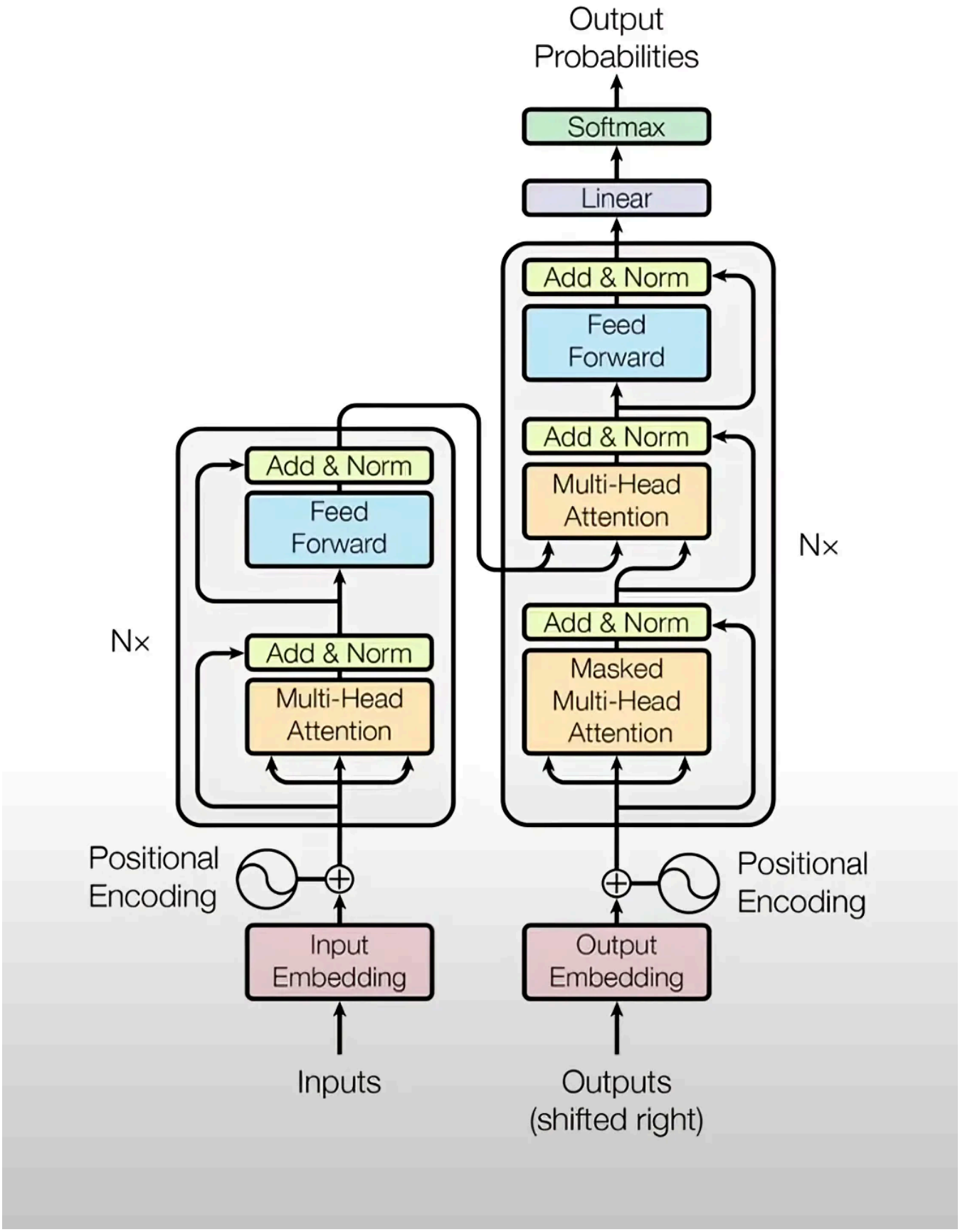
Transformer from scratch using Pytorch

BavalpreetSinghh · Follow

22 min read · Jun 15, 2024

In today's blog we will go through the understanding of transformers architecture. Transformers have revolutionized the field of Natural Language Processing (NLP) by introducing a novel mechanism for capturing dependencies within sequences through attention mechanisms. Let's break it down, implement it from scratch using PyTorch.



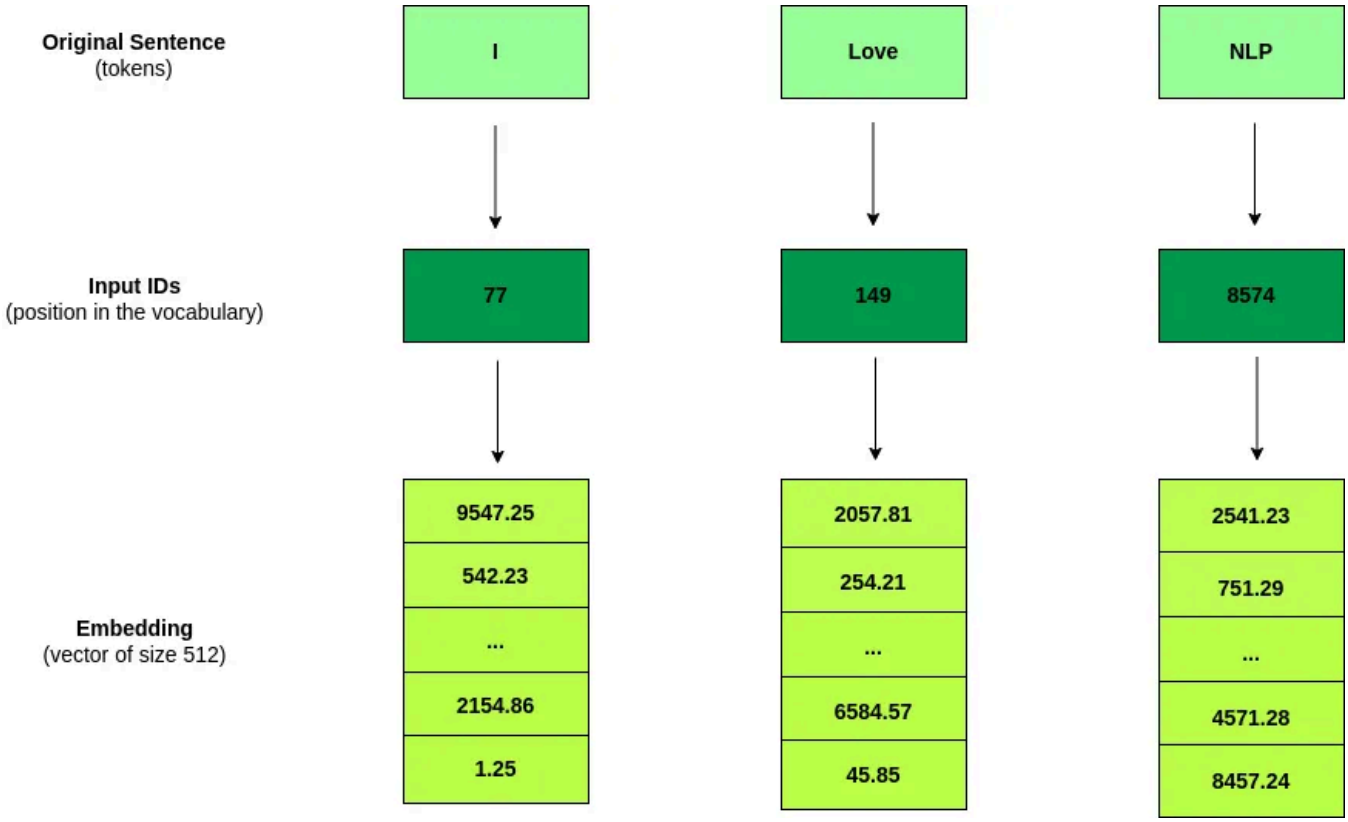
source: paper

```
import torch
import torch.nn as nn
import math
```

- `torch`: The main PyTorch library.
- `torch.nn`: Provides neural network components.
- `math`: Provides mathematical functions.

Input Embedding

It allows to convert the original sentence into a vector of X dimensions (The original Transformer model uses 512 as a size of dimension(d_model) for the base version and d_model = 1024 for the larger version.).



Source: Author

Purpose of `__init__()` Method

The primary purposes of the `__init__()` method are:

- To initialize the state of an object (i.e., to set up initial values for the object's attributes).

- To define the layers and components that the neural network module will use.
- To ensure that any necessary setup or initialization code is executed when an object is created.

`super()`

The `super()` function is used to call a method from the parent class.

- `super()` returns a temporary object of the superclass that allows you to call its methods.
- In the case of `super().__init__()`, it calls the `__init__` method of the parent class (`nn.Module`).

