the basis of the CIDEr score, which quantifies the overall alignment of the generated text with the expected output.

Let's check an example. Consider the assumption below:

Model-generated summary: "The quick brown fox jumps over the lazy dog."
Reference summaries:

- "A quick brown fox jumps over a lazy dog."
- "Quick brown fox jumping over lazy dogs."
- "Brown fox jumps over the lazy dog quickly."

1. Calculate TF-IDF for Each Word in Each Summary:
   For simplicity, assume that words appear with equal frequency across the reference texts and each word has equal TF-IDF weight.
2. Count Common Words:
   Common words between the model-generated summary and each reference text are counted, such as "quick", "brown", "fox", "jumps", "over", "lazy", "dog".
3. Calculate Cosine Similarity:
   Cosine similarity between the TF-IDF vectors of the model-generated summary and each reference summary is calculated.

[5]

3.5

METEOR (Metric for Evaluation of Translation with Explicit ORdering): Considers synonyms and stemming when comparing machine translation to human translation. Perplexity: Represents how well a probability model predicts a sample and is especially crucial for evaluation metrics for language models.

How is the METEOR metric computed?

Here is the steps to compute the METEOR metric.

Computing an Alighment

An alignment between the generated text and the reference can be done by matching word for word, or by using tools for similarity such as word embeddings, dictionary and so on.

/

A chunk in an alignment is an adjacent set of words that map to adjacent set of words in the reference. The above alignment has three chunks.
While there are multiple alignments possible we want to pick that alignment where the number of chunks is the lowest.
For instance, take two sentences, "the cat likes the bone" (generated text) and "the cat loves the bone" (Reference). The word "the" in the gnerated text can be mapped to either of the two "the"s in the reference, but mapping it to the first makes more sense since it leads to just one chunk since all words in generated text map to reference consecutively: The – The, cat – cat, likes – loves, the – the, bone -bone.

Computing the F-score

METEOR takes into account both the precision and recall while evaluating a match. The following formulas are used to compute the evaluation metrics for translation quality:

Precision:

$$P = \frac{m}{w_t}$$

Recall:

$$R = \frac{m}{w_r}$$

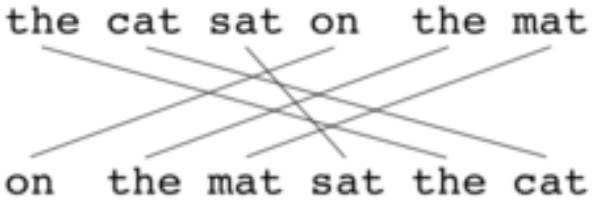
F-measure:

$$F_{\text{mean}} = \frac{10 \cdot P \cdot R}{R + 9 \cdot P}$$

where:

- m: Number of unigrams in the candidate translation also found in reference
- w_t: Number of unigrams in candidate translation
- w_r: Number of unigrams in reference translation

Take a look at the following example: Since all words of candidates are in the reference and all words of reference are in the candidate, we get a perfect score of 1 with the above definition based on the F score.



Computing Chunk Penalty

A chunk is a set of consecutive words. Typically we note that chunks of words in the source map to chunks of words in the target. The chunk penalty gives a penalty based on the number of chunks in candidate that map to chunks in target or the reference.

In an ideal case, if the candidate and the reference are the same, all words in candidate map to all words in reference consecutively and we just have one chunk. But in reality, we have more chunks in a less than ideal match.

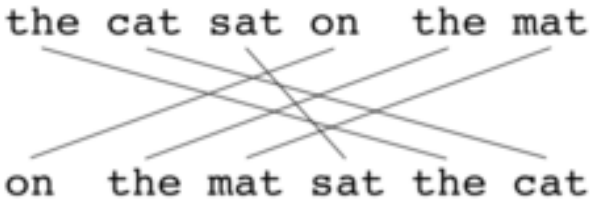
The chunk penalty is computed as follows:

$$p = 0.5 \left(\frac{c}{u_m} \right)^3$$

where:

- c: Number of chunks in candidate
- u_m: Unigrams in candidate

Once again consider an example with the following alighment:



There are three distinct chunks in the candidate ("the cat", "sat", "on the mat") that map to different chunks in the reference. While there are six unigrams in the candidate. Hence, C is 3 and $\mathrm{U}_m is 6 to compute the formula above$.

Computing the overall METEOR score

$$M = F_{\text{mean}}(1 - p)$$

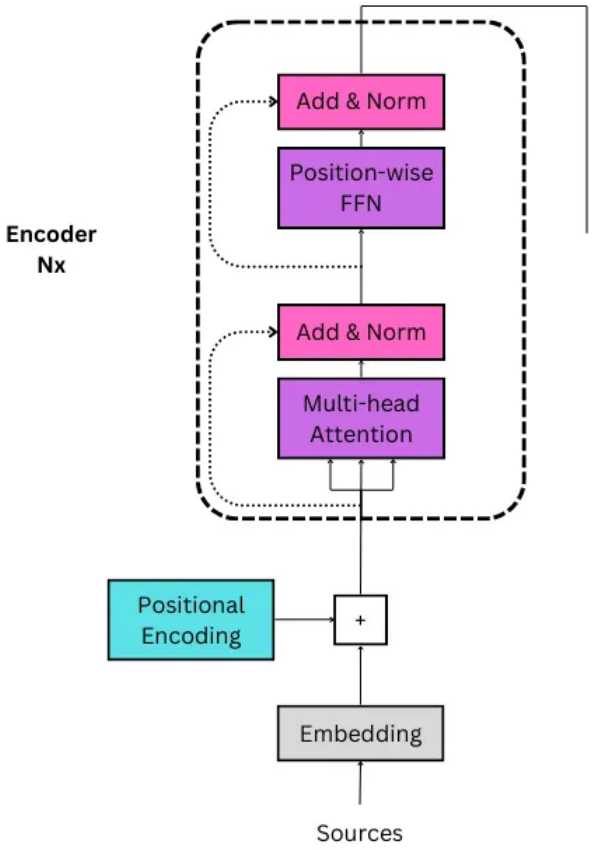
where:

- F_{mean} represents the F-measure or the harmonic mean of precision and recall.
- p is a probability factor derived from the model's evaluation.

