Hi everyone, welcome to the part 3 of the course “Implementing Transformer from Scratch with PyTorch”.

---

In part 2, we have discussed all building blocks of transformer, the model proposed in 2017 by Vaswani et al. in the paper “Attention is all you need”. This model has revolutionized not only the field of Natural Language Processing but also the whole world of artificial intelligence.

In this part we will talk about the implementation part of this model.

---

**What will we learn in this part?**

We'll begin with an introduction to the training process implementation, providing an overview of the fundamental steps and best practices.

Following this, we'll learn how to implement a transformer model as a Python package, ensuring modularity and reusability in your projects.

We'll then cover the procedures for calling both the training and inference processes, highlighting key functions and their interactions.

In addition to this, we'll explore how to experiment with these implementations using Jupyter notebooks, offering a practical, hands-on approach to refining and testing your implementations.

---

Before we start coding, let me introduce the How-to Guides for Implementation.

First, I will show you the source code structure to help you see the big picture. I will explain which file contains which function or class.

After that, I will walk you through coding all the Python functions and classes step by step.

Throughout the implementation process, I will present notebook files with concrete examples of the functions and classes.

Finally, I will show you the expected outputs to validate that the developed functions and classes work properly as expected.

Combining these four elements will allow us to gain a good understanding of the implementation.

---

Let’s look at our plan for the transformer implementation. There are two main parts.

In the first part, we will implement the training process of the transformer model. This process involves taking source sentences in one language and target sentences in another language as inputs. We will develop classes and functions that allow the model to be trained through multiple iterations and weight updates. After that, we will run the training scripts to obtain the trained model.

In the second part, once the model is trained, we will implement and test the inference scripts. In this step, we input a sentence in one language, such as English, into the trained model, and it outputs the translated sentence in another language, such as French.

---

Let’s first handle the implementation of the training part.

This process enables the model to learn features from data. For example, in the case of machine translation task, it learns from both source and target sentences. It then adjusts its weight values to enable the model to generate sentences that closely resemble the target sentence, as closely as possible.

--

For training process, I propose 2 main steps:

First, we load the configuration. It is a Json file containing a Python Dictionary, where the keys are name of different parameters serving different steps in training and inference. The values can be string or numerical values.

Then, we go into the training engine, which includes 3 main steps: pre-processing data, create and initiate transformer model, and training loop. We will dive deeper into each of the sub-steps.

---

For preprocessing data, we will prepare the functions like get\_dataset, get\_tokenizer, create encoder mask, padding mask, causal mask, decoder mask and the class like DataPreprocessor. We also use the class DataLoader from PyTorch framework.

---

For building a transformer model, we need to build the classes: InputEmbedding, PositionalEncoding, Encoder, Decoder, and Projection. For Encoder and Decoder, we need to build EncoderLayer and DecoderLayer respectively. The layers are derived from the atomic classes: MultiHeadAttention, FeedForward, ResidualConnection and LayerNormalization.

---

For training loop, on one hand we prepare loss function, optimizer, etc, on the other hand we need to prepare evaluation functions for validation purpose during training.

---

Now putting it all together to see the whole picture. It seems quite complicated to successfully run the training process of transformer but by breaking the complex task into small modular functions and classes, we can easily move on and turn the complex thing into multiple simple sub-components.

Well now you have a high level and structured overview of what will be implemented, let’s jump into the exciting parts.

---

Well we now look at the source code structure. I will build transformer source code as a Python package.

For this, we create a folder: transformer. Inside this folder, there are 3 main groups:

* \_\_init\_\_.py file, this is a default file to let Python know that its parent folder is a Python package
* Then, there is a group of Python files for Python functions
* And another group of Python files for Python classes

---

For the Data Preparation steps, we will be implementing:

* The DataPreprocessor class in the Python file data.
* Functions like load\_config, get\_dataset, get\_tokenizer, and preprocess\_data in the Python file utils.
* Functions to create the encoder mask, padding mask, causal mask, and decoder mask in the Python file mask.

---

For Transformer Building Blocks, we will be implementing the following classes in the Python file layer: InputEmbedding, PositionalEncoding, MultiHeadAttention, FeedForward, LayerNorm, RisudualConnection, and Projection

---

For creating Transformer model, we will be implementing EncoderLayer and Encoder classes in the Python file encoder, DecoderLayer and Decoder in the Python file decoder, Transformer class in the Python file Transformer and then the function create\_transformer\_model in the Python file utils

---

For Training process, we will be implementing the function infer\_training, evaluation and train in the Python file engine

---

For Inference process, we will be implementing the function inference in the Python file engine.

---

**Get Config**

First, we talk about the load\_config function:

This Python function is used to load the configuration file in JSON format.

1. First, we import the **json** library. The function load\_config takes one argument config\_file\_path, which is the path to the configuration file.
2. It uses Python's built-in ‘open’ function to open the file in read mode ("r"). The ‘with’ statement is used for clean resource management, automatically closing the file after the nested coding block is executed.
3. The **json.load** function is used to parse the JSON file and convert it into a Python dictionary. This dictionary is stored in the **config** variable.
4. The function then returns the **config** variable. This variable will be used in the rest of the program to access configuration values.

---

**Get Config - Notebook**

Now, to test the load\_config function, we are going to play with the dedicated notebook file,

First of all, We import the load\_config function. Because the notebook file stays in the notebooks folder, and the transformer package stays in the parent directory of the notebook file, these two lines of code allow to append the parent directory to the system path to access the transformer modules.

Then, we add the input argument as the path to the config.json file, then the output is assigned to the variable **config** as a Python dictionary type.

Afterwards, we can use it to access to the key and value in the configuration data.

For example, we can print some keys and values like: source language, target language, batch size, number of epochs like this.

Finally, we can see what results we can obtain at the end.

Great! We can now load and access to the configuration data, to use it for the whole program later.

---

**Get Dataset**

Now we are going to talk about the get\_dataset function.

This function is used to load a dataset for a machine translation task.

1. Firstly, we import load\_dataset function from the datasets library. This is the library developed by Hugging Face, allowing us to access and download the dataset that is available in the Hugging Face hub.

In our case, we will use the news\_commentary dataset in the Hugging Face hub.

Now let’s see how to write this function.

1. The function takes one argument, that is the config variable.
2. The function first retrieves the name of the dataset, the source and target languages. It also retrieves the dataset split option (such as "train", "validation", or "test") from the configuration.
3. It then constructs the language pair string by concatenating the source and target languages with a hyphen.
4. Afterwards, it uses the load\_dataset function that takes the name of the dataset, the language pair, the split as arguments. This function returns a DatasetDict, which is a dictionary-like object.
5. Finally, the function returns the loaded dataset.

**Get Dataset - Notebook**

Now, to test the get\_dataset function, we are going to play with the dedicated notebook file,

1. Firstly, the script starts by importing necessary modules and functions.
2. Then, the load\_config function is used to load the configuration settings from a JSON file.
3. Afterwards, we call the get\_dataset function to load the dataset based on the configuration settings.
4. Finally, we can print out the first two items from the 'translation' field of the dataset. We can see that this field typically contains the source and target sentences for the machine translation task.

In summary, this notebook file allows us to verify that the function works as expected.

**Get Tokenizer**

Now we are going to talk about the get\_tokenizer function.

This function is used to load or train a tokenizer for a specific language.

Let’s see how we can implement this function;

1. Firstly, we load the necessary package and functions
2. The function takes three arguments as inputs: config, dataset and language
3. The function first retrieves the name of the tokenizer from the configuration and constructs the path to the tokenizer file by appending the language to the tokenizer name.
4. If the tokenizer file exists, it loads the tokenizer from the file using the Tokenizer.from\_file method.
5. If the tokenizer file does not exist, we create a new tokenizer with a word-level model and we set the unknown token as ([UNK]). We also set the pre-tokenizer to split the text into words based on whitespace.

we then create a trainer with a minimum frequency of 2, meaning that a word must appear at least twice in the text corpus to be considered as part of the vocabulary. Words that appear only once will be ignored. We set special tokens for padding ([PAD]), unknown words ([UNK]), start of sentence ([SOS]), and end of sentence ([EOS]).

The tokenizer is then trained on the dataset using the train\_from\_iterator method. The dataset is assumed to be a dictionary with a key "translation" that contains a list of dictionaries, each with a key for each language and the corresponding text as the value.

1. After training the tokenizer, it saves the tokenizer to a file so that it can be loaded quickly in the future.
2. Finally, the function returns the tokenizer. This tokenizer can be used to convert text to tokens and vice versa.

