**Language Models**

* A language model is a probabilistic framework that assigns a probability to a sequence of words.
* It predicts the likelihood of a given word based on the words that precede it.
* This is typically done by representing the words in the sequence as numerical values, such as real-valued vectors, and the
* n using the vectors as input to a mathematical model, such as a neural network.

**Types of Language Models**

1. Statistical N gram Model
2. Neural Network Models

**Statistical N gram Model**

* N-grams, a fundamental concept in NLP, play a pivotal role in capturing patterns and relationships within a sequence of words.
* N-grams are contiguous sequences of ’n’ items, typically words in the context of NLP. These items can be characters, words, or even syllables, depending on the granularity desired. The value of ’n’ determines the order of the N-gram.

**Examples:**

* Unigrams (1-grams): Single words, e.g., “cat,” “dog.”
* Bigrams (2-grams): Pairs of consecutive words, e.g., “natural language,” “deep learning.”
* Trigrams (3-grams): Triplets of consecutive words, e.g., “machine learning model,” “data science approach.”
* 4-grams, 5-grams, etc.: Sequences of four, five, or more consecutive words.

**Code**

import nltk

from nltk import ngrams

from nltk.tokenize import word\_tokenize

from collections import Counter

# Download necessary resources

nltk.download('punkt')

nltk.download('punkt\_tab')

# Example sentence

sentence = "N-grams enhance language processing tasks. N-grams are useful for NLP."

# Tokenize the sentence

tokens = word\_tokenize(sentence)

# Generate bigrams

bigrams = list(ngrams(tokens, 2))

# Count unigrams (individual words)

unigram\_counts = Counter(tokens)

# Count bigrams

bigram\_counts = Counter(bigrams)

# Calculate bigram probabilities

bigram\_probabilities = {}

for bigram, count in bigram\_counts.items():

first\_word = bigram[0]

bigram\_probabilities[bigram] = count / unigram\_counts[first\_word]

# Print results

print("Bigrams and their probabilities:")

for bigram, prob in bigram\_probabilities.items():

print(f"P({bigram[1]} | {bigram[0]}) = {prob:.4f}")

**Problem**

**Define the Corpus**

**The given text corpus consists of three sentences:**

**"The cat sat on the mat"**

**"The cat saw a rat"**

**"The rat ran away"**

**Our goal is to extract bigrams and compute their probabilities.**

**Step1: Generate Bigrams**

A bigram is a sequence of two consecutive words. We extract all bigrams from the corpus:

Extracted Bigrams with Counts

|  |  |
| --- | --- |
| **Bigram** | **Count** |
| "The cat" | 2 |
| "cat sat" | 1 |
| "sat on" | 1 |
| "on the" | 1 |
| "the mat" | 1 |
| "cat saw" | 1 |
| "saw a" | 1 |
| "a rat" | 1 |
| "The rat" | 1 |
| "rat ran" | 1 |
| "ran away" | 1 |

**Step2: Count the Occurrences of Each First Word**

To compute the probability of a bigram P(w2∣w1), we divide the frequency of the bigram (w1,w2) by the total occurrences of w1w\_1w1​ in the corpus.

|  |  |
| --- | --- |
| **First Word** | **Count** |
| "The" | 3 |
| "cat" | 2 |
| "sat" | 1 |
| "on" | 1 |
| "the" | 1 |
| "saw" | 1 |
| "a" | 1 |
| "rat" | 2 |
| "ran" | 1 |

**Step 3: Compute Bigram Probabilities**

We calculate P(w2∣w1) using the formula:

P(w2∣w1)= Count(w1, w2)/ Count(w1)

