Rushil Dhingra
Axios - Machine Learning Wing
29 August, 2025

Neural Machine Translation: Hinglish → English Transformer

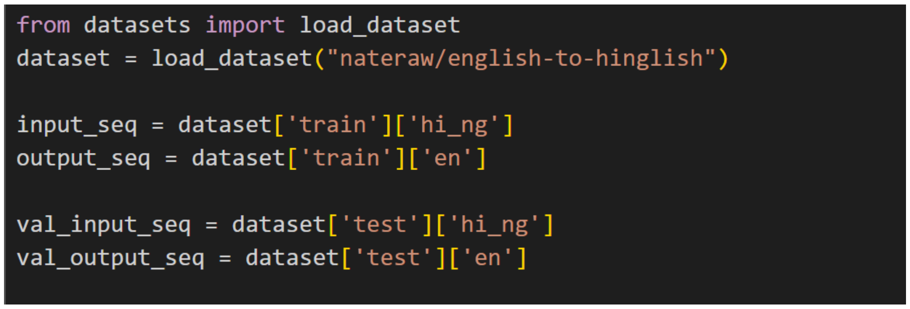
It all started when I dived into the world of Transformers and stumbled upon the research paper *"Attention Is All You Need."* The simplicity yet power of the **self-attention mechanism** really impressed me — it felt like a complete shift in how we think about language and sequence to sequence processing models. Out of curiosity (and a bit of obsession), I decided to implement the entire architecture from scratch in **PyTorch** and train it on an **Hinglish → English translation task**. This report explains that journey: from taking the first steps to fully implementing the Transformers architecture in PyTorch from scratch.

1. Problem Statement

The goal of this project was to build a **Neural Machine Translation (NMT) model** capable of translating **Hinglish** into **English**. Hinglish is an inherently noisy, code-mixed language, making it a challenging NLP task.

To tackle this, I built a **Transformer-based sequence-to-sequence model** from scratch using **PyTorch**, trained on the **nateraw/english-to-hinglish** dataset from Hugging Face.

I start by loading the dataset into my working directory using the load_dataset function from the Hugging Face datasets library



```
from datasets import load_dataset
dataset = load_dataset("nateraw/english-to-hinglish")

input_seq = dataset['train']['hi_ng']
output_seq = dataset['train']['en']

val_input_seq = dataset['test']['hi_ng']
val_output_seq = dataset['test']['en']
```

2. Tech Stack

Programming Language

- Python 3.x
- PyTorch
- Numpy

Text Preprocessing Modules

- BeautifulSoup
- re (Regular Expressions)

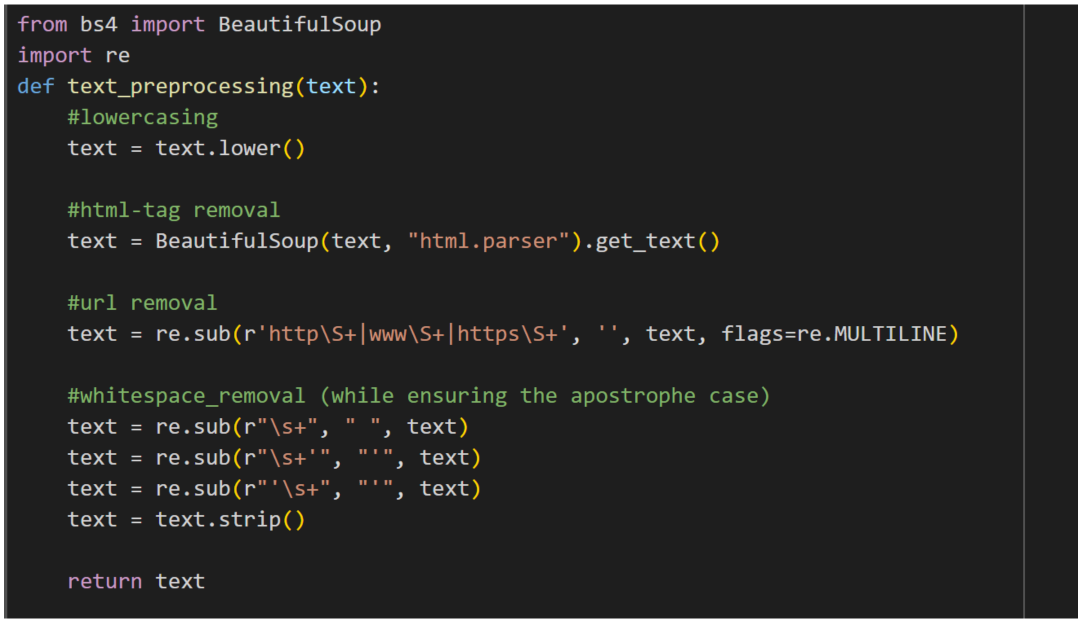
Hardware

- **GPU Acceleration** – cuda support (tested on P100)

3. Preprocessing

The language data was noisy, so it needed cleaning. My text preprocessing pipeline included:

- **Lowercasing** – Standardized the entire corpus for consistency.
- **HTML Tag Removal** – Removed HTML tags using the BeautifulSoup module.
- **URL Removal** – Defined general patterns of what a URL might look like and removed all matches.
- **Whitespace Removal** – Removed extra whitespaces to make the corpus clean and noiseless.