[20]

Chapter 4

Core Components



4.1 Multi-Head Self-Attention

[21] Attention is used in the Transformer in three places:

- Self-attention in the Encoder — the input sequence pays attention to itself
- Self-attention in the Decoder — the target sequence pays attention to itself
- Encoder-Decoder-attention in the Decoder — the target sequence pays attention to the input sequence

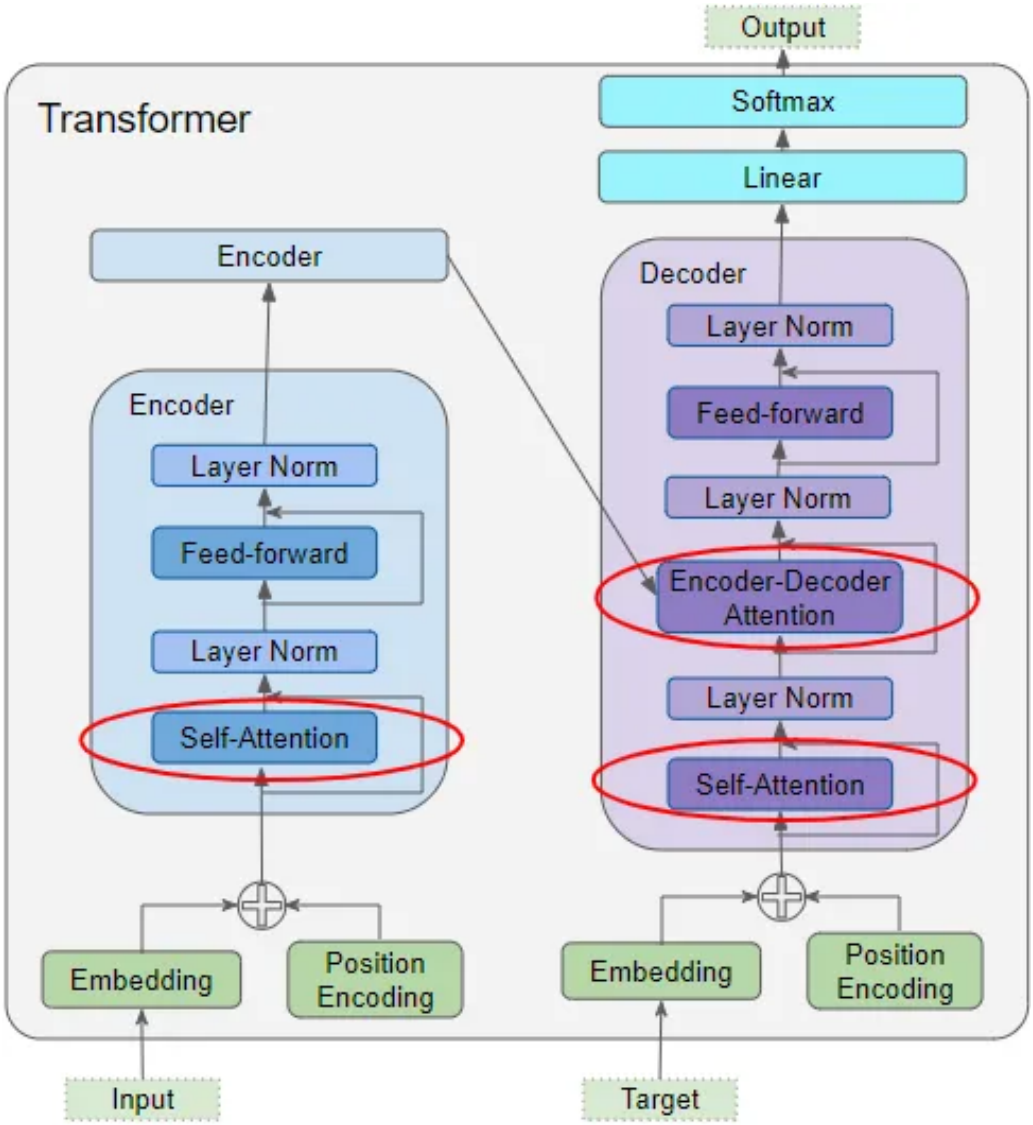


Figure 4.1: Caption

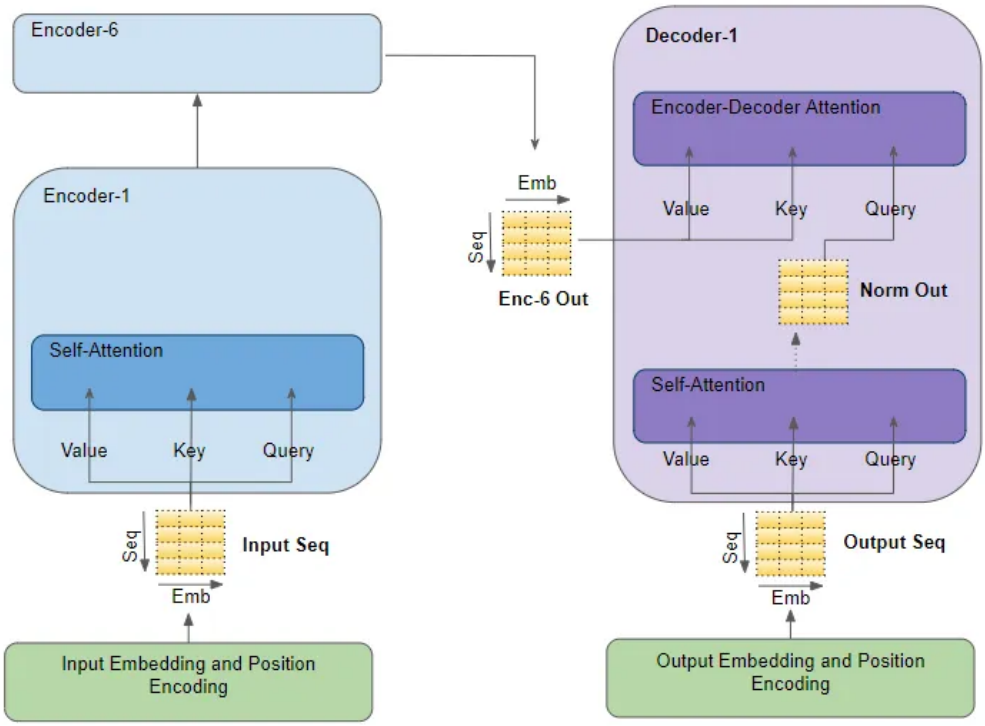
The Attention layer takes its input in the form of three parameters, known as the Query, Key, and Value. All three parameters are similar in structure, with each word in the sequence represented by a vector.

4.1.1 Encoder Self-Attention

The input sequence is fed to the Input Embedding and Position Encoding, which produces an encoded representation for each word in the input sequence that captures the meaning and position of each word. This is fed to all three parameters, Query, Key, and Value in the Self-Attention in the first Encoder which then also produces an encoded representation for each word in the input sequence, that now incorporates the attention scores for each word as well. As this passes through all the Encoders in the stack, each Self-Attention module also adds its own attention scores into each word's representation.

4.1.2 Decoder Self-Attention

Coming to the Decoder stack, the target sequence is fed to the Output Embedding and Position Encoding, which produces an encoded representation for each word in the target sequence that captures the meaning and position of each word. This is fed to all three parameters, Query, Key, and Value in the Self-Attention in the first Decoder which then also produces an encoded representation for each word in the target sequence, which now incorporates the attention scores for each word as well.
After passing through the Layer Norm, this is fed to the Query parameter in the Encoder-Decoder Attention in the first Decoder.