P(cat | The) = Count("The cat") / Count("The") = 2 / 3 ≈ 0.67

P(sat | cat) = Count("cat sat") / Count("cat") = 1 / 2 = 0.5

P(on | sat) = Count("sat on") / Count("sat") = 1 / 1 = 1

P(the | on) = Count("on the") / Count("on") = 1 / 1 = 1

P(mat | the) = Count("the mat") / Count("the") = 1 / 1 = 1

P(saw | cat) = Count("cat saw") / Count("cat") = 1 / 2 = 0.5

P(a | saw) = Count("saw a") / Count("saw") = 1 / 1 = 1

P(rat | a) = Count("a rat") / Count("a") = 1 / 1 = 1

P(rat | The) = Count("The rat") / Count("The") = 1 / 3 ≈ 0.33

P(ran | rat) = Count("rat ran") / Count("rat") = 1 / 2 = 0.5

P(away | ran) = Count("ran away") / Count("ran") = 1 / 1 = 1

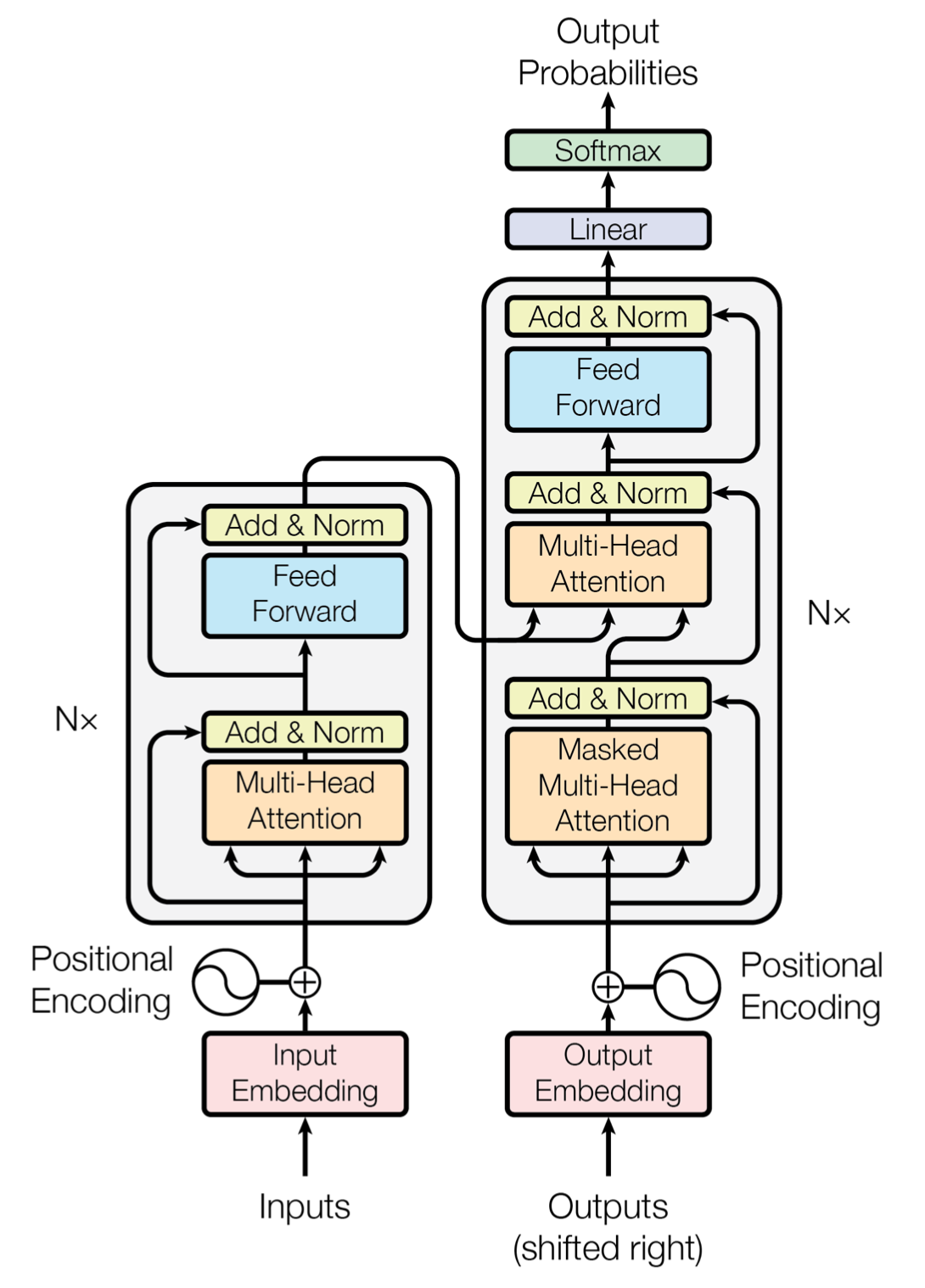
* A probability of 1 means that whenever the first word appears, the second word always follows.
* A probability less than 1 means that the first word can be followed by multiple different words.

**DIY: Corpus**

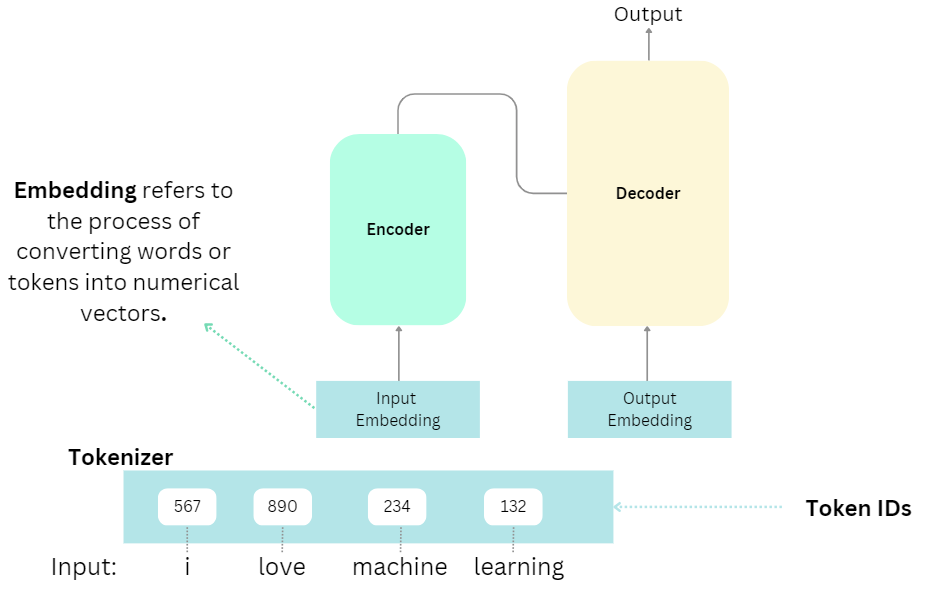
**"The dog barked loudly"**

**"The dog chased the cat"**

**"The cat climbed the tree"**



* The transformer architecture consists of two main components: the encoder and the decoder.
* The Encoder processes the input sequence, breaking it down into meaningful representations. On the other hand, a Decoder takes these representations and generates the output sequence, like a translation or a text continuation.