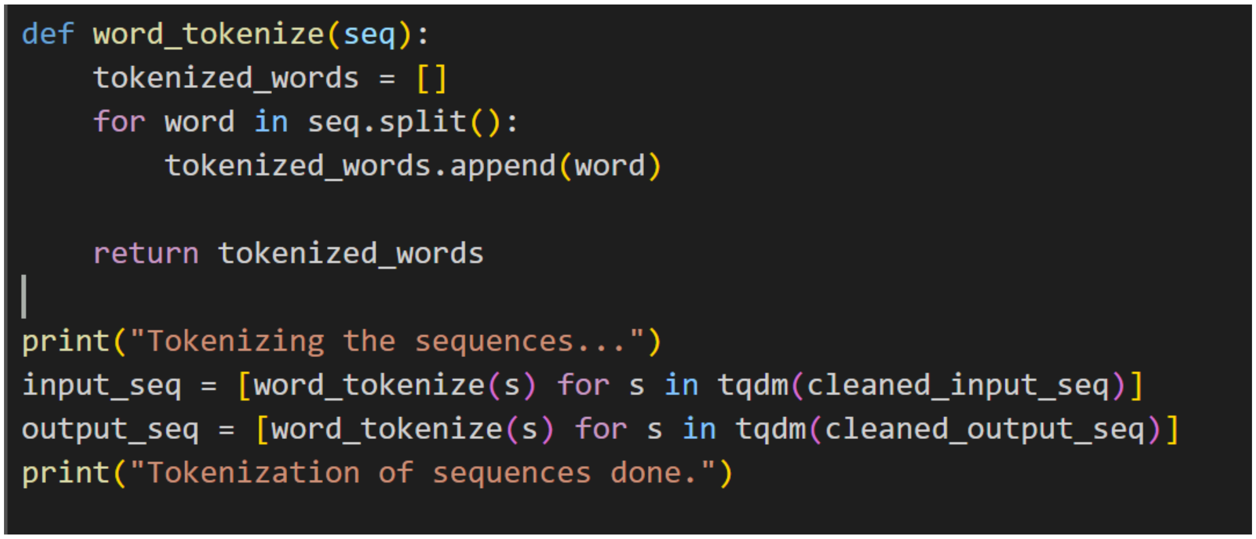


```python
from bs4 import BeautifulSoup
import re
def text_preprocessing(text):
    #lowercasing
    text = text.lower()

    #html-tag removal
    text = BeautifulSoup(text, "html.parser").get_text()

    #url removal
    text = re.sub(r'http\S+|www\S+|https\S+', '', text, flags=re.MULTILINE)

    #whitespace_removal (while ensuring the apostrophe case)
    text = re.sub(r"\s+", " ", text)
    text = re.sub(r"\s+'", "'", text)
    text = re.sub(r"'\s+", "'", text)
    text = text.strip()

    return text
```

4. Tokenization

- After preprocessing, the text was converted into tokens for model processing.
- A simple whitespace-based tokenization approach was used.
- Each sentence was split into words based on spaces.
- Separate tokenized lists were created for:

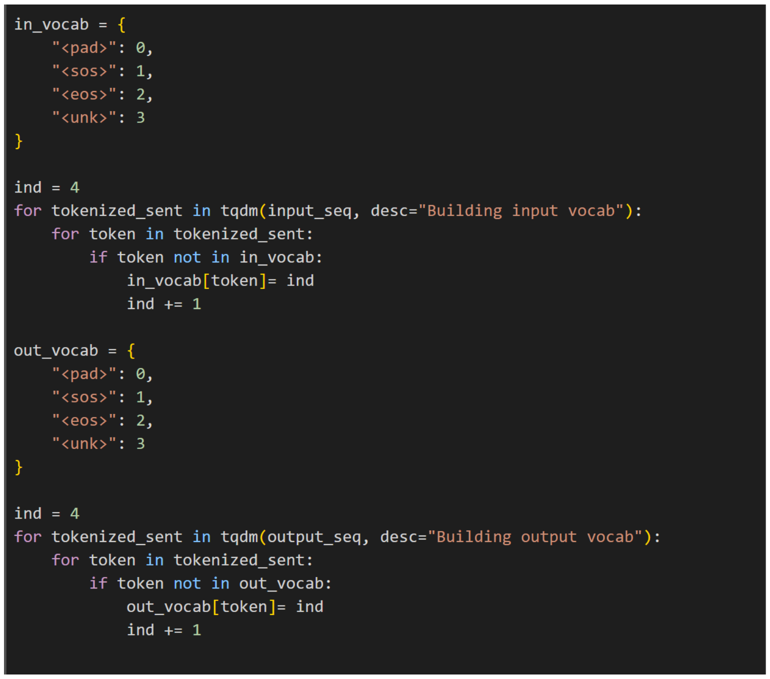
- **Input sequences (Hinglish)**
- **Output sequences (English)**



```
def word_tokenize(seq):
    tokenized_words = []
    for word in seq.split():
        tokenized_words.append(word)

    return tokenized_words

print("Tokenizing the sequences...")
input_seq = [word_tokenize(s) for s in tqdm(cleaned_input_seq)]
output_seq = [word_tokenize(s) for s in tqdm(cleaned_output_seq)]
print("Tokenization of sequences done.")
```