Using `super()` is especially important in the context of inheritance because it ensures that the initialization code of the base class runs, setting up any necessary internal structures and attributes that the subclass might rely on.

```
class InputEmbeddings(nn.Module):

    def __init__(self, d_model: int, vocab_size: int) -> None:
        super().__init__()
        self.d_model = d_model
        self.vocab_size = vocab_size
        self.embedding = nn.Embedding(vocab_size, d_model)

    def forward(self, x):
        # (batch, seq_len) --> (batch, seq_len, d_model)
        # Multiply by sqrt(d_model) to scale the embeddings according to the pa
        return self.embedding(x) * math.sqrt(self.d_model)
```

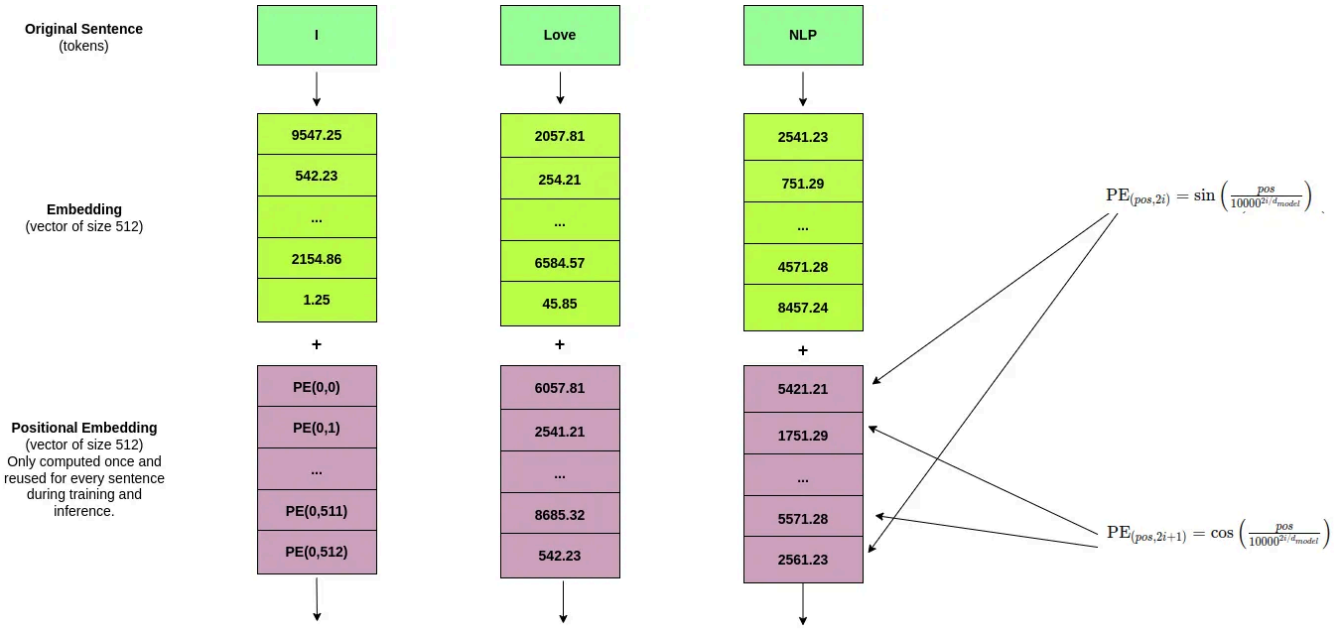
- `nn.Embedding(vocab_size, d_model)`: This creates an embedding layer that maps indices(usually representing words) to a `d_model`-dimensional vector. The embedding layer is initialized randomly and these vectors are learned during training. So given a number it will provide you the same vector everytime. To learn more please refer this link.
- `self.embedding(x)`: Here, `x` is a tensor of token indices. The embedding layer looks up the vector for each token index in `x`. For example, if `x` is [0, 1, 2], it

looks up the vectors for the indices 0, 1, and 2.

- `* math.sqrt(self.d_model)`: This scales the embeddings by the square root of `d_model`. This is often done to maintain the variance of the embeddings when they are passed through the network, helping with training stability. To read more about this please refer to section 3.4 of the paper.

PositionalEncoding Class

Positional encoding is a crucial component in transformer models, which helps the model understand the position of each word in a sentence. Since transformers do not inherently process tokens in a sequential manner like RNNs (Recurrent Neural Networks), they need a way to incorporate the order of tokens. This is achieved through positional encodings, which are vectors added to the word embeddings.



Source: Author

Here's an explanation of how positional encoding works and why it's important:

Importance of Positional Encoding

Transformers process the entire sequence of tokens simultaneously, which allows for efficient parallel computation but also means they lack information about the order of tokens. Positional encoding provides a way to introduce this sequential information.

How Positional Encoding Works

1. **Embedding Size:** The positional encoding vectors have the same size as the word embeddings, typically 512 dimensions (in the case of the original Transformer model). This ensures that the positional information can be seamlessly added to the embeddings without altering their dimension.

2. **Creating Positional Encodings:** The positional encoding vectors are designed to represent the position of each word in the sequence. These vectors are created using a combination of sine and cosine functions of different frequencies. This allows the model to learn and distinguish between different positions in a way that is smooth and continuous.

3. **Adding Positional Encodings:** Once the positional encoding vectors are generated, they are added element-wise to the corresponding word embeddings. This combined representation incorporates both the semantic information from the word embeddings and the positional information from the positional encodings.

Mathematical Formulation

For a given position pos and embedding dimension i:

$$\text{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$
$$\text{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

Where:

- $\text{PE}_{(pos,2i)}$ is the value of the positional encoding at position pos for the even dimension $2i$.
- $\text{PE}_{(pos,2i+1)}$ is the value of the positional encoding at position pos for the odd dimension $2i + 1$.
- d_{model} is the dimension of the embeddings (e.g., 512).

```
class PositionalEncoding(nn.Module):

    def __init__(self, d_model: int, seq_len: int, dropout: float) -> None:
        super().__init__()
        self.d_model = d_model
        self.seq_len = seq_len
        self.dropout = nn.Dropout(dropout)
        # Create a matrix of shape (seq_len, d_model)
        pe = torch.zeros(seq_len, d_model)
```

```
        # Create a vector of shape (seq_len)
        position = torch.arange(0, seq_len, dtype=torch.float).unsqueeze(1) # (
        # Create a vector of shape (d_model)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(1
        # Apply sine to even indices
        pe[:, 0::2] = torch.sin(position * div_term) # sin(position * (10000 **
        # Apply cosine to odd indices
        pe[:, 1::2] = torch.cos(position * div_term) # cos(position * (10000 **
        # Add a batch dimension to the positional encoding
        pe = pe.unsqueeze(0) # (1, seq_len, d_model)
        # Register the positional encoding as a buffer
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x + (self.pe[:, :x.shape[1], :]).requires_grad_(False) # (batch, se
        return self.dropout(x)
```

`__init__` method:

- Initializes with `d_model` (dimension of the embeddings), `seq_len` (maximum sequence length), and `dropout` (dropout rate).
- Creates a matrix of shape (`seq_len, d_model`), then position variable is calculating the numerator and div_term the denominator(we have calculated it in log space for numerical stability. For more refer this link.
- Creates a positional encoding matrix `pe` with sine and cosine functions to encode positional information. ***In embedding of every word sin will be used at even positions and cosine for the odd ones.***
- Because there will be batch of sentences therefore we need to add the batch dimension to the tensor (pe = pe.unsqueeze(0)), the shape will now become (seq_len, d_model) to (1, seq_len, d_model)
- Uses `register_buffer` to store `pe` without making it a learnable parameter.

`forward` method:

- Adds positional encodings to the input embeddings `x` and applies dropout. We also tell model to not to learn these pos encoding because they are fixed and will always be same using `requires_grad_(False)` .

LayerNormalization Class

Layer normalization is a technique used to improve the training of deep neural networks by normalizing the inputs across the features for each training example. It helps in stabilizing the learning process, improving convergence, and reducing the dependence on careful initialization of parameters.

Understanding the Input x

- **Word Embedding:** In the context of transformer models, each word or token in a sentence is converted into a dense vector (embedding) of fixed size (e.g., 512 dimensions). These embeddings capture semantic information about the words.
- **Sentence Representation:** Sentences are represented as a sequence of these word embeddings.
- These vectors are multidimensional representations capturing various aspects of the word's meaning and syntactic role.
- **The dimensions of the word embeddings (e.g., 512 dimensions) are referred to as "features."**
- Each feature corresponds to a specific aspect of the word's representation. These could include syntactic roles, semantic meanings, contextual usage, etc.
- When applying layer normalization in transformer models, the input x is usually a 3-dimensional tensor with the shape (*batch_size,seq_len,d_model)*, where:
- **batch_size**: Number of sentences or sequences in a batch.
- **seq_len**: Number of tokens in each sentence or sequence.
- **d_model**: Dimensionality of the word embeddings (e.g., 512).

Example

Consider the example where you have a batch of sentences, each sentence represented by a sequence of word embeddings:

```
# Create dummy input with shape (batch_size, seq_len, d_model)
input_data = torch.randn(32, 10, 512)  # 32 sentences, each with 10 tokens, eac
```

Here:

- Each element in the input tensor x corresponds to the embedding of a word in a sentence.
- Layer normalization will be applied across the last dimension (the embedding dimension or we can say features) for each word embedding.

Normalization Across Features:

- For each token embedding (a vector of size d_model), compute the mean and variance across its features.
- Normalize the embedding by subtracting the mean and dividing by the standard deviation.
- This process is done for each token independently, ensuring each token's embedding is normalized based on its own feature values.

Here's the mathematical formulation for layer normalization:

1. **Compute Mean and Variance**:
   For each input x in the layer, compute the mean μ and variance σ^2 across the features.

$$\mu = \frac{1}{H}\sum_{i=1}^{H} x_i$$

$$\sigma^2 = \frac{1}{H}\sum_{i=1}^{H}(x_i - \mu)^2$$

   where H is the number of features.

2. **Normalize**:
   Normalize the input using the computed mean and variance.

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

   Here, ϵ is a small constant added for numerical stability.

3. **Scale and Shift**:
   Apply learned parameters γ (scale) and β (shift) to the normalized input.

$$y_i = \gamma\hat{x}_i + \beta$$

Source : GPT 3.5 Turbo

Parameters: Gamma (γ) and Beta (β)

- **Gamma (γ):** This is a learnable scale parameter. After normalization, the data is centered around zero with unit variance. The gamma parameter allows the model to scale the normalized data.
- **Beta (β):** This is a learnable shift parameter. After scaling the data with gamma, the beta parameter allows the model to shift the normalized data.