**Tokenizer Notebook**

Now, to examine the get\_tokenizer function, we are going to play with the dedicated notebook file.

1. First, we import necessary modules and functions. The two lines of code allow to append the parent directory of the notebook file to the system path to access the transformer module in different directory.
2. Then, we load the configuration settings from a JSON file using the load\_config function.
3. After that, the get\_dataset function is used to load the dataset based on the configuration settings.
4. Next, we use the get\_tokenizer function to load the source and target tokenizers, corresponding to the source and target language string arguments, respectively.
5. Finally, we print out various properties of the tokenizers, such as the vocabulary size, how they encode text and decode token,

In summary, this notebook file allows us to verify that the function works as expected.

**Create masks**

Now we are going to see how to implement the mask functions, including create encoder mask, padding mask, causal mask and decoder mask. Let’s dive deep into each of them one by one.

**Create encoder mask**

Let’s see how we can implement the function: create encoder mask

This function is used to create a mask for the encoder inputs. The mask is used to inform the model which tokens should be attended to (usually the non-padding tokens) and which ones should be ignored (usually the padding tokens).

* The operation **encoder\_input\_ids != pad\_token\_id** creates a boolean tensor where each element is True if the corresponding element in encoder\_input\_ids is not equal to pad\_token\_id (i.e., it's not a padding token), and False otherwise.
* **.unsqueeze(0).unsqueeze(0)** : These two unsqueeze methods add two extra dimensions at the beginning of the tensor. This is done to match the expected input shape for the attention mask in transformer models, which is typically (batch\_size, 1, 1, sequence\_length).
* **.int()** : The int() method converts the boolean tensor to an integer tensor, because the attention mask in transformer models is usually of type int.

**Playground – Create encoder mask**

Now let’s test our function create\_encoder\_mask

1. First of all, we import necessary packages and functions
2. Then, we create a tensor en\_input\_ids representing the input sequence to the encoder. Each number in the tensor corresponds to the ID of a token in the sequence. The pad\_id tensor is also created, representing the ID of the padding token.
3. The create\_encoder\_mask function from the transformer.mask module is called with en\_input\_ids and pad\_id as arguments.
4. The script then prints the input IDs, their shape, the encoder mask, and its shape.
5. Finally, we can check that the output result is the one that we are expected.

**Create padding mask**

Let’s see how we can implement the function: create\_padding\_mask. This function creates a padding mask for the decoder inputs. It works similarly to create\_encoder\_mask, but add only one extra dimension to the mask.

**Playground – Create padding mask**

Now, to examine the create\_padding\_mask function, we are going to play with this dedicated notebook file.

1. We first import the PyTorch library and all functions from the transformer.mask module.
2. Then we initiate the two tensors decoder\_input\_ids and padding\_id
3. After that we call the function create\_padding\_mask on the two input tensors
4. Finaly we can print out some results like the input IDs, their shape, the padding mask, and its shape.

Well the results justify our function works as expected.

**Create causal mask**

Well, now we are going to see how to implement the function create\_causal\_mask.

torch.ones((1, seq\_len, seq\_len)): This creates a 3D tensor filled with ones, with shape (1, seq\_len, seq\_len). This tensor represents a square matrix of size seq\_len x seq\_len with all elements being 1.

* torch.triu(...): This function returns the upper triangular part of the matrix, setting elements below the diagonal to zero. The diagonal=1 argument shifts the diagonal up by one position, so the diagonal elements themselves are also set to zero.
* .type(torch.int): This changes the data type of the tensor to integer.
* ... == 0: This creates a boolean mask where each element is True if the corresponding element in the tensor is 0, and False otherwise.

To summarize, this function creates a causal mask, which is used to ensure that each position in the decoder can only attend to earlier positions in the sequence. It takes the sequence length as an argument and creates a square mask of that size, with Trues below and on the diagonal and Falses above the diagonal.

**Playground – Create causal mask**

Now, to test the create\_causal\_mask function, we are going to play with this dedicated notebook file.

1. We first import the Torch library and all functions from the transformer.mask module.
2. Then we initiate the two tensors decoder\_input\_ids, padding\_id and the parameter sequence length
3. After that we call the function create\_causal\_mask on the three input tensors
4. Finaly we can print out some results like the decoder input IDs, their shape, the causal mask, and its shape.

Well the results allow us to verify that our function works as expected.

**Create decoder mask**

Well, now we are going to see how to implement the function create\_decoder\_mask:

This function creates a mask for the decoder inputs by combining a padding mask and a causal mask.

It takes the decoder input IDs, the ID of the padding token, and the sequence length as arguments.

The resulting mask contains a 1 at each position where the corresponding input ID is not a padding token and the position is not ahead in the sequence, and a 0 otherwise.

**Playground – Create decoder mask**

Now, to test the create\_decoder\_mask function, we are going to play with this dedicated notebook file.

1. We first import the Torch library and all functions from the transformer.mask module.
2. Then we initiate the two tensors decoder\_input\_ids, padding\_id and the parameter sequence length
3. After that we call the function create\_decoder\_mask on these input arguments
4. Finaly we can print out some results like the decoder input IDs tensor, the causal mask, and their shapes, like this.

Well the results allow us to verify that our function works as expected.

**Data Preprocessor Class**

Next, I am going to show you how to implement DataPreprocessor class.

This class inherits from PyTorch's Dataset class. The purpose of this class is to preprocess data for a transformer model.

1. First of all, the \_\_init\_\_ method initializes the class with the dataset, source and target tokenizers, source and target languages, and the maximum sequence length. It also retrieves the token IDs for the special tokens [SOS], [EOS], and [PAD].
2. The \_\_len\_\_ method returns the number of items in the dataset.
3. The \_\_getitem\_\_ method preprocesses the data for the item at the given index. It performs the following steps:
   * Retrieves the source and target texts from the dataset.
   * Next, it tokenizes the source and target texts.
   * Then, it calculates the number of padding tokens needed for the encoder and decoder inputs.
     1. To calculate the number of padding tokens for the encoder input, it subtracts the length of the encoder tokens (len(encoder\_token\_ids)) and 2 from the desired sequence length (self.seq\_len). In this case, 2 represents two special tokens: a start-of-sequence token and an end-of-sequence token.
     2. To calculate the number of padding tokens for the decoder input, it subtracts the length of the decoder tokens (len(decoder\_token\_ids)) and 1 from the desired sequence length (self.seq\_len). In this case, 1 represents one special token: start-of-sequence token.
   * Afterwards, we create the encoder inputs, decoder inputs and the decoder target.
     1. The encoder inputs start with the [SOS] token, followed by the tokenized text, the [EOS] token (for the encoder input), and the necessary number of [PAD] tokens.
     2. The decoder inputs start with the [SOS] token, followed by the tokenized text, and the necessary number of [PAD] tokens. For decoder inputs, we do not use EOS token
     3. The decoder target start with the tokenized text followed by the [EOS] token and the necessary number of [PAD] tokens.
   * Creates the masks for the encoder and decoder inputs using the create\_encoder\_mask and create\_decoder\_mask functions. These masks are used in the transformer model to ignore the padding tokens during the attention calculations.
   * Returns a dictionary containing the preprocessed data, including source and target texts, encoder input ids, decoder input ids, decoder target ids, encoder mask and decoder mask.

In summary, this class is used to preprocess data for a transformer model. It will be used for a PyTorch DataLoader later to provides batches of preprocessed data to the transformer model during training.

**Data Preprocessing**

Well now I am going to show you how to implement the preprocessing\_data function. This function is used to prepare the data for training, validation, and testing of a Transformer model in a machine translation task.

This function takes one argument ‘config’ as input.

1. **Get Dataset**: First of all, it calls the get\_dataset function to retrieve the raw dataset based on the configuration.
2. **Tokenization**: Then, it calls the get\_tokenizer function to get the source and target tokenizers. These tokenizers are used to convert the source and target texts into sequences of token IDs.
3. **Split Dataset**: After that, it splits the raw dataset into training, validation, and testing datasets. The split is 70% for training, 20% for validation, and 10% for testing.
4. **Data Preprocessing**: Afterwards, we use the DataPreprocessor class (explained in the previous slide) to convert the raw datasets into a format that can be directly used by the Transformer model. This includes tokenization, addition of special tokens, and creation of masks.
5. **Data Loading**: Next, we create PyTorch DataLoader objects for the training, validation, and testing datasets. These DataLoader objects are used to iterate over the datasets in batches. The batch size for the training dataset is specified in the configuration, while the batch size for the validation and testing datasets is set to 1. The data is shuffled to ensure that the model is not biased by the order of the examples.
6. **Return**: Finally, it returns the DataLoader objects for the training, validation, and testing datasets, as well as the source and target tokenizers.