4.1.3 Encoder-Decoder Attention

Along with that, the output of the final Encoder in the stack is passed to the Value and Key parameters in the Encoder-Decoder Attention.
The Encoder-Decoder Attention is therefore getting a representation of both the target sequence (from the Decoder Self-Attention) and a representation of the input sequence (from the Encoder stack). It, therefore, produces a representation with the attention scores for each target sequence word that captures the influence of the attention scores from the input sequence as well.
As this passes through all the Decoders in the stack, each Self-Attention and each Encoder-Decoder Attention also add their own attention scores into each word's representation.

4.1.4 Multiple Attention Heads

In the Transformer, the Attention module repeats its computations multiple times in parallel. Each of these is called an Attention Head. The Attention module splits its Query, Key, and Value parameters N-ways and passes each split independently through a separate Head. All of these similar Attention calculations are then combined together to produce a final Attention score. This is called Multi-head attention and gives the Transformer greater power to encode multiple relationships and nuances for each word.

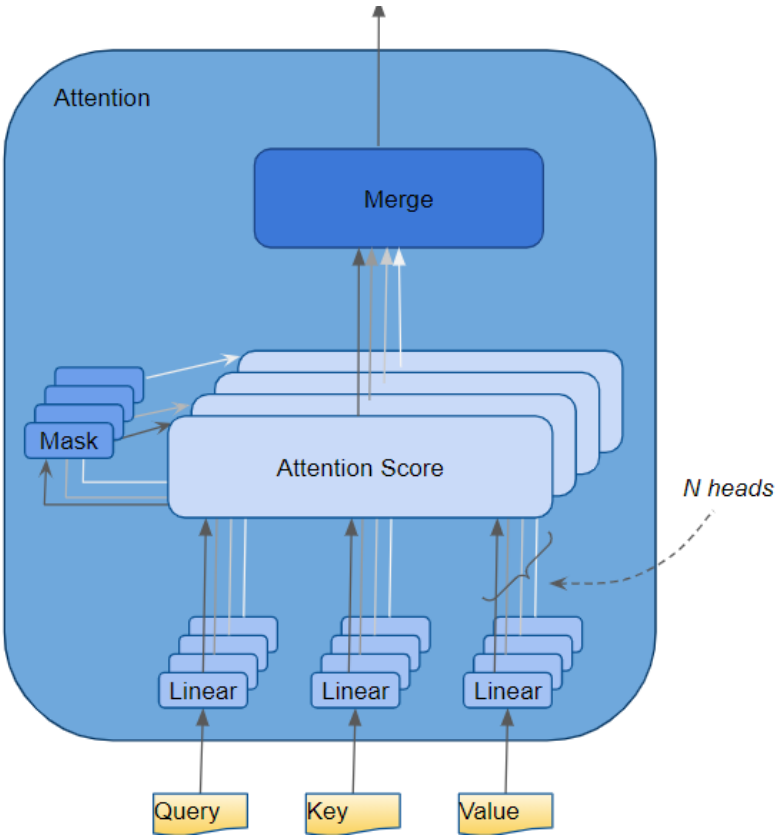


Figure 4.2: Caption

To understand exactly how the data is processed internally, let's walk through the working of the Attention module while we are training the Transformer to solve a translation problem. We'll use one sample of our training data which consists of an input sequence ('You are welcome' in English) and a target sequence ('De nada' in Spanish).

4.1.5 Attention Hyperparameters

There are three hyperparameters that determine the data dimensions:

- Embedding Size — width of the embedding vector (we use a width of 6 in our example). This dimension is carried forward throughout the Transformer model and hence is sometimes referred to by other names like 'model size' etc.
- Query Size (equal to Key and Value size)— the size of the weights used by three Linear layers to produce the Query, Key, and Value matrices respectively (we use a Query size of 3 in our example)
- Number of Attention heads (we use 2 heads in our example)

In addition, we also have the Batch size, giving us one dimension for the number of samples.

4.1.6 Input Layers

The Input Embedding and Position Encoding layers produce a matrix of shape (Number of Samples, Sequence Length, Embedding Size) which is fed to the Query, Key, and Value of the first Encoder in the stack.

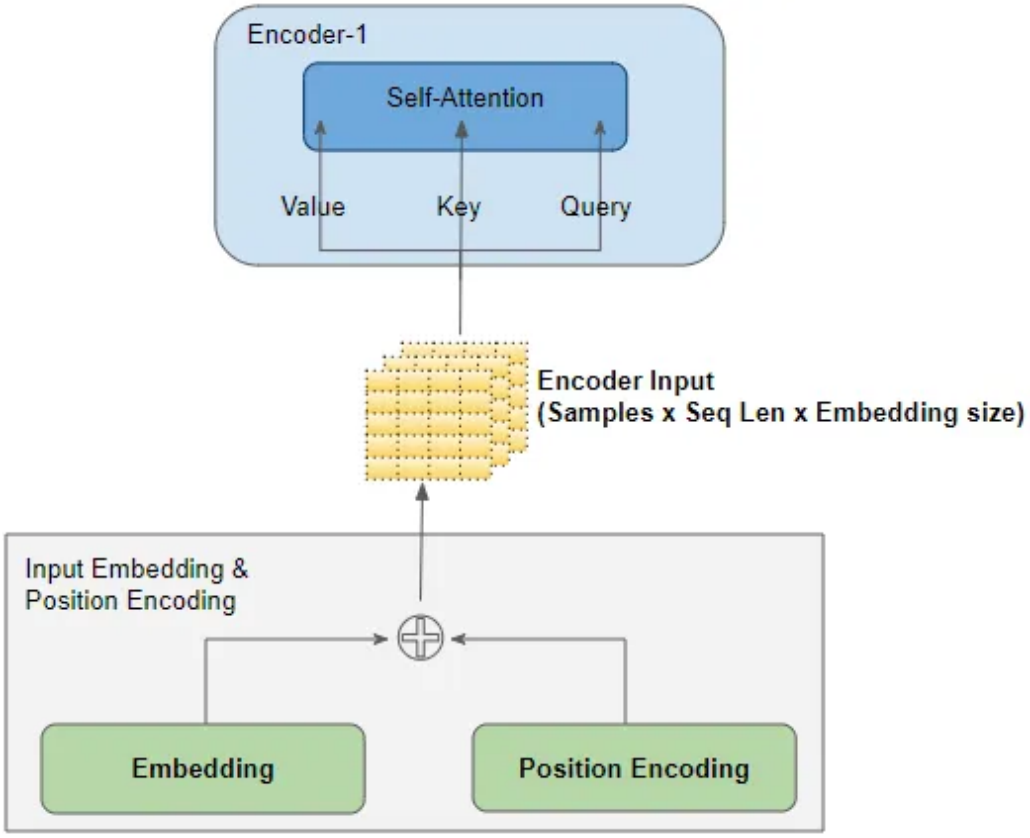


Figure 4.3: InputLayer

To make it simple to visualize, we will drop the Batch dimension in our pictures and focus on the remaining dimensions.

