Sunday, February 2, 2025 6:18 PM

📌 Breakdown of the Imports:

1. **import torch** – This imports PyTorch, which we'll use for tensor computations, autograd, and model building.
2. **import torch.nn as nn** – This brings in PyTorch's neural network module (nn), which contains layers like nn.Linear, nn.Transformer, etc.
3. **import torch.optim as optim** – This imports PyTorch's optimization library, which helps in training the model (like Adam, SGD).
4. **import gzip** – This is a built-in Python library to handle .gz compressed files. (Maybe we are dealing with a compressed dataset?)
5. **import time** – Used for measuring execution time, probably for tracking training speed.
6. **import math** – Might be used for mathematical operations like scaling, exponentials, etc. (often used in the Transformer's positional encoding).
7. **import spacy** – A popular NLP library, often used for tokenization. It helps in processing text (like splitting sentences into words).
8. **from torch.utils.data import Dataset, DataLoader** –
    - Dataset helps define a custom dataset.
    - DataLoader is used to load data in batches efficiently.
9. **from torch.nn.utils.rnn import pad_sequence** – Helps pad sequences to the same length (important for batch processing in NLP models).
10. **from tqdm import tqdm** – A progress bar library, used for displaying progress during training or data processing.

🔍 What This Tells Us About the Code?

- This is an **NLP-based Transformer model** (because of spacy and pad_sequence).
- It will involve **batch processing** (because of Dataset and DataLoader).
- It might be **trained from scratch** (since we are not importing Hugging Face's Transformer library).
- Positional encoding or some math-based scaling might be involved (math library).
- The dataset might be compressed (gzip), so we'll likely load .gz files.

📌 Why Use a Compressed .gz File?

1. **Saves Space** 🗜️ – Large text datasets take up a lot of disk space. Compressing them reduces storage requirements.
2. **Faster Loading** 🚀 – Instead of storing raw text, compressed files can be read directly while training, reducing I/O overhead.
3. **Common in NLP Datasets** 📚 – Many public datasets (like **WikiText, OpenWebText, or large corpora**) come in .gz format to keep downloads manageable.
4. **Efficient for Streaming** ⏳ – We can read data in chunks rather than loading everything into memory at once.

💡 **Example of Why It's Used:**
Let's say we have a dataset of **500MB** in raw text. If we compress it using .gz, it might become **50MB**, making it easier to download and load into memory while training.

💬 **Next:** If you share the code where the .gz file is actually being read, I can explain exactly how it's being handled!

📌 What These Commands Do?

1. **!python -m spacy download de_core_news_sm**
   - Downloads the **German (de) small language model** for spaCy.
   - "core_news" means it's designed for **news-style text**.
   - "sm" means **small model** (faster, but less accurate).
2. **!python -m spacy download en_core_web_sm**
   - Downloads the **English (en) small language model**.
   - "core_web" means it's trained on **web text** (more general-purpose).
   - "sm" means **small model**, suitable for lightweight processing.

🔍 Why Are We Downloading These?

- These models **help with tokenization, lemmatization, and POS tagging**.
- In **NLP pipelines**, we often need to **split text into words**, remove unnecessary elements, and extract useful features.
- Since we are working on a **Transformer model**, we might need to preprocess text **before passing it into the model**.

💡 **What's Next?** If your code is using spaCy for tokenization, share that part, and I'll explain exactly how it fits into your Transformer pipeline! 🔥

📌 What Does This Code Do?

```
spacy_de = spacy.load('de_core_news_sm')
spacy_en = spacy.load('en_core_web_sm')
```

- This **loads the pre-trained spaCy language models** for **German (de)** and **English (en)**.
- These models contain **pre-trained tokenizers, part-of-speech (POS) taggers, lemmatizers, and named entity recognition (NER) systems**.
- The **variable names** (spacy_de and spacy_en) indicate that these will be used for **processing German and English text**, respectively.

🔍 Why Are We Using spaCy?

Since we are building a **Transformer model**, we need to **process text before feeding it into the model**. spaCy helps us:

1. **Tokenize Sentences** – Converts text into a list of words (tokens).
2. **Normalize Text** – Handles punctuation, special characters, etc.
3. **Lemmatization (if needed)** – Converts words to their base form (e.g., "running" → "run").
4. **Language-Specific Rules** – Helps deal with differences in German and English grammar.

🔗 Example of How spaCy Works

```
text = "Hallo, wie geht es dir?"
tokens = [tok.text for tok in spacy_de(text)]
print(tokens)
```

Output:

```
['Hallo', ',', 'wie', 'geht', 'es', 'dir', '?']
```

As you can see, it **splits the sentence into individual words and punctuation**, which is important for NLP tasks.

💡 **Next Step:** Your Transformer code will likely use spaCy for tokenization before converting text into numerical data (embeddings). If that part is next, send it over, and I'll break it down! 🚀🔥

📌 What Does This Code Do?

SEED = 123
torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True

1. **SEED = 123**
    - This sets a **random seed**. In machine learning, using a fixed random seed ensures **reproducibility**—so every time you run your code, you'll get the same results. It's essential for debugging and comparing experiments.
2. **torch.manual_seed(SEED)**
    - This sets the **seed for PyTorch's random number generator**. This ensures that any random operations (like weight initialization, shuffling data, etc.) give the same results across runs.
    - This is especially useful if you're doing experiments or training models, and you want to make sure that your results are consistent.
3. **torch.backends.cudnn.deterministic = True**
    - This is related to **CUDA (GPU)**. When training on a GPU, some operations can be **non-deterministic** (i.e., produce slightly different results each time due to optimization for speed).
    - By setting this flag to **True**, we make the operations deterministic on **NVIDIA GPUs** using cuDNN, ensuring the results are **exactly the same** every time (given the same input, same seed).

🔍 Why Is This Important?

- **Reproducibility** is a key aspect of machine learning. Setting a fixed seed ensures that:
    1. If you or anyone else runs the code multiple times, you'll get **consistent results** (same output, same metrics).
    2. Helps **debug** effectively, because if there's a bug or performance drop, you can isolate it and know it's not due to randomness.
- **CUDA Determinism**: When working on GPUs, it's easy to get tiny differences in results because of parallel operations. This setting ensures **exact replication** of results, making it easier to compare experiments.

💡 Next Step: If you have more code that involves model building or training, feel free to share it! I can help explain how all these pieces come together in the Transformer model. 🚀

📌 What Does This Code Do?

This part of the code defines a custom Dataset class called Multi30kDataset. This class is responsible for loading, transforming, and providing access to the data in a convenient way for training the model.

1. class Multi30kDataset(Dataset):

- This is creating a **custom dataset** class that inherits from torch.utils.data.Dataset, which is the standard way of working with datasets in PyTorch.
- The Multi30k dataset is likely related to **machine translation**, where the source and target languages are provided in separate files.

2. def __init__(self, src_file, trg_file, src_transform=None, trg_transform=None):

- **Constructor** to initialize the dataset.
    - **src_file**: Path to the source language file (e.g., German).
    - **trg_file**: Path to the target language file (e.g., English).
    - **src_transform**: Optional transformation for the source sentences (like tokenization or padding).

- **trg_transform**: Optional transformation for the target sentences.

3. self.src_data = self.load_data(src_file) and self.trg_data = self.load_data(trg_file)

- **Loading the Data**:
  - The function **load_data** is called for both the source (src_file) and target (trg_file) data files.
  - The **gzip.open** method reads the **compressed files** (.gz format), and the data is loaded line-by-line as a list of sentences.
  - The strip() method removes any unnecessary whitespace or newline characters from the sentences.

4. def load_data(self, file_path):

- This method opens and reads the contents of the dataset files.
- The dataset is expected to be **one sentence per line** in the .gz compressed files.

5. def __len__(self):

- Returns the **length of the dataset**, i.e., the number of source sentences (assuming source and target files are of equal length).

6. def __getitem__(self, idx):

- The **getitem** method allows us to index into the dataset and get a specific item (sentence pair) by its index idx.
- It returns a dictionary with:
  - **"src"**: The source sentence.
  - **"trg"**: The target sentence.
- **Transforms**: If any transformations (like tokenization, lowercasing, or padding) were provided, they are applied to both the source and target sentences before returning them.

🔍 Why Is This Important?

1. **Custom Dataset Class**: This class makes it easy to work with custom data by using PyTorch's DataLoader. The dataset can be used in training loops, and since it supports transformations, you can preprocess data (like tokenization or padding) before it reaches the model.
2. **Efficient Data Handling**:
   - The data is read **line by line** from a **compressed file**, saving both **disk space** and **loading time**.
   - By using **gzip** and transformations, you ensure that the data is ready for feeding into a neural network without extra overhead.
3. **Language Pairs**: This is likely part of a **machine translation task** where the source language (e.g., German) is mapped to a target language (e.g., English).

💡 Next Step:

If you're using this dataset for a Transformer, you'll likely need to tokenize these sentences and convert them into **numerical formats** (like word IDs). If there's more code, share it, and I can help explain how everything fits together! 🔥

Steps to Run the Code:

1. **Install the Required Libraries:**
   - You need torch and spaCy for this to work. Install them using pip if you haven't already:
     pip install torch spacy tqdm

2. **Download the Language Models:**
   - Download the language models as per your earlier code:
     python -m spacy download de_core_news_sm
     python -m spacy download en_core_web_sm
3. **Prepare Sample Data:**
   - If you don't have the actual .gz files (src_file and trg_file), you can create **dummy text files**

for testing:

- Create two simple text files src.txt (for German) and trg.txt (for English).
- Put a few sentences in each file to simulate the dataset (e.g., German sentence in src.txt and corresponding English sentence in trg.txt).