In conclusion, this function is typically called at the beginning of the training script to prepare the data for the training loop.

---

Now we are going to implement building blocks of transformer under the form of Python class, in which we will talk about Input Embedding, Positional Encoding, Multi-Head-Attention, Feed-Forward, Layer Normalization, Residual Connection, and Projection layer

---

**Input Embedding**

Well now I am going to show you how to implement This InputEmbedding class. This class is specifically used for embedding the input tokens.

* It inherits from nn.Module, which is the base class for all neural network modules in PyTorch.
* The \_\_init\_\_ method initializes the class. It takes two parameters: vocab\_size and d\_model. vocab\_size is the size of the vocabulary, and d\_model is the dimension of the embeddings.

Inside this method, an embedding layer is created using nn.Embedding with vocab\_size and d\_model as parameters. The d\_model is also stored as an instance variable.

* The forward method defines the forward pass of the module. It takes an input x which is expected to be a 2D tensor with shape [batch\_size, seq\_len], where batch\_size is the number of sequences in a batch and seq\_len is the length of each sequence. The method returns the embedded input, scaled by the square root of d\_model. The scaling is a part of the original Transformer model, which helps in stabilizing the gradients.

**Input Embedding – Notebook**

Now, to test the InputEmbedding class, we are going to play with this dedicated notebook file.

This class convert sequences of word indices into continuous vector representations (embeddings).

1. First of all, Import the necessary modules: PyTorch and the InputEmbedding class.
2. Then, we define the vocabulary size (vocab\_size) and the embedding dimension (d\_model). For example, the vocabulary size is 10 (i.e., there are 10 unique words in the vocabulary), and the embedding dimension is 5 (i.e., each word will be represented as a 5-dimensional vector).
3. Next, we create an instance of the InputEmbedding class, passing the vocabulary size and the embedding dimension as arguments.
4. In this example, we define a batch of 2 sequences, each sequence containing 4 words. The words are represented by their indices in the vocabulary. This is done using a PyTorch tensor.
5. Aftezwards, we pass the sequences through the InputEmbedding instance. This converts the word indices into their corresponding embeddings and scales them by the square root of d\_model.
6. Finally, we can print out the resulting tensor and its shape. The output shape is [2, 4, 5], which corresponds to [batch\_size, seq\_len, d\_model]. This means there are 2 sequences in the batch, each sequence has 4 words, and each word is represented as a 5-dimensional vector.

Well the results allow us to verify that our class works as expected.

**Positional Encoding**

Well now I am going to show you how to implement The PositionalEncoding class. This class is used for adding positional information to the input embeddings. This is necessary because the Transformer does not have any inherent notion of the position of words in a sequence.

* First of all, this class inherits from nn.Module, which is the base class for all neural network modules in PyTorch.
* The \_\_init\_\_ method initializes the class. It takes three parameters: d\_model, seq\_len, and dropout. d\_model is the dimension of the embeddings, seq\_len is the maximum length of the sequences, and dropout is the dropout rate used for regularization. Inside this method, a dropout layer is created using nn.Dropout,
* The positional encoding matrix pe is calculated using a specific formula that uses sine and cosine functions of different frequencies. The formula is designed in such a way that the positional encoding for each position varies smoothly from -1 to 1. The positional encoding matrix pe is calculated and registered as a buffer. The positional encoding matrix pe is registered as a buffer because it's a part of the model's state that needs to be saved and loaded along with the model, but it's not a parameter that is learned during training. Positional encodings are a way to give the model some information about the relative positions of the tokens in the input sequence. They are typically calculated once and then used repeatedly, rather than being updated during training like the model's parameters.
* The forward method defines the forward pass of the module. It takes an input x which is expected to be a 3D tensor with shape [batch\_size, seq\_len, d\_model], representing the embedded input sequences. The method adds the positional encoding to the input embeddings (with the positional encoding adapted to the sequence length of x), applies dropout, and returns the result.
* The output of the forward method is a 3D tensor with shape [batch\_size, seq\_len, d\_model], representing the embedded input sequences with positional information added.

**Positional Encoding – Notebook**

Now, to test the PositionalEncoding class, we are going to play with this dedicated notebook file.

This use case demonstrates how to use the PositionalEncoding class to add positional information to the input embeddings.

1. Import the necessary modules: sys (to modify the Python path), PyTorch, and the PositionalEncoding class.
2. Define the sequence length (seq\_len), the embedding dimension (d\_model), and the dropout rate (dropout). In this case, the sequence length is 3 (i.e., each sequence has 3 words), the embedding dimension is 6 (i.e., each word is represented as a 6-dimensional vector), batch size is equal to 1 and the dropout rate is 0.1.
3. Create an instance of the PositionalEncoding class, passing the embedding dimension, the sequence length, and the dropout rate as arguments.
4. Then, we create a random input embedding tensor, having the shape of (batch\_size, sequence\_length, d\_model). In this case, a batch of 1 sequences, each sequence containing 3 words. The words are represented by their embeddings, which are 6-dimensional vectors. This is done using a PyTorch tensor filled with random numbers.
5. Next, we pass the sequences through the PositionalEncoding instance. This adds positional information to the input embeddings and applies dropout.
6. Finally, we can print out the resulting tensor and its shape.

Well the results allow us to verify that our class works as expected.

**Multi-Head Attention**

Well now I am going to show you how to implement The Multi-Head Attention class.

This class implements the Multi-Head Attention mechanism, a key component of the Transformer model.

1. \_\_init\_\_: This method initializes the class. It takes three parameters: d\_model (the feature length of a token), h (the number of attention heads), and dropout (the dropout rate). It creates four linear layers (w\_q, w\_k, w\_v, w\_o) and a dropout layer. The linear layers are used to transform the input sequences into queries, keys, and values, and to transform the output of the attention mechanism. The dropout layer is used to apply dropout to the attention scores.
2. attention: This static method calculates the attention scores and the output of the attention mechanism for a given query, key, and value.

* The function takes six parameters: query\_k, key\_k, value\_k, d\_k, and optionally mask and dropout. mask is an optional parameter used to prevent attention to certain positions. dropout is a dropout layer used for regularization.
* The attention score is calculated by taking the dot product of the query and key (with the key transposed), and dividing by the square root of d\_k. This results in a tensor of shape [batch\_size, h, seq\_len, seq\_len].
* If a mask is provided, the function applies the mask to the attention scores. This is done by replacing the scores at the masked positions with a very large negative number (-1e9 minus one times ten to the power of nine), which becomes close to zero after applying the softmax function.
* The softmax function is applied to the attention scores to convert them into probabilities. The probabilities are then passed through the dropout layer for regularization.
* Finally, the function returns the weighted sum of the value vectors (calculated by taking the dot product of the attention scores and the value vectors), and the attention scores. The weighted sum represents the output of the attention mechanism, and the attention scores can be used for visualization or other purposes.

The output of the function is a pair of tensors with shapes [batch\_size, h, seq\_len, d\_k] and [batch\_size, h, seq\_len, seq\_len], respectively.

1. forward: This method applies the multi-head attention mechanism to the input sequences. It takes four parameters: query, key, value (the input sequences), and mask (an optional mask). It first transforms the input sequences into queries, keys, and values using the linear layers. Then, it reshapes these tensors to have a separate dimension for the attention heads. After that, it calls the attention method to calculate the attention scores and the output of the attention mechanism. Finally, it concatenates the outputs of the different attention heads and transforms the result using the w\_o linear layer.

The output of the forward method is a tensor of the same shape as the input sequences, but the values have been updated to reflect the attention mechanism. This tensor can be used as input to the next layer of the Transformer model.

**Attention – Notebook**

**Usage of Attention without Mask**

Well to check the attention mechanism without using mask, we are going to play with this dedicated notebook file.

This code is using the attention function from the MultiHeadAttention class to compute the attention scores and the output of the attention mechanism for a given batch of sequences. In this case, no mask is applied, so all positions can attend to all other positions.

1. Import the necessary modules: sys (to modify the Python path), PyTorch, and the MultiHeadAttention class.
2. Define the batch size, number of attention heads, sequence length, and dimension of the key/query vectors. In this case, there is 1 sequence in the batch, the sequence has 4 words, there is 1 attention head, and the dimension of the key/query vectors is 64.
3. Create an input tensor filled with random numbers. This tensor represents the input sequence.
4. Create the query, key, and value vectors by cloning the input tensor. These vectors are used as input to the attention mechanism.
5. Call the attention function, passing the query, key, and value vectors, the dimension of the key vectors, and a dropout layer as arguments. No mask is passed, so all positions can attend to all other positions.
6. Print the shape of the output tensor, the shape of the attention scores tensor, and the attention scores themselves. The output tensor is a weighted sum of the value vectors, and the attention scores represent how much each word in the sequence should attend to each other word in the sequence.