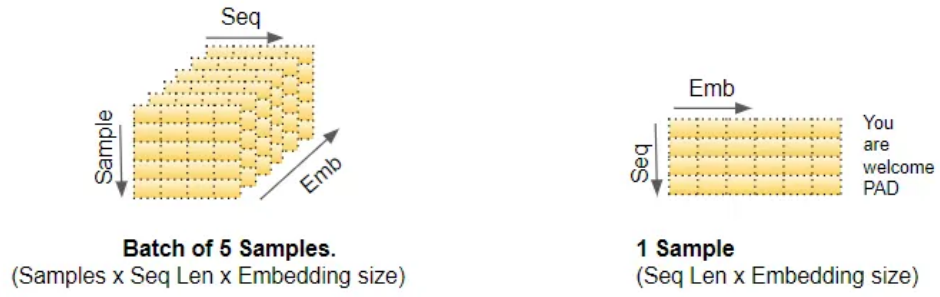
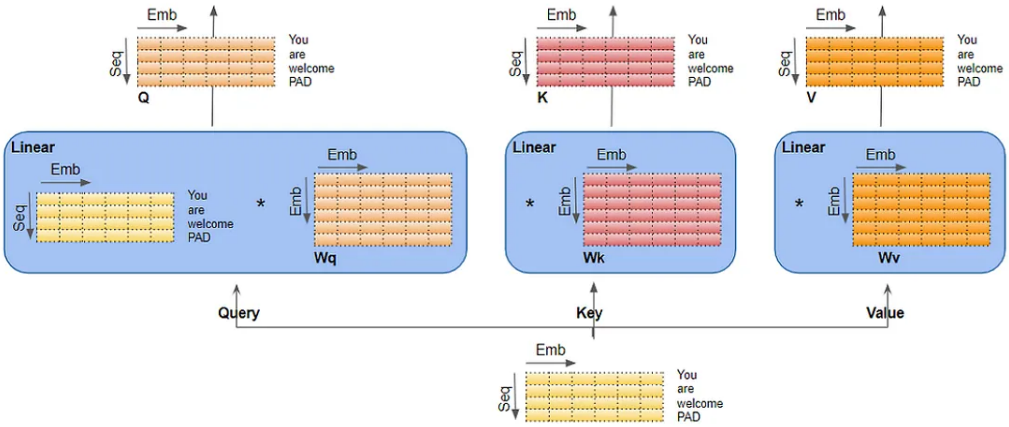


Figure 4.4: Caption

4.1.7 Linear Layers

There are three separate Linear layers for the Query, Key, and Value. Each Linear layer has its own weights. The input is passed through these Linear layers to produce the Q, K, and V matrices.



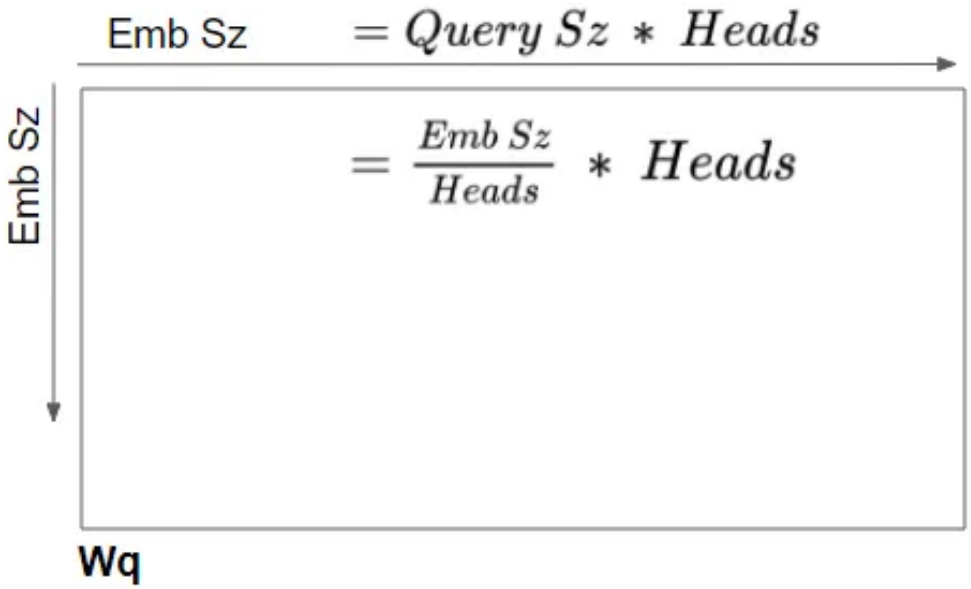
4.1.8 Splitting data across Attention heads

Now the data gets split across the multiple Attention heads so that each can process it independently. However, the important thing to understand is that this is a logical split only. The Query, Key, and Value are not physically split into separate matrices, one for each Attention head. A single data matrix is used for the Query, Key, and Value, respectively, with logically separate sections of the matrix for each Attention head. Similarly, there are not separate Linear layers, one for each Attention head. All the Attention heads share the same Linear layer but simply operate on their 'own' logical section of the data matrix.

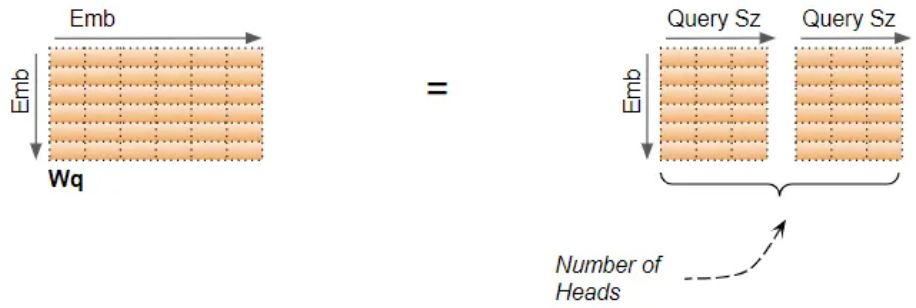
4.1.9 Linear layer weights are logically partitioned per head

This logical split is done by partitioning the input data as well as the Linear layer weights uniformly across the Attention heads. We can achieve this by choosing the Query Size as below:

$$\text{Query Size} = \frac{\text{Embedding Size}}{\text{Number of Heads}}$$



In our example, that is why the Query Size = 6/2 = 3. Even though the layer weight (and input data) is a single matrix we can think of it as 'stacking together' the separate layer weights for each head.



The computations for all Heads can be therefore be achieved via a single matrix operation rather than requiring N separate operations. This makes the computations more efficient and keeps the model simple because fewer Linear layers are required, while still achieving the power of the independent Attention heads.

4.1.10 Reshaping the Q, K, and V matrices

The Q, K, and V matrices output by the Linear layers are reshaped to include an explicit Head dimension. Now each 'slice' corresponds to a matrix per head.
This matrix is reshaped again by swapping the Head and Sequence dimensions. Although the Batch dimension is not drawn, the dimensions of Q are now (Batch, Head, Sequence, Query size).