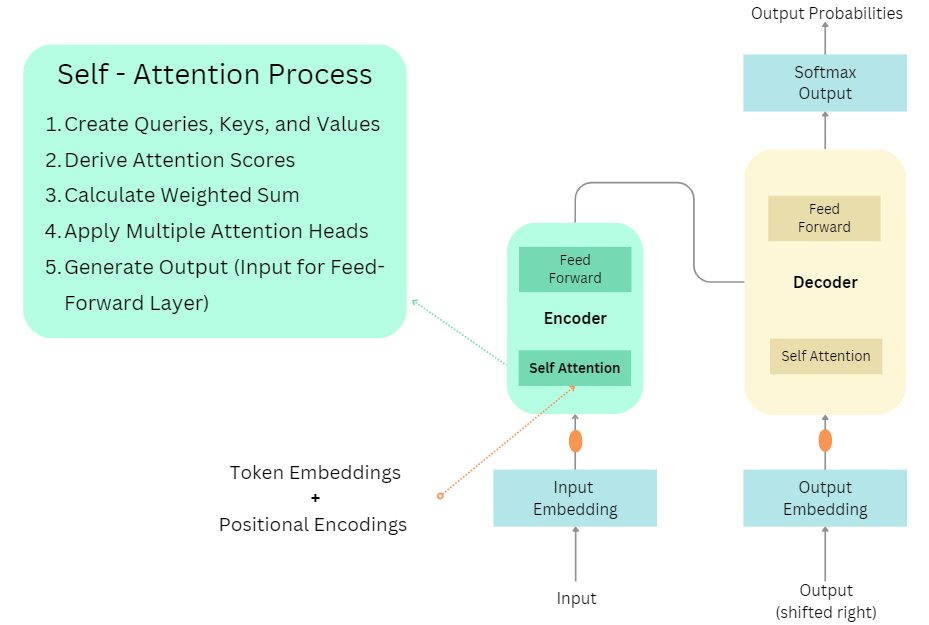
1.**Word to Vector Conversion**: Each word or token in the input text is assigned a unique numerical vector. This vector represents the word’s meaning and context within the given language. These word vectors are often pre-trained on vast text corpora and capture semantic relationships between words.

2. **Embedding Layer**: These word vectors pass through an “embedding layer” in the model. This layer acts as a lookup table, associating each word with its corresponding vector.

Next, we assign Positional Encodings.

Positional encoding is a technique used to provide the model with information about the position or order of words in a sequence. Since transformers process words in parallel rather than sequentially.

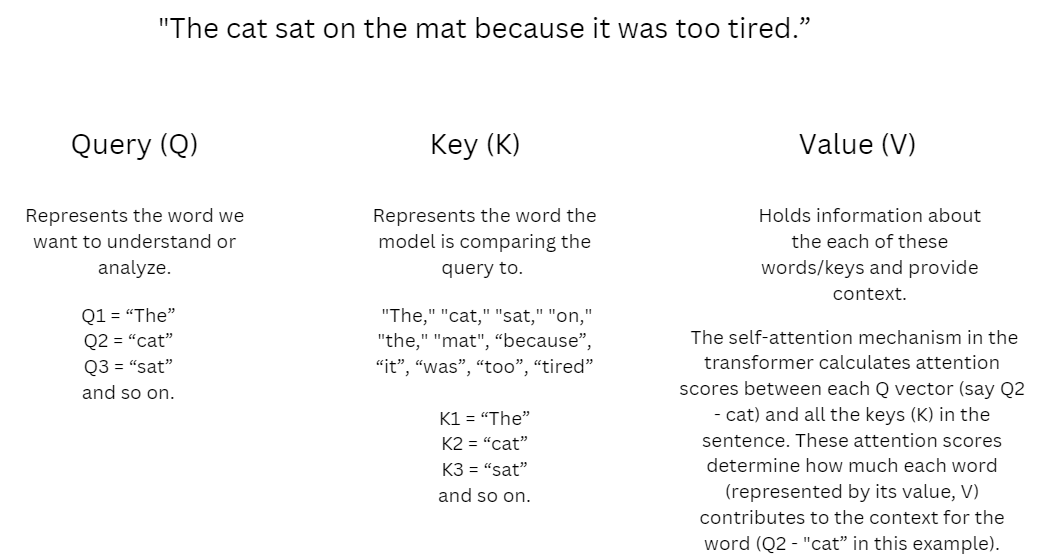
3. **Self-Attention Layer:** The **self-attention layer** is a pivotal component of the transformer architecture, and it is responsible for capturing relationships and dependencies between words in a sequence.



**1. Create Queries, Keys, and Values:** To understand how words relate to each other, the self-attention layer creates three sets of vectors for each word in the input sequence:

* **Query (Q)**: Represents the word we are currently focusing on. Each word has its corresponding Query vector.
* **Key (K)**: Represents all words we want to pull information from to help determine the relevance of each word to the Query.
* **Value (V)**: Contains the information of words that we’ll extract when there’s a match between Query and Key.

Let’s outline the Q, K, and V values for the following sentence.



**2. Derive Attention Scores:** Next, the self-attention mechanism calculates attention scores between the Query vectors (Q) and the Key vectors (K). These scores indicate how much each word should pay attention to other words. Higher scores imply higher attention, suggesting that the word is more relevant to the Query. Lower scores suggest lower relevance or importance to the Query.