5. Vocabulary Construction

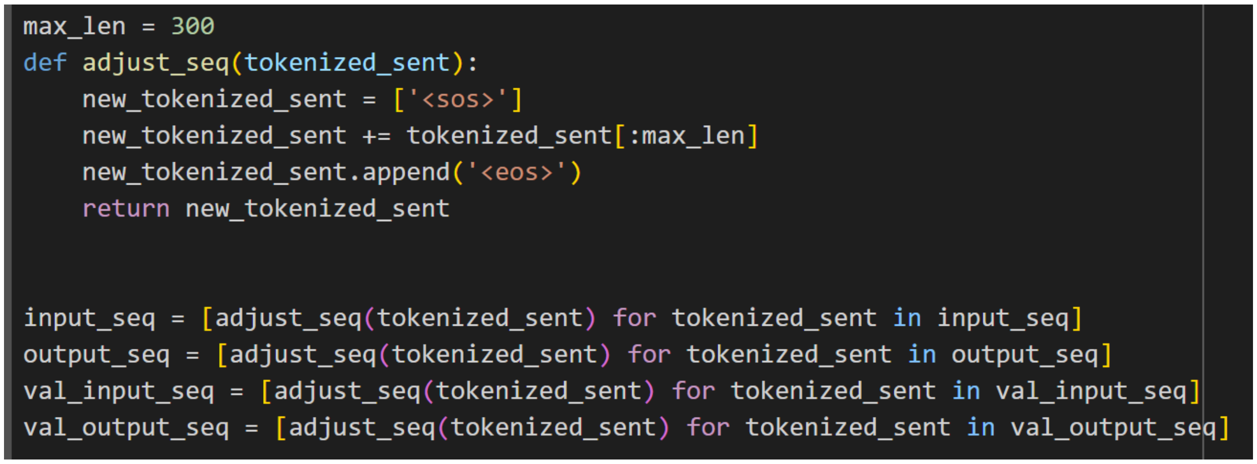
- Built separate vocabularies for the input sequences (Hinglish corpus) and output sequences (English corpus)
- Reserved token IDs [0 - 3] : ‘<pad>’, ‘<sos>’, ‘<eos>’, ‘<unk>’ : where,
  <pad> : This is the padding token ID used in sequences whose length is less than the expected length for a sequence (written as max_len in the code).
  <sos> : ‘Start of Sequence’ this is added to the beginning of every sequence.
  <eos> : ‘End of Sequence’ this is added to the end of every sequence.
- All the sequences were looped through and all the unique words were assigned their respective ‘token ID’ and stored inside ‘in_vocab’ and ‘out_vocab’ respectively.



```
in_vocab = {
    "<pad>": 0,
    "<sos>": 1,
    "<eos>": 2,
    "<unk>": 3
}

ind = 4
for tokenized_sent in tqdm(input_seq, desc="Building input vocab"):
    for token in tokenized_sent:
        if token not in in_vocab:
            in_vocab[token]= ind
            ind += 1

out_vocab = {
    "<pad>": 0,
    "<sos>": 1,
    "<eos>": 2,
    "<unk>": 3
}

ind = 4
for tokenized_sent in tqdm(output_seq, desc="Building output vocab"):
    for token in tokenized_sent:
        if token not in out_vocab:
            out_vocab[token]= ind
            ind += 1
```

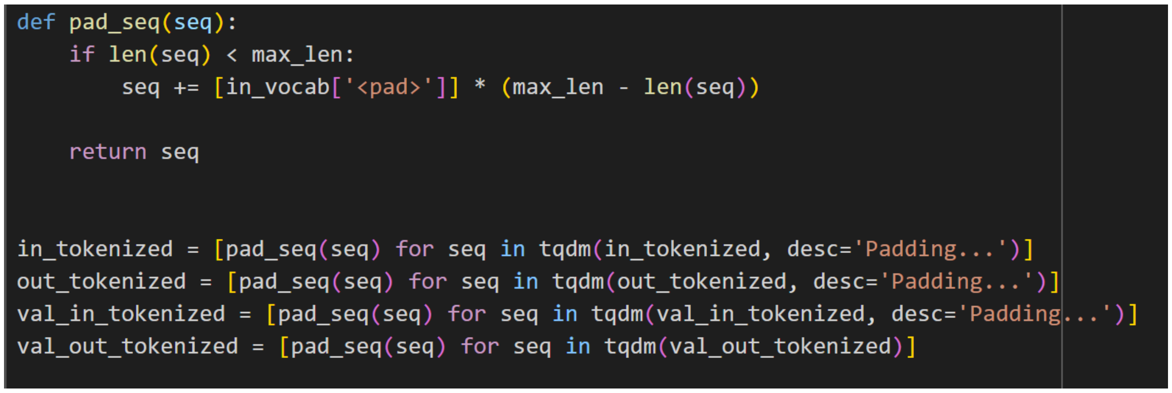
6. Sequence Handling

- Added <sos> and <eos> tokens at the start and end of each sequence, respectively.
- For sequences longer than 300 tokens, they were truncated to this length, and the <eos> token was appended at the new end.



```python
max_len = 300
def adjust_seq(tokenized_sent):
    new_tokenized_sent = ['<sos>']
    new_tokenized_sent += tokenized_sent[:max_len]
    new_tokenized_sent.append('<eos>')
    return new_tokenized_sent


input_seq = [adjust_seq(tokenized_sent) for tokenized_sent in input_seq]
output_seq = [adjust_seq(tokenized_sent) for tokenized_sent in output_seq]
val_input_seq = [adjust_seq(tokenized_sent) for tokenized_sent in val_input_seq]
val_output_seq = [adjust_seq(tokenized_sent) for tokenized_sent in val_output_seq]
```