The output of the attention function is a pair of tensors: the output of the attention mechanism and the attention scores. The shapes of these tensors are printed to verify that they are as expected (the output tensor should have the same shape as the input tensor, and the attention scores tensor should have a shape of [batch\_size, h, seq\_len, seq\_len]).

## Example of Attention with Mask

Well to check the attention mechanism using mask, we are going to play with this dedicated notebook file.

This code is similar to the previous one, but it introduces a mask to the attention mechanism. The mask is used to prevent certain positions from attending to other positions. This is useful in certain scenarios, such as when you want to prevent future positions from being used in the prediction of the current position (causal masking), or when you want to ignore padding positions.

1. Firstly, we import the necessary packages and functions
2. Then, we initialize the configuration setting values
3. Next, we initialize the input tensor and clone it to the query, key and value tensor
4. After that, we initialize the decoder mask. We have discussed about this before.
5. Then, we call the attention function, passing the query, key, and value vectors, the dimension of the key vectors, the mask, and a dropout layer as arguments.
6. Finally, we can print out the mask, the shape of the output tensor, the shape of the attention scores tensor, and the attention scores themselves.

The output of the attention function is a pair of tensors: the output of the attention mechanism and the attention scores. The shapes of these tensors are printed to verify that they are as expected. The mask is also printed for reference.

**Multi-Head Attention Notebook**

Well let’s check how the MultiHeadAttention class works, we are going to play with this dedicated notebook file.

This code is using the MultiHeadAttention class to apply the multi-head attention mechanism to a batch of sequences. Here's a step-by-step explanation:

1. Import the necessary modules
2. Then, we initialize the configuration values
3. Create an instance of the MultiHeadAttention class, passing the dimensions of the model, the number of attention heads, and the dropout rate as arguments.
4. Create a random tensor to represent a batch of sequences. This tensor has a shape of [batch\_size, seq\_len, d\_model], which corresponds to a batch of 1 sequences, each with 5 words, and each word represented by a 6-dimensional vector.
5. Create the query, key, and value tensors by cloning the input tensor. These tensors are used as input to the multi-head attention mechanism.
6. Pass the query, key, and value tensors through the multi-head attention layer by calling the forward method of the MultiHeadAttention instance. This applies the multi-head attention mechanism to the input sequences and returns a new tensor that represents the output of the attention mechanism. In this example, we use the default mask as None.
7. Print the shape of the output tensor. This tensor has the same shape as the input tensor ([batch\_size, seq\_len, d\_model]), but the values have been updated to reflect the attention mechanism. This tensor can be used as input to the next layer of the Transformer model.

**Feed Forward Layer**

Well now I am going to show you how to implement The FeedForward layer. This includes two linear transformations with a ReLU activation in between.

1. The FeedForward class is a subclass of nn.Module, which means it's a custom PyTorch module.
2. In the \_\_init\_\_ method, three layers are defined: linear1, linear2, and dropout. linear1 and linear2 are linear (fully connected) layers, and dropout is a dropout layer for regularization. The dimensions of the input and output of the linear layers (d\_model and d\_ff) are passed as arguments to the \_\_init\_\_ method.
3. The forward method is where the actual computation happens. It takes an input tensor x of shape [batch\_size, seq\_len, d\_model].
4. The input tensor is first passed through linear1, which applies a linear transformation to every element of the input.
5. The output of linear1 is then passed through a ReLU activation function. The ReLU function is applied element-wise, and it replaces negative values with zeros.
6. The output of the ReLU function is then passed through the dropout layer, which randomly sets some elements to zero with a probability equal to dropout. This helps prevent overfitting.
7. The output of the dropout layer is then passed through linear2, which applies another linear transformation.
8. The result is a tensor of the same shape as the input tensor, but with the values transformed by two linear layers and a ReLU activation function. This tensor is returned as the output of the layer.

**Feed Forward Layer Notebook**

Well to check the FeedForward  class, we are going to play with this dedicated notebook file.

This code is to apply a feed-forward neural network to a batch of sequences.

1. Import the necessary modules: sys (to modify the Python path), PyTorch, and the FeedForward class.
2. Then, we initialize the configuration setting values
3. Create an instance of the FeedForward class, passing the dimensions of the model, the dimensions of the feed-forward network, and the dropout rate as arguments.
4. Set a seed for PyTorch's random number generator to ensure that the same random numbers are generated every time the program is run.
5. Create a random tensor to represent a batch of sequences. This tensor has a shape of [batch\_size, seq\_len, d\_model], which corresponds to a batch of 1 sequence, with 4 words, and each word represented by a 5-dimensional vector.
6. Pass the tensor through the feed-forward network by calling the forward method of the FeedForward instance. This applies the feed-forward network to the input sequences and returns a new tensor that represents the output of the network.
7. Print the shape of the output tensor. This tensor has the same shape as the input tensor ([batch\_size, seq\_len, d\_model]), but the values have been transformed by the feed-forward network.
8. Print the output tensor. This tensor represents the output of the feed-forward network for each word in each sequence in the batch.

**Normalization Layer**

Well now I am going to show you how to implement the Layer Normalization layer.

Layer normalization is a type of normalization technique like Batch Normalization, but instead of normalizing the features across the batch, it normalizes the features across the feature dimension (i.e., for each individual sample in the batch).

1. The LayerNorm class is a subclass of nn.Module, which means it's a custom PyTorch module.
2. In the \_\_init\_\_ method, two parameters are defined: para\_mul and para\_bias. These are learnable parameters that will be optimized during training. They are initialized to ones and zeros, respectively. The eps (epsilon) value is a small constant used for numerical stability to avoid division by zero.
3. The forward method is where the actual normalization happens. It takes an input tensor x of shape [batch\_size, seq\_len, d\_model].
4. The mean and standard deviation of the input tensor are computed along the last dimension (i.e., the feature dimension). The keepdim=True argument ensures that the output tensors have the same number of dimensions as the input tensor.
5. The input tensor is then normalized by subtracting the mean and dividing by the standard deviation. The eps value is added to the denominator to prevent division by zero.
6. The normalized tensor is then scaled by para\_mul and shifted by para\_bias. These are learnable parameters that allow the layer to learn an optimal scale and mean for the outputs.
7. The result is a tensor of the same shape as the input tensor, but with normalized values. This tensor is returned as the output of the layer.

**Normalization Layer Notebook**

Well to check the LayerNorm  class, we are going to play with this dedicated notebook file.

This example is to apply layer normalization to a batch of sequences.

1. Import the necessary modules: sys (to modify the Python path), PyTorch, and the LayerNorm class.
2. Define the feature length (d\_model). In this case, each word is represented by a 3-dimensional vector.
3. Create an instance of the LayerNorm class, passing the feature length as an argument.
4. Set a seed for PyTorch's random number generator to ensure that the same random numbers are generated every time the program is run.
5. Create a random tensor to represent a batch of sequences. This tensor has a shape of [1, 3, 5], which corresponds to a batch of 1 sequence, with 3 words, and each word represented by a 5-dimensional vector.
6. Pass the tensor through the layer normalization layer by calling the forward method of the LayerNorm instance. This applies layer normalization to the input sequences and returns a new tensor that represents the normalized sequences.
7. Print the shape of the output tensor. This tensor has the same shape as the input tensor ([1, 4, 7]), but the values have been normalized.
8. Print the normalized sequences. These sequences have the same shape as the input sequences, but their values have been scaled and shifted to have a mean of 0 and a standard deviation of 1 (approximately, due to the learnable parameters and the epsilon value used for numerical stability).

**Residual Connection**

Well now I am going to show you how to implement the ResidualConnection layer.

The ResidualConnection layer applies a sublayer to the input, and then adds the original input back to the result. This "shortcut" helps to mitigate the problem of vanishing gradients in deep networks, allowing the model to learn more effectively.