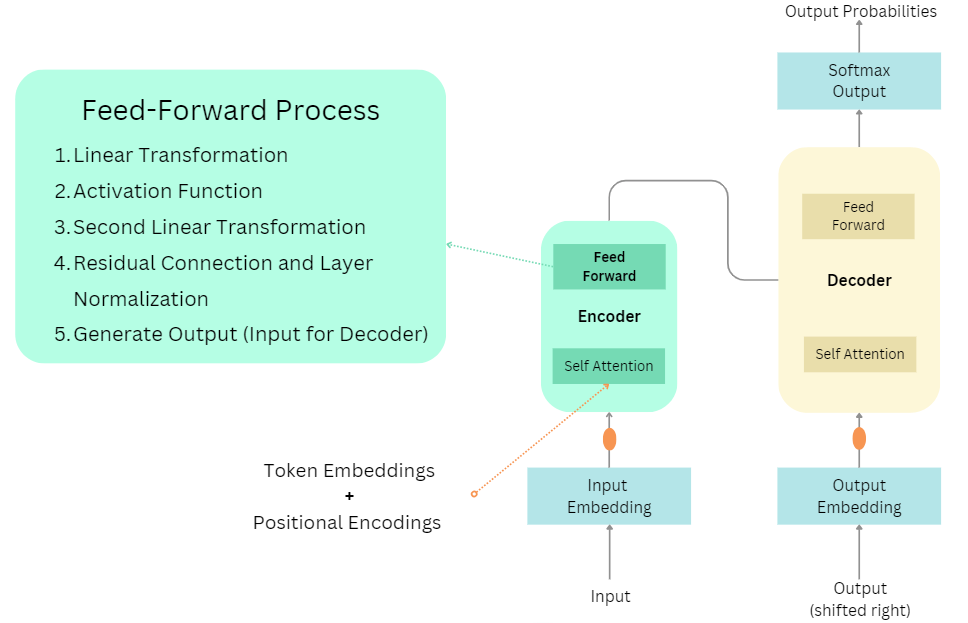
**3.** **Calculat**e **Weighted Sum:** Using the calculated attention scores, the self-attention layer computes a weighted sum of the value vectors (V). Words with higher attention scores contribute more to this weighted sum. This step effectively combines the context of all words in the sequence for the Query word.

**4. Apply Multiple Attention Heads:** To enhance its understanding of context, the transformer often employs multiple sets of Queries, Keys, and Values, known as “attention heads.” These heads allow the model to focus on different aspects of the sequence simultaneously. It’s like having multiple people in a group discussion, each paying attention to different parts of the conversation.

**5. Generate Output:** The output contains contextualized representations of the words in the input sequence, taking into account their relationships with other words. This output now becomes the input for the Feed-Forward layer.

**Feed-Forward Layer**

The feed-forward layer in the transformer architecture is a neural network layer that is applied to the output of the self-attention layer. It is a position-wise transformation, meaning that it is applied to each position in the sequence independently of the other positions.



The feed-forward layer plays a multifaceted role in the transformer architecture, facilitating the modeling of long-range dependencies, introducing non-linearity for complex relationships, and boosting the model’s capacity to process and extract meaningful information from input sequences.

**Residual Connection and Layer Normalization:** The transformer architecture addresses challenges like the vanishing gradient problem by using a residual connection. This means that the output of a layer is added to the initial input, allowing the model to learn to only make small changes to the input. Layer normalization is applied before adding the residual connection, promoting stability in training and ensuring consistent gradients throughout the layers.

1. Residual Connection (or Skip Connection): Imagine you’re trying to solve a math problem, and your friend suggests a intricate method to solve it. Instead of completely discarding your initial approach, you synergize both: your method and your friend’s, to derive the answer.

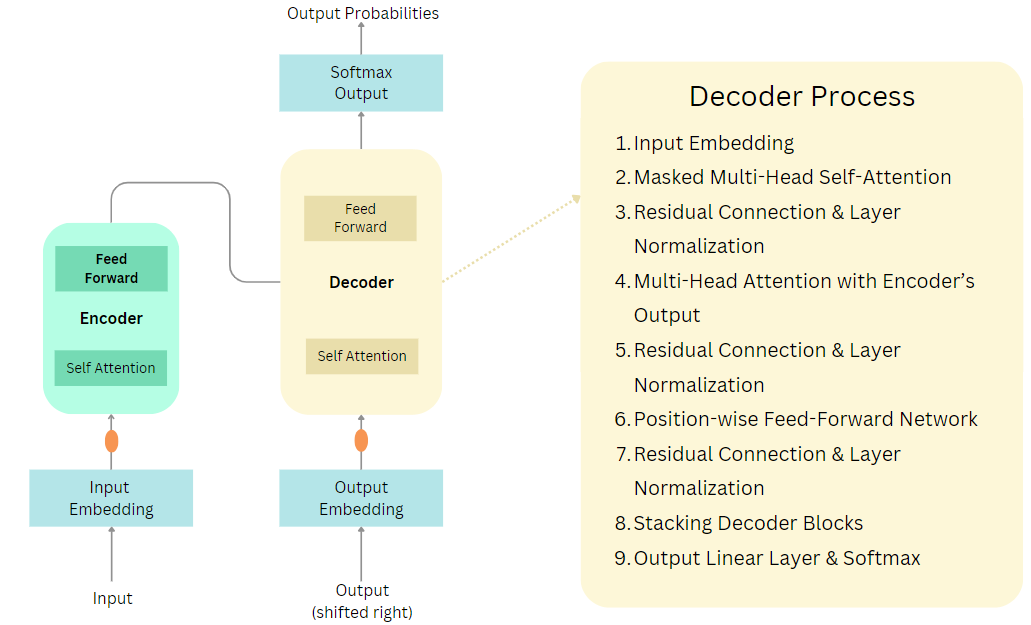
Similarly, in deep learning, as we stack more layers, the learning process might become challenging. Residual connections come to the rescue by letting the input of a layer bypass some intermediate layers and directly merge with the output. In essence, the network can blend the original information (akin to your method) with the more refined information (similar to your friend’s approach).