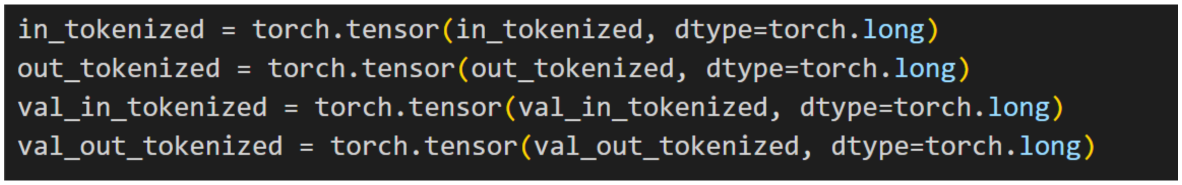
- The maximum length of 300 was chosen to ensure that no sequences in the training data were unnecessarily compromised while keeping computation manageable.
- Padding was applied by iterating through the input and output sequences and adding padding tokens where necessary to match the required sequence length.



```python
def pad_seq(seq):
    if len(seq) < max_len:
        seq += [in_vocab['<pad>']] * (max_len - len(seq))

    return seq


in_tokenized = [pad_seq(seq) for seq in tqdm(in_tokenized, desc='Padding...')]
out_tokenized = [pad_seq(seq) for seq in tqdm(out_tokenized, desc='Padding...')]
val_in_tokenized = [pad_seq(seq) for seq in tqdm(val_in_tokenized, desc='Padding...')]
val_out_tokenized = [pad_seq(seq) for seq in tqdm(val_out_tokenized)]
```

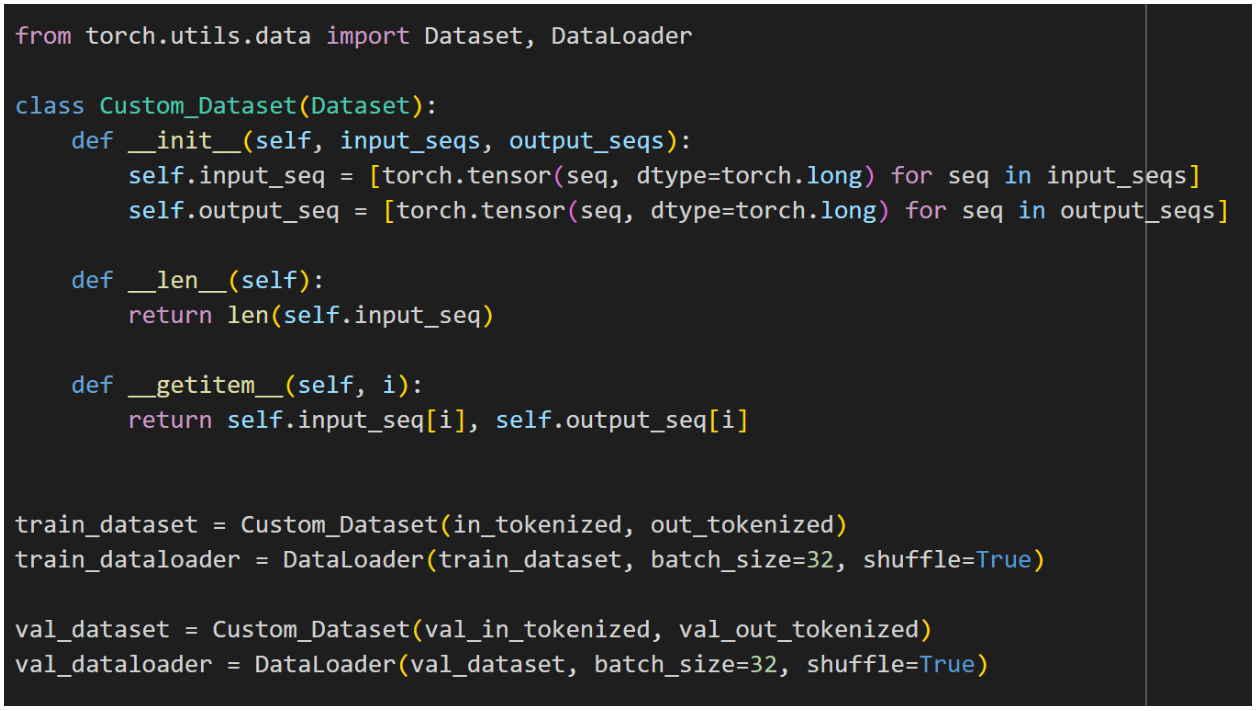
7. Dataset + DataLoader

- The current input sequences and output sequences are of the form python lists but for further processing they needed to be converted to pytorch tensors.



```python
in_tokenized = torch.tensor(in_tokenized, dtype=torch.long)
out_tokenized = torch.tensor(out_tokenized, dtype=torch.long)
val_in_tokenized = torch.tensor(val_in_tokenized, dtype=torch.long)
val_out_tokenized = torch.tensor(val_out_tokenized, dtype=torch.long)
```

- I created a custom dataset to process the input sequences and feed them to the dataloader to form batches and send in as batches for parallelizing the training.