4. **Code for Testing the Dataset:**
   - Here's how you can test it with sample files:

```
import torch
import spacy
from torch.utils.data import DataLoader

# Load the language models
spacy_de = spacy.load('de_core_news_sm')
spacy_en = spacy.load('en_core_web_sm')

# Define a simple transformation (e.g., tokenize)
def tokenize_de(sentence):
    return [token.text for token in spacy_de(sentence)]

def tokenize_en(sentence):
    return [token.text for token in spacy_en(sentence)]

# Assuming you have `src.txt` and `trg.txt` files
src_file = 'src.txt'  # Replace with actual path
trg_file = 'trg.txt'  # Replace with actual path

# Create the dataset
dataset = Multi30kDataset(src_file, trg_file, src_transform=tokenize_de,
trg_transform=tokenize_en)

# Create a DataLoader to batch and shuffle the dataset
data_loader = DataLoader(dataset, batch_size=2, shuffle=True)

# Fetch one batch of data
for batch in data_loader:
    print(batch)
```

5. **Run the Code:**
   - Run the script in your Python environment. It will:
     1. Load the dataset.
     2. Apply tokenization using spaCy.
     3. Print out a batch of tokenized sentences.

Expected Output:

After running the script, you should see output like this (example with a batch size of 2):
{'src': [['Hallo', ',', 'wie', 'geht', 'es', 'dir', '?'], ['Guten', 'Morgen', '!']], 'trg': [['Hello', ',', 'how', 'are', 'you', '?'], ['Good', 'morning', '!']]}

Tip:

If you encounter issues, feel free to share the error message here, and I'll help you debug it!

What These Functions Do?

def tokenize_de(text):

```
    return [token.text.lower() for token in spacy_de.tokenizer(text)]
def tokenize_en(text):
    return [token.text.lower() for token in spacy_en.tokenizer(text)]
```

1. tokenize_de(text)

- This function takes a **German text** (text) as input and processes it through the spaCy tokenizer for German (spacy_de).
- **.tokenizer(text)**: Tokenizes the sentence into individual words or subword units based on spaCy's built-in rules.
- **.lower()**: Converts every token to lowercase, ensuring that the model doesn't treat words like **"Hello"** and **"hello"** as separate entities. This can help in reducing vocabulary size and improving generalization.

2. tokenize_en(text)

- This is the same as tokenize_de, but it works for **English** text using the English tokenizer (spacy_en).

🔍 Why Lowercasing?

- **Consistency**: Lowercasing makes sure that the model doesn't treat words with different cases (e.g., "Apple" and "apple") as separate entities. This is particularly useful for small datasets or when you don't want to overfit to the case of words.
- **Simplification**: Helps reduce the size of the vocabulary, as the model doesn't have to learn separate embeddings for upper- and lowercase versions of the same word.

💡 Example:

Let's assume we pass a sentence like **"Hallo, Wie geht es dir?"** through tokenize_de and **"Hello, how are you?"** through tokenize_en.

```
# Test tokenization
german_text = "Hallo, Wie geht es dir?"
english_text = "Hello, how are you?"
print(tokenize_de(german_text))  # Output: ['hallo', ',', 'wie', 'geht', 'es', 'dir', '?']
print(tokenize_en(english_text))  # Output: ['hello', ',', 'how', 'are', 'you', '?']
```

Next Step:

If you're using these functions in your Multi30kDataset class (as the transform functions), your dataset will now return **tokenized and lowercase sentences**. If you have more parts of the code (like padding, batch processing, or model definitions), feel free to share them! I'll help explain how they all work together. 🔥

📌 What Does This Code Do?

1. Loading Datasets:

```
train_data = Multi30kDataset(train_de_path, train_en_path, src_transform=tokenize_de,
trg_transform=tokenize_en)
val_data = Multi30kDataset(val_de_path, val_en_path, src_transform=tokenize_de,
trg_transform=tokenize_en)
test_data = Multi30kDataset(test_de_path, test_en_path, src_transform=tokenize_de,
trg_transform=tokenize_en)
```

- **Loading Datasets**:
    - These lines are creating three instances of the Multi30kDataset class for the **training**, **validation**, and **test datasets**.
    - **train_de_path, train_en_path**: These variables store the paths to the **German** and **English** training data files (same for validation and test data).
    - **src_transform=tokenize_de**: This applies the tokenize_de function to the German sentences. This means every sentence in the source language will be tokenized into

individual words and converted to lowercase.
    - **trg_transform=tokenize_en**: Similarly, the tokenize_en function is applied to the English sentences.
      Result: After loading the dataset, the Multi30kDataset class will return a list of tokenized sentences in both German and English when you index into it.

2. Defining Special Tokens:

PAD_TOKEN = '<pad>'
SOS_TOKEN = '<sos>'
EOS_TOKEN = '<eos>'
UNK_TOKEN = '<unk>'

These lines define **special tokens** that will be used during the model training and tokenization process. Let's break them down:

1. **PAD_TOKEN = '<pad>'**:
    - This is the **padding token**. Padding is used when sentences of varying lengths are batched together for training.
    - The model uses the padding token to "fill in" shorter sequences, so all sequences in the batch are of equal length (usually the length of the longest sentence in the batch).
2. **SOS_TOKEN = '<sos>'**:
    - This stands for **Start Of Sequence**. It is typically added at the beginning of a sequence in **sequence-to-sequence models** (like machine translation) to signal the start of the target sentence.
    - In **machine translation**, the model generates tokens one by one, starting with the <sos> token.
3. **EOS_TOKEN = '<eos>'**:
    - This stands for **End Of Sequence**. It is used to mark the end of a generated sequence.
    - When the model predicts the <eos> token, it stops generating further tokens.
4. **UNK_TOKEN = '<unk>'**:
    - This stands for **Unknown**. It's used when the model encounters a word that is not in its vocabulary.
    - The model replaces any unknown or rare words with this token, preventing errors during processing.

🔍 Why Are These Special Tokens Important?

- **Padding (<pad>)**: Ensures that all sequences in a batch have the same length, making them easier to process in parallel. It's ignored by the model during training.
- **Start and End Tokens (<sos>, <eos>)**: These help the model know when to start and stop generating output. It's essential for training and testing models like **machine translation** or **summarization**.
- **Unknown Token (<unk>)**: This is a fallback mechanism to handle out-of-vocabulary (OOV) words, preventing the model from crashing or producing nonsense if it encounters an unseen word during inference.

💡 Next Step:

Now that you've set up your dataset and special tokens, you'll likely need to **build a vocabulary** and convert these tokens into **numerical IDs** for training the model. Let me know if you want me to walk you through that process, or if you need help with any other part of the code! 🔥

📌 What Does This Function Do?

This function builds a **vocabulary** from a list of tokenized sentences. The vocabulary maps each unique token (word) to a unique **index**. Additionally, it ensures that the special tokens (like <pad>, <sos>, etc.)

are included in the vocabulary.
Here's the code you shared:

```
def create_vocab(tokenized_sentences, special_tokens):
  vocab = {token: idx for idx, token in enumerate(special_tokens)}  # Step 1

  for sentence in tokenized_sentences:  # Step 2
    for token in sentence:
      if token not in vocab:  # Step 3
        vocab[token] = len(vocab)  # Step 4
  return vocab
```

Step-by-Step Explanation:

1. **Initialize Vocabulary with Special Tokens**:
   vocab = {token: idx for idx, token in enumerate(special_tokens)}
   - This initializes the vocabulary with the **special tokens** that you defined earlier (<pad>, <sos>, <eos>, <unk>).
   - enumerate(special_tokens) gives each token an index starting from 0. The special tokens are assigned indices in the order in which they appear in the special_tokens list.
     For example, if special_tokens = ['<pad>', '<sos>', '<eos>', '<unk>'], the initial vocabulary would look like this:
     vocab = {'<pad>': 0, '<sos>': 1, '<eos>': 2, '<unk>': 3}
2. **Iterate Over the Tokenized Sentences**:
   for sentence in tokenized_sentences:
     for token in sentence:
   - The function loops through each sentence in the tokenized_sentences list, where each sentence is a list of tokens (words).
   - For each sentence, it iterates over the individual tokens (words).
3. **Add New Tokens to Vocabulary**:
   if token not in vocab:
     vocab[token] = len(vocab)
   - This checks if a token is **not already in the vocabulary**. If it's a new token (that wasn't in the original list of special tokens), it assigns the token a unique index.
   - The index is the **current length of the vocabulary**, ensuring that each new token gets a unique ID that's one greater than the previous token's index.
     Example:
     Let's say you have the following tokenized sentence:
     tokenized_sentences = [['hello', 'world'], ['how', 'are', 'you']]

     Initially, the vocabulary is:
     vocab = {'<pad>': 0, '<sos>': 1, '<eos>': 2, '<unk>': 3}

     After processing all the sentences, the vocabulary will become:
     vocab = {'<pad>': 0, '<sos>': 1, '<eos>': 2, '<unk>': 3, 'hello': 4, 'world': 5, 'how': 6, 'are': 7, 'you': 8}

4. **Return the Vocabulary**:
   return vocab
   - Finally, the function returns the completed vocabulary, which contains both the special tokens and any additional tokens found in the sentences.