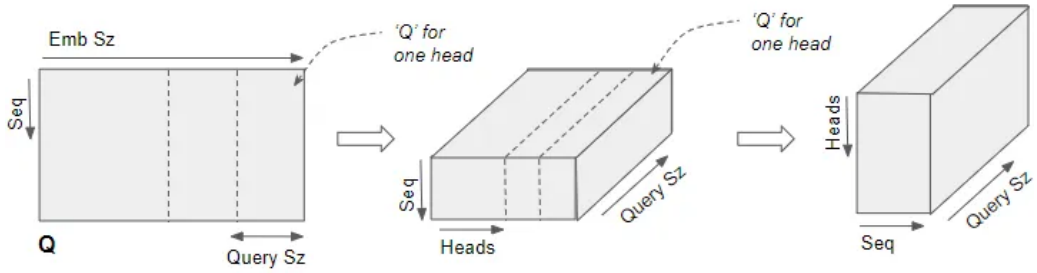
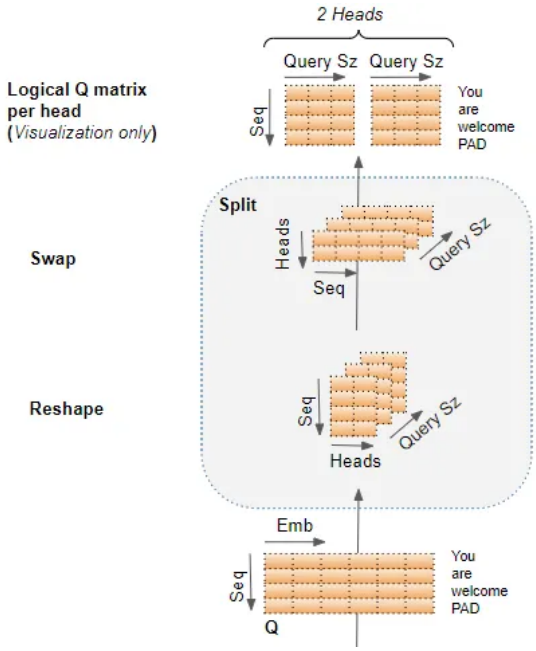


Figure 4.5: The Q matrix is reshaped to include a Head dimension and then reshaped again by swapping the Head and Sequencd dimensions.

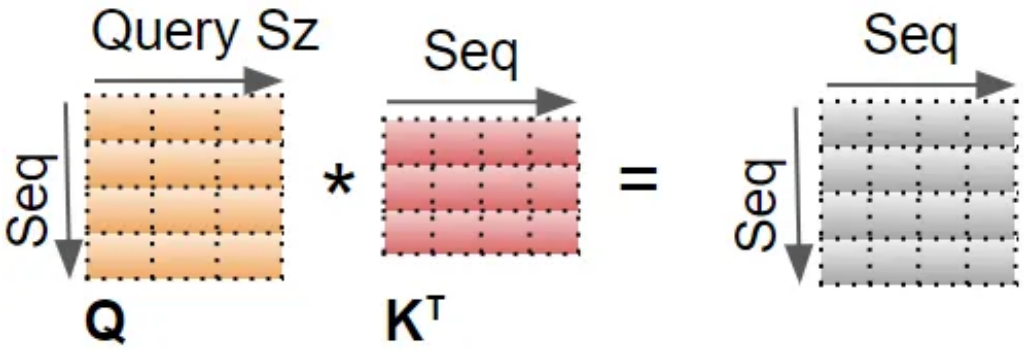
In the picture below, we can see the complete process of splitting our example Q matrix, after coming out of the Linear layer.
The final stage is for visualization only — although the Q matrix is a single matrix, we can think of it as a logically separate Q matrix per head.



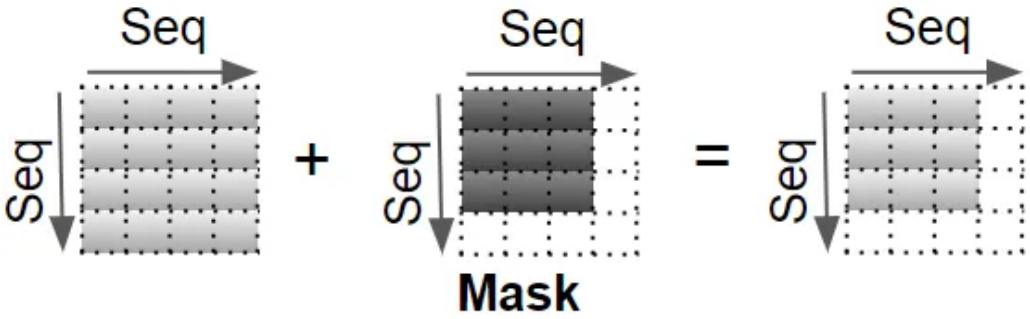
We are ready to compute the Attention Score.

4.1.11 Compute the Attention Score for each head

We now have the 3 matrices, Q, K, and V, split across the heads. These are used to compute the Attention Score.
We will show the computations for a single head using just the last two dimensions (Sequence and Query size) and skip the first two dimensions (Batch and Head). Essentially, we can imagine that the computations we're looking at are getting 'repeated' for each head and for each sample in the batch (although, obviously, they are happening as a single matrix operation, and not as a loop).
The first step is to do a matrix multiplication between Q and K.



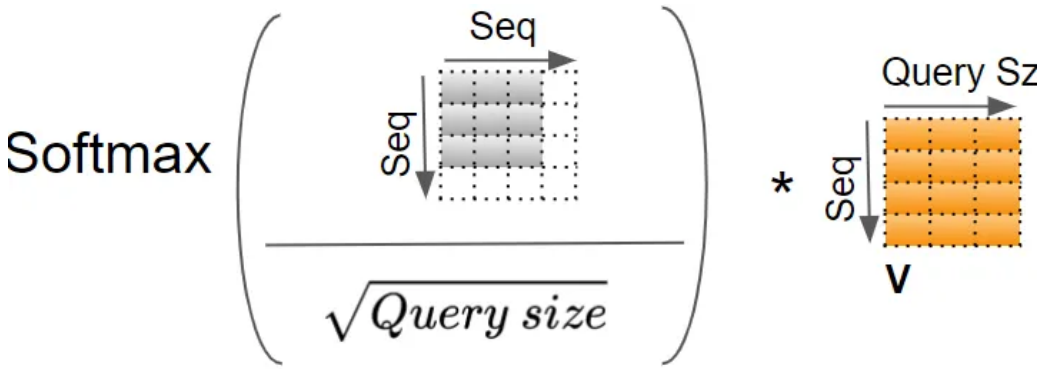
A Mask value is now added to the result. In the Encoder Self-attention, the mask is used to mask out the Padding values so that they don't participate in the Attention Score.
Different masks are applied in the Decoder Self-attention and in the Decoder Encoder-Attention which we'll come to a little later in the flow.