Note — Gamma is multiplicative and Beta is additive, why we use them?
Because we want the model to have a possibilty to amplify these values when he needs these values to be amplified. the model will learn to multiply this gamma by these values in such a way to amplify the values that it wants to be amplified.

```
class LayerNormalization(nn.Module):

    def __init__(self, features: int, eps:float=10**-6) -> None:
        super().__init__()
        self.eps = eps
        self.alpha = nn.Parameter(torch.ones(features)) # alpha is a learnable
        self.bias = nn.Parameter(torch.zeros(features)) # bias is a learnable p

    def forward(self, x):
        # x: (batch, seq_len, hidden_size)
         # Keep the dimension for broadcasting
        mean = x.mean(dim = -1, keepdim = True) # (batch, seq_len, 1)
        # Keep the dimension for broadcasting
        std = x.std(dim = -1, keepdim = True) # (batch, seq_len, 1)
        # eps is to prevent dividing by zero or when std is very small
        return self.alpha * (x - mean) / (std + self.eps) + self.bias__init__ m
```

- Initializes with a small epsilon value `eps` to prevent division by zero.
- Defines `alpha` (scale parameter) and `bias` (shift parameter) as learnable parameters.(*`nn.Parameter` makes them learnable*)

> *Often denoted as `gamma` and `beta` in layer normalization literature*

`forward` method:

- Normalizes the input `x` by subtracting the mean and dividing by the standard deviation, then applies `alpha` and `bias`.

FeedForwardBlock Class

FeedForward is basically a fully connected layer, that transformer uses in both encoder and decoder. It consists of two linear transformations with a ReLU activation in between. This helps in adding non-linearity to the model, allowing it to learn more complex patterns.

Linear Layers: `self.linear_1` and `self.linear_2` are linear transformations that project the input to a higher-dimensional space (`d_ff`) and back to the original dimensionality (`d_model`).[*You will see this in code below*]

Mathematical Formulation

Given an input x from the previous layer, the feedforward network performs the following operations:

1. Apply a linear transformation: x1 = W1x+b1
2. Apply a ReLU activation: x2 = ReLU(x1)
3. Apply another linear transformation: y = W2x2+b2

Here, W1 and W2 are weight matrices, and b1 and b2 are bias vectors.

```
class FeedForwardBlock(nn.Module):

    def __init__(self, d_model: int, d_ff: int, dropout: float) -> None:
        super().__init__()
        self.linear_1 = nn.Linear(d_model, d_ff) # w1 and b1
        self.dropout = nn.Dropout(dropout)
        self.linear_2 = nn.Linear(d_ff, d_model) # w2 and b2

    def forward(self, x):
        # (batch, seq_len, d_model) --> (batch, seq_len, d_ff) --> (batch, seq_
        return self.linear_2(self.dropout(torch.relu(self.linear_1(x))))
```

`__init__` method:

- Initializes with `d_model` (input dimension), `d_ff` (dimension of the feedforward layer), and `dropout` (dropout rate).
- Defines two linear layers with a ReLU(`torch.relu(x)` introduces non-linearity into the model.) activation and dropout(`self.dropout(x)` is used to prevent overfitting by randomly setting a fraction of the input units to zero during training.) in between.

`forward` method:

- Applies the first linear layer, ReLU activation, dropout, and then the second linear layer.

MultiHeadAttentionBlock Class

Multi-head attention is a core component of the transformer architecture, enabling the model to focus on different parts of the input sequence simultaneously. Let's break down how multi-head attention works and why it is essential.

1. Self-Attention Mechanism

Before understanding multi-head attention, it's crucial to understand self-attention. The self-attention mechanism allows each position in the sequence to attend to all other positions, providing a weighted sum of these positions. This helps the model capture dependencies regardless of their distance in the sequence.

Self-Attention Process

1. **Input Vectors:** Let's say we have a sequence of input vectors X=[x1,x2,…,xn], where each x*i* is a d-dimensional vector.
2. **Query, Key, Value Matrices:** For each input vector xi, we compute three vectors: the query qi, key ki and value vi. These are obtained using learned linear transformation:
   Q`= XWQ, K` = XWK, V` = XWV

Where WQ, WK, and WV are learned weight matrices.

1. **Scaled Dot-Product Attention:** The attention scores are computed using the query and key vectors. The scores determine how much focus each position should place on other positions:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

Here, dk is the dimension of the key vectors, and the division by √dk is a scaling factor to prevent the dot products from growing too large.

2. Multi-Head Attention

Multi-head attention extends the self-attention mechanism by allowing the model to jointly attend to information from different representation subspaces at different positions. Instead of performing a single attention function, the model performs hhh attention functions (heads) in parallel.

Multi-Head Attention Process

- **Multiple Linear Projections:** The input vectors are linearly projected h times to create multiple sets of queries, keys, and values:

$$\mathbf{Q}_i = \mathbf{X}\mathbf{W}_i^Q, \quad \mathbf{K}_i = \mathbf{X}\mathbf{W}_i^K, \quad \mathbf{V}_i = \mathbf{X}\mathbf{W}_i^V \quad \text{for} \quad i = 1, 2, ..., h$$

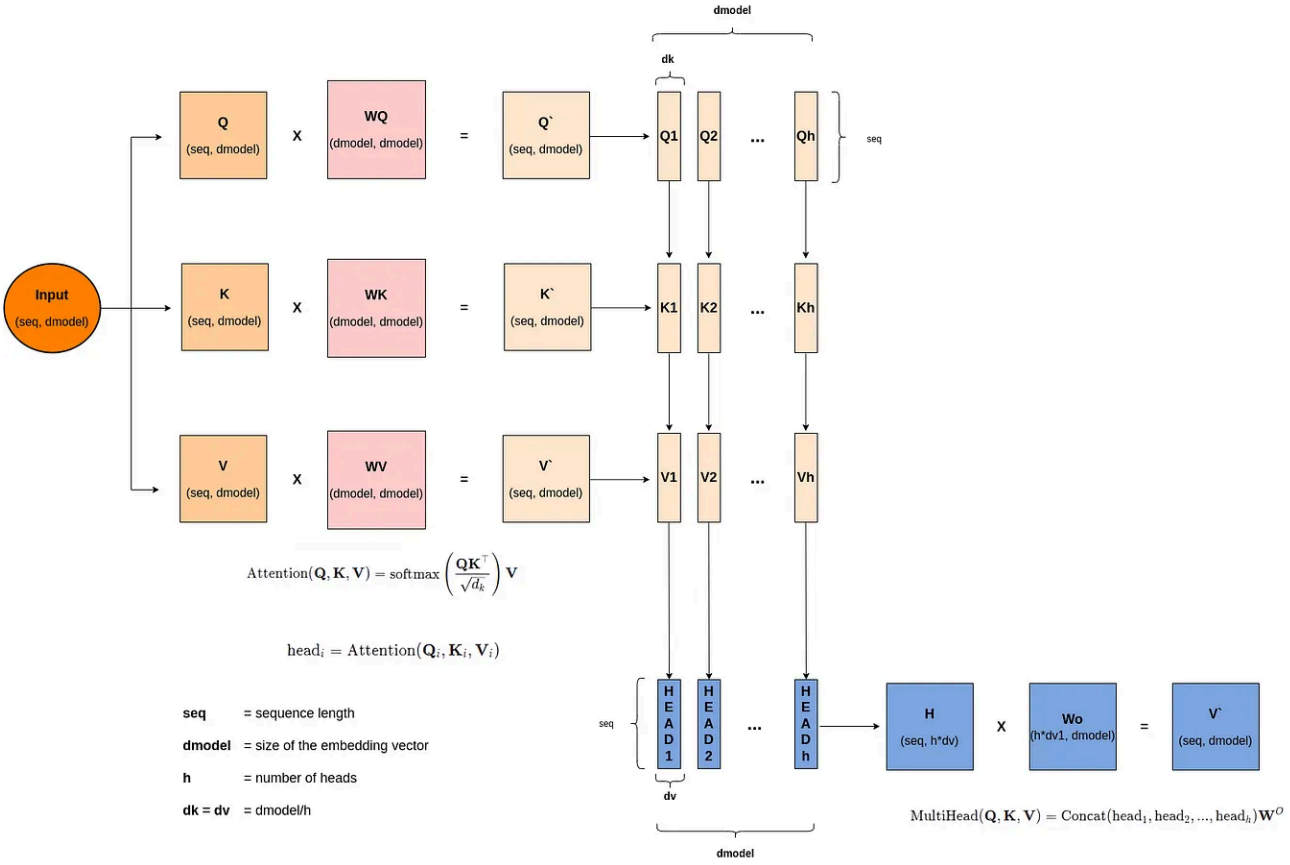
- **Scaled Dot-Product Attention for Each Head:** Each set of queries, keys, and values is used to compute attention scores and outputs

$$\text{head}_i = \text{Attention}(\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i)$$

- **Concatenation of Heads:** The outputs of the hhh attention heads are concatenated:

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, \text{head}_2, ..., \text{head}_h)\mathbf{W}^O$$

where WO is a learned weight matrix used to project the concatenated outputs back to the desired dimension.