Mathematically, it looks like this: Output = Input + Transformed Input

2. Layer Normalization: Let’s say you’re baking cookies by following a specific recipe. But every time you bake, your oven’s temperature fluctuates a lot. This makes it hard to consistently bake good cookies. To solve this, you decide to use an oven thermometer to ensure the temperature is consistent every time.

Layer normalization functions much like that oven thermometer. In deep learning, the values (like the oven’s temperature) can become too big or too small as they pass through layers. This can cause problems and make the network harder to train. Layer normalization, thus, standardizes these values across layers, ensuring consistency. Specifically, it adjusts inputs to a layer such that their average is 0 and their variability (standard deviation) is 1. This equilibration not only accelerates the learning process but also fortifies the model’s adaptability to unfamiliar data.

Assuming the output has been passed to the Decoder block, the diagram illustrates the steps encompassed within the Decoder process.



1. **Input**: The decoder takes in a part of the English sentence that has already been generated (in the beginning, this is just a start token).
2. **Self-Attention**: Just like people often think about how words in a sentence relate to each other when translating, the decoder checks how each word in the English input relates to the other words. This helps in making sure the translation flows naturally and maintains context. This step is similar to the self-attention mechanism in the encoder but is focused on the target sentence.
3. **Cross-Attention**: Next, the decoder pays “attention” to the French sentence (the encoded version). It’s like when you glance back at the original French text to make sure you’re translating correctly. The decoder checks which parts of the French sentence are most relevant to the word it’s currently trying to produce in English.
4. **Feed Forward and Output**: The decoder then processes the information it has gathered from the above steps, decides on the next English word (or token), and adds it to the output sequence.
5. **Repeat**: The decoder keeps doing this, generating one word at a time, until it decides the translation is complete and produces an end token.

The decoder in the transformer architecture is a multi-step process.

1. **Input Embedding:**Just as with the Encoder, the input to the Decoder (which is the target sequence during training) is first embedded into continuous vectors. This embedded input is then added to the positional encoding to incorporate the sequence’s order information.
2. **Masked Multi-Head Self-Attention:**The Decoder employs a masked version of the self-attention mechanism, ensuring each position can only attend to positions before it (or to itself) in the sequence. This masking is essential during training to prevent a word from “seeing” future words, maintaining the autoregressive property of the Decoder. It’s important to note that this masking is only applied during training. During inference, the decoder can attend to all words in the target sequence, including future words.
3. **Residual Connection & Layer Normalization (post Self-Attention):** A residual connection adds the output of the self-attention layer to its input. Layer normalization is then applied to stabilize and scale the activations.
4. **Multi-Head Attention over Encoder’s Output:**This multi-head attention mechanism uses the Encoder’s output as its keys and values and the output from the Decoder’s self-attention as its queries. It allows the Decoder to focus on relevant parts of the source sequence while generating the target sequence.
5. **Residual Connection & Layer Normalization:**The output from the Multi-Head attention layer is combined with its input using a residual connection. This is followed by layer normalization.
6. **Position-wise Feed-Forward Network:**The Decoder also has a position-wise Feed-Forward layer, similar to the one in the Encoder. This network consists of two linear (dense) layers with a ReLU activation function in between. It operates on each position independently, adding more expressive power to the Decoder.
7. **Residual Connection & Layer Normalization**: The output of the Feed-Forward network is combined with its input via a residual connection. Another layer normalization is applied.
8. **Stacking of Decoder Blocks:**Just as multiple Encoder blocks are stacked in the Transformer, multiple Decoder blocks are stacked as well. The output from one block serves as the input to the subsequent block, passing through the above operations repeatedly.
9. **Output Linear Layer & Softmax:**Once the data passes through all the Decoder blocks, it goes through a final linear layer which maps it to the desired output vocabulary size. A softmax function is then applied to produce the probability distribution over the target vocabulary, generating the final output sequence.

**Training of Large language Models: Concept of Epoch and batch size**

1. Epoch

* An epoch refers to one complete pass through the entire training dataset during the training process.
* During each epoch, the model gets exposed to every sample in the training set once, and it updates its internal weights and parameters based on the loss computed from those samples.
* Multiple epochs are typically required to train a model effectively, as one pass through the data may not be enough to converge to a good solution.
* After each epoch, the model parameters are updated, and the process continues for the next epoch, gradually improving the model's performance.

For example:

If you have 1,000 training samples, an epoch means the model has seen and processed all 1,000 samples once.

Usually, the number of epochs is a hyperparameter chosen by the model designer, and training may stop either after a fixed number of epochs or when performance on the validation set stops improving.

2. Batch Size

* The batch size refers to the number of training samples that are processed at one time before the model's weights are updated.
* Instead of using all the training data at once (which would be computationally expensive), the data is divided into smaller subsets, known as batches. The model processes each batch, computes the loss, and updates the weights after each batch.
* Batch size is a hyperparameter that can significantly affect training speed and the generalization ability of the model.

We split the training set into many batches. When we run the algorithm, it requires one epoch to analyze the full training set. An epoch is composed of many iterations (or batches).

Iterations: the number of batches needed to complete one Epoch.

Batch Size: The number of training samples used in one iteration.

Epoch: one full cycle through the training dataset. A cycle is composed of many iterations.