1. The ResidualConnection class is a subclass of nn.Module, which means it's a custom PyTorch module.
2. In the \_\_init\_\_ method, a Layer Normalization (LayerNorm) and a Dropout layer are defined. The Layer Normalization normalizes the features across the feature dimension, and the Dropout layer randomly sets some elements to zero with a probability equal to dropout for regularization.
3. The forward method is where the actual computation happens. It takes an input tensor x of shape [batch\_size, seq\_len, d\_model] and a sublayer function sublayer.
4. The input tensor is first normalized using the Layer Normalization.
5. The normalized tensor is then passed through the sublayer function. This could be any function that takes a tensor as input and returns a tensor of the same shape, such as a feed-forward network or a self-attention mechanism.
6. The output of the sublayer function is then passed through the Dropout layer.
7. The original input tensor x is added to the output of the Dropout layer. This is the "residual connection" that gives the layer its name.
8. The result is a tensor of the same shape as the input tensor, but with the values transformed by the sublayer and then added back to the original input. This tensor is returned as the output of the layer.

**Residual Connection Notebook**

Well to check the ResidualConnection  class, we are going to play with this dedicated notebook file.

This notebook code is to apply a residual connection to a batch of sequences, with a feed-forward network as the sublayer.

1. Import the necessary modules: sys (to modify the Python path), PyTorch, and the FeedForward, LayerNorm, and ResidualConnection classes.
2. Define the dimensions of the model (d\_model), the dimensions of the feed-forward network (d\_ff), the dropout rate, the batch size, and the sequence length. In this case, each word is represented by a 5-dimensional vector, the feed-forward network has 2048 dimensions, the dropout rate is 0.1, the batch size is 1, and each sequence has 3 words.
3. Create an instance of the ResidualConnection class, passing the dimensions of the model and the dropout rate as arguments.
4. Create an instance of the FeedForward class, passing the dimensions of the model, the dimensions of the feed-forward network, and the dropout rate as arguments.
5. Set a seed for PyTorch's random number generator to ensure that the same random numbers are generated every time the program is run.
6. Create a random tensor to represent a batch of sequences. This tensor has a shape of [batch\_size, seq\_len, d\_model], which corresponds to a batch of 1 sequence, with 3 words, and each word represented by a 5-dimensional vector.
7. Pass the tensor through the residual connection by calling the forward method of the ResidualConnection instance, with the FeedForward instance as the sublayer. This applies the feed-forward network to the normalized input sequences, applies dropout, and then adds the original input back to the result.
8. Print the original input tensor, the shape of the output tensor, and the output tensor. The output tensor has the same shape as the input tensor ([batch\_size, seq\_len, d\_model]), but the values have been transformed by the feed-forward network and then added back to the original input.

**Projection Layer**

Well now I am going to show you how to implement the Projection layer.

The Projection layer is a simple fully connected neural network that is used to transform the output of the Transformer's decoder into prediction scores for each possible output token in the vocabulary.

1. The Projection class is a subclass of nn.Module, which means it's a custom PyTorch module.
2. In the \_\_init\_\_ method, a linear layer named projection is defined. This layer will map from the dimensionality of the model (d\_model) to the size of the output vocabulary (vocab\_size).
3. The forward method takes an input tensor x of shape [batch\_size, seq\_len, d\_model].
4. The input tensor is passed through the projection layer, which applies a linear transformation to every element of the input.
5. The result is a tensor of shape [batch\_size, seq\_len, vocab\_size]. This tensor represents the prediction scores for each possible output token in the vocabulary, for each position in each sequence in the batch.
6. This tensor is returned as the output of the layer. In a full Transformer model, these scores would typically be passed through a softmax function to produce a probability distribution over the vocabulary for each position in each sequence.

**Projection Notebook**

Well to check the Projection class, we are going to play with this dedicated notebook file.

This notebook code is to apply a projection to a batch of sequences.

1. Import the necessary modules: sys (to modify the Python path), PyTorch, and the Projection class.
2. Define the dimensions of the model (d\_model), the size of the output vocabulary (vocab\_size), the batch size, and the sequence length.
3. Create an instance of the Projection class, passing the dimensions of the model and the size of the output vocabulary as arguments.
4. Set a seed for PyTorch's random number generator to ensure that the same random numbers are generated every time the program is run.
5. Create a random tensor to represent a batch of sequences. This tensor has a shape of [batch\_size, seq\_len, d\_model], which corresponds to a batch of 1 sequence, with 3 words, and each word represented by a 5-dimensional vector.
6. Pass the tensor through the projection by calling the forward method of the Projection instance. This applies the projection to the input sequences and returns a new tensor that represents the prediction scores for each possible output token in the vocabulary.
7. Print the original input tensor, the shape of the output tensor, and the output tensor. The output tensor has a shape of [batch\_size, seq\_len, vocab\_size], which means it contains prediction scores for 7 possible output tokens for each of the 3 words in each of the 1 sequence in the batch.

---

Now I am going to implement the class and function to create transformers architecture, in which I will show you Encoder Layer, Encoder, Decoder Layer, Decoder, Transformer class, and finally the function create transformer model that combines all the building blocks into the final transformer model. Let’s deep dive in.

---

**Encoder Layer**

Well now I am going to show you how to implement the EncoderLayer  layer.

Each EncoderLayer consists of two main parts: a multi-head self-attention mechanism and a position-wise feed-forward network. These parts are connected by residual connections and followed by layer normalization.

1. The EncoderLayer class is a subclass of nn.Module, which means it's a custom PyTorch module.
2. In the \_\_init\_\_ method, the multi-head self-attention mechanism, the feed-forward network, and the residual connections are defined. The residual connections are a list of two ResidualConnection modules, which are used to add the input of each part to its output.
3. The forward method takes an input tensor x of shape [batch\_size, seq\_len, d\_model] and an optional source mask mask\_scr.
4. The input tensor is first passed through the first residual connection and the self-attention mechanism. The lambda function is used to apply the same input as the query, key, and value in the self-attention mechanism.
5. The output of the self-attention mechanism is then passed through the second residual connection and the feed-forward network. The feed-forward network applies the same transformation to each position in the sequence independently.
6. The output of the feed-forward network is returned as the output of the layer. This tensor has the same shape as the input tensor and can be passed to the next EncoderLayer in the sequence.

In a full Transformer model, several EncoderLayers are stacked on top of each other to form the encoder.

**Encoder**

Well now I am going to show you how to implement the Encoder layer.

The Encoder consists of a stack of identical layers, each of which is an instance of the EncoderLayer class.

1. The Encoder class is a subclass of nn.Module, which means it's a custom PyTorch module.
2. In the \_\_init\_\_ method, a list of EncoderLayer instances is created. The number of layers is specified by the num\_layers argument. A layer normalization module (LayerNorm) is also created. This is used to normalize the output of the last layer.
3. The forward method takes an input tensor x of shape [batch\_size, seq\_len, d\_model] and an optional source mask mask\_scr.
4. The input tensor is passed through each EncoderLayer in the list. Each layer applies a multi-head self-attention mechanism and a feed-forward network to the input, and adds the result to the original input (residual connection). The output of each layer is used as the input to the next layer.
5. After all layers have processed the input, the output of the last layer is normalized using the layer normalization module. This helps to stabilize the learning process and reduces the training time.
6. The normalized output is returned as the output of the encoder. This tensor has the same shape as the input tensor and can be passed to the decoder of the Transformer model.

In a full Transformer model, the Encoder processes the source sequences and produces a set of high-level features that represent the content of the sequences. These features are then used by the Decoder to generate the target sequences.

**Encoder Notebook**

Well to check the Encoder  class, we are going to play with this dedicated notebook file.

This notebook code is using an Encoder to process a batch of sequences.

1. Import the necessary modules and classes.
2. Define the configuration parameters: the dimensions of the model (d\_model), the dimensions of the feed-forward network (d\_ff), the number of heads in the multi-head attention mechanism (h), the batch size, the sequence length, the dropout ratio, and the number of layers in the encoder (num\_layers).
3. Create instances of the MultiHeadAttention and FeedForward classes. These are the main components of the EncoderLayer.
4. Create an instance of the EncoderLayer class, passing the dimensions of the model, the multi-head attention mechanism, the feed-forward network, and the dropout ratio as arguments.
5. Create an instance of the Encoder class, passing the dimensions of the model, the EncoderLayer instance, and the number of layers as arguments. This creates an encoder with num\_layers identical layers.
6. Set a seed for PyTorch's random number generator to ensure that the same random numbers are generated every time the program is run.
7. Create a random tensor to represent a batch of sequences. This tensor has a shape of [batch\_size, seq\_len, d\_model], which corresponds to a batch of 1 sequence, with 4 words, and each word represented by an 6-dimensional vector.
8. Pass the tensor through the encoder by calling the forward method of the Encoder instance. This applies the multi-head attention mechanism and the feed-forward network to the input sequences and returns a new tensor that represents the output of the encoder.
9. Print the original input tensor, the shape of the output tensor, and the output tensor. The output tensor has the same shape as the input tensor, which means it can be passed to the decoder of the Transformer model.