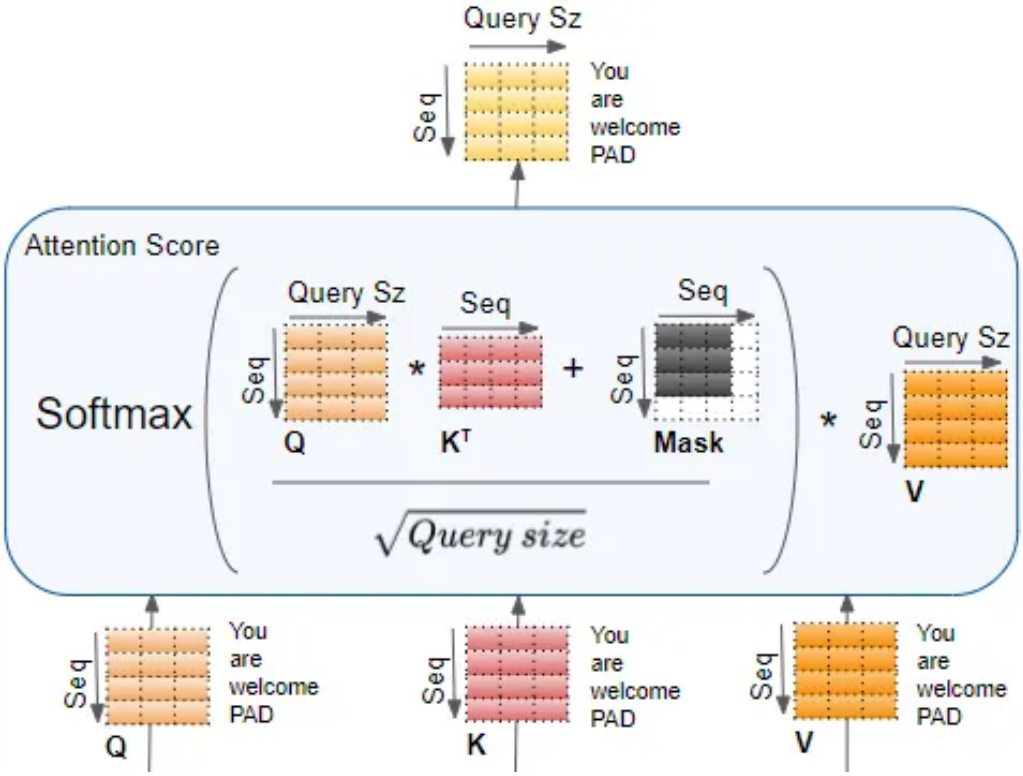
The result is now scaled by dividing by the square root of the Query size, and then a Softmax is applied to it.

$$\text{Softmax}\left(\frac{\text{Seq} \times \text{Seq}}{\sqrt{Query\ size}}\right)$$

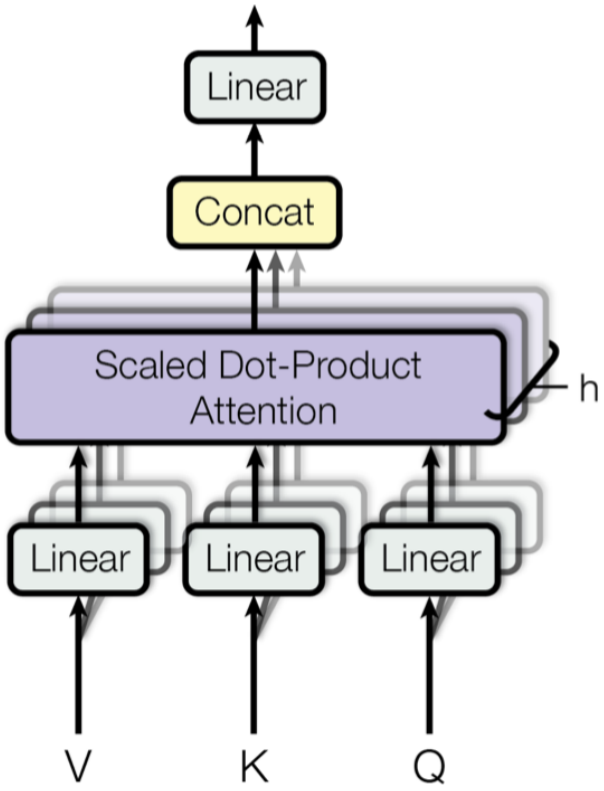
Another matrix multiplication is performed between the output of the Softmax and the V matrix.



The complete Attention Score calculation in the Encoder Self-attention is as below:



4.2 Multi-Head Attention Calculation



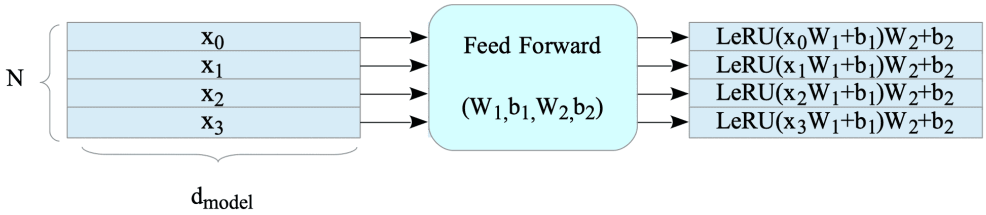
Multi-head Attention is a module for attention mechanisms which runs through an attention mechanism several times in parallel. The independent attention outputs are then concatenated and linearly transformed into the expected dimension. Intuitively, multiple attention heads allows for attending to parts of the sequence differently (e.g. longer-term dependencies versus shorter-term dependencies)[22].

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = [\text{head}_1, \ldots, \text{head}_h]\mathbf{W}_0$$

$$\text{where head}_i = \text{Attention}\left(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V\right)$$

Above W are all learnable parameter matrices.

4.3 Position-wise Feed-Forward Network (FFN)



[23]

The FFN consists of two fully connected layers. The number of dimensions in the hidden layer d_{ff} is generally set to around four times that of the token embedding d_{model}. Therefore, it is sometimes referred to as the

expand-and-contract network.

An activation function is applied at the hidden layer, typically using the ReLU (Rectified Linear Unit) activation function:

$$\text{ReLU}(x) = \max(0, x)$$

Thus, the FFN function can be expressed as:

$$\text{FFN}(x, W_1, W_2, b_1, b_2) = \max(0, xW_1 + b_1)W_2 + b_2$$

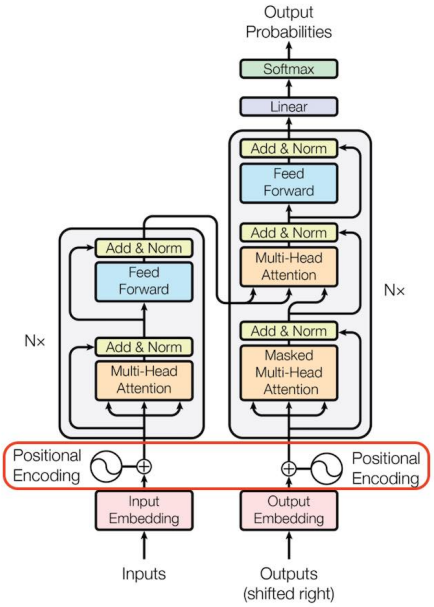
where W_1, W_2, b_1, and b_2 are all learnable parameters.