🔍 Why Is This Function Important?

- **Mapping Words to Indices**: Neural networks work with numerical data, so you need to convert words into numbers. This vocabulary serves as a lookup table for converting tokens into indices

and vice versa.

- **Handling Special Tokens**: Special tokens like <pad>, <sos>, etc., are crucial for sequence processing. Padding ensures sequences are of equal length, and the start and end tokens help with generation tasks.
- **Extending Vocabulary**: The function also allows you to dynamically add any new words that appear in the training data to the vocabulary. This is useful for models that need to handle **out-of-vocabulary words**.

💡 Next Step:

Once you have this vocabulary, you'll need to convert both the source and target sentences into sequences of indices (numerical representation) before passing them to the model. Let me know if you want to move forward with that part or if you have any more questions! 🔥

📌 What is Positional Encoding?

Transformers **don't have recurrence (like RNNs)** or any notion of sequence order. To help them understand **the order of words** in a sequence, we add **positional encodings** to the input embeddings. Each position in the sequence gets a unique **sinusoidal (sine & cosine) encoding**.

📜 Step-by-Step Breakdown

Here's the code you shared:

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=1000):
        super().__init__()
```

- This is a PyTorch module (nn.Module), meaning it can be used like a layer in a neural network.
- **d_model**: The embedding size (how many features per token).
- **max_len**: Maximum length of input sequences. Default is 1000, meaning we assume a sentence can be at most 1000 tokens long.

1️⃣ Create a Positional Encoding Matrix

```
pe = torch.zeros(max_len, d_model)
```

- Creates a max_len × d_model matrix filled with zeros.
- Each row represents a **position** (1st word, 2nd word, etc.).
- Each column represents a **dimension of the embedding**.

2️⃣ Generate Position Indices

```
position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
```

- torch.arange(0, max_len) creates a **vector of positions**: [0, 1, 2, ..., max_len-1]
- .unsqueeze(1) changes its shape from [max_len] → [max_len, 1], making it a **column vector**.

Example for max_len=5:

```
position = [[0],
            [1],
            [2],
            [3],
            [4]]
```

3️⃣ Compute Frequency Scaling Term

```
div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) / d_model))
```

- This calculates different frequency scales for sine and cosine functions.
- The 10000.0 factor helps distribute values smoothly.

- The arange(0, d_model, 2) selects every **even** index (for sine) and every **odd** index (for cosine).

Example when d_model=6:
[0, 2, 4] → for sine
[1, 3, 5] → for cosine

Each position gets different sinusoidal values based on these scales.

4 Apply Sine & Cosine

```
pe[:, 0::2] = torch.sin(position * div_term)  # Even indices → sine
pe[:, 1::2] = torch.cos(position * div_term)  # Odd indices → cosine
```

- **Even columns** (0, 2, 4, …) get **sine** values.
- **Odd columns** (1, 3, 5, …) get **cosine** values.
- This ensures **different frequencies** for different dimensions, giving a **rich encoding**.

Example output for **first 3 positions** (d_model=6):
Position 0: [0.000, 1.000, 0.000, 1.000, 0.000, 1.000]
Position 1: [0.841, 0.540, 0.909, 0.416, 0.951, 0.309]
Position 2: [0.909, -0.416, 0.755, -0.655, 0.588, -0.809]

- Notice how sine values **oscillate** between -1 and 1 as the position increases.

5 Reshape and Store in Buffer

```
pe = pe.unsqueeze(0).transpose(0, 1)
```

- .unsqueeze(0): Adds a **batch dimension** → [1, max_len, d_model]
- .transpose(0, 1): Swaps first two dimensions → [max_len, 1, d_model]
    - This helps in adding positional encoding **across batches**.

```
self.register_buffer('pe', pe)
```

- register_buffer stores pe **inside the model**, but it **won't be updated during training** (since it's not a learnable parameter).

6 Forward Pass

```
def forward(self, x):
   return x + self.pe[:x.size(0), :]
```

- **Takes x (word embeddings) as input**.
- **Adds the positional encoding** (pe[:x.size(0), :]) to it.
- The x.size(0) ensures we **only take the first N positions** corresponding to the input length.

Why Use Sin & Cos Functions?

1. **Periodic Nature**: Words at similar positions get similar encodings, helping the model learn position-related patterns.
2. **Unique Representation**: Each position has a distinct encoding due to different sine & cosine frequencies.
3. **Extrapolation**: Unlike learned embeddings, sinusoidal encodings can generalize to unseen lengths.

Summary

- This **adds positional information** to word embeddings in a Transformer model.
- Uses **sinusoidal encodings** (sine for even, cosine for odd).
- Stored as a **buffer** so it doesn't update during training.
- During forward(), **adds positional encodings** to input embeddings.

What's Next?

This is a key part of Transformer models. If you're building an **encoder-decoder model**, next you'll likely define the **embedding layer** and **self-attention mechanism**. Let me know if you need help with that!

Example: Understanding Positional Encoding

Let's **manually** run the PositionalEncoding class for a small example and see how it modifies an embedding.

Step 1: Define Positional Encoding Class (Same as Before)

```
import torch
import math
import torch.nn as nn
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=10):  # Keeping max_len small for easy understanding
        super().__init__()
        pe = torch.zeros(max_len, d_model)
position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)  # Shape: [max_len, 1]
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) / d_model))
pe[:, 0::2] = torch.sin(position * div_term)  # Apply sin to even indices
        pe[:, 1::2] = torch.cos(position * div_term)  # Apply cos to odd indices
pe = pe.unsqueeze(0)  # Shape becomes [1, max_len, d_model] (for batch compatibility)
        self.register_buffer('pe', pe)
def forward(self, x):
        return x + self.pe[:x.size(1), :]
```

Step 2: Create a Dummy Input

Let's assume we have a **sentence with 5 words**, and each word is represented by a **4-dimensional embedding** (d_model=4).

```
# Define embedding size
d_model = 4
max_len = 10  # Maximum sentence length
# Create Positional Encoding Layer
pos_encoder = PositionalEncoding(d_model, max_len)
# Create a dummy embedding for a 5-word sentence (batch size = 1, seq_len = 5, d_model = 4)
dummy_embedding = torch.zeros(1, 5, d_model)  # Shape: [batch, seq_len, d_model]
# Apply positional encoding
encoded_embedding = pos_encoder(dummy_embedding)
print(encoded_embedding)
```

Step 3: Understanding the Output

Before Positional Encoding (Original dummy_embedding):

```
[[[0.0000, 0.0000, 0.0000, 0.0000],  # Word 1
  [0.0000, 0.0000, 0.0000, 0.0000],  # Word 2
  [0.0000, 0.0000, 0.0000, 0.0000],  # Word 3
  [0.0000, 0.0000, 0.0000, 0.0000],  # Word 4
  [0.0000, 0.0000, 0.0000, 0.0000]]] # Word 5
```

Since it was initialized with all zeros, each word starts with **no unique information**.

After Applying Positional Encoding:

```
[[[ 0.0000,  1.0000,  0.0000,  1.0000],  # Position 0
```

```
[ 0.8415,  0.5403,  0.0909,  0.9959],  # Position 1
[ 0.9093, -0.4161,  0.1818,  0.9839],  # Position 2
[ 0.1411, -0.9900,  0.2727,  0.9640],  # Position 3
[-0.7568, -0.6536,  0.3636,  0.9361]]] # Position 4
```

What's Happening Here?

- The values are **no longer zeros**; each word in the sequence **now has a unique positional encoding**.
- Even indices (0, 2) → **Sinusoidal values** (sin function).
- Odd indices (1, 3) → **Cosine values** (cos function).
- **Patterns emerge** in the position encodings, which the Transformer uses to understand word order.

Summary

1. **Before**: All embeddings were zero, meaning **no positional information**.
2. **After**: Each word got **a unique position-based modification**, allowing the Transformer to differentiate between positions.
3. **Now**, when these encoded embeddings go into the self-attention mechanism, the model understands their relative positions without recurrence.

Next Steps

Now that we have **positional encodings**, the next steps in a Transformer would be: ☑ **Word Embeddings** (instead of zero vectors).

☑ **Self-Attention Mechanism**.

☑ **Feed-Forward Network**.

Code for forward() Function

```
def forward(self, x):
   return x + self.pe[:x.size(0), :]
```

Breakdown Step-by-Step

1 x is the Input Embedding

```
def forward(self, x):
```

- x yaha ek **word embedding tensor** hoga.
- Shape: [seq_len, batch_size, d_model]
    - seq_len = Kitne words hain sequence me
    - batch_size = Kitne sentences ek saath pass ho rahe hain
    - d_model = Embedding size (jitne features per word)

For example, agar ek batch me **3 sentences** hain, har ek **5 words ka** aur embedding size **4** hai, to x ka shape hoga:

x.shape = [5, 3, 4]

Mtlb:

- **5 words** per sentence
- **3 sentences** ek saath
- **4-dimensional embeddings** per word

2 Add Positional Encoding

```
return x + self.pe[:x.size(0), :]
```

- self.pe positional encoding matrix hai (size [max_len, 1, d_model]).

- x.size(0) mtlb jitna x ka sequence length hoga utna hi **pe ka part select hoga**.
- **Broadcasting hoti hai**, yani har batch ke liye positional encoding automatically apply ho jata hai.

🔥 Example Run

Agar x ek **random embedding** ho:

```
x = torch.tensor([
   [ [1.0, 2.0, 3.0, 4.0] ],  # Word 1
   [ [5.0, 6.0, 7.0, 8.0] ],  # Word 2
   [ [9.0, 10.0, 11.0, 12.0] ] # Word 3
])  # Shape: [3, 1, 4] (3 words, batch=1, 4 features per word)
```

Aur positional encoding **pe** kuch aisa ho:

```
pe = [
   [ 0.0,  1.0,  0.0,  1.0],  # Position 1
   [ 0.8,  0.5,  0.1,  0.9],  # Position 2
   [ 0.9, -0.4,  0.2,  0.8]   # Position 3
]  # Shape: [3, 1, 4]
```

To **final output** hoga:

```
x + pe = [
   [ 1.0+0.0,  2.0+1.0,  3.0+0.0,  4.0+1.0 ],  # Word 1
   [ 5.0+0.8,  6.0+0.5,  7.0+0.1,  8.0+0.9 ],  # Word 2
   [ 9.0+0.9, 10.0-0.4, 11.0+0.2, 12.0+0.8 ]   # Word 3
]
```

Final Result:

```
[
   [1.0, 3.0, 3.0, 5.0],
   [5.8, 6.5, 7.1, 8.9],
   [9.9, 9.6, 11.2, 12.8]
]
```

🔑 Summary

☑ x hota hai **word embeddings**.
☑ self.pe hota hai **precomputed positional encodings**.
☑ x + self.pe[:x.size(0), :] har **word embedding me positional encoding add** kar deta hai.
☑ Ye model ko **word ka position samjhne me madad karta hai** bina recurrence ke!

📌 Multi-Head Attention Overview

◇ **Attention ka kaam kya hai?**

👉 Ye **decides karta hai ki input ke kaunse words important hai** while processing a sentence.

◇ **Multi-Head Attention ka kya fayda hai?**

👉 Ek hi input me **multiple attention mechanisms parallelly kaam karte hain** to capture different relationships.

1️⃣ Constructor (__init__) - Initialization

```
class MultiHeadAttention(nn.Module):
   def __init__(self, d_model, num_heads):
      super().__init__()
      assert d_model % num_heads == 0, "d_model must be divisible by num_heads"
self.d_model = d_model
      self.num_heads = num_heads
```

```
        self.d_k = d_model // num_heads  # Each head ka dimension
self.W_q = nn.Linear(d_model, d_model)  # Query matrix
        self.W_k = nn.Linear(d_model, d_model)  # Key matrix
        self.W_v = nn.Linear(d_model, d_model)  # Value matrix
        self.W_o = nn.Linear(d_model, d_model)  # Final output linear layer
```

Explanation

1. d_model → Input embedding ka size (usually **512** or **768**).
2. num_heads → Kitne attention heads chahiye (e.g. **8 heads**).
3. **Each head ke liye dimension**:
   ```
   self.d_k = d_model // num_heads  # e.g. 512/8 = 64
   ```
   Har **head ka size** d_k hota hai (64 in this case).
4. W_q, W_k, W_v → Ye **Query (Q), Key (K), aur Value (V) vectors generate karte hain**.
   - **Query (Q) → Kis word ko dhyan dena hai?**
   - **Key (K) → Kaunse words important hai?**
   - **Value (V) → Actual information jo pass hogi.**

2 scaled_dot_product_attention() - Compute Attention

```
def scaled_dot_product_attention(self, Q, K, V, mask=None):
    attn_scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)
    if mask is not None:
        attn_scores = attn_scores.masked_fill(mask == 0, -1e9)  # Ignore padding tokens
attn_probs = torch.softmax(attn_scores, dim=-1)  # Normalize scores
    output = torch.matmul(attn_probs, V)  # Multiply with Values
    return output
```

Explanation

1. **Dot Product between Q and K**
   ```
   attn_scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)
   ```
   - **Query aur Key ka dot product le rahe hain.**
   - **Scaling factor** 1/sqrt(d_k) ensure karta hai ki values **stable rahe** (NaN na aaye).
2. **Apply Mask (if needed)**
   ```
   if mask is not None:
       attn_scores = attn_scores.masked_fill(mask == 0, -1e9)
   ```
   - **Padding ya future words ignore karne ke liye mask use hota hai**.
3. **Softmax Normalization**
   ```
   attn_probs = torch.softmax(attn_scores, dim=-1)
   ```
   - Ye probability distribution generate karta hai.
4. **Multiply with Values**
   ```
   output = torch.matmul(attn_probs, V)
   ```
   - **Important words** ki values ko **weightage** milta hai.

3 forward() - Complete Flow

```
def forward(self, Q, K, V, mask=None):
    batch_size = Q.size(0)
Q = self.W_q(Q).view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)
    K = self.W_k(K).view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)
    V = self.W_v(V).view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)
output = self.scaled_dot_product_attention(Q, K, V, mask)
output = output.transpose(1, 2).contiguous().view(batch_size, -1, self.d_model)
    return self.W_o(output)
```

Explanation

1. **Q, K, V Generate**
   Q = self.W_q(Q).view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)
   - W_q(Q): Query generate kiya
   - .view(...): Multi-heads me reshape kiya
   - .transpose(1, 2): Batch aur heads swap kiya (shape [batch, heads, seq_len, d_k] ban gaya)
2. **Apply Scaled Dot Product Attention**
   output = self.scaled_dot_product_attention(Q, K, V, mask)
   - Ye **attention weights calculate karke values multiply karta hai**.
3. **Concatenate Multi-Heads**
   output = output.transpose(1, 2).contiguous().view(batch_size, -1, self.d_model)
   - **Heads ko combine karke wapas ek single vector banaya**.
   - .contiguous() memory ko optimize karta hai.
4. **Final Linear Layer**
   return self.W_o(output)
   - Ye final output layer hai jo next transformer block me jayega.

Example - Running Multi-Head Attention

```
# Define Attention layer
mha = MultiHeadAttention(d_model=8, num_heads=2)
# Dummy Input (batch=1, seq_len=4, d_model=8)
x = torch.rand(1, 4, 8)
# Run forward pass
output = mha(x, x, x)
print(output.shape)  # Output: [1, 4, 8]
```

What Happens Here?

1. **Q, K, V banega** (embedding ke liye projection layer lagegi).
2. **Multi-head attention compute hoga** (scaled dot-product attention).
3. **Concatenation hoke final output milega** (size d_model ka hi hoga).

Summary

Step	Explanation
1. Q, K, V Generate	Input embeddings ko query, key, aur value vectors me map kiya
2. Dot-Product Attention	Q aur K ka dot product leke attention scores nikala
3. Softmax & Value Multiplication	Softmax apply kiya aur V ke saath multiply kiya
4. Multi-Heads Apply	Multiple heads ko parallelly apply kiya
5. Concatenation & Output Linear Layer	Heads ka output combine karke linear layer se pass kiya

Final Thoughts

☑ **Multi-Head Attention se model ek hi sentence me multiple relationships detect kar sakta hai!**
☑ **Ye Transformer ka most important layer hai, jo self-attention ko enhance karta hai.**
☑ **Iske bina Transformer properly kaam nahi karega.**

📌 Example Sentence

💬 Suppose, we have this English sentence:

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

1️⃣ Input Embeddings (Converting Words → Vectors)

Har word ek vector ban jayega **(embedding size = 8)**.
Maan lo, **word embeddings** kuch aise hain:

Word	**Embedding (Size 8)**
"The"	[0.1, 0.3, 0.5, ...]
"cat"	[0.2, 0.4, 0.6, ...]
"sat"	[0.7, 0.1, 0.9, ...]
"on"	[0.3, 0.6, 0.8, ...]
"the"	[0.1, 0.3, 0.5, ...]
"mat"	[0.4, 0.7, 0.2, ...]

Iska tensor representation hoga:

```
import torch
# Batch = 1, Seq_Len = 6, d_model = 8
x = torch.rand(1, 6, 8)  # (Random embeddings)
```

2️⃣ Query (Q), Key (K), Value (V) Matrices Generate

```
import torch.nn as nn
mha = MultiHeadAttention(d_model=8, num_heads=2)  # 2 heads
Q = mha.W_q(x)  # Query
K = mha.W_k(x)  # Key
V = mha.W_v(x)  # Value
```

🖊️ **What happens?**

- Query (Q): Kis word ko **focus** karna hai?
- Key (K): Kaunse words **important** hai?
- Value (V): Jo **actual information** pass hogi.

3️⃣ Dot-Product Attention (Similarity Calculation)

```
attn_scores = torch.matmul(Q, K.transpose(-2, -1)) / (8 ** 0.5)
```

🛠️ **Example Output:**

Word	**"The"**	**"cat"**	**"sat"**	**"on"**	**"the"**	**"mat"**
The	1.0	0.7	0.5	0.3	1.0	0.4
cat	0.7	1.0	0.8	0.4	0.7	0.5
sat	0.5	0.8	1.0	0.6	0.5	0.7
on	0.3	0.4	0.6	1.0	0.3	0.8
the	1.0	0.7	0.5	0.3	1.0	0.4
mat	0.4	0.5	0.7	0.8	0.4	1.0

💡 **High Attention Values → Strong Relationship**

- "cat" ne "sat" par **zyada attention** diya (0.8) ☑️
- "on" ne "mat" par **zyada attention** diya (0.8) ☑️

4 Softmax & Weighted Sum (Final Attention Output)