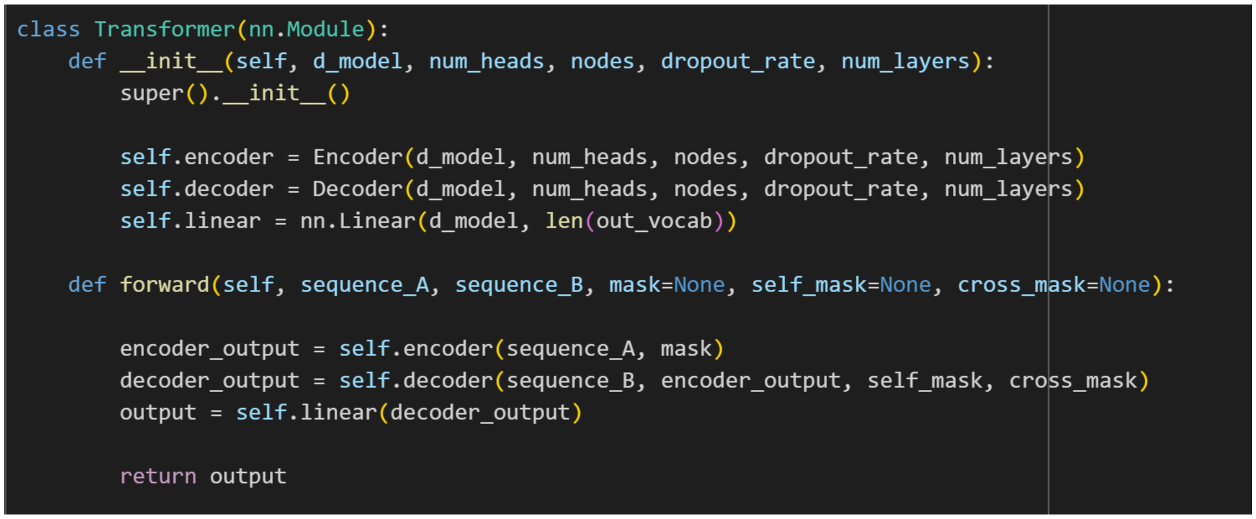
```python
from torch.utils.data import Dataset, DataLoader

class Custom_Dataset(Dataset):
    def __init__(self, input_seqs, output_seqs):
        self.input_seq = [torch.tensor(seq, dtype=torch.long) for seq in input_seqs]
        self.output_seq = [torch.tensor(seq, dtype=torch.long) for seq in output_seqs]

    def __len__(self):
        return len(self.input_seq)

    def __getitem__(self, i):
        return self.input_seq[i], self.output_seq[i]


train_dataset = Custom_Dataset(in_tokenized, out_tokenized)
train_dataloader = DataLoader(train_dataset, batch_size=32, shuffle=True)

val_dataset = Custom_Dataset(val_in_tokenized, val_out_tokenized)
val_dataloader = DataLoader(val_dataset, batch_size=32, shuffle=True)
```

8. Transformer Architecture

While building this architecture, I followed an **outwards-to-inwards approach**: starting with the most basic components of the model and gradually exploring the deeper, more complex parts of the Transformer as I progressed.

Transformer Wrapper

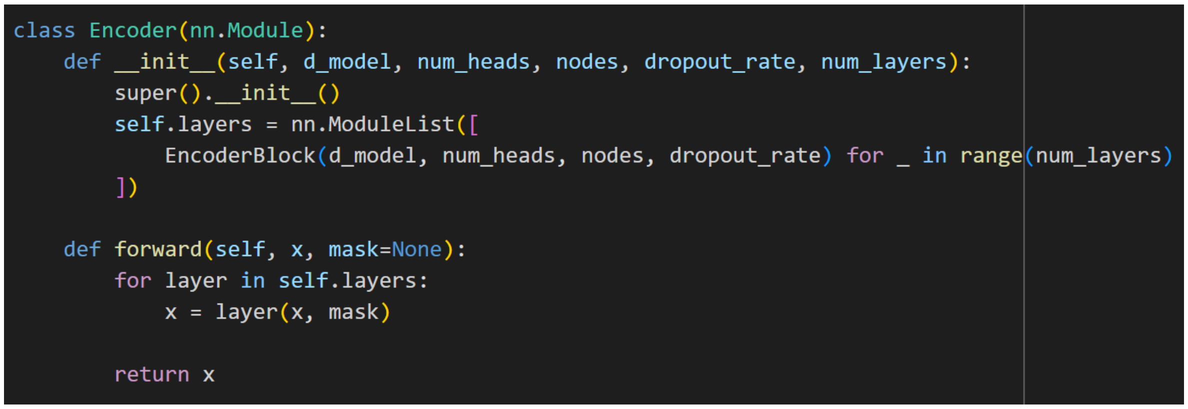
- This is the outermost wrapper of the model, encompassing the **encoder**, **decoder**, and the final **fully connected output layer**.
- The encoder processes the input sequences, and the decoder generates the output sequences conditioned on the encoder’s output.
- The final linear layer maps the decoder’s output to the vocabulary size to produce token predictions.
- During training, the **train_dataloader** provides batches of input and output sequences of size 32, while the **val_dataloader** provides validation batches of the same size.
-



```python
class Transformer(nn.Module):
    def __init__(self, d_model, num_heads, nodes, dropout_rate, num_layers):
        super().__init__()

        self.encoder = Encoder(d_model, num_heads, nodes, dropout_rate, num_layers)
        self.decoder = Decoder(d_model, num_heads, nodes, dropout_rate, num_layers)
        self.linear = nn.Linear(d_model, len(out_vocab))

    def forward(self, sequence_A, sequence_B, mask=None, self_mask=None, cross_mask=None):

        encoder_output = self.encoder(sequence_A, mask)
        decoder_output = self.decoder(sequence_B, encoder_output, self_mask, cross_mask)
        output = self.linear(decoder_output)

        return output
```