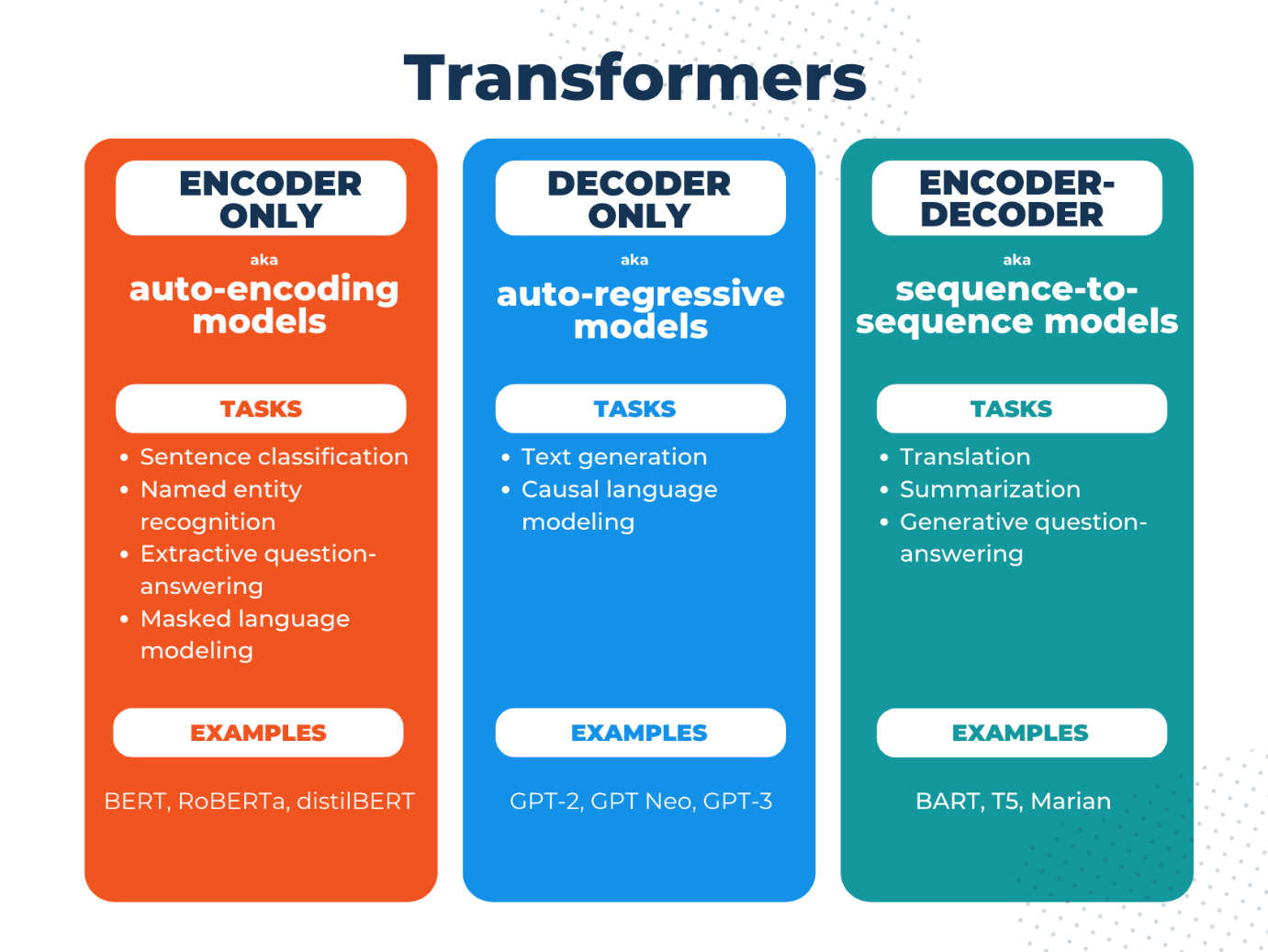
Number of Steps per Epoch = (Total Number of Training Samples) / (Batch Size)