**---**

**Decoder Layer**

Well now I am going to show you how to implement the DecoderLayer  class.

Each DecoderLayer consists of three main parts: a self-attention mechanism, an encoder-decoder attention mechanism, and a position-wise feed-forward network. These parts are connected by residual connections and followed by layer normalization.

1. The DecoderLayer class is a subclass of nn.Module, which means it's a custom PyTorch module.
2. In the \_\_init\_\_ method, the self-attention mechanism, the encoder-decoder attention mechanism, the feed-forward network, and the residual connections are defined. The residual connections are a list of three ResidualConnection modules, which are used to add the input of each part to its output.
3. The forward method is where the actual computation happens. It takes an input tensor x of shape [batch\_size, seq\_len, d\_model], an encoder\_output tensor of the same shape, and optional source and target masks mask\_src and mask\_tgt.
4. The input tensor is first passed through the first residual connection and the self-attention mechanism. The lambda function is used to ensure that the same input is used for the query, key, and value in the self-attention mechanism. The target mask is used to prevent the decoder from looking at future tokens.
5. The output of the self-attention mechanism is then passed through the second residual connection and the encoder-decoder attention mechanism. The source mask is used to prevent the decoder from looking at padding tokens.
6. The output of the encoder-decoder attention mechanism is then passed through the third residual connection and the feed-forward network. The feed-forward network applies the same transformation to each position in the sequence independently.
7. The output of the feed-forward network is returned as the output of the layer. This tensor has the same shape as the input tensor and can be passed to the next DecoderLayer in the sequence.

In a full Transformer model, several DecoderLayers are stacked on top of each other to form the decoder.

**Decoder**

Well now I am going to show you how to implement the Decoder class.

The Decoder consists of a stack of identical layers, each of which is a DecoderLayer, and a final layer normalization.

1. The Decoder class is a subclass of nn.Module, which means it's a custom PyTorch module.
2. In the \_\_init\_\_ method, the layers of the decoder and the final layer normalization are defined. The layers are a list of DecoderLayer instances, and the layer normalization is an instance of the LayerNorm class.
3. The forward method takes an input tensor x of shape [batch\_size, seq\_len, d\_model], an encoder\_output tensor of the same shape, and optional source and target masks mask\_src and mask\_tgt.
4. The input tensor is passed through each layer in the decoder in turn. Each layer applies the self-attention mechanism, the encoder-decoder attention mechanism, and the feed-forward network to the input and encoder output, and returns a new tensor that represents the output of the layer.
5. After all layers have processed the input, the final layer normalization is applied. This ensures that the output has a mean of 0 and a standard deviation of 1, which can help to stabilize the learning process.
6. The output of the layer normalization is returned as the output of the decoder. This tensor has the same shape as the input tensor and can be passed to the next component of the Transformer model.

In a full Transformer model, the Decoder takes the output of the Encoder and generates a sequence of output tokens.

**Decoder Notebook**

Well to check the Decoder class, we are going to play with this dedicated notebook file.

This notebook code is using a Decoder to process a batch of sequences.

1. Import the necessary modules and classes.
2. Define the configuration parameters: the dimensions of the model (d\_model), the dimensions of the feed-forward network (d\_ff), the number of heads in the multi-head attention mechanism (h), the batch size, the sequence length, the dropout ratio, and the number of layers in the decoder (num\_layers).
3. Create instances of the MultiHeadAttention and FeedForward classes. These are the main components of the DecoderLayer.
4. Create an instance of the DecoderLayer class, passing the dimensions of the model, the self-attention mechanism, the encoder-decoder attention mechanism, the feed-forward network, and the dropout ratio as arguments.
5. Create an instance of the Decoder class, passing the dimensions of the model, the DecoderLayer instance, and the number of layers as arguments.
6. Set a seed for PyTorch's random number generator to ensure that the same random numbers are generated every time the program is run.
7. Create a random tensor to represent a batch of sequences. This tensor has a shape of [batch\_size, seq\_len, d\_model], which corresponds to a batch of 1 sequence, with 4 words, and each word represented by an 6-dimensional vector.
8. Create another random tensor to represent the output of the encoder. This tensor has the same shape as the input tensor.
9. Pass the input tensor and the encoder output through the decoder by calling the forward method of the Decoder instance. This applies the decoder layers and the final layer normalization to the input and encoder output, and returns a new tensor that represents the output of the decoder.
10. Print the original input tensor, the encoder output tensor, the shape of the output tensor, and the output tensor. The output tensor has the same shape as the input tensor, which means it can be passed to the next component of the Transformer model.

**---**

**Transformer Model Class**

Well now I am going to show you how to implement the Transformer class, which is the core part in our implementation.

1. Initialization (\_\_init\_\_): The class is initialized with an encoder, a decoder, source and target input embeddings, source and target positional encodings, and a projection layer. These are all stored as attributes of the class.
2. Encoding (encode): This method takes a source sequence and an optional source mask as input. It first applies the source input embedding and the source positional encoding to the source sequence, and then passes the result through the encoder. The output of the encoder is returned.
3. Decoding (decode): This method takes a target sequence, the output of the encoder, and optional source and target masks as input. It first applies the target input embedding and the target positional encoding to the target sequence, and then passes the result, along with the encoder output and the masks, through the decoder. The output of the decoder is returned.
4. Projection (project): This method takes a sequence (typically the output of the decoder) as input and passes it through the projection layer. The output of the projection layer, which has the same length as the input sequence but a depth equal to the target vocabulary size, is returned.

This class is typically used to create a Transformer model, which can then be trained on a sequence-to-sequence task. The encode, decode, and project methods correspond to the main steps of the Transformer model: encoding the source sequence, decoding the target sequence, and projecting the decoder output to the target vocabulary size.

**Create Transformer model**

Until now we have implemented all building blocks of transformer model. Next we are am going to combine all of the blocks to build the transformer model. Let’s see how to implement the function create\_transformer\_model to do this.

This function is used to create a Transformer model for a machine translation task.

1. **Extract configuration parameters**: The function first extracts the necessary parameters from the configuration, such as the dimension of the model (d\_model), the number of layers (num\_layers), the number of attention heads (h), the dimension of the feed-forward network (d\_ff), the dropout rate (dropout), and the sequence length (seq\_len).
2. **Initialize embedding layers**: It initializes the source and target embedding layers using the InputEmbedding class. These layers convert the input tokens into vectors of dimension d\_model.
3. **Initialize positional encoding layers**: It initializes the source and target positional encoding layers using the PositionalEncoding class. These layers add positional information to the input embeddings.
4. **Initialize encoder**: It initializes the self-attention and feed-forward layers for the encoder using the MultiHeadAttention and FeedForward classes, respectively. It then creates an encoder layer using the EncoderLayer class and repeats this layer num\_layers times to create the encoder.
5. **Initialize decoder**: It initializes the self-attention, encoder-decoder attention, and feed-forward layers for the decoder. It then creates a decoder layer using the DecoderLayer class and repeats this layer num\_layers times to create the decoder.
6. **Initialize projection layer**: It initializes the projection layer using the Projection class. This layer converts the output of the decoder into logits for each possible output token.
7. **Initialize transformer model**: It creates the Transformer model using the Transformer class, passing in the encoder, decoder, embedding layers, positional encoding layers, and projection layer.
8. **Initialize model parameters**: It initializes the parameters of the model using the Xavier uniform initialization. This is a common initialization method for neural networks.
9. **Return transformer model**: Finally, the function returns the created Transformer model. This model can be used to train and evaluate a machine translation task.

**Put it all together - Playground Transformer**

Well to check the create\_transformer\_model  function, we are going to play with this dedicated notebook file.