Encoder

- This is the outer wrapper of the encoder, which stacks a collection of num_layers **EncoderBlocks**.
- Each EncoderBlock performs self-attention and feed-forward operations.
- The layers are applied sequentially: the output of one layer becomes the input to the next.
- The final output of the encoder is passed to the decoder for further processing.



```python
class Encoder(nn.Module):
    def __init__(self, d_model, num_heads, nodes, dropout_rate, num_layers):
        super().__init__()
        self.layers = nn.ModuleList([
            EncoderBlock(d_model, num_heads, nodes, dropout_rate) for _ in range(num_layers)
        ])

    def forward(self, x, mask=None):
        for layer in self.layers:
            x = layer(x, mask)

        return x
```

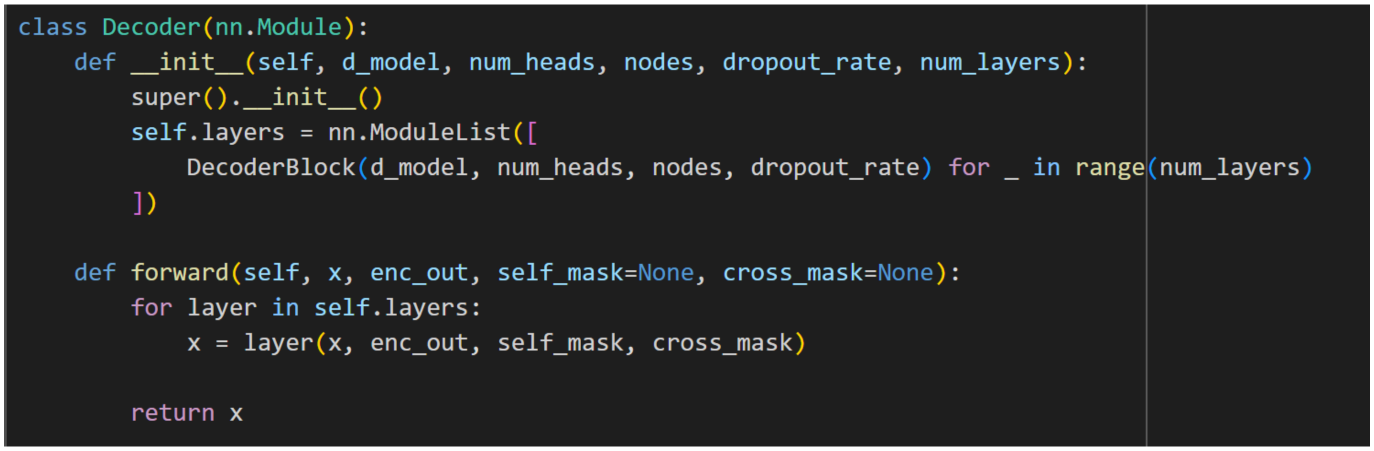
Decoder

- This is the outer wrapper of the decoder, which stacks a collection of num_layers **DecoderBlocks**.

- Each DecoderBlock performs masked self-attention on the target sequence, followed by cross-attention over the encoder's output, and a feed-forward operation.

- The layers are applied sequentially: the output of one decoder block is fed into the next.

- The final output of the decoder is passed to the linear layer to generate predictions over the vocabulary



```
class Decoder(nn.Module):
    def __init__(self, d_model, num_heads, nodes, dropout_rate, num_layers):
        super().__init__()
        self.layers = nn.ModuleList([
            DecoderBlock(d_model, num_heads, nodes, dropout_rate) for _ in range(num_layers)
        ])

    def forward(self, x, enc_out, self_mask=None, cross_mask=None):
        for layer in self.layers:
            x = layer(x, enc_out, self_mask, cross_mask)

        return x
```

<u>Encoder Block</u>

This is the block with the main functionality for the encoder.

- This block contains the main functionality of the encoder.

- Each **EncoderBlock** consists of:

    - **Multi-Head Self-Attention Layer** – captures relationships between tokens in the input sequence.
    - **Dropout Layer** – prevents overfitting.
    - **Layer Normalization** – normalizes the output for stable training.
    - **Feed-Forward Neural Network Layer** – applies pointwise transformations.
    - **Dropout + Layer Normalization** – follows the feed-forward layer.