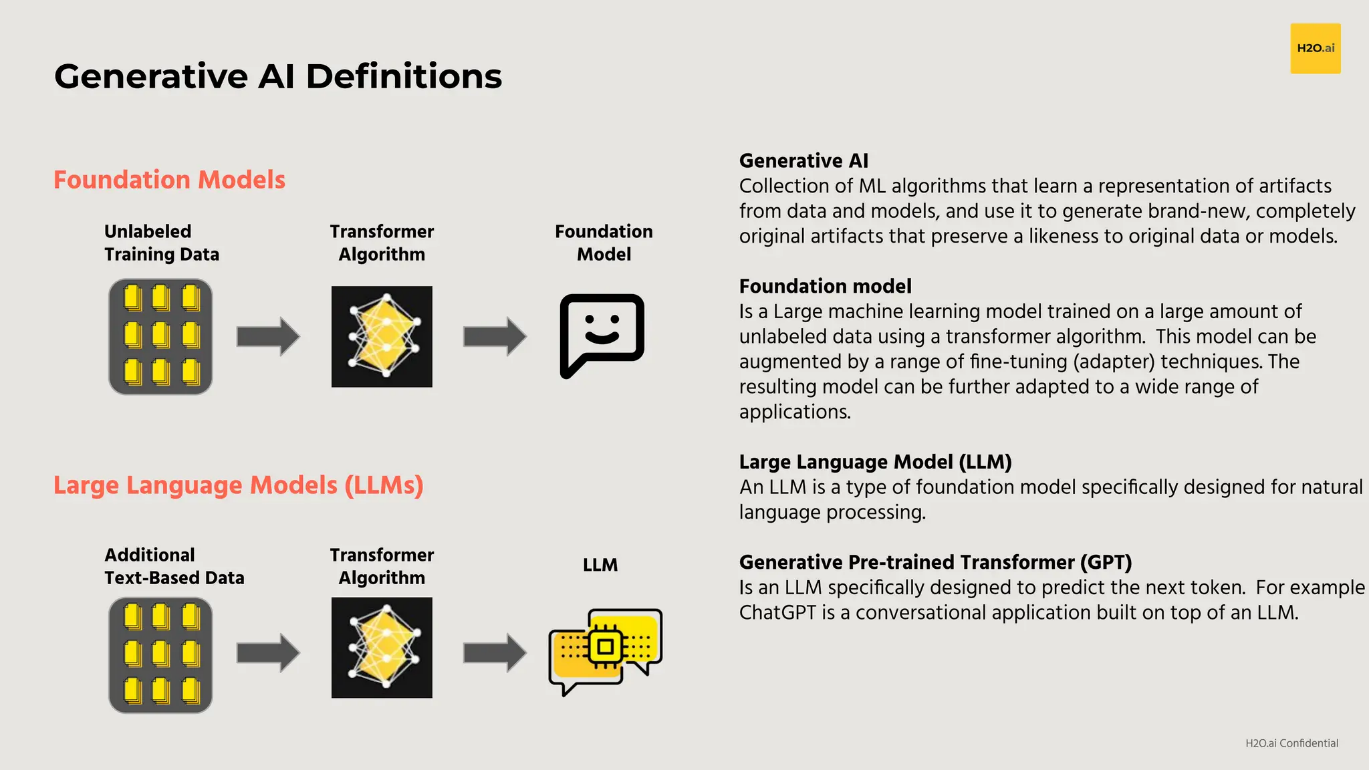
Example

Training Set = 2,000 images

Batch Size = 10

Number of Steps per Epoch = 2,000 / 10 = 200 steps



****

**The Journey of Training LLMs**

There are two key stages in training LLMs:

1. Pretraining: The model learns general knowledge and patterns from large datasets.
2. Fine-Tuning: The pretrained model is refined for niche tasks or domains to better serve our specific needs.

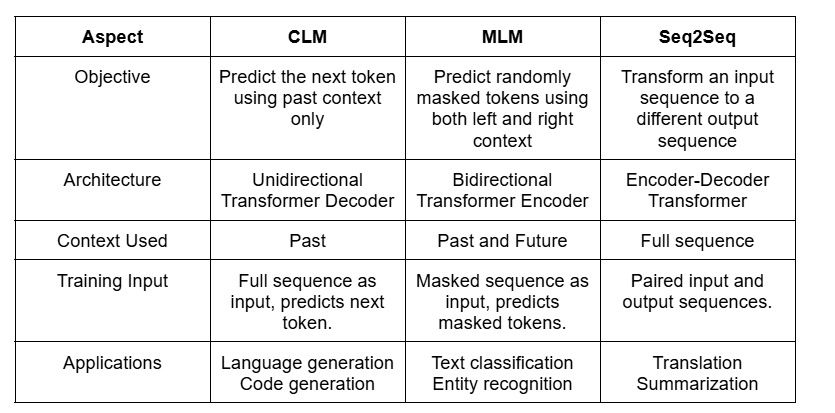
**Pretraining**

* Pre-training of Large Language Models (LLMs) refers to the initial phase of training where a model learns to process and generate language by exposing it to vast amounts of text data.
* During this phase, the model is not yet trained for specific tasks (like answering questions, translation, or summarization).
* Instead, it learns general language patterns, grammar, word associations, and semantic relationships that allow it to perform a wide range of natural language processing (NLP) tasks later on.
* Pre-training is typically unsupervised, meaning the model doesn’t require labeled data (i.e., data with explicit input-output pairs), but instead learns from large corpora of text by predicting parts of the text, such as the next word in a sentence or a missing word in a passage.

**Why Pretrain?**

Well it is obvious why big open source companies like OpenAI, Meta, Google etc., pretrain LLMs. Because someone has to right?

**Techniques for Pretraining**



**Steps involved in Pre-training**

1. **Data Collection:** Data collection for pre-training can be done by collecting/scrapping data from books, codes, articles, wiki and web pages.
2. **Tokenization:** Text data is split into smaller chunks called tokens. For example, a token could be a word or part of a word. These tokens are then converted into numerical representations (vectors) that the model can process.
3. **Model Initialization:** In this stage, we change the model architecture according to our requirements.

* Model Downscaling — We need this method if we have to use small models. Here, we take existing models and remove layers from middle and train the model.
* Depth Upscaling — We need this method if we have to use large models. Here, we take two existing models and train them on dataset. Later, we concatenate few layers from each model to create one model. This model is trained again on the dataset.

1. **Learning Objective:** The model uses a pre-training objective to learn.

For example, in causal language modeling (CLM), the model learns to predict the next word in a sequence. In masked language modeling (MLM), it learns to predict missing words.

1. **Training Process:** The model undergoes numerous iterations (epochs) over the text corpus, adjusting its internal parameters based on how accurately it predicts the missing or next words. This process involves optimization techniques like gradient descent and backpropagation to minimize errors in prediction.
2. **Evaluation:** In this stage, we compare the performance of our model with other models. This is done by evaluating our model on common benchmark datasets.