In some cases, the GELU (Gaussian Error Linear Unit) activation function is also utilized instead of ReLU:

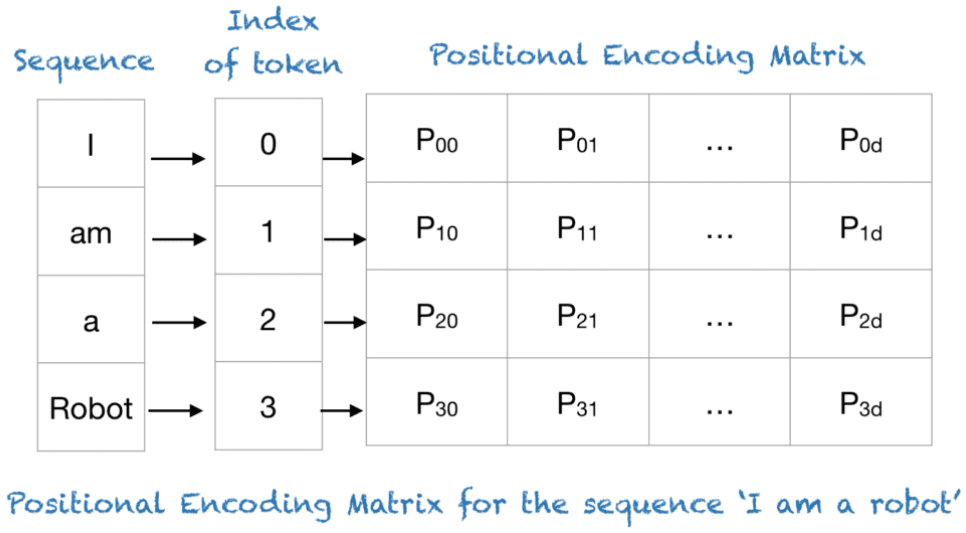
$$\text{GELU}(x) = x\Phi(x)$$

where $\Phi(x) = P(X \leq x), X \sim N(0, 1)$. [24]

4.4 Positional Encoding



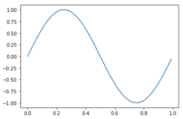
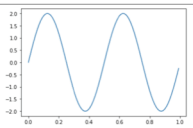
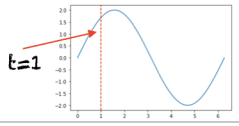
As we disscussed , Positional encoding describes the location or position of an entity in a sequence so that each position is assigned a unique representation. There are many reasons why a single number, such as the index value, is not used to represent an item's position in transformer models. For long sequences, the indices can grow large in magnitude. If you normalize the index value to lie between 0 and 1, it can create problems for variable length sequences as they would be normalized differently.
Transformers use a smart positional encoding scheme, where each position/index is mapped to a vector. Hence, the output of the positional encoding layer is a matrix, where each row of the matrix represents an encoded object of the sequence summed with its positional information. An example of the matrix that encodes only the positional information is shown in the figure below.



Positional Encoding Matrix for the sequence 'I am a robot'

A Quick Run-Through of the Trigonometric Sine Function

This is a quick recap of sine functions; you can work equivalently with cosine functions. The function's range is [-1,+1]. The frequency of this waveform is the number of cycles completed in one second. The wavelength is the distance over which the waveform repeats itself. The wavelength and frequency for different waveforms are shown below:

Equation	Graph	Frequency	Wavelength
$\sin(2\pi t)$		1	1
$\sin(2 * 2\pi t)$		2	1/2
$\sin(t)$		$1/2\pi$	2π
$\sin(ct)$	Depends on c	$c/2\pi$	$2\pi/c$

4.4.1 Positional Encoding Layer in Transformers

Let's dive straight into this. Suppose you have an input sequence of length n and require the position of the i-th object within this sequence. The positional encoding is given by sine and cosine functions of varying frequencies:

$$PE(i,d) = \begin{cases} \sin\left(\frac{i}{10000^{\frac{2d}{d_{\text{model}}}}}\right) & \text{if } d \text{ is even} \\ \cos\left(\frac{i}{10000^{\frac{2d}{d_{\text{model}}}}}\right) & \text{if } d \text{ is odd} \end{cases}$$

Here:

- i: Position of an object in the input sequence, where $0 \leq i < n$.
- d: Dimension of the output embedding space.

- $PE(i)$: Position function for mapping a position i in the input sequence to the index d of the positional matrix.
- scale = 10000: User-defined scalar, set to 10,000 by the authors of *Attention Is All You Need*[4].
- d: Used for mapping to column indices d, with a single value of d mapping to both sine and cosine functions.

In the above expression, you can see that even positions correspond to a sine function and odd positions correspond to cosine functions. [25] **Example**

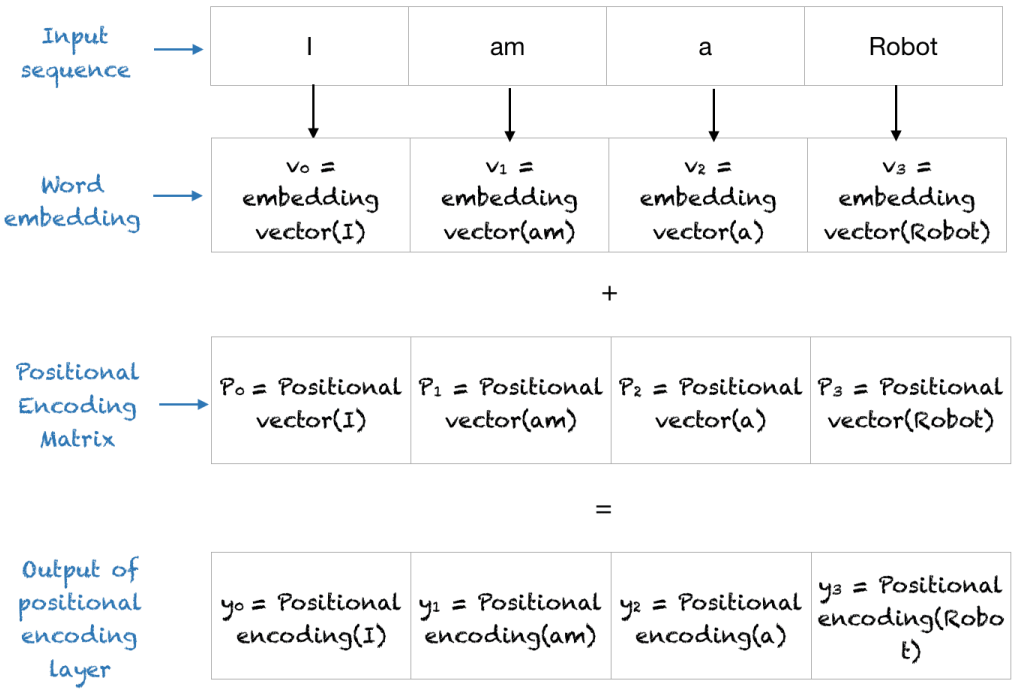
To understand the above expression, let's take an example of the phrase "I am a robot," with n=100 and d=4. The following table shows the positional encoding matrix for this phrase. In fact, the positional encoding matrix would be the same for any four-letter phrase with n=100 and d=4.