This notebook code is using a Transformer model to encode a batch of source sequences, decode a batch of target sequences, project the decoder output to the vocabulary size, and then decode the projected output into words

1. Import the necessary modules and functions.
2. Load the configuration file, dataset, and tokenizers for the source and target languages.
3. Define the vocabulary sizes for the source and target languages, the sequence length, and the batch size.
4. Create the Transformer model using the create\_tranformer\_model function.
5. Set a seed for PyTorch's random number generator to ensure that the same random numbers are generated every time the program is run.
6. Create a random tensor to represent a batch of source sequences and pass it through the encoder of the Transformer model.
7. Create another random tensor to represent a batch of target sequences and pass it along with the encoder output through the decoder of the Transformer model.
8. Pass the decoder output through the projection layer of the Transformer model. This transforms the decoder output into prediction scores for each possible output token in the vocabulary.
9. Find the token with the highest prediction score in each position of each sequence. This is done by calling torch.max on the projected output and taking the indices of the maximum values. The result is a tensor of shape [batch\_size, seq\_len], where each element is the index of the most probable token.
10. Convert the tensor of predicted tokens into a NumPy array. This is done by first removing any extra dimensions with squeeze, then detaching the tensor from the computation graph with detach, and finally converting it to a NumPy array with cpu().numpy().
11. Find the index of the first end-of-sentence (EOS) token in the predicted tokens. This is done by iterating over the predicted tokens and breaking the loop when the EOS token (which has an index of 3) is found.
12. Slice the predicted tokens up to the EOS token. This gives the predicted token sequence.
13. Decode the predicted token sequence into words using the target tokenizer. This converts the indices back into words.
14. Print the predicted sequence of words. This is the output of the Transformer model for the given input sequences.

---

Now I’m going to present to you the functions for launching training process, which includes the train, evaluation and inference during training function. Let’s deep dive in.

---

**Softmax**

As you can see that to make transformer work in the machine translation task, we have used Softmax function at the final step. What it is actually and how to implement it?

The softmax function is a function used in machine learning, especially in the field of neural networks, to map a vector of arbitrary real-valued scores to a vector of probabilities.

Let’s see a concrete usage example of this function and what is its expected result.

The softmax function takes a K-dimensional vector of real numbers and transforms it into a vector of real number in range (0, 1) which add up to 1. This property makes it suitable to interpret the outputs of the model as probabilities.

Given a vector X of length K, the softmax function  is to transform X to obtain the vector Y of the same length.

The element i of Y is calculated by this softmax function, in which the numerator will be the exponential of X\_i with base of the natural logarithm, and the denominator will be the sum of the exponential of all components of X.

In other words, the softmax function exponentiates its input (making all components positive) and then normalizes it (making the sum of all components equal to 1).

**Softmax notebook**

Now let’s play with the notebook file for softmax function

This Python script applies the softmax function to an input tensor using PyTorch.

1. The script first imports the necessary libraries: PyTorch and NumPy.
2. It then creates an input tensor with three elements: 0.123, 0.264, and -np.inf. The -np.inf represents negative infinity.
3. The script applies the softmax function to the input tensor along dimension 0 (since the tensor is 1-dimensional, this is the only dimension).
4. Finally, the script prints the input tensor and the output tensor. The output tensor should contain the softmax-transformed values of the input tensor. Note that the softmax of -np.inf is 0, as the softmax function effectively "squashes" very negative numbers to 0.

---

**Loss function – Cross Entropy Loss**

Before we dive into the training process, I would like to explain a key element of the training process: calculating the loss function. In transformers, they use CrossEntropyLoss to calculate the loss. I will walk you through a concrete example to show how to implement this loss function and what the expected result is.

Given a machine translation task, developing a transformer to translate a French sentence to an English sentence, we configure the sequence length as 3 and the English vocabulary size as 4. The English dictionary in the tokenizer has 4 key-value pairs: "am" for 0, "fine" for 1, "I" for 2, and "Hi" for 3.

I will show the calculation of the cross-entropy loss function in three steps.

In the first step, we will model and predict. Given an input sequence “Je vais bien” in French, the transformer model will predict a tensor of probabilities with shape (sequence\_length, vocabulary\_size). In this case, the prediction is a tensor of shape (3,4), meaning our model predicts 3 tokens, each represented by a 4-dimensional vector. Meanwhile, we know the target sentence is “I am fine.” Based on the English tokenizer, we can deduce that the target tensor of indices is [2,0,1].

In the second step, we calculate the softmax of the predicted probability outputs. Then, we select the probability of each token at the index of the correct label. In this example, we select the value 0.2887 at the index of 2 for the first predicted token, then the value 0.2887 at the index of 0 for the second predicted token, and finally the value 0.3096 at the index of 1 for the last predicted token.

In the third step, we compute the negative log of all the selected probabilities, then take the average of the outputs. We finally obtain the result of the cross-entropy loss.

We have now gone through three simple steps to calculate the cross-entropy loss function for a real-world problem. Now, we will see how to implement it in practice.

**Cross Entropy Loss Notebook**

Now let’s work with this notebook file to implement the cross-entropy loss function. We will calculate the cross-entropy loss between the model's prediction and the target label indices. I will present two ways of implementing this, both of which will obtain the same result.

In the first implementation, we will follow the exact steps presented in the example. First, we import torch and the F function from PyTorch. Suppose we have a prediction and a target tensor as follows. The next step is to calculate the softmax function of the probability prediction, then select the probability for each token at the target index for that token. After that, we calculate the negative log-likelihood loss for all the selected probabilities and take the mean function at the end. Finally, we can print out the results. We can see that our cross-entropy loss value is 1.21921, etc.

In the second implementation, we will use an abstract function, CrossEntropyLoss, from the neural network module in PyTorch. After initializing the prediction and target tensor, we simply call the loss function on the two tensors and obtain the loss result. Finally, we can print out the result. We can verify that we obtain the same cross-entropy loss result as the previous implementation, but this time with just a few lines of code.

In this demonstration, we have shown two ways of implementing the cross-entropy loss function. In the first implementation, we built it from scratch to understand each step of this function. In the second implementation, we used an abstract function from the neural network module of PyTorch to achieve the same result. During the training process of the transformer, we will use the second method due to its convenience.

---

**Train**

Well now I am going to show you how to implement the train function

This function is responsible for training a Transformer model for a machine translation task

1. **Device Assignment**: It assigns the device (GPU if available, else CPU) for computation.
2. **Tensorboard Writer**: It initiates a Tensorboard writer for logging training metrics.
3. **Model Directory**: It checks if a directory for saving model weights exists, if not, it creates one.
4. **Data Preprocessing**: It calls the preprocessing\_data function to load and preprocess the dataset.
5. **Vocabulary Size**: It gets the vocabulary size for the source and target languages.
6. **Model Creation**: It creates the Transformer model using the create\_transformer\_model function.
7. **Optimizer Initialization**: It initializes the Adam optimizer.
8. **Checkpoint Loading**: If a checkpoint exists, it loads the model and optimizer states from the checkpoint, and sets the initial epoch and global step accordingly.
9. **Loss Function**: It defines the CrossEntropyLoss function, ignoring the padding token and applying label smoothing.
10. **Training Loop**: It runs a training loop for a specified number of epochs. For each batch in each epoch, it:
    * Moves the inputs and targets to the device
    * Passes the inputs through the model to get the output
    * Calculates the loss between the output and the target
    * Logs the loss to Tensorboard
    * Performs backpropagation and updates the model parameters
    * Increments the global step
11. **Validation**: At the end of each epoch, it runs a validation step using the evaluation\_step function.
12. **Checkpoint Saving**: It saves a checkpoint of the model and optimizer states, the current epoch, and the global step.

This function is typically called to start the training process. It handles all aspects of training, including data loading, model creation, training loop, validation, and checkpointing.

**Evaluation during training**

Well now I am going to show you how to implement the evaluation function

This function is used to evaluate a trained Transformer model on a validation dataset.

1. **Model Evaluation Mode**: It sets the model to evaluation mode.
2. **Token IDs**: It gets the IDs of the start-of-sequence (SOS) and end-of-sequence (EOS) tokens.
3. **Text Lists**: It initializes lists to store the source texts, target texts, and predicted texts.
4. **Evaluation Loop**: It runs a loop over the validation dataloader. For each batch in the dataloader, it:
   * Moves the encoder input and mask to the device.
   * Asserts that the batch size is 1.
   * Calls the infer\_training function to generate a sequence of tokens from the model.
   * Decodes the source text, target text, and predicted text.
   * Appends the texts to the respective lists.
   * Prints the source text, target text, and predicted text.
   * Breaks the loop if the number of evaluated samples reaches the specified limit.
5. **Metrics Calculation**: If logs are enabled, it calculates the BLEU score, word error rate (WER), and character error rate (CER) between the predicted texts and target texts, and logs these metrics.

This function is typically used in the validation stage of a machine translation task, where the goal is to evaluate the performance of the trained model on unseen data. The evaluation metrics provide a quantitative measure of the model's performance.

---

**Inference during training**

Well now I am going to show you how to implement the infer\_training function

This function performs inference using a provided Transformer model.