```
import torch.nn.functional as F
attn_probs = F.softmax(attn_scores, dim=-1)  # Normalize
output = torch.matmul(attn_probs, V)
```

🛠 **Example Softmax Output:**

Word	**"The"**	**"cat"**	**"sat"**	**"on"**	**"the"**	**"mat"**
The	0.25	0.20	0.15	0.10	0.25	0.05
cat	0.20	0.30	0.25	0.10	0.10	0.05
sat	0.15	0.25	0.30	0.15	0.05	0.10
on	0.10	0.10	0.15	0.35	0.10	0.20
the	0.25	0.20	0.15	0.10	0.25	0.05
mat	0.05	0.05	0.10	0.20	0.05	0.55

Kaise interpret kare?

- "cat" ka **focus zyada hai "sat" (0.25) aur khud par (0.30)**
- "on" ka **focus "mat" (0.20) aur khud par (0.35)**

🥏 **Yahi reason hai ki Self-Attention sentence me relationships ko samajhne me madad karta hai!**

5 Multi-Head Attention (Parallel Heads)

Ek attention **multiple heads me parallelly kaam karta hai**.

- ◇ **1st Head** → Focuses on **subject-object relations**
- ◇ **2nd Head** → Focuses on **positional dependencies**

```
output = output.view(1, 6, 8)  # Merge heads
final_output = mha.W_o(output)  # Final linear layer
```

💡 **Final Output bhi 8-dimension ka hota hai (same as input size).**

✳ Summary

Step	**Explanation**
1. Create Q, K, V	Words ko Query, Key, aur Value vectors me convert kiya
2. Dot-Product Attention	Words ka similarity score calculate kiya
3. Softmax Normalize	Scores ko probabilities me convert kiya
4. Weighted Sum with Values	Important words ka final output nikala
5. Multi-Head Attention	Multiple perspectives add kiye

💡 Real-World Use Case

🛠 **Machine Translation Example (English → German)**

English Input: "The cat sat on the mat."

German Output: "Die Katze saß auf der Matte."

Multi-Head Attention ensure karega ki: ☑ "The" ka **attention "Die" par ho**

☑ "cat" ka **attention "Katze" par ho**

☑ "sat" ka **attention "saß" par ho**

💡 **Without Attention, ye alignment problematic ho sakti thi.** 🦉

🔥 Final Thoughts

☑ **Multi-Head Attention Transformer ka core hai!**
☑ **Words ke relationships detect karne me help karta hai.**
☑ **Ye Self-Attention ko aur powerful banata hai.**

Class Overview:

The **feedforward network** is an important component of the Transformer architecture. It is applied **independently** to each position in the sequence (hence "position-wise"). This means that the same feedforward neural network is applied to each token in the sequence, one at a time, without any interaction between them.
Now, let's dive into the code:

```
class PositionwiseFeedforward(nn.Module):
   def __init__(self, d_model, d_ff):
      super().__init__()
      self.fc1 = nn.Linear(d_model, d_ff)  # First linear layer
      self.fc2 = nn.Linear(d_ff, d_model)  # Second linear layer
      self.relu = nn.ReLU()  # ReLU activation function
def forward(self, x):
      return self.fc2(self.relu(self.fc1(x)))  # Apply two linear layers with ReLU activation
```

Explanation of Code:

1. **Constructor (__init__ method):**
   - d_model: This is the dimensionality of the input (and output) of the model. In the Transformer, this corresponds to the size of the input vector for each token.
   - d_ff: This is the dimensionality of the **intermediate layer** in the feedforward network, i.e., how many neurons there will be in the hidden layer between the two linear transformations.
     The **feedforward network** consists of two linear layers:
   - **self.fc1**: The first linear transformation takes the input of dimension d_model and transforms it to the higher dimensional space of d_ff.
   - **self.fc2**: The second linear transformation projects the result from d_ff back to d_model.
   - **self.relu**: A ReLU activation function is applied between the two linear transformations. This introduces non-linearity into the model, which is crucial for learning complex patterns.

1. **Forward Method:**
   - The forward method defines how the input tensor x (which is passed through this layer) is processed.
   - **Flow:**
     1. The input x passes through the first linear layer fc1, which transforms the input from d_model dimensions to d_ff dimensions.
     2. The output of fc1 is passed through the **ReLU** activation function (self.relu). This step adds non-linearity to the transformation.
     3. Then, the activated output is passed through the second linear layer fc2, which brings it back to the original dimension d_model.
        So, this feedforward layer takes the input, applies a transformation, adds non-linearity, and transforms it again back to the original dimension.

Role of Positionwise Feedforward in Transformer:

In the Transformer model:

- This **position-wise feedforward layer** is applied **independently** to each token (or position) in the sequence.
- The reason it's called "position-wise" is that each token is treated separately and is transformed

independently of the others, unlike the attention layers, which allow tokens to attend to each other.

This operation is repeated for each token in the sequence after the **multi-head attention** block. This allows each token to have its representation refined.

Summary:

- **PositionwiseFeedforward** is a simple two-layer feedforward network used in the Transformer.
- It processes each token's representation independently, applying a non-linear transformation.
- It consists of two linear layers with a **ReLU** activation function between them.

Let me know if you need more details or examples on this! 😊

This class EncoderLayer represents one **layer of the encoder** in the Transformer model. The encoder layer is the building block of the **transformer encoder**, and it consists of two main components: a **multi-head self-attention layer** and a **position-wise feedforward layer**.

Let's break down the code step by step.

Class Breakdown:

```
class EncoderLayer(nn.Module):
  def __init__(self, d_model, num_heads, d_ff, dropout):
    super().__init__()
    self.self_attn = MultiHeadAttention(d_model, num_heads)  # Multi-head attention layer
    self.feed_forward = PositionwiseFeedforward(d_model, d_ff)  # Position-wise feedforward layer
    self.norm1 = nn.LayerNorm(d_model)  # Layer normalization for the first component (attention)
    self.norm2 = nn.LayerNorm(d_model)  # Layer normalization for the second component (feedforward)
    self.dropout = nn.Dropout(dropout)  # Dropout to prevent overfitting
def forward(self, x, mask):
    # Step 1: Self-attention
    attn_output = self.self_attn(x, x, x, mask)  # Query, Key, Value all come from 'x' (self-attention)
    x = self.norm1(x + self.dropout(attn_output))  # Apply residual connection, dropout, and normalization
# Step 2: Feedforward layer
    ff_output = self.feed_forward(x)  # Apply position-wise feedforward network
    x = self.norm2(x + self.dropout(ff_output))  # Apply residual connection, dropout, and normalization
return x
```

Explanation of Components:

1. **self.self_attn = MultiHeadAttention(d_model, num_heads)**
   - This initializes the **multi-head attention layer**. It takes the input x (which represents the tokens' embeddings) as the **query**, **key**, and **value**. The self-attention mechanism allows each token to attend to all other tokens in the sequence.
2. **self.feed_forward = PositionwiseFeedforward(d_model, d_ff)**
   - This initializes the **position-wise feedforward layer**. It is applied after the attention mechanism, independently to each token, as explained earlier.
3. **self.norm1 = nn.LayerNorm(d_model)**
   - **Layer normalization** is applied after the **self-attention** block. Layer normalization helps stabilize and speed up training by normalizing the activations.
   - It ensures that the output of the attention layer has zero mean and unit variance across each batch.
4. **self.norm2 = nn.LayerNorm(d_model)**
   - Similarly, **layer normalization** is applied after the **feedforward layer** for stability.
5. **self.dropout = nn.Dropout(dropout)**

- **Dropout** is a regularization technique applied to the outputs of the attention and feedforward layers to prevent overfitting during training. dropout is the probability of setting a unit to zero during training.

Forward Method:

The forward method defines how the data flows through the encoder layer. Here's how it works:

1. **Self-Attention**:
   - The input x passes through the **multi-head attention** mechanism (self.self_attn(x, x, x, mask)).
   - The **mask** can be used to prevent certain tokens from attending to others (e.g., padding tokens or future tokens).
   - The result of the attention (attn_output) is then added to the original input x as a **residual connection**. This allows the model to learn both the attention output and the original input in parallel.
   - **Dropout** is applied to the attention output, and then **layer normalization** is applied on top of the sum (x + attn_output).
2. **Feedforward Layer**:
   - The output from the attention mechanism (after normalization) passes through the **position-wise feedforward layer** (self.feed_forward(x)).
   - Again, a **residual connection** is applied by adding the original input x to the output of the feedforward layer. This helps the network retain information from earlier stages.
   - **Dropout** is applied again, and **layer normalization** is done on the final output.
3. **Return**:
   - The final output x after the attention and feedforward operations is returned, which is passed to the next layer of the encoder (or used for further processing, depending on the architecture).

Summary:

- The EncoderLayer consists of two main components:
  1. **Self-Attention**: Helps the model focus on different parts of the input sequence.
  2. **Feedforward Layer**: Processes each token independently after attention.
- **Residual Connections**: After each of the two operations, the input is added to the output (residual connection). This helps with **gradient flow** during backpropagation and prevents vanishing/exploding gradients.
- **Layer Normalization**: Normalizes the output after each operation (attention and feedforward) for stable training.
- **Dropout**: Applied to prevent overfitting during training.

This architecture is repeated in the **encoder** stack, with each layer applying self-attention and feedforward processing.

This class DecoderLayer implements a **decoder layer** for the **Transformer model**. The decoder in the Transformer consists of three main components:

1. **Self-attention mechanism** (similar to the encoder but with a mask to prevent future tokens from attending to previous ones)
2. **Cross-attention mechanism** (to attend to the encoder's output)
3. **Feedforward layer** (applied independently to each position in the sequence)

Let's break down the code step by step to understand how it works.

Class Breakdown:

```
class DecoderLayer(nn.Module):
  def __init__(self, d_model, num_heads, d_ff, dropout):
    super().__init__()
    self.self_attn = MultiHeadAttention(d_model, num_heads)  # Self-attention mechanism
```

```
        self.cross_attn = MultiHeadAttention(d_model, num_heads)  # Cross-attention mechanism
        self.feed_forward = PositionwiseFeedforward(d_model, d_ff)  # Feedforward network
        self.norm1 = nn.LayerNorm(d_model)  # Layer normalization after self-attention
        self.norm2 = nn.LayerNorm(d_model)  # Layer normalization after cross-attention
        self.norm3 = nn.LayerNorm(d_model)  # Layer normalization after feedforward
        self.dropout = nn.Dropout(dropout)  # Dropout to prevent overfitting
def forward(self, x, enc_output, src_mask, trg_mask):
        # Step 1: Self-attention (using the target sequence with masking)
        attn_output = self.self_attn(x, x, x, trg_mask)  # Query, Key, Value = x (self-attention)
        x = self.norm1(x + self.dropout(attn_output))  # Apply residual connection, dropout, and
normalization
# Step 2: Cross-attention (using the encoder output with source masking)
        attn_output = self.cross_attn(x, enc_output, enc_output, src_mask)  # Query = x, Key/Value =
encoder output
        x = self.norm2(x + self.dropout(attn_output))  # Apply residual connection, dropout, and
normalization
# Step 3: Feedforward layer
        ff_output = self.feed_forward(x)  # Apply position-wise feedforward network
        x = self.norm3(x + self.dropout(ff_output))  # Apply residual connection, dropout, and normalization
return x
```

Explanation of Components:

1. **self.self_attn = MultiHeadAttention(d_model, num_heads)**:
    - This initializes the **multi-head self-attention** layer. It processes the target sequence (x) to allow the model to attend to all previous tokens in the sequence (during training) while preventing attending to future tokens (via masking).
    - This step ensures that during training, the model doesn't cheat by looking at future tokens.
2. **self.cross_attn = MultiHeadAttention(d_model, num_heads)**:
    - This is the **cross-attention** mechanism. Here, the model attends to the output of the **encoder** (enc_output) while processing the target sequence (x).
    - The **query** comes from the target sequence (x), while the **key** and **value** come from the encoder's output (enc_output).
    - This mechanism allows the decoder to focus on relevant parts of the input (source) sequence.
3. **self.feed_forward = PositionwiseFeedforward(d_model, d_ff)**:
    - This is the **position-wise feedforward network**, applied to each token independently, as described in previous explanations.
    - This component is responsible for learning more complex representations after the attention steps.
4. **self.norm1, self.norm2, self.norm3**:
    - **Layer normalization** is applied after each of the three steps:
        1. After self-attention (to normalize the result of self-attention)
        2. After cross-attention (to normalize the result of cross-attention)
        3. After the feedforward network (to normalize the output of the feedforward network)
    - This normalization helps stabilize training and allows for faster convergence.
5. **self.dropout**:
    - **Dropout** is applied to prevent overfitting during training. It randomly sets some values to zero in the output, forcing the model to generalize better.

Forward Method:

The forward method defines the data flow through the decoder layer. Here's how it works:

1. **Self-Attention**:

- The input x (the target sequence) passes through the **self-attention** mechanism.
- The **trg_mask** (target mask) is used to prevent attending to future tokens during training, ensuring that the model predicts each token only using previous tokens (auto-regressive property).
- After applying the attention, a **residual connection** is added (x + attn_output), followed by **dropout** and **layer normalization**.

2. **Cross-Attention**:
   - The output of the self-attention mechanism (after normalization) is passed to the **cross-attention** mechanism.
   - Here, the **encoder output** (enc_output) is used as the **key** and **value**, while the **target sequence** (x) is used as the **query**.
   - This allows the decoder to focus on relevant parts of the source (input) sequence.
   - Again, a **residual connection**, followed by **dropout** and **layer normalization**, is applied.
3. **Feedforward Layer**:
   - The output of the cross-attention mechanism (after normalization) passes through the **position-wise feedforward network**.
   - This further refines the representations for each token in the sequence.
   - A final **residual connection**, followed by **dropout** and **layer normalization**, is applied.
4. **Return**:
   - The final output x is returned, which represents the processed target sequence after attention and feedforward processing. This output is passed to the next decoder layer (or the output layer if it's the last decoder layer).

Summary:

- **Self-Attention**: Allows the decoder to focus on relevant tokens in the target sequence while ensuring that future tokens aren't attended to (via masking).
- **Cross-Attention**: Allows the decoder to attend to relevant parts of the source sequence (from the encoder).
- **Feedforward Layer**: Refines the representations of each token independently.
- **Residual Connections**: Help with gradient flow and prevent vanishing/exploding gradients.
- **Layer Normalization**: Stabilizes the training by normalizing the output at each stage.
- **Dropout**: Prevents overfitting by randomly setting values to zero during training.

This architecture is repeated across multiple decoder layers to process and refine the target sequence's representation. Each decoder layer refines the sequence based on both the target sequence (via self-attention) and the source sequence (via cross-attention).

This code defines the Transformer model, which is the complete architecture that combines the encoder and decoder layers with additional components such as embeddings, positional encoding, and the final output layer.

Class Breakdown:

__init__ Method:

This method initializes all the components of the Transformer model, including embeddings, positional encoding, layers, and final output layer.

```
def __init__(self, src_vocab_size, trg_vocab_size, d_model, num_heads, num_layers, d_ff,
max_seq_length, dropout):
    super().__init__()

    # Embeddings for source and target sequences
    self.encoder_embedding = nn.Embedding(src_vocab_size, d_model)  # Source sequence embedding
    self.decoder_embedding = nn.Embedding(trg_vocab_size, d_model)  # Target sequence embedding

    # Positional encoding to add positional information to the embeddings
```

```
    self.positional_encoding = PositionalEncoding(d_model, max_seq_length)

    # Stacks of encoder and decoder layers
    self.encoder_layers = nn.ModuleList([EncoderLayer(d_model, num_heads, d_ff, dropout) for _ in range(num_layers)])
    self.decoder_layers = nn.ModuleList([DecoderLayer(d_model, num_heads, d_ff, dropout) for _ in range(num_layers)])

    # Final output layer (to produce predictions for each token)
    self.fc_out = nn.Linear(d_model, trg_vocab_size)

    # Dropout to prevent overfitting
    self.dropout = nn.Dropout(dropout)

    # Scaling factor for the embeddings
    self.scale = torch.sqrt(torch.FloatTensor([d_model]))
```

Explanation of Each Component:

1. **encoder_embedding and decoder_embedding**:
   - These are **embedding layers** for the source (src) and target (trg) sequences. These layers transform input tokens (words) into dense vector representations of size d_model.
2. **positional_encoding**:
   - **Positional Encoding** is added to the embeddings to incorporate the order of tokens in the sequence. Since the Transformer doesn't inherently have any sense of order, this encoding adds information about the position of each token in the sequence.
3. **encoder_layers and decoder_layers**:
   - These are **lists of encoder and decoder layers**, which are instances of the EncoderLayer and DecoderLayer classes, respectively. The number of layers in the encoder and decoder is determined by num_layers.
4. **fc_out**:
   - This is a **fully connected layer** that maps the output of the decoder to the target vocabulary size (trg_vocab_size). It produces logits for each word in the vocabulary at each position in the sequence.
5. **dropout**:
   - A **dropout layer** is used for regularization, which randomly drops a proportion of neurons during training to prevent overfitting.
6. **scale**:
   - The embeddings are scaled by a factor of the square root of the model dimension (d_model). This scaling helps stabilize the training.

generate_mask Method:

```
def generate_mask(self, src, trg):
    # Mask for the source sequence (to ignore padding tokens in the source)
    src_mask = (src != SRC_VOCAB[PAD_TOKEN]).unsqueeze(1).unsqueeze(2)

    # Mask for the target sequence (to ignore padding tokens in the target)
    trg_mask = (trg != TRG_VOCAB[PAD_TOKEN]).unsqueeze(1).unsqueeze(3)

    # Create a "no-peak" mask for the target sequence (to prevent attending to future tokens in the decoder)
    seq_length = trg.shape[1]
    nopeak_mask = (1 - torch.triu(torch.ones(1, seq_length, seq_length), diagonal=1)).bool()
    trg_mask = trg_mask & nopeak_mask  # Combine padding mask and no-peak mask
```

```
return src_mask, trg_mask
```

- **src_mask**: This mask ensures that padding tokens in the source sequence are ignored during attention computation in both the encoder and decoder.
- **trg_mask**: This mask ensures that padding tokens in the target sequence are ignored during attention computation in the decoder.
- **nopeak_mask**: This mask ensures that during training, the decoder can only attend to earlier positions and not future positions in the target sequence (causing the model to generate tokens in an auto-regressive manner). It's a key part of the **causal masking** in the decoder.

forward Method:

```
def forward(self, src, trg):
    src_mask, trg_mask = self.generate_mask(src, trg)

    # Embedding and positional encoding for the source and target sequences
    src_embedded = self.dropout(self.positional_encoding(self.encoder_embedding(src) * self.scale))
    trg_embedded = self.dropout(self.positional_encoding(self.decoder_embedding(trg) * self.scale))

    # Encoder forward pass
    enc_output = src_embedded
    for enc_layer in self.encoder_layers:
        enc_output = enc_layer(enc_output, src_mask)

    # Decoder forward pass
    dec_output = trg_embedded
    for dec_layer in self.decoder_layers:
        dec_output = dec_layer(dec_output, enc_output, src_mask, trg_mask)

    # Final output (logits for each token in the target sequence)
    output = self.fc_out(dec_output)
    return output
```

1. **Generate Masks**:
    - The generate_mask function is called to create masks for the source and target sequences.
2. **Embedding and Positional Encoding**:
    - The input src (source) and trg (target) sequences are first passed through their respective embedding layers and then multiplied by the scaling factor (scale).
    - Positional encoding is added to the embeddings to introduce information about the position of each token in the sequence.
    - **Dropout** is applied for regularization.
3. **Encoder Forward Pass**:
    - The embedded source sequence is passed through the stack of **encoder layers**. The src_mask ensures that padding tokens are ignored during attention calculations.
4. **Decoder Forward Pass**:
    - The embedded target sequence is passed through the stack of **decoder layers**.
    - Both the encoder output (enc_output) and the masks (src_mask and trg_mask) are passed to the decoder layers.
5. **Final Output**:
    - The decoder output is passed through the final **fully connected layer (fc_out)** to produce the logits for each token in the target vocabulary.
    - These logits are used to predict the next token in the sequence.

Summary:

- The Transformer model consists of **embedding layers** for both the source and target sequences, **positional encoding**, and a stack of **encoder** and **decoder** layers.

- The generate_mask function creates the necessary masks to ignore padding and prevent the model from attending to future tokens during training.
- The forward pass involves embedding the input sequences, passing them through the encoder and decoder layers, and producing logits for the target sequence.

This Transformer model can be trained to perform tasks like machine translation, where the goal is to translate a source sentence into a target sentence.

Let's break down the code step by step:

Optimizer Definition:

optimizer = optim.Adam(model.parameters(), lr=0.0001, betas=(0.9, 0.98), eps=1e-9)

- **optim.Adam**: This is the **Adam optimizer** used for training the model. Adam is an adaptive learning rate optimization algorithm that computes individual learning rates for different parameters based on the first and second moments of the gradients.
- **model.parameters()**: This refers to the parameters of the model (which is your Transformer model). These are the weights and biases that will be optimized during training.
- **lr=0.0001**: This is the **learning rate**, which controls how much the model's weights are adjusted with respect to the loss gradient. A smaller learning rate can lead to more stable training but may take longer to converge.
- **betas=(0.9, 0.98)**: These are the **beta values** used in the calculation of running averages for the first and second moments of the gradients. In Adam, beta1 (0.9) is typically used for the moving average of the first moment (mean), and beta2 (0.98) is used for the second moment (uncentered variance).
- **eps=1e-9**: This is a small constant added to avoid division by zero during the computation of the learning rate update. It's a safeguard to ensure stability.

Loss Function Definition:

PAD_IDX = SRC_VOCAB[PAD_TOKEN]
criterion = nn.CrossEntropyLoss(ignore_index=PAD_IDX)

- **PAD_IDX = SRC_VOCAB[PAD_TOKEN]**:
  - This gets the index of the padding token (<pad>) in the source vocabulary (SRC_VOCAB). Padding tokens are used to make all sentences in a batch have the same length, but they don't contribute to the actual content of the sentence.
- **criterion = nn.CrossEntropyLoss(ignore_index=PAD_IDX)**:
  - **CrossEntropyLoss** is a commonly used loss function for multi-class classification tasks. In the context of sequence generation (like translation), it computes the loss for each predicted token (logits) against the true token labels in the target sequence.
  - **ignore_index=PAD_IDX**: This ensures that padding tokens (which do not carry any meaningful information) are ignored during the loss calculation. That means if the model predicts a padding token, it won’t contribute to the loss and won't affect training.

Summary:

- The **Adam optimizer** is used to update the parameters of the Transformer model during training with an appropriate learning rate and gradient moment settings.
- The **CrossEntropyLoss** is used to measure the performance of the model by comparing its predictions to the actual target sequence. Padding tokens are ignored during loss computation to ensure that they don't negatively affect the model's training.

With this setup, you can now proceed to train the model by using these components.

Let's break down the code step by step:

Collate Function for DataLoader:

```
def collate_fn(batch):
    src_batch, trg_batch = [], []
    for sample in batch:
```

```
    src_batch.append(torch.tensor([SRC_VOCAB.get(token, SRC_VOCAB[UNK_TOKEN]) for token in [SOS_TOKEN] + sample['src'] + [EOS_TOKEN]]))
    trg_batch.append(torch.tensor([TRG_VOCAB.get(token, TRG_VOCAB[UNK_TOKEN]) for token in [SOS_TOKEN] + sample['trg'] + [EOS_TOKEN]]))
src_batch = pad_sequence(src_batch, padding_value=SRC_VOCAB[PAD_TOKEN])
  trg_batch = pad_sequence(trg_batch, padding_value=TRG_VOCAB[PAD_TOKEN])
return src_batch.transpose(0, 1), trg_batch.transpose(0, 1)
```

- **Purpose**: This is a **collate function** used by the DataLoader to batch the data.
    - It takes a list of individual samples (each containing source and target sentences) and processes them into batches.
    - **Padding**: It pads the sequences so that all sequences in the batch are the same length. Padding is done using the <pad> token from the vocabulary.
- **Steps**:
    - **Add special tokens**: Adds SOS_TOKEN and EOS_TOKEN to the source (src) and target (trg) sentences.
    - **Convert tokens to indices**: The sentences are converted to indices using the SRC_VOCAB and TRG_VOCAB vocabularies. If a token is not found, it is replaced with <unk> (unknown token).
    - **Pad sequences**: Sequences are padded using pad_sequence so they are all of equal length within the batch.
    - **Return**: It returns the padded source and target batches, transposed to fit the model's input requirements.

Training Function:

```
def train(model, iterator, optimizer, criterion, clip):
  model.train()
  epoch_loss = 0
  print(len(iterator))
  for src, trg in tqdm(iterator, desc="Training", leave=False):
    optimizer.zero_grad()
output = model(src, trg[:, :-1])
output_dim = output.shape[-1]
output = output.contiguous().view(-1, output_dim)
    trg = trg[:, 1:].contiguous().view(-1)
loss = criterion(output, trg)
loss.backward()
torch.nn.utils.clip_grad_norm_(model.parameters(), clip)
optimizer.step()
epoch_loss += loss.item()
return epoch_loss / len(iterator)
```

- **Purpose**: This function trains the model for one epoch (one pass through the entire dataset).
- **Steps**:
    - **Model in training mode**: model.train() puts the model in training mode (so dropout layers, etc., are active).
    - **Iterate through batches**: For each batch (src, trg), it performs the following:
        - **Forward pass**: Passes the source and target sequences (excluding the last token in target, trg[:, :-1]) through the model.
        - **Reshape outputs**: The output tensor is reshaped and flattened to calculate the loss for each token.
        - **Loss calculation**: The loss is calculated using the criterion (CrossEntropyLoss).
        - **Backpropagation**: The loss is backpropagated, and the model parameters are updated.
        - **Gradient clipping**: To avoid exploding gradients, gradient clipping is applied

(clip_grad_norm_).
        - **Update weights**: The optimizer steps forward and updates the model's weights.
    - The function returns the average loss for the epoch.

Evaluation Function:

```
def evaluate(model, iterator, criterion):
  model.eval()
  epoch_loss = 0
  with torch.no_grad():
    for i, batch in enumerate(iterator):
      src, trg = batch
output = model(src, trg[:, :-1])
output_dim = output.shape[-1]
output = output.contiguous().view(-1, output_dim)
      trg = trg[:, 1:].contiguous().view(-1)
loss = criterion(output, trg)
epoch_loss += loss.item()
return epoch_loss / len(iterator)
```

- **Purpose**: This function evaluates the model performance on the validation or test set. It's similar to the training loop but without backpropagation.
- **Steps**:
    - **Model in evaluation mode**: model.eval() sets the model to evaluation mode (disables dropout).
    - **Iterate through validation/test batches**: For each batch, it calculates the loss the same way as in training, but without updating model weights (no backpropagation).
    - **No gradient computation**: torch.no_grad() disables the computation of gradients, which saves memory and computation time during inference.
    - It returns the average loss for the epoch.

Translation Function:

```
def translate_sentence(sentence, src_vocab, trg_vocab, model, device, max_len=50):
  model.eval()
tokens = [SOS_TOKEN] + tokenize_de(sentence) + [EOS_TOKEN]
src_indexes = [src_vocab.get(token, src_vocab[UNK_TOKEN]) for token in tokens]
src_tensor = torch.LongTensor(src_indexes).unsqueeze(0).to(device)
src_mask = model.generate_mask(src_tensor, src_tensor)
with torch.no_grad():
    enc_src = model.encoder_embedding(src_tensor)
    for enc_layer in model.encoder_layers:
      enc_src = enc_layer(enc_src, src_mask[0])
trg_indexes = [trg_vocab[SOS_TOKEN]]
for i in range(max_len):
    trg_tensor = torch.LongTensor(trg_indexes).unsqueeze(0).to(device)
trg_mask = model.generate_mask(src_tensor, trg_tensor)
with torch.no_grad():
      output = model.decoder_embedding(trg_tensor)
      for dec_layer in model.decoder_layers:
        output = dec_layer(output, enc_src, src_mask[0], trg_mask[1])
      output = model.fc_out(output)
pred_token = output.argmax(2)[:,-1].item()
trg_indexes.append(pred_token)
if pred_token == trg_vocab[EOS_TOKEN]:
```

```
        break
trg_tokens = [list(trg_vocab.keys())[list(trg_vocab.values()).index(i)] for i in trg_indexes]
return trg_tokens[1:-1]
```

- **Purpose**: This function translates a given sentence from the source language to the target language using the trained model.
- **Steps**:
  - **Tokenize and prepare source sentence**: The source sentence is tokenized, and special tokens (SOS_TOKEN, EOS_TOKEN) are added. It is then converted into token indices using the src_vocab.
  - **Generate source mask**: A mask is generated to prevent attention to padding tokens during encoding.
  - **Encoder**: The input sequence is passed through the encoder layers.
  - **Initialize target sequence**: The target sequence starts with the SOS_TOKEN.
  - **Iteratively generate target tokens**: The model generates one token at a time. The target sequence is extended by the predicted token.
  - **Stop at EOS**: The process stops when the model predicts the EOS_TOKEN.
  - **Convert token indices to words**: Finally, the indices of the generated tokens are mapped back to words using the target vocabulary.

Summary:

- **Collate Function**: Prepares batches by adding special tokens and padding sequences.
- **Training Function**: Trains the model by performing forward passes, loss calculation, backpropagation, and optimization.
- **Evaluation Function**: Evaluates the model on validation/test data without updating the model parameters.
- **Translation Function**: Uses the trained model to generate translations for a given input sentence.

The vocab (vocabulary) is critical because it is the bridge between words (or tokens) and their corresponding integer indices, which the model can process. Let's look at **where and how the vocab is used**:

Where is vocab used in the code?

1. **In the Collate Function:** The vocab is used when preparing the batches of data (in the collate_fn function) before passing them into the model.
   src_batch.append(torch.tensor([SRC_VOCAB.get(token, SRC_VOCAB[UNK_TOKEN]) for token in [SOS_TOKEN] + sample['src'] + [EOS_TOKEN]]))
   trg_batch.append(torch.tensor([TRG_VOCAB.get(token, TRG_VOCAB[UNK_TOKEN]) for token in [SOS_TOKEN] + sample['trg'] + [EOS_TOKEN]]))
   - **Explanation**:
     - **SRC_VOCAB.get(token, SRC_VOCAB[UNK_TOKEN])**: For each token in the source sentence (with added SOS_TOKEN and EOS_TOKEN), we look up the token in the SRC_VOCAB dictionary. If the token is not found, we replace it with the <unk> token index (using SRC_VOCAB[UNK_TOKEN]).
     - **TRG_VOCAB.get(token, TRG_VOCAB[UNK_TOKEN])**: Similarly, for the target sentence, we look up the token in the TRG_VOCAB dictionary. If the token is not found, it's replaced with the <unk> token index.
2. **During Translation (translate_sentence function):** The vocab is used to convert tokens into their respective indices (before feeding them into the model) and then convert the predicted indices back into tokens.
   src_indexes = [src_vocab.get(token, src_vocab[UNK_TOKEN]) for token in tokens]
   - **Explanation**: The sentence is tokenized, and each token (including SOS_TOKEN and EOS_TOKEN) is converted into an index using src_vocab (for the source language). This is what the model understands—numerical representations of words.

After the model predicts indices (for the target language), we use the target vocab to convert those indices back into words:
trg_tokens = [list(trg_vocab.keys())[list(trg_vocab.values()).index(i)] for i in trg_indexes]

- **Explanation**: This part maps the predicted target token indices (trg_indexes) back to their corresponding words using trg_vocab.

3. **When Training:** In the training loop (both train and evaluate functions), vocab is indirectly used in the loss calculation:
loss = criterion(output, trg)
   - The output from the model contains predicted indices for the target sequence. These predicted indices are compared to the **true target indices** (trg) to calculate the loss.
   - The vocab (especially TRG_VOCAB) is used to convert tokens to indices before the model processes them and to map predicted indices back to tokens during evaluation or translation.

Why is vocab important in this context?

- **Tokens to Indices**: The neural network can only process numerical values, so we need to convert words (tokens) into numerical representations (indices). This conversion is done using vocab.
- **Handling Unknown Tokens**: The vocab allows us to handle unknown words by mapping them to a special <unk> token.
- **Decoding the Output**: The model generates indices, and the vocab helps us map those indices back to readable words for evaluation or translation.

Summary:

- **SRC_VOCAB** and **TRG_VOCAB** are used to **convert words into indices** (before feeding them to the model) and **convert predicted indices back into words** (after the model generates them).
- **In the collate function**, vocab helps prepare the batches.
- **In the training and evaluation loops**, vocab helps convert between tokens and indices to calculate loss.
- **In the translation function**, vocab helps convert tokens into indices before passing them to the model and then converts predicted indices back to readable words.

The code you've shared implements the **training loop** for training a Transformer-based model for machine translation. Let me break it down for you step by step:

1. Hyperparameters Setup

N_EPOCHS = 1
CLIP = 1.0
BATCH_SIZE = 32

- **N_EPOCHS**: Number of training epochs (1 in this case). This means the model will go through the entire training data once.
- **CLIP**: The gradient clipping value. This helps avoid exploding gradients by limiting the size of gradients during backpropagation.
- **BATCH_SIZE**: Number of examples in each batch when training.

2. Data Loaders

train_dataloader = DataLoader(train_data, batch_size=BATCH_SIZE, shuffle=True, collate_fn=collate_fn)
val_dataloader = DataLoader(val_data, batch_size=BATCH_SIZE, collate_fn=collate_fn)

- **DataLoader**: This is used to load the data in batches.
   - **train_dataloader**: Loads the training data in batches of size BATCH_SIZE and shuffles the data.
   - **val_dataloader**: Loads the validation data in batches. It does not shuffle the data as it's only used for evaluation during training.
- **collate_fn**: This is used to prepare batches. It takes care of padding sequences to ensure all

sentences in a batch are of the same length (since transformer models require fixed-length inputs).

3. Training Loop (For Each Epoch)

```
for epoch in range(N_EPOCHS):
   start_time = time.time()
train_loss = train(model, train_dataloader, optimizer, criterion, CLIP)
   valid_loss = evaluate(model, val_dataloader, criterion)
end_time = time.time()
```

- **start_time** and **end_time**: Measure the time taken to complete one epoch.
- The loop runs for the number of epochs defined by N_EPOCHS.
- During each epoch:
  - **train_loss**: The loss computed on the training data by calling the train function.
  - **valid_loss**: The loss computed on the validation data by calling the evaluate function.

4. Saving the Best Model

```
if valid_loss < best_valid_loss:
   best_valid_loss = valid_loss
   torch.save(model.state_dict(), 'transformer-translation-model.pt')
```

- If the current **valid_loss** (validation loss) is lower than the **best_valid_loss**, the model is saved using torch.save(model.state_dict(), 'transformer-translation-model.pt'). This ensures that the best-performing model (with the lowest validation loss) is saved and can be loaded later for inference.

5. Logging Epoch Information

```
epoch_mins, epoch_secs = divmod(end_time - start_time, 60)
print(f'Epoch: {epoch+1:02} | Time: {epoch_mins}m {epoch_secs}s')
print(f'\tTrain Loss: {train_loss:.3f} | Train PPL: {math.exp(train_loss):7.3f}')
print(f'\t Val. Loss: {valid_loss:.3f} |  Val. PPL: {math.exp(valid_loss):7.3f}')
```

- **epoch_mins, epoch_secs**: These compute the total time for the epoch (in minutes and seconds).
- The loss values are printed for both **training** and **validation** sets.
  - **PPL (Perplexity)** is printed as math.exp(train_loss) and math.exp(valid_loss). Perplexity is a common metric for evaluating language models. It's the exponentiation of the cross-entropy loss, and a lower perplexity means better performance.

6. Loading the Best Model

```
model.load_state_dict(torch.load('transformer-translation-model.pt'))
```

- After training, the **best model** (the one with the lowest validation loss) is loaded using torch.load. This model is now ready for evaluation or inference.

Key Takeaways:

- **Training and Validation Loss**: During each epoch, the model is trained on the training set and evaluated on the validation set.
- **Model Saving**: The best model (based on validation loss) is saved.
- **Logging**: Loss and perplexity metrics are printed for both training and validation.
- **Loading the Best Model**: After training, the best model is loaded for further evaluation or inference.

In summary:

This training loop trains your Transformer model for a defined number of epochs. It tracks the training loss and validation loss, and after each epoch, it saves the best model if it improves on the validation data. The final model with the lowest validation loss is loaded and saved for future use.