Source : Author

Benefits of Multi-Head Attention

- **Enhanced Representational Capacity:** By attending to different parts of the sequence simultaneously, the model can capture more complex relationships.
- **Mitigation of Information Loss:** Multiple heads ensure that even if some heads fail to capture certain dependencies, others might succeed, providing a more robust representation.

```
class MultiHeadAttentionBlock(nn.Module):

    def __init__(self, d_model: int, h: int, dropout: float) -> None:
        super().__init__()
        self.d_model = d_model # Embedding vector size
        self.h = h # Number of heads
        # Make sure d_model is divisible by h
        assert d_model % h == 0, "d_model is not divisible by h"

        self.d_k = d_model // h # Dimension of vector seen by each head
        self.w_q = nn.Linear(d_model, d_model, bias=False) # Wq
        self.w_k = nn.Linear(d_model, d_model, bias=False) # Wk
        self.w_v = nn.Linear(d_model, d_model, bias=False) # Wv
        self.w_o = nn.Linear(d_model, d_model, bias=False) # Wo
```

```
        self.dropout = nn.Dropout(dropout)

    @staticmethod
    def attention(query, key, value, mask, dropout: nn.Dropout):
        d_k = query.shape[-1]
        # Just apply the formula from the paper
        # (batch, h, seq_len, d_k) --> (batch, h, seq_len, seq_len)
        attention_scores = (query @ key.transpose(-2, -1)) / math.sqrt(d_k)
        if mask is not None:
            # Write a very low value (indicating -inf) to the positions where m
            attention_scores.masked_fill_(mask == 0, -1e9)
        attention_scores = attention_scores.softmax(dim=-1) # (batch, h, seq_le
        if dropout is not None:
            attention_scores = dropout(attention_scores)
        # (batch, h, seq_len, seq_len) --> (batch, h, seq_len, d_k)
        # return attention scores which can be used for visualization
        return (attention_scores @ value), attention_scores

    def forward(self, q, k, v, mask):
        query = self.w_q(q) # (batch, seq_len, d_model) --> (batch, seq_len, d_
        key = self.w_k(k) # (batch, seq_len, d_model) --> (batch, seq_len, d_mc
        value = self.w_v(v) # (batch, seq_len, d_model) --> (batch, seq_len, d_

        # (batch, seq_len, d_model) --> (batch, seq_len, h, d_k) --> (batch, h,
        query = query.view(query.shape[0], query.shape[1], self.h, self.d_k).tr
        key = key.view(key.shape[0], key.shape[1], self.h, self.d_k).transpose(
        value = value.view(value.shape[0], value.shape[1], self.h, self.d_k).tr

        # Calculate attention
        x, self.attention_scores = MultiHeadAttentionBlock.attention(query, key

        # Combine all the heads together
        # (batch, h, seq_len, d_k) --> (batch, seq_len, h, d_k) --> (batch, sec
        x = x.transpose(1, 2).contiguous().view(x.shape[0], -1, self.h * self.d

        # Multiply by Wo
        # (batch, seq_len, d_model) --> (batch, seq_len, d_model)
        return self.w_o(x)
```

Let's understand the code above and one by one breakdown why we are performing these steps **Linear Projections, Reshape and Transpose, Scaled Dot-Product Attention, Reshape and Concatenate and Final Linear Layer.**

`__init__` method:

- Initializes with `d_model` (dimension of the model), `h` (number of attention heads), and `dropout` (dropout rate).

- Ensures `d_model` is divisible by `h`.
- Defines linear layers for query (`w_q`), key (`w_k`), value (`w_v`), and output (`w_o`).

`attention` static method:

- Computes scaled dot-product attention.
- Applies mask if provided, computes softmax over attention scores, and optionally applies dropout.

`forward` method:

1. **Linear Projections:**

- `query = self.w_q(q) key = self.w_k(k) value = self.w_v(v)`
- These lines apply linear transformations to the input tensors q, k, and v using learnable weight matrices Wq, Wk, and Wv. This is the first step in computing the query, key, and value vectors for attention.

2. Reshape and Transpose for Multi-Head Attention:

- `query = query.view(query.shape[0], query.shape[1], self.h, self.d_k).transpose(1, 2) key = key.view(key.shape[0], key.shape[1], self.h, self.d_k).transpose(1, 2) value = value.view(value.shape[0], value.shape[1], self.h, self.d_k).transpose(1, 2)`
- Here, we reshape and transpose the query, key, and value tensors to prepare them for multi-head attention.
- `query.view(query.shape[0], query.shape[1], self.h, self.d_k)` reshapes the tensor to have a separate dimension for the number of heads (`h`). The dimensions become `(Batch, Seq_len, h, d_k)`.
- `.transpose(1, 2)` swaps the sequence length and head dimensions, resulting in a tensor of shape `(Batch, h, Seq_len, d_k)`.
- This allows each attention head to work on its portion of the `d_model` dimensions separately.

3. Scaled Dot-Product Attention:

- `x, self.attention_scores = MultiHeadAttentionBlock.attention(query, key, value, mask, self.dropout)`
- This line calls the static method `attention` to compute the attention scores and weighted sum of values.
- The `attention` method performs scaled dot-product attention using the query, key, value tensors, and the mask.

4. Reshape and Concatenate Heads:

- `x = x.transpose(1, 2).contiguous().view(x.shape[0], -1, self.h * self.d_k)`
- `x.transpose(1, 2)` changes the shape from `(Batch, h, Seq_len, d_k)` back to `(Batch, Seq_len, h, d_k)`.
- `.contiguous().view(x.shape[0], -1, self.h * self.d_k)` reshapes the tensor to combine the head dimensions, resulting in a shape of `(Batch, Seq_len, d_model)`, where `d_model = h * d_k`.
- `contiguous()` ensures that the tensor's memory layout is continuous, which can be important for efficiency in certain operations.

5. Final Linear Layer:

- `return self.w_o(x)`
- This line applies the final linear transformation using the weight matrix WoW_oWo.
- The output has the shape `(Batch, Seq_len, d_model)`, which is the same as the input shape.

What is Mask?

> *In the provided `MultiHeadAttentionBlock` class, the type of mask used in the `attention` method can be either a padding mask or a look-ahead mask (causal mask), depending on how it is applied when calling the `forward` method. The mask is passed as an argument to the `forward` method, and then it is used in the `attention` static method.*

- **Padding Mask**: is used to ensure that padding tokens in the input sequence do not influence the attention mechanism. Padding tokens are added to sequences to ensure they all have the same length within a batch, but these tokens do not contain meaningful information and should be ignored by the model during training and inference.Ensures that padding tokens do not influence the attention mechanism by setting their attention scores to a very large negative value, effectively ignoring them. **Used In**: Both encoder and decoder blocks of the Transformer model.

> ***Encoder****: Prevents the attention mechanism from considering padding tokens in the input sequences.*
>
> ***Decoder****:*
>
> ***Self-Attention****: Prevents padding tokens in the target sequences from affecting the attention mechanism.*
>
> ***Encoder-Decoder Attention****: Prevents padding tokens in the input sequences from influencing the attention to the encoder's outputs.*

Example of Padding Mask:

Consider a batch of sequences with different lengths that have been padded:python

```
import torch
```

```
# Example input sequences (batch_size=2, seq_len=5)
input_sequences = [
    [1, 2, 3, 0, 0],  # Sequence 1, padded with 0s
    [4, 5, 6, 7, 8]   # Sequence 2, no padding
]

# Convert to tensor
input_tensor = torch.tensor(input_sequences)

# Create a padding mask where 1 indicates a valid position and 0
indicates padding
padding_mask = (input_tensor != 0).unsqueeze(1).unsqueeze(2)

print("Input Tensor:")
print(input_tensor)
```

```
print("Padding Mask:")
print(padding_mask)
```

```
#output
Input Tensor:
tensor([[1, 2, 3, 0, 0],
        [4, 5, 6, 7, 8]])

Padding Mask:
tensor([[[[ True,  True,  True, False, False]]],
        [[[ True,  True,  True,  True,  True]]]])
```

- **Look-Ahead Mask (Causal Mask)**: Its purpose is to ensure that during training and inference, each position in the output sequence can only attend to positions before it and the current position, but not to any future positions. This is crucial in autoregressive tasks, such as language modeling and text generation, where the model predicts the next token in a sequence based on the previous tokens. **Used In**: Decoder block of the Transformer model.

Example of Look-Ahead Mask in a Decoder:

```
import torch
# Create a look-ahead mask
seq_len = 5
look_ahead_mask = torch.triu(torch.ones((seq_len, seq_len)), diagonal=1).bool()
print("Look-Ahead Mask:")
print(look_ahead_mask)
```

```
Look-Ahead Mask:
tensor([[False,  True,  True,  True,  True],
        [False, False,  True,  True,  True],
        [False, False, False,  True,  True],
        [False, False, False, False,  True],
        [False, False, False, False, False]])
```

In this mask:

- The `True` values represent positions that should not be attended to (future positions).
- The `False` values represent positions that can be attended to (current and past positions).

ResidualConnection Class

Purpose of Residual Connection: Residual connections, or skip connections, are used to help with the training of deep neural networks by allowing gradients to flow more easily through the network. They essentially "skip" one or more layers, directly connecting the input of a layer to the output of another layer further down the stack. This helps mitigate the problem of vanishing gradients and allows for more effective training of very deep networks.

Typical Structure in Transformer:

In a Transformer model, residual connections are used together with layer normalization. After the multi-head attention or feed-forward layers, the original input to the layer is added back to the output (after applying the layer), and then layer normalization is applied.

```
class ResidualConnection(nn.Module):
    def __init__(self, dropout: float) -> None:
        super().__init__()
        self.dropout = nn.Dropout(dropout)
        self.norm = LayerNormalization()

    def forward(self, x, sublayer):
        return x + self.dropout(sublayer(self.norm(x)))
```

`__init__` method:

- Initializes with `dropout` (dropout rate).
- Defines dropout and layer normalization.

`forward` method:

- Applies layer normalization, dropout, and adds the input `x` (residual connection).

EncoderBlock Class

Now we will create the encoder block which will contain one multi-head attention, two Add and Norm & one feed forward layer.

```
class EncoderBlock(nn.Module):
    def __init__(self, self_attention_block: MultiHeadAttentionBlock, feed_forw
        super().__init__()
        self.self_attention_block = self_attention_block
        self.feed_forward_block = feed_forward_block
        self.residual_connections = nn.ModuleList([ResidualConnection(dropout)

    def forward(self, x, src_mask):
        x = self.residual_connections[0](x, lambda x: self.self_attention_block
        x = self.residual_connections[1](x, self.feed_forward_block)
        return x
```

`__init__` method:

- Initializes with `self_attention_block` (an instance of `MultiHeadAttentionBlock`), `feed_forward_block` (an instance of `FeedForwardBlock`), and `dropout`.
- Creates two residual connections for self-attention and feed-forward blocks.