1. The function takes several arguments: a Transformer model, an input tensor for the encoder, a mask for the encoder input, the IDs of the start and end of sequence tokens, the length of the sequence, and the device to perform computations on.
2. It first encodes the input using the model's encode method, which returns the encoder output.
3. It then initializes the decoder input with the start of sequence token. The decoder input is a tensor of shape (1, 1).
4. The function enters a loop that continues until the sequence length of the decoder input equals the sequence length of the encoder input.
5. Inside the loop, it creates a causal mask for the decoder input. This mask is used in the Transformer model to prevent the decoder from "seeing" future tokens in the input.
6. It decodes the encoder output using the model's decode method, which returns the decoder output.
7. It selects the last token from the sequence length dimension of the decoder output.
8. It projects this token's output to the target vocabulary size using the model's project method.
9. It finds the token with the maximum value in the projected output, which is the predicted token.
10. It appends this predicted token to the decoder input.
11. If the predicted token is the end of sequence token, it breaks the loop.
12. Finally, it returns the decoder input, removing the batch dimension.

This function is used during the training of the Transformer model to perform inference on the training data. It generates a sequence of tokens from the model, starting with the start of sequence token and ending when the model predicts the end of sequence token or when the sequence length reaches the specified limit.

---

**Training in action**

Well, now we have built the whole transformer package. It’s time to call the training engine in action.

We need to write this small code snippet to do it.

This script is used to train a Transformer model.

1. It first imports the necessary functions: train from the transformer.engine module and load\_config from the transformer.utils module.
2. The script then checks if it is being run as a standalone program (as opposed to being imported as a module) with the if \_\_name\_\_ == "\_\_main\_\_": statement.
3. Inside this conditional block, it sets the path of the configuration file to "config.json".
4. It loads the configuration from this file using the load\_config function. This function presumably reads the JSON file and returns a dictionary containing the configuration parameters.
5. Finally, it calls the train function with the loaded configuration as an argument. This function presumably trains the Transformer model according to the specified configuration.

In summary, this script is a driver script for training a Transformer model. It reads the training configuration from a JSON file and then calls a function to perform the training.

---

After that, we call the train.py on the terminal to launch the training.

The training process starts with checking the available device for training.

Then if there exists a training checkpoint since the last training, if so, it will load the weight from the last checkpoint and continue to train for a new checkpoint. If it does not exist the checkpoint, it will start training from the first epoch.

After one epoch is trained successfully, it will print several inference examples based on the trained model until that epoch. Then, the checkpoint model will saved in the model folder

When we finish training over all epochs, we will obtain all the checkpoint model like this.

At this point, we are ready to start inferencing and testing the latest trained model.

Let’s continue to this excited part.

---

**Intro to Inference**

Well, now I am going to show you how we can do inference on the trained model.

Given the task of machine translation in transformer, the inference process

* Firstly load the latest trained model, and the necessary configuration setting, like target tokenizers, etc
* Then it takes a sentence in one language, in this case a English sentence, as input.
* Finally, it output a translated sentence in French.

Let’s deep dive into how we can obtain the inference process.

---

**Model Inference**

To do the inference process, we need to implement the inference function in the engine file.

I am going to show you how to implement the inference function.

This function, inference, translates a source text using a Transformer model.

1. The function takes several arguments: the source text to translate, a Transformer model, tokenizers for the source and target languages, the maximum sequence length, and the device to perform computations on.
2. It first sets the model to evaluation mode with model.eval(). This is necessary because the model behaves differently during training and evaluation.
3. It then disables gradient calculation with torch.no\_grad(). This is done because gradients are not needed during inference, and disabling their calculation saves memory.
4. It tokenizes the source text using the source language tokenizer's encode method.
5. It calculates the number of padding tokens needed for the encoder input. This is done by subtracting the length of the tokenized source text and 2 (for the start and end of sequence tokens) from the maximum sequence length.
6. It gets the IDs of the start of sequence, end of sequence, and padding tokens using the source language tokenizer's token\_to\_id method.
7. It creates the encoder input by concatenating the start of sequence token, the tokenized source text, the end of sequence token, and the necessary number of padding tokens.
8. It creates the encoder mask, which is used in the Transformer model to ignore the padding tokens during the attention calculation.
9. It encodes the input using the model's encode method, which returns the encoder output.
10. It initializes the decoder input with the start of sequence token.
11. It enters a loop that continues until the sequence length of the decoder input equals the maximum sequence length or the end of sequence token is predicted.
12. Inside the loop, it creates a causal mask for the decoder input. This mask is used in the Transformer model to prevent the decoder from "seeing" future tokens in the input.
13. It decodes the encoder output using the model's decode method, which returns the decoder output.
14. It selects the last token from the sequence length dimension of the decoder output.
15. It projects this token's output to the target vocabulary size using the model's project method.
16. It finds the token with the maximum value in the projected output, which is the predicted token.
17. It appends this predicted token to the decoder input.
18. If the predicted token is the end of sequence token, it breaks the loop.
19. Finally, it decodes the decoder input using the target language tokenizer's decode method to get the translated text, and returns this text.

This function is used during the inference phase of the Transformer model to translate a source text to the target language. It generates a sequence of tokens from the model, starting with the start of sequence token and ending when the model predicts the end of sequence token or when the sequence length reaches the specified limit.

**Playground inference**

Now let’s check the inference function with the trained Transformer model using this dedicated notebook file.

This final example is to translate a source text using a pre-trained Transformer model.

1. It first imports the necessary modules and functions.
2. It then assigns the device for computation. If a GPU is available, it uses that; otherwise, it uses the CPU.
3. It loads the configuration from a JSON file using the load\_config function.
4. It gets the dataset using the get\_dataset function and the configuration.
5. It gets the tokenizers for the source and target languages using the get\_tokenizer function, the configuration, the dataset, and the respective language codes from the configuration.
6. It gets the vocabulary sizes for the source and target languages using the get\_vocab\_size method of the respective tokenizers.
7. It gets the maximum sequence length from the configuration.
8. It creates the Transformer model using the create\_tranformer\_model function, the configuration, and the vocabulary sizes. It moves the model to the assigned device.
9. It gets the path of the latest checkpoint using the get\_checkpoint\_path function and the configuration.
10. It loads the state of the model from the checkpoint using torch.load.
11. It assigns the loaded state to the Transformer model using the load\_state\_dict method.
12. It sets the source text to translate.
13. It calls the inference function with the source text, the model, the tokenizers, the maximum sequence length, and the device. This function translates the source text using the model and returns the translated text.
14. Finally, it prints the source text and the translated text.

In summary, this script loads a pre-trained Transformer model and uses it to translate a source text. It demonstrates how to use a Transformer model for inference.

--

In summary, in this long video, I have walked you through the implementation, training, and inference of the Transformer model for the machine translation task. Here are the summaries of our journey:

* Firstly, we implemented all building blocks of the Transformer.
* Then, we built the Transformer as a package.
* Next, we wrote a script to launch the training with several lines of code.
* After that, we tested the inference with the trained model for the use case of a machine translation task.
* In addition, we provided a playground with notebooks for all functions and classes.

Title:

Implementing Transformer in PyTorch: Full Source Code and Notebook Playground

In this video, I walk you through implementing transformer in PyTorch. I will present the implementation by building a Python package for transformer. I will present source code for each class and function, along with the notebook file to playground with.

The source code can be cloned here:

Hope you find it helpful.

Following are some key stations in the video.

00:00 Introduction to transformer implementation

06:44 Source code structure

09:00 Introduction to data preparation

09:22 Load config function

11:52 Get dataset function

14:22 Get Tokenizer function

18:19 Encoder mask function

21:00 Padding mask function

22:10 Causal mask function

24:51 Decoder mask function

26:21 Data Preprocessor class

30:18 Preprocessing data function

33:20 Input Embedding class

37:31 Positional Encoding class

43:11 Multi-head Attention class

54:56 Feed forward class

59:39 LayerNorm class

01:05:21 Residual Connection class

01:11:09 Projection class

01:16:03 Intro to encoder, decoder, model classes

01:16:28 Encoder Layer class

01:19:06 Encoder class

01:24:42 Decoder Layer class

01:28:26 Decoder

01:34:30 Transformer class

01:37:31 Create-transformer function

01:45:06 Softmax function

01:48:01 Cross Entropy Loss function

01:54:13 Intro to engine classes

01:54:27 Train engine

01:57:31 Evaluation engine

01:59:53 Inference in training engine

02:03:01 Script to launch training

02:06:19 Intro to inference

02:07:06 Inference engine

02:12:25 Inference playground

02:15:24 Thank you