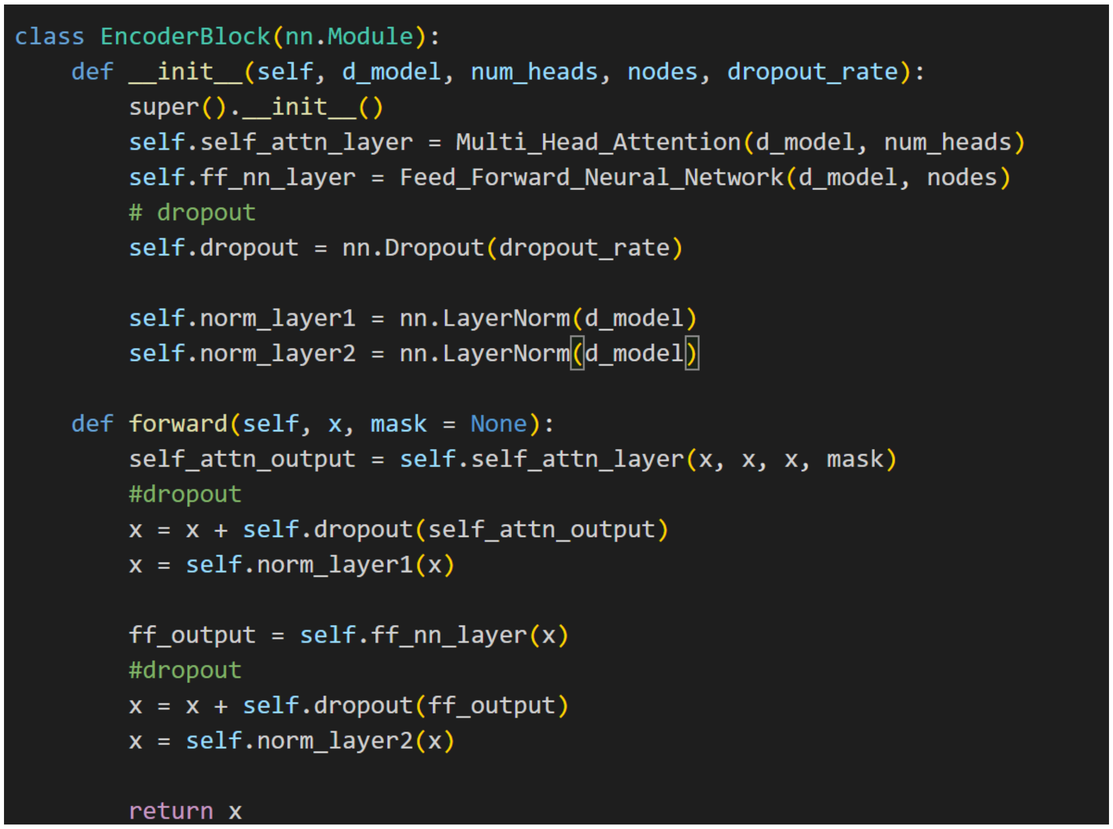
- The **constructor (__init__)** defines all the layers required for the block.

- The **forward function** performs the actual data processing:

    - The input x passes through the self-attention layer, producing self_attn_output.

  - Dropout is applied, and the result is added to the original input (**residual connection**).
  - Layer normalization is applied to stabilize training.
  - The output is passed through the feed-forward network, followed by another residual connection and layer normalization.

- The final output of the block is forwarded to the next encoder block or to the decoder.

- 

```python
class EncoderBlock(nn.Module):
    def __init__(self, d_model, num_heads, nodes, dropout_rate):
        super().__init__()
        self.self_attn_layer = Multi_Head_Attention(d_model, num_heads)
        self.ff_nn_layer = Feed_Forward_Neural_Network(d_model, nodes)
        # dropout
        self.dropout = nn.Dropout(dropout_rate)

        self.norm_layer1 = nn.LayerNorm(d_model)
        self.norm_layer2 = nn.LayerNorm(d_model)

    def forward(self, x, mask = None):
        self_attn_output = self.self_attn_layer(x, x, x, mask)
        #dropout
        x = x + self.dropout(self_attn_output)
        x = self.norm_layer1(x)

        ff_output = self.ff_nn_layer(x)
        #dropout
        x = x + self.dropout(ff_output)
        x = self.norm_layer2(x)

        return x
```

Decoder Block

- The **DecoderBlock** is the main functional unit of the decoder.

- Each block consists of three major components:

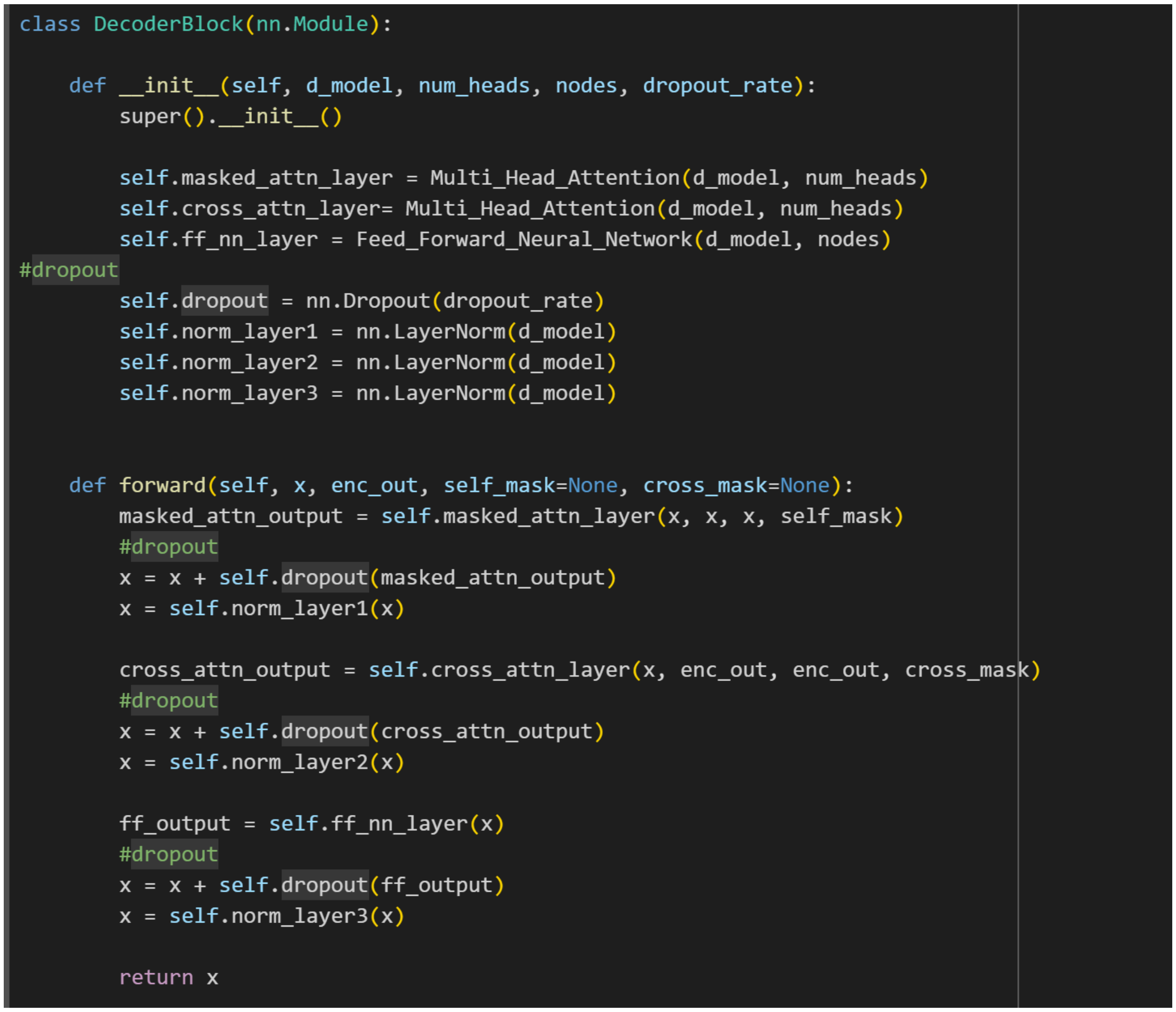
  - **Masked Self-Attention** – allows the decoder to attend to previous tokens in the target sequence while preventing future tokens from being seen.

- ○ **Cross-Attention** – attends to the encoder's output to incorporate information from the input sequence.
- ○ **Feed-Forward Neural Network (FFN)** – applies pointwise transformations.

- Each layer is followed by **dropout** and **layer normalization**, with **residual connections** applied after each sub-layer.

Constructor (__init__)

- Defines all the layers required for the block: masked self-attention, cross-attention, feed-forward network, three layer normalization layers, and dropout.



```
class DecoderBlock(nn.Module):

    def __init__(self, d_model, num_heads, nodes, dropout_rate):
        super().__init__()

        self.masked_attn_layer = Multi_Head_Attention(d_model, num_heads)
        self.cross_attn_layer= Multi_Head_Attention(d_model, num_heads)
        self.ff_nn_layer = Feed_Forward_Neural_Network(d_model, nodes)
#dropout
        self.dropout = nn.Dropout(dropout_rate)
        self.norm_layer1 = nn.LayerNorm(d_model)
        self.norm_layer2 = nn.LayerNorm(d_model)
        self.norm_layer3 = nn.LayerNorm(d_model)


    def forward(self, x, enc_out, self_mask=None, cross_mask=None):
        masked_attn_output = self.masked_attn_layer(x, x, x, self_mask)
        #dropout
        x = x + self.dropout(masked_attn_output)
        x = self.norm_layer1(x)

        cross_attn_output = self.cross_attn_layer(x, enc_out, enc_out, cross_mask)
        #dropout
        x = x + self.dropout(cross_attn_output)
        x = self.norm_layer2(x)

        ff_output = self.ff_nn_layer(x)
        #dropout
        x = x + self.dropout(ff_output)
        x = self.norm_layer3(x)

        return x
```

Forward Function

1. Input `x` passes through the **masked self-attention** layer.

2. Dropout is applied, and a **residual connection** adds the original input to the output, followed by layer normalization.

3. The result is passed to the **cross-attention** layer along with the encoder's output. Dropout, residual connection, and layer normalization are applied again.

4. The output is then passed through the **feed-forward network**, followed by dropout, residual connection, and layer normalization.

5. The final output is returned and later processed by the linear layer and softmax to produce a probability distribution over the output vocabulary.

<u>Multi Head Attention</u>

- This is the part which was the game-changer from the old lstm based seq2seq models.When I first encountered Multi-Head Attention, it honestly felt overwhelming. The equations looked intimidating, and I kept wondering *"why complicate it with multiple heads when one attention should be enough?"* But as I wrestled with the idea and implemented it myself, the pieces started falling into place.