`forward` method:

- Applies self-attention block with residual connection. So here we are applying a self-attention block with a residual connection. Here, the self-attention block takes the input xxx as the query (q), key (k), and value (v), which is why it's called "self-attention." Essentially, the input x is used to attend to itself. This means that each word in a sentence interacts with every other word in the same sentence.
- In the decoder, the attention mechanism works differently because of cross-attention. In cross-attention, the query comes from the decoder, while the key and value come from the encoder. This allows the decoder to focus on relevant parts of the input sequence produced by the encoder, rather than only attending to its own output.
- Thus, self-attention enables intra-sentence interactions, whereas cross-attention facilitates interactions between the encoder's output and the decoder's input.

- Applies feed-forward block with residual connection.

```
class Encoder(nn.Module):
    def __init__(self, layers: nn.ModuleList) -> None:
        super().__init__()
        self.layers = layers
        self.norm = LayerNormalization()

    def forward(self, x, mask):
        for layer in self.layers:
            x = layer(x, mask)
        return self.norm(x)
```

DecoderBlock Class

The `DecoderBlock` class represents a single block of the Transformer decoder. Each decoder block contains a self-attention mechanism, a cross-attention mechanism (attending to the encoder's output), and a feed-forward network, all surrounded by residual connections and layer normalization.

```
class DecoderBlock(nn.Module):
    def __init__(self, self_attention_block: MultiHeadAttentionBlock, cross_att
        super().__init__()
        self.self_attention_block = self_attention_block
        self.cross_attention_block = cross_attention_block
        self.feed_forward_block = feed_forward_block
        self.residual_connections = nn.ModuleList([ResidualConnection(dropout)

    def forward(self, x, encoder_output, src_mask, tgt_mask):
        x = self.residual_connections[0](x, lambda x: self.self_attention_block
        x = self.residual_connections[1](x, lambda x: self.cross_attention_bloc
        x = self.residual_connections[2](x, self.feed_forward_block)
        return x
```

`__init__` method:

- **Parameters:**

- `self_attention_block`: Instance of `MultiHeadAttentionBlock` for self-attention.

- `cross_attention_block`: Instance of `MultiHeadAttentionBlock` for cross-attention.
- `feed_forward_block`: Instance of `FeedForwardBlock`.
- `dropout`: Dropout rate for regularization.

`self.residual_connections`: A list of three `ResidualConnection` instances for each sub-layer in the decoder block.

`forward` Method:

Parameters:

- `x`: The input tensor (decoder input).
- `encoder_output`: The output from the encoder.
- `src_mask`: Source mask to prevent the model from attending to padding tokens in the source input.
- `tgt_mask`: Target mask to prevent the model from attending to future tokens in the target sequence (look-ahead mask).

Decoder Class

The `Decoder` class consists of a stack of decoder blocks. It sequentially applies these blocks to the input, followed by a layer normalization step.

```
class Decoder(nn.Module):

    def __init__(self, features: int, layers: nn.ModuleList) -> None:
        super().__init__()
        self.layers = layers
        self.norm = LayerNormalization(features)

    def forward(self, x, encoder_output, src_mask, tgt_mask):
        for layer in self.layers:
            x = layer(x, encoder_output, src_mask, tgt_mask)
        return self.norm(x)
```

`__init__` Method

Parameters:

- `features`: The number of features (dimensionality) of the input and output.
- `layers`: A list of `DecoderBlock` instances that make up the decoder.

Attributes:

- `self.layers`: Stores the list of decoder blocks.
- `self.norm`: Applies layer normalization to the output of the decoder stack.

`forward` Method

Parameters:

- `x`: The input tensor (decoder input).
- `encoder_output`: The output from the encoder.
- `src_mask`: Source mask to prevent the model from attending to padding tokens in the source input.
- `tgt_mask`: Target mask to prevent the model from attending to future tokens in the target sequence (look-ahead mask).

ProjectionLayer Class

The `ProjectionLayer` class is used to convert the high-dimensional vectors (output of the decoder) into logits over the vocabulary. This projection is typically the last layer in the decoder of a transformer model.

`__init__` Method

Parameters:

- `d_model`: The dimensionality of the model's internal representations (i.e., the hidden size).
- `vocab_size`: The size of the vocabulary, indicating the number of possible output tokens.

Attributes:

- `self.proj` : An instance of `nn.Linear` that maps from `d_model` dimensions to `vocab_size` dimensions.

This layer takes in a tensor of shape `(batch_size, seq_len, d_model)` and applies a linear transformation to each position in the sequence, resulting in a tensor of shape `(batch_size, seq_len, vocab_size)`. This tensor can then be used to compute the probability distribution over the vocabulary for each position in the sequence.

`forward` Method

Parameters:

- `x` : A tensor of shape `(batch_size, seq_len, d_model)`, where `batch_size` is the number of sequences in a batch, `seq_len` is the length of each sequence, and `d_model` is the dimensionality of the model's hidden states.

`self.proj(x)` : Applies the linear transformation to project the `d_model`-dimensional vectors to `vocab_size`-dimensional logits for each position in the sequence.

```
class ProjectionLayer(nn.Module):

    def __init__(self, d_model, vocab_size) -> None:
        super().__init__()
        self.proj = nn.Linear(d_model, vocab_size)

    def forward(self, x) -> None:
        # (batch, seq_len, d_model) --> (batch, seq_len, vocab_size)
        return self.proj(x)
```

The `ProjectionLayer` is a crucial part of the transformer model as it translates the model's internal high-dimensional representations into a format suitable for predicting tokens in the vocabulary. This class is typically used at the end of the decoder to produce logits that can be converted into probabilities over the vocabulary using a softmax function.

Transformer Class

The `Transformer` class encapsulates the entire transformer model, integrating both the encoder and decoder components along with embedding layers and positional encodings. Here's a detailed explanation of each part of the class:

```
class Transformer(nn.Module):

    def __init__(self, encoder: Encoder, decoder: Decoder, src_embed: InputEmbe
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed
        self.tgt_embed = tgt_embed
        self.src_pos = src_pos
        self.tgt_pos = tgt_pos
        self.projection_layer = projection_layer

    def encode(self, src, src_mask):
        # (batch, seq_len, d_model)
        src = self.src_embed(src)
        src = self.src_pos(src)
        return self.encoder(src, src_mask)

    def decode(self, encoder_output: torch.Tensor, src_mask: torch.Tensor, tgt:
        # (batch, seq_len, d_model)
        tgt = self.tgt_embed(tgt)
        tgt = self.tgt_pos(tgt)
        return self.decoder(tgt, encoder_output, src_mask, tgt_mask)

    def project(self, x):
        # (batch, seq_len, vocab_size)
        return self.projection_layer(x)
```

Parameters:

- `encoder`: An instance of the `Encoder` class, responsible for encoding the source sequence.
- `decoder`: An instance of the `Decoder` class, responsible for decoding the encoded representation and generating the output sequence.
- `src_embed`: An instance of `InputEmbeddings` for embedding the source sequence tokens.
- `tgt_embed`: An instance of `InputEmbeddings` for embedding the target sequence tokens.
- `src_pos`: An instance of `PositionalEncoding` to add positional information to the source embeddings.

- `tgt_pos`: An instance of `PositionalEncoding` to add positional information to the target embeddings.
- `projection_layer`: An instance of `ProjectionLayer` to project the decoder outputs to the vocabulary size.

`encode` Method

Parameters:

- `src`: The source sequence tensor of shape `(batch_size, seq_len)`.
- `src_mask`: The source mask tensor to handle padding tokens in the source sequence.

Execution:

- Apply source embeddings to the input tensor `src`.
- Add positional encodings to the embedded source tensor.
- Pass the resulting tensor through the encoder along with the source mask.

Output: Returns the encoded representation of the source sequence of shape `(batch_size, seq_len, d_model)`.

`decode` Method

Parameters:

- `encoder_output`: The encoded representation of the source sequence from the encoder.
- `src_mask`: The source mask tensor used during encoding.
- `tgt`: The target sequence tensor of shape `(batch_size, seq_len)`.
- `tgt_mask`: The target mask tensor to handle padding and future tokens in the target sequence.

Execution:

1. Apply target embeddings to the input tensor `tgt`.
2. Add positional encodings to the embedded target tensor.
3. Pass the resulting tensor through the decoder along with the encoder output and the respective masks.

Output: Returns the decoded representation of the target sequence of shape `(batch_size, seq_len, d_model)`.

`project` Method

Parameters:

- `x`: The output tensor from the decoder of shape `(batch_size, seq_len, d_model)`.

Execution:

- Apply the projection layer to map the `d_model` dimensional output to `vocab_size` dimensional logits.

Output: Returns the logits tensor of shape `(batch_size, seq_len, vocab_size)`.