Sequence	Index of token, k	Positional Encoding Matrix with d=4, n=100			
		i=0	i=0	i=1	i=1
I	0	P_{00}=sin(0) = 0	P_{01}=cos(0) = 1	P_{02}=sin(0) = 0	P_{03}=cos(0) = 1
am	1	P_{10}=sin(1/1) = 0.84	P_{11}=cos(1/1) = 0.54	P_{12}=sin(1/10) = 0.10	P_{13}=cos(1/10) = 1.0
a	2	P_{20}=sin(2/1) = 0.91	P_{21}=cos(2/1) = -0.42	P_{22}=sin(2/10) = 0.20	P_{23}=cos(2/10) = 0.98
Robot	3	P_{30}=sin(3/1) = 0.14	P_{31}=cos(3/1) = -0.99	P_{32}=sin(3/10) = 0.30	P_{33}=cos(3/10) = 0.96

Positional Encoding Matrix for the sequence 'I am a robot'

What Is the Final Output of the Positional Encoding Layer?

The positional encoding layer sums the positional vector with the word encoding and outputs this matrix for the subsequent layers. The entire process is shown below.



Chapter 5

History

The Transformer architecture, since its inception in 2017, has spurred a multitude of variations and innovations across various aspects of neural network design and application. Below is an overview of prominent Transformer-based models, organized chronologically based on their time of release. This list highlights key developments that have had significant impacts on the field of machine lea
rning, particularly in natural language processing (NLP) and beyond.

2017

Transformer (Vaswani et al., "Attention is All You Need")

Introduced the original Transformer model, which uses self-attention mechanisms for both the encoder and the decoder, effectively revolutionizing sequence modeling tasks.

2018

BERT (Bidirectional Encoder Representations from Transformers) (Devlin et al.)

Developed by Google, BERT introduced the concept of bidirectional training of Transformer models to improve context understanding for tasks like question answering and sentiment analysis.

2019

GPT-2 (Generative Pre-trained Transformer 2) (Radford et al., OpenAI)

An extension of GPT that uses a larger and more complex model with a more extensive pre-training process, enhancing its ability to generate coherent and diverse text.

RoBERTa (Robustly Optimized BERT Approach) (Liu et al., Facebook)

A variant of BERT optimized through more robust training methods, including training on a larger dataset and longer, which improved its performance across multiple benchmarks.

DistilBERT (Sanh et al., Hugging Face)

A distilled version of BERT that aims to retain most of the performance while significantly reducing the size of the model, making it more efficient.

2020

T5 (Text-to-Text Transfer Transformer) (Raffel et al., Google)

A versatile model that reframes all text-based language tasks into a unified text-to-text format, improving consistency and flexibility in handling various tasks.

GPT-3 (Generative Pre-trained Transformer 3) (Brown et al., OpenAI)

An even larger iteration of GPT-2, featuring more parameters and better performance, pushing the boundaries of what autoregressive language models can achieve.

ELECTRA (Clark et al., Google)

Introduces a new pre-training method that's more efficient than the masking approach used in BERT, involving a discriminator model that distinguishes between "real" and "fake" input tokens.

2021

Switch Transformers (Fedus et al., Google)

An extension of the T5 model that incorporates a Mixture of Experts (MoE), allowing the model to scale up significantly by specializing certain parts of the model to particular tasks.

2022

PaLM (Pathways Language Model) (Chowdhery et al., Google)

A scaled-up language model using the Pathways system, which allows training very large models more efficiently by dynamically allocating computational resources.

2023

ChatGPT (based on GPT technology) (OpenAI)

A specialized version of GPT fine-tuned for conversational applications, showing remarkable capabilities in generating human-like responses in dialogues.

These models represent key milestones in the development and application of the Transformer architecture. Each has contributed uniquely, either by introducing new training techniques, scaling up the architecture, or applying the model to new domains or tasks. The list provided focuses on influential releases, and there are certainly other models and variations that contribute to the rich and rapidly evolving landscape of Transformer-based networks.

References

[1] Utkarsh Ankit.

[2] Punyakeerthi BL.

[3] Sasirekha Cota. Deep learning basics — part 7 — feed forward neural networks (ffnn).

[4] Minh-Thang Luong Hieu Pham Christopher D. Manning. Effective approaches to attention-based neural machine translation. 2017.

[5] Chat GPT-4.

[6] Giuliano Giacaglia. https://towardsdatascience.com/transformers-141e32e69591.

[7] Jean Nyandwi.

[8] https://aws.amazon.com/what-is/recurrent-neural network/.

[9] https://www.ibm.com/topics/convolutional-neural networks.

[10] 1 Darshan Bulsara 2 Kashu Yamazaki, 1 Viet-Khoa Vo-Ho and * Ngan Le1. Spiking neural networks and their applications: A review.

[11] Navneet Singh Arora. https://navneet-singh-arora.medium.com/siamese-neural-network-snn.

[12] Ketan Doshi https://towardsdatascience.com/transformers-explained-visually-part-1-overview-of-functionality-95a6dd460452: :text=The https://towardsdatascience.com/transformers-explained-visually-part-1-overview-of-functionality-95a6dd460452: :text=the

[13] Amanatullah.

[14] https://medium.com/@priyankads/evaluation-metrics-in-natural-language-processing-bleu dc3cfa8faaa5. https://medium.com/@priyankads/evaluation-metrics-in-natural-language-processing-bleu-dc3cfa8faaa5.

[15] https://medium.com/@priyankads/rouge-your-nlp-results b2feba61053a. https://medium.com/@priyankads/rouge-your-nlp-results-b2feba61053a.

[16] Kathleen Kenealy Abigail See Jon Deaton, Austin Jacobs. Transformers and pointer-generator networks for abstractive summarizatio. 2019.

[17] https://medium.com/@priyankads/perplexity-of-language-models 41160427ed72.

[18] Sofia Serrano Zander Brumbaugh Noah A. Smith. Language models: A guide for the perplexed. 2023.

[19] https://huggingface.co/docs/transformers/en/perplexity. https://huggingface.co/docs/transformers/en/perplexity.

[20] https://machinelearninginterview.com/topics/machine-learning/meteor-for-machine translation/. https://machinelearninginterview.com/topics/machine-learning/meteor-for-machine-translation/.

[21] https://towardsdatascience.com/transformers-explained-visually-part-3-multi-head-attention-deep-dive 1c1ff1024853. https://towardsdatascience.com/transformers-explained-visually-part-3-multi-head-attention-deep-dive-1c1ff1024853.

[22] https://paperswithcode.com/method/multi-head attention.

[23]

[24] https://nn.labml.ai/transformers/feed$_f orward.html$.

[25] https://machinelearningmastery.com/a-gentle-introduction-to-positional-encoding-in-transformer-models-part 1/.