**Outcome of Pre-training**

**General Language Understanding:**

* By the end of pre-training, the model develops a rich understanding of language, allowing it to recognize relationships between words, entities, and concepts. This helps it generalize well to various downstream tasks.
* Learned Parameters: The model has adjusted millions or billions of parameters (weights) to capture linguistic patterns, which can now be fine-tuned for specific tasks (like question answering, summarization, or sentiment analysis).

**Advantages of Pretraining**

1. **Generalized Language Understanding**
   * The model learns a **wide range of linguistic patterns**, making it versatile for multiple applications.
2. **No Need for Labeled Data**
   * Uses **self-supervised learning**, so there’s no need for expensive human-labeled datasets.
3. **Reusable Across Tasks**
   * A single pretrained model (e.g., GPT-3, BERT) can be **fine-tuned** for different applications like chatbots, summarization, or medical diagnosis.
4. **Captures World Knowledge**
   * Trained on massive text corpora, LLMs can answer **general knowledge questions** effectively.
5. **Scalability**
   * Pretrained models can **scale up** using more data and larger architectures to improve performance.

**Disadvantages of Pretraining**

1. **Extremely Resource-Intensive**
   * Requires massive **compute power (TPUs, GPUs)**, making it **expensive** to train from scratch.
2. **Lacks Task-Specific Expertise**
   * A pretrained model may not perform well on specialized tasks (e.g., legal or medical texts) **without fine-tuning**.
3. **Bias and Ethical Issues**
   * Training on unfiltered internet data can introduce **biases, misinformation, or harmful outputs**.
4. **Long Training Time**
   * Pretraining large models can take **weeks or months** even with high-end hardware.

**Finetuning**

* Fine-tuning of Large Language Models (LLMs) is the process of taking a pre-trained model and adapting it to a specific task by continuing training on a smaller, task-specific dataset.
* Data for fine-tuning is human generated or machine (LLM) generated as it requires data in instruction format.

**Steps involved in Finetuning**

**1. Pre-trained Model Selection:**

* You begin with a pre-trained LLM (such as GPT, BERT, T5, or another transformer-based model) that has already learned the basics of language. The choice of model depends on the task; for instance, BERT might be chosen for tasks that require deep understanding of context (like question answering), while GPT is often used for generative tasks.

**2. Preparing Task-Specific Data:**

Fine-tuning requires a labeled dataset that is specific to the task.

* For example:For text classification, you would need a set of documents labeled with specific categories.
* For named entity recognition (NER), the dataset would contain texts with labeled entities (e.g., people, places, organizations).
* For machine translation, you would need a parallel dataset with source and target language pairs.

**3.Adapting the Model Architecture:**

* Depending on the task, you may need to adjust the model slightly.
* For example:For classification tasks, a final classification layer (e.g., a softmax layer) might be added on top of the pre-trained model.
* For question answering, the model may be adapted to return a specific span of text as the answer.

**4. Training the Model on the Task Data:**

* Fine-tuning is done by training the pre-trained model on the labeled task-specific data. During this process, the model’s parameters are adjusted to minimize the error in predictions for the specific task.

**5. Learning rate:**

* Fine-tuning is typically done with a lower learning rate than pre-training, since the model already has learned general language patterns and only needs to adjust to the new task.
* Epochs:The model is trained for a smaller number of epochs compared to pre-training, as it is primarily making task-specific adjustments rather than learning from scratch.

**6. Evaluation and Validation:**

* After fine-tuning, the model’s performance is evaluated on a separate validation or test set that the model has not seen during training.
* Metrics used for evaluation depend on the task:
* Accuracy or F1 score for classification tasks.
* BLEU score for machine translation.
* ROUGE score for summarization.

**Advantages of Fine-Tuning**

1. **Task-Specific Performance Boost**
   * Fine-tuned models perform **much better** than generic models on specific tasks (e.g., sentiment analysis, medical diagnosis).
2. **Less Data & Compute Needed**
   * Fine-tuning only requires **a fraction of the dataset** compared to pretraining.
3. **Can Overcome Biases**
   * With domain-specific fine-tuning, the model can be adjusted to reduce biases present in the original pretrained model.
4. **Efficient Transfer Learning**
   * Since the model is already pretrained, fine-tuning adapts it **quickly** for new domains.
5. **Customization for Organizations**
   * Companies can fine-tune LLMs using **their own private datasets**, making the model more relevant to their use case.

**Disadvantages of Fine-Tuning**

1. **Requires Labeled Data**
   * Unlike pretraining, fine-tuning **needs labeled datasets**, which can be costly and time-consuming to create.
2. **Risk of Overfitting**
   * Fine-tuning on a **small dataset** may lead to overfitting, reducing the model’s ability to generalize.
3. **Catastrophic Forgetting**
   * If not fine-tuned carefully, the model may **forget** the general knowledge it learned during pretraining.
4. **Ethical Risks in Customization**
   * Fine-tuning on biased or poor-quality data can make the model **even more biased** than the original pretrained version.
5. **Need for Hyperparameter Tuning**
   * Fine-tuning requires careful tuning of learning rates, batch sizes, and other parameters to get the best results.

|  |  |  |
| --- | --- | --- |
| **Feature** | **Pretraining** | **Fine-Tuning** |
| Goal | Learn general language patterns | Adapt to a specific task/domain |
| Dataset Type | Large-scale, unlabeled text | Small, labeled task-specific data |
| Training Type | Self-supervised learning | Supervised learning |
| Model Output | Generic LLM (GPT, BERT, LLaMA) | Specialized model (chatbots, medical AI) |
| Training Time | Weeks to months | Hours to days |
| Computational Cost | Very high (TPUs, clusters) | Lower (single GPU can work) |