Build Transformer Method

`build_transformer` constructs a full Transformer model by putting together its various components, such as embedding layers, positional encoding, encoder and decoder blocks, and a final projection layer.

```
def build_transformer(src_vocab_size: int, tgt_vocab_size: int, src_seq_len: ir
    # Create the embedding layers
    src_embed = InputEmbeddings(d_model, src_vocab_size)
    tgt_embed = InputEmbeddings(d_model, tgt_vocab_size)

    # Create the positional encoding layers
    src_pos = PositionalEncoding(d_model, src_seq_len, dropout)
    tgt_pos = PositionalEncoding(d_model, tgt_seq_len, dropout)

    # Create the encoder blocks
    encoder_blocks = []
    for _ in range(N):
        encoder_self_attention_block = MultiHeadAttentionBlock(d_model, h, drop
        feed_forward_block = FeedForwardBlock(d_model, d_ff, dropout)
        encoder_block = EncoderBlock(d_model, encoder_self_attention_block, fee
        encoder_blocks.append(encoder_block)
```

```
    # Create the decoder blocks
    decoder_blocks = []
    for _ in range(N):
        decoder_self_attention_block = MultiHeadAttentionBlock(d_model, h, drop
        decoder_cross_attention_block = MultiHeadAttentionBlock(d_model, h, drc
        feed_forward_block = FeedForwardBlock(d_model, d_ff, dropout)
        decoder_block = DecoderBlock(d_model, decoder_self_attention_block, dec
        decoder_blocks.append(decoder_block)

    # Create the encoder and decoder
    encoder = Encoder(d_model, nn.ModuleList(encoder_blocks))
    decoder = Decoder(d_model, nn.ModuleList(decoder_blocks))

    # Create the projection layer
    projection_layer = ProjectionLayer(d_model, tgt_vocab_size)

    # Create the transformer
    transformer = Transformer(encoder, decoder, src_embed, tgt_embed, src_pos,

    # Initialize the parameters
    for p in transformer.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)

    return transformer
```

Here is the URL for the Complete code notebook — Link

Thanks for your patience hope it helps, in next blog I will share how we can use it for any one usecase.

References

Chefer, H., Gur, S., & Wolf, L. (2021). Generic attention-model explainability for interpreting bi-modal and encoder-decoder Transformers. In *arXiv [cs.CV]*. http://arxiv.org/abs/2103.15679

Jain, S., & Wallace, B. C. (2019). Attention is not Explanation. In *arXiv [cs.CL]*. http://arxiv.org/abs/1902.10186

Lynn-Evans, S. (2018, September 27). *How to code The Transformer in Pytorch*. Towards Data Science. https://towardsdatascience.com/how-to-code-the-transformer-in-pytorch-24db27c8f9ec

Jamil, U. [@umarjamilai]. (2023, May 25). *Coding a Transformer from scratch on PyTorch, with full explanation, training and inference*. Youtube. https://www.youtube.com/watch?v=ISNdQcPhsts

Vaswani, A., Shazeer, N., Parmar, N., & Uszkoreit, J. (n.d.). *Attention is all you need*. Arxiv.org. Retrieved June 15, 2024, from http://arxiv.org/abs/1706.03762

(N.d.). Datacamp.com. Retrieved June 15, 2024, from https://www.datacamp.com/tutorial/building-a-transformer-with-py-torch

Transformers Scratch NLP Architecture Implementation

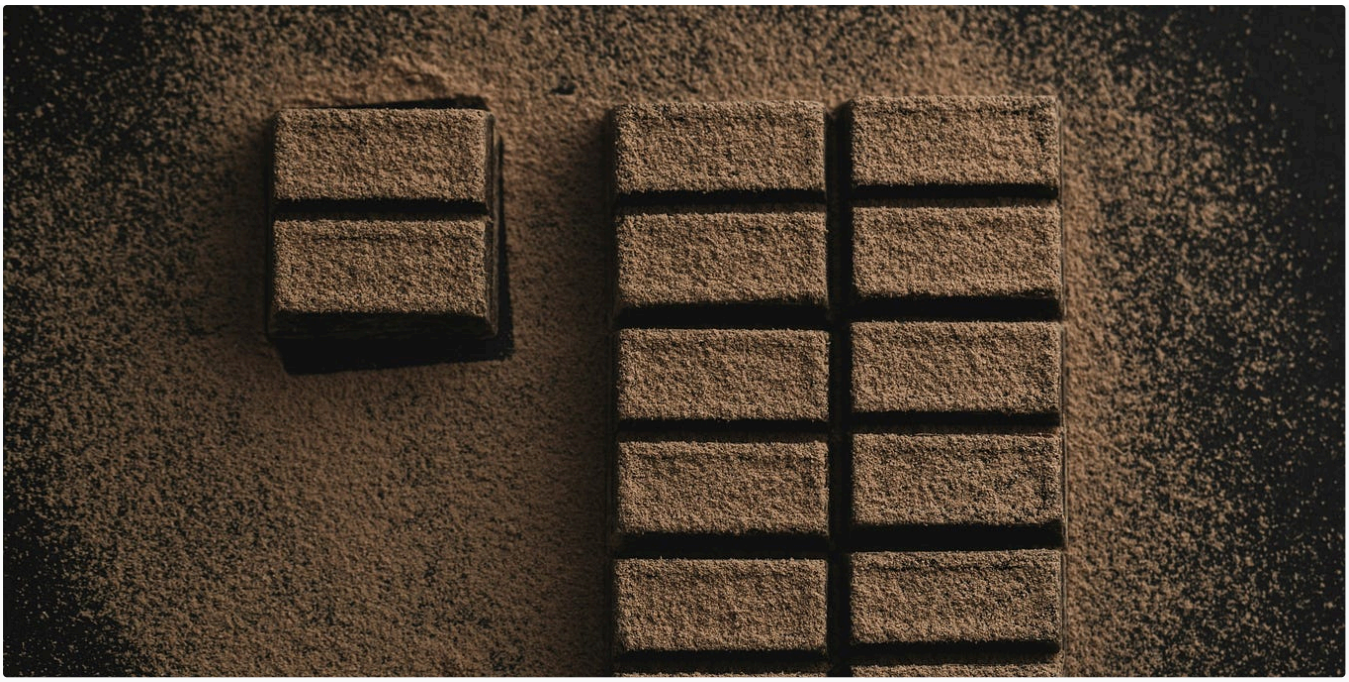
Follow

Written by BavalpreetSinghh

161 Followers · 40 Following

Consultant Data Scientist and AI ML Engineer @ CloudCosmos | Ex Data Scientist at Tatras Data | Reseacher @ Humber College | Ex Consultant @ SL2

More from BavalpreetSinghh

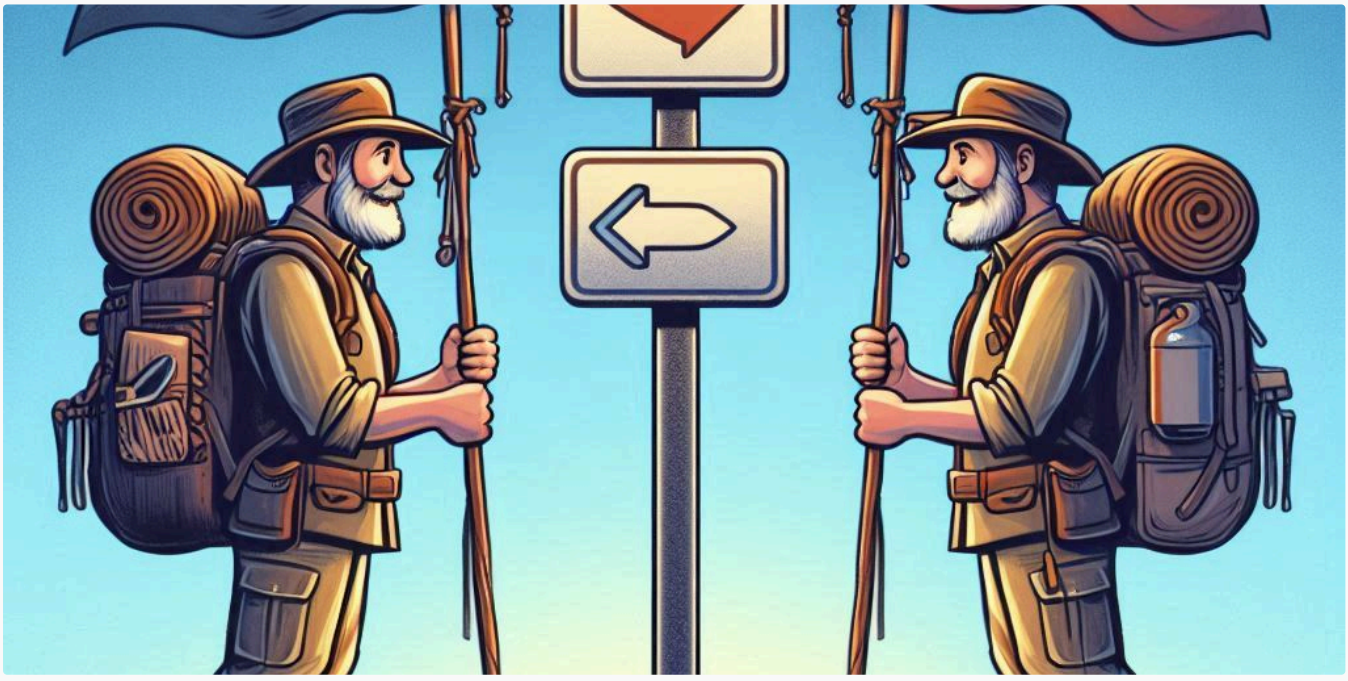


BavalpreetSinghh

LlamaIndex: Chunking Strategies for Large Language Models. Part—1

In the previous blog, we delved into the intricacies of constructing and querying document indexes with Llama-Index. It's important to...

Mar 3 253 2

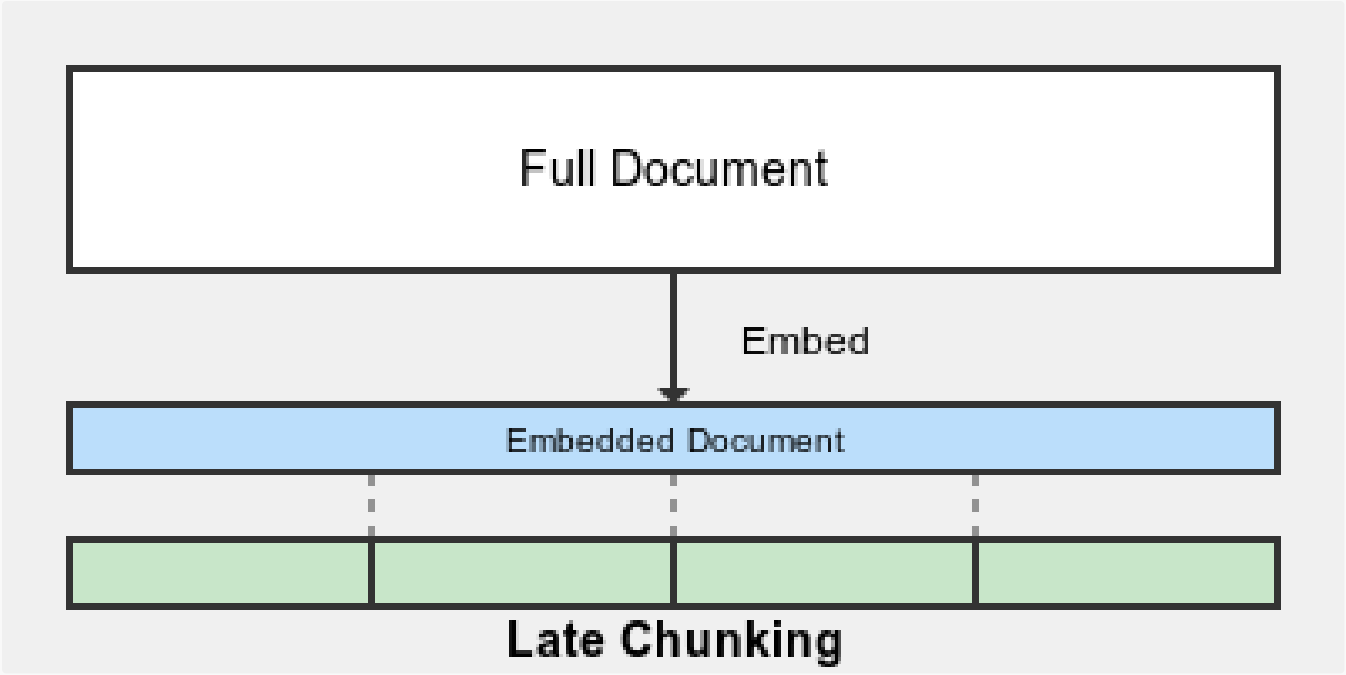


BavalpreetSinghh

RLHF(PPO) vs DPO

Although large-scale unsupervisly trained language models (LLMs) gain broad world knowledge and some reasoning abilities, precisely...

Jun 9 182

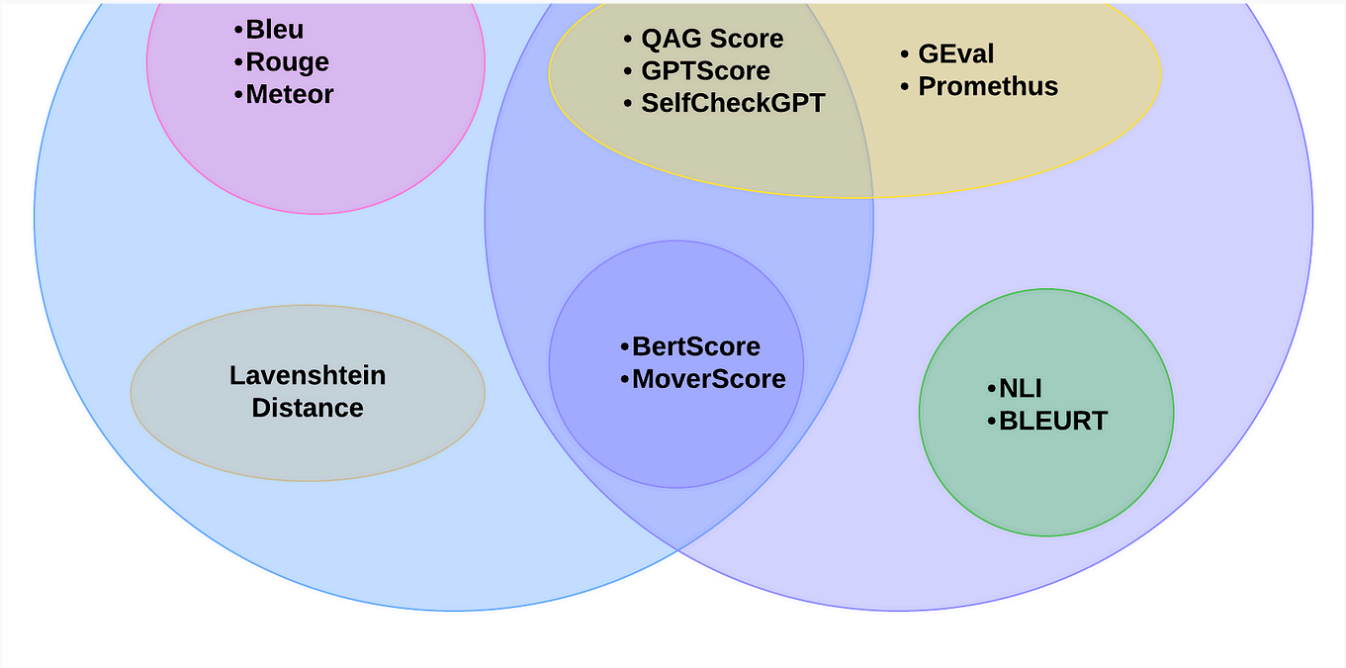


BavalpreetSinghh

Late Chunking: Embedding First Chunk Later—Long-Context Retrieval in RAG Applications

In the rapidly evolving landscape of large-scale Retrieval-Augmented Generation (RAG) applications, developers face a complex set of...

Sep 21 372 1



BavalpreetSinghh

RAG and LLM Evaluation Metrics

LlamaIndex RAG/LLM Evaluators

Aug 9 286

See all from BavalpreetSinghh

Recommended from Medium

noplaxochia

Multihead attention from scratch

Pytorch without using nn.MultiheadAttention.

Jul 10 11

In Stackademic by Abdur Rahman

Python is No More The King of Data Science

5 Reasons Why Python is Losing Its Crown

Oct 23 6.7K 29

Lists

Natural Language Processing

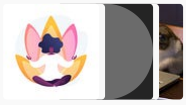
1815 stories · 1428 saves

The New Chatbots: ChatGPT, Bard, and Beyond

12 stories · 507 saves

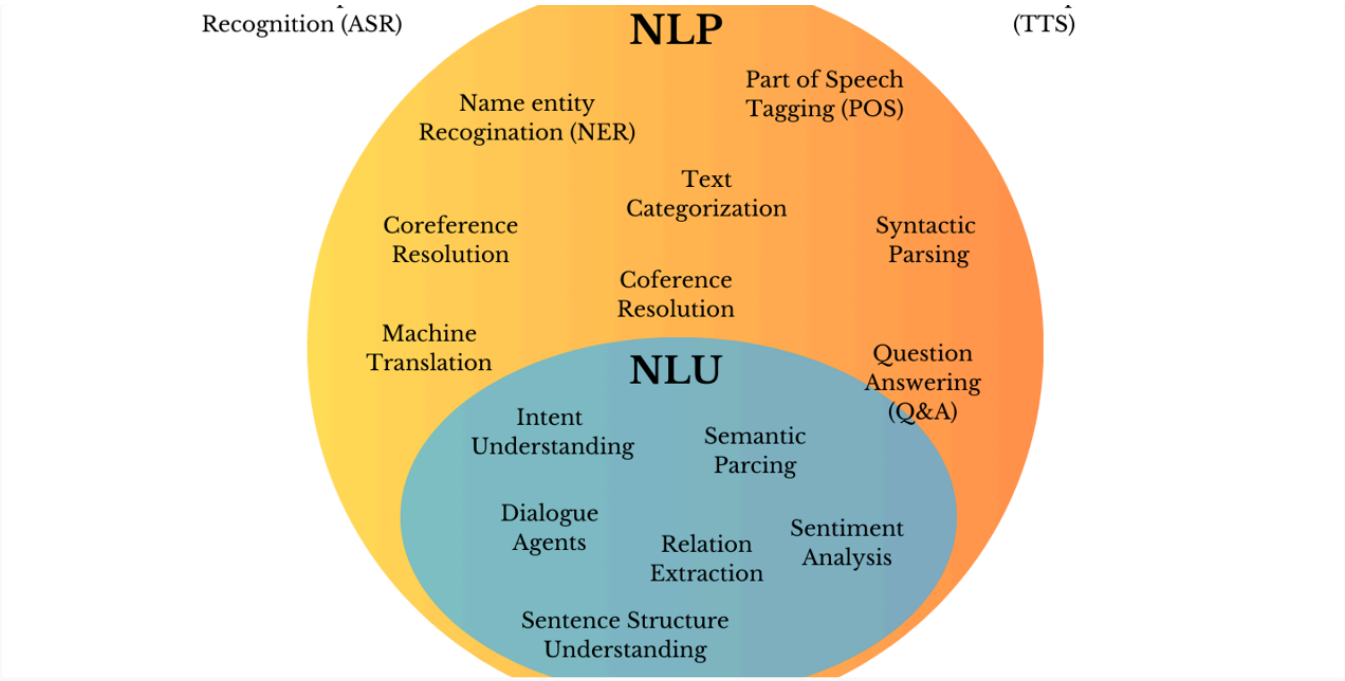
data science and AI

40 stories · 284 saves



Stories to Help You Grow as a Software Developer

19 stories · 1474 saves

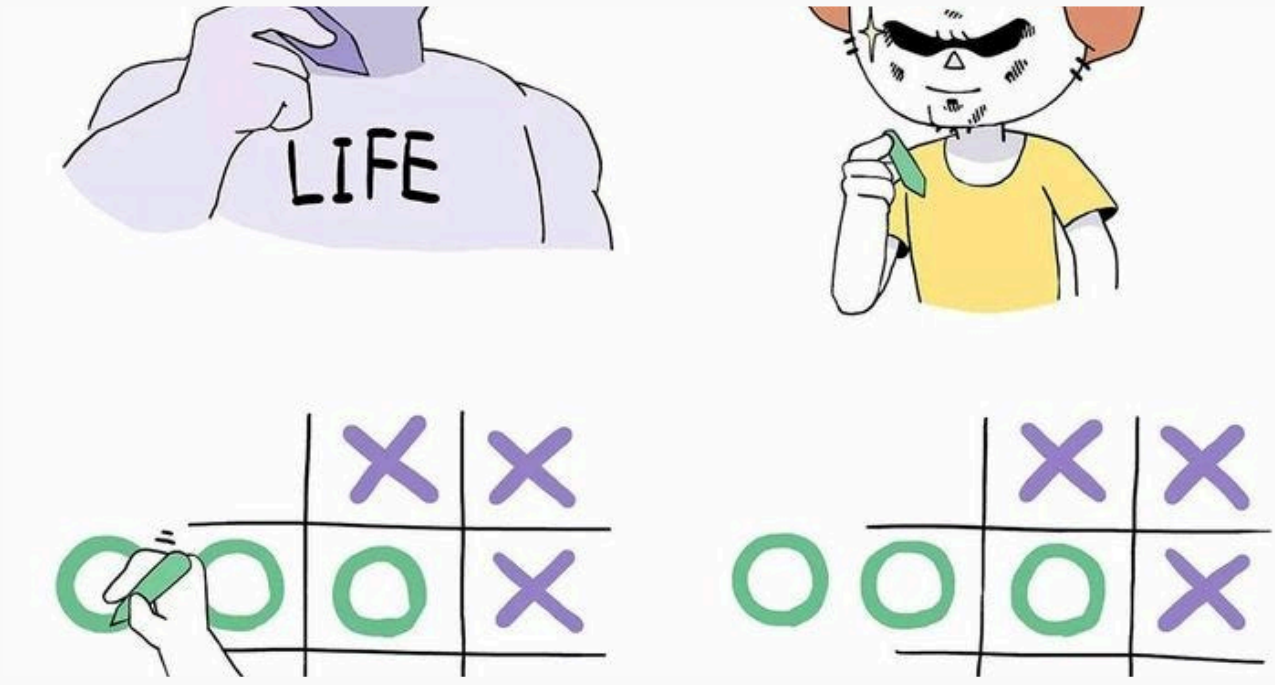


Vipra Singh

LLM Architectures Explained: NLP Fundamentals (Part 1)

Deep Dive into the architecture & building of real-world applications leveraging NLP Models starting from RNN to the Transformers.

Aug 15 1.92K 13

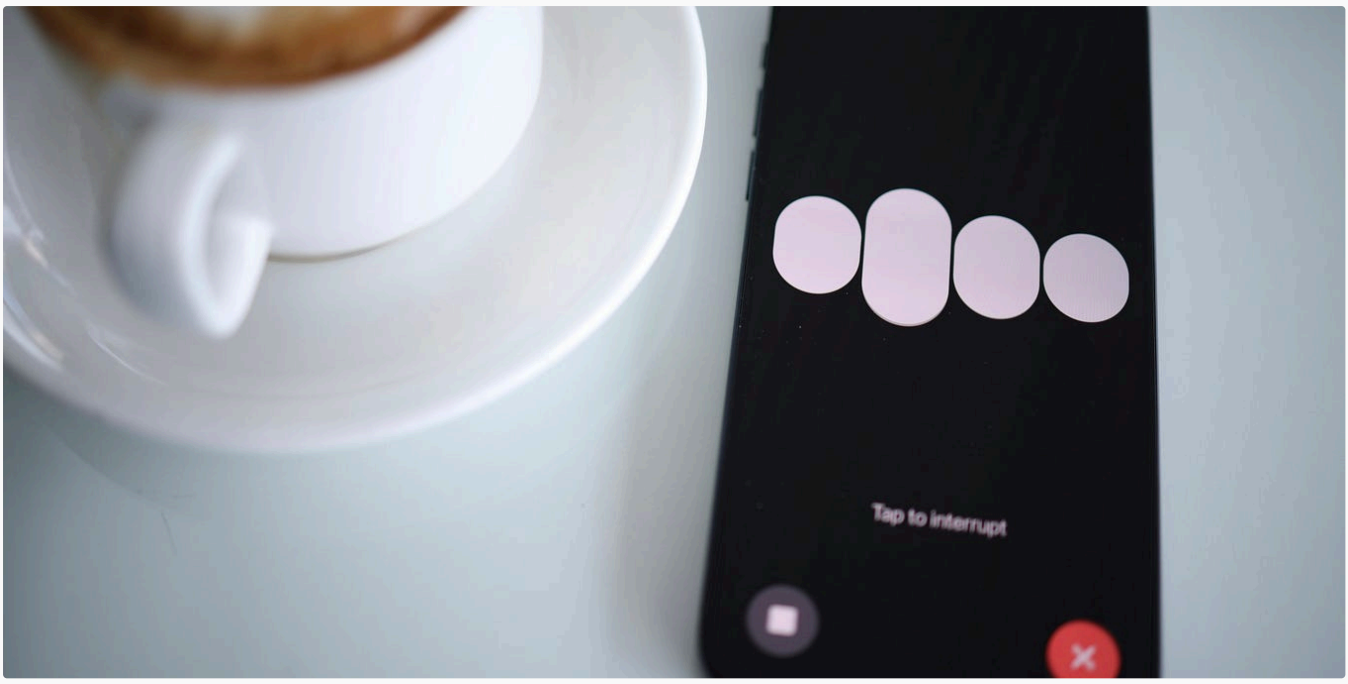


Amit Yadav

Computer Vision Project Ideas With Code

Not a Medium member? Read the full story by clicking here.

Aug 7 103

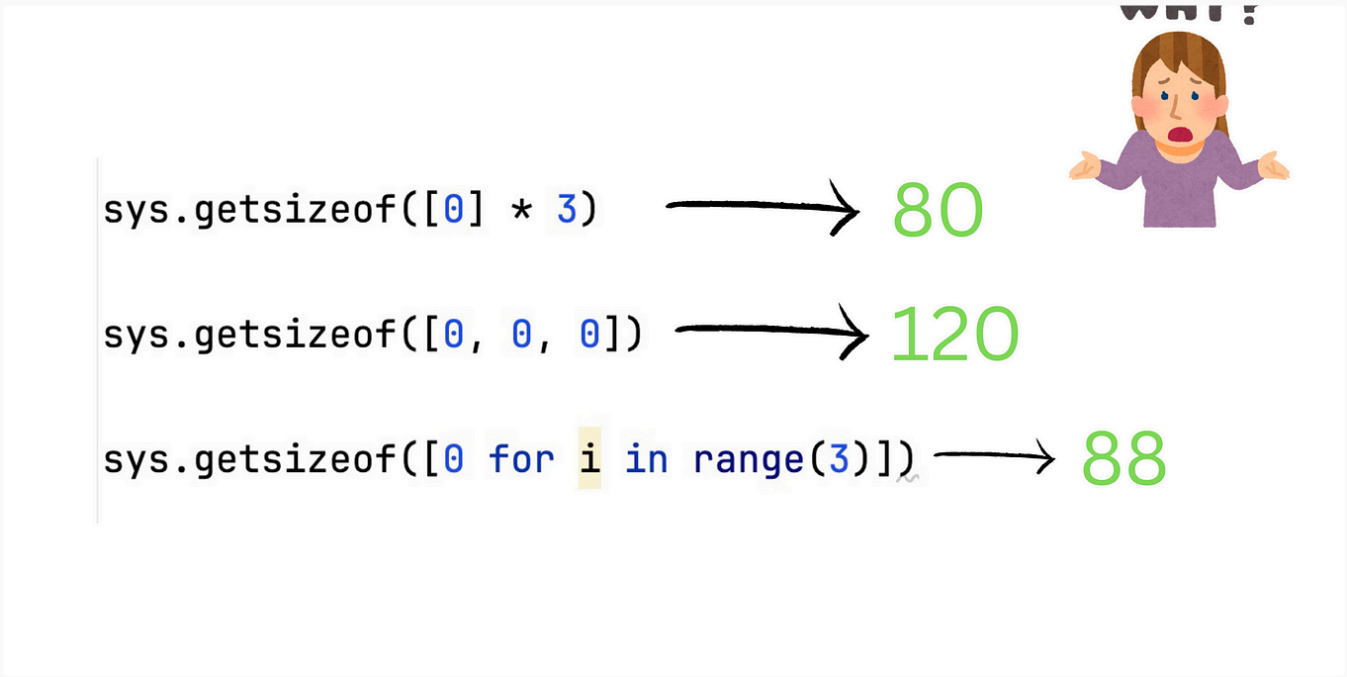


Don Lim

What is 1-bit LLM?—Bitnet.cpp may eliminate GPUs

Microsoft introduces Bitnet.cpp, a lightweight AI model that can run efficiently on a portable device.

Oct 19 415 4



In Programming Domain by Shuai Li

Most Developers Failed with this Senior-Level Python Interview Question

What is the difference between [0] * 3 and [0, 0, 0]?

Aug 9 3.6K 51

See more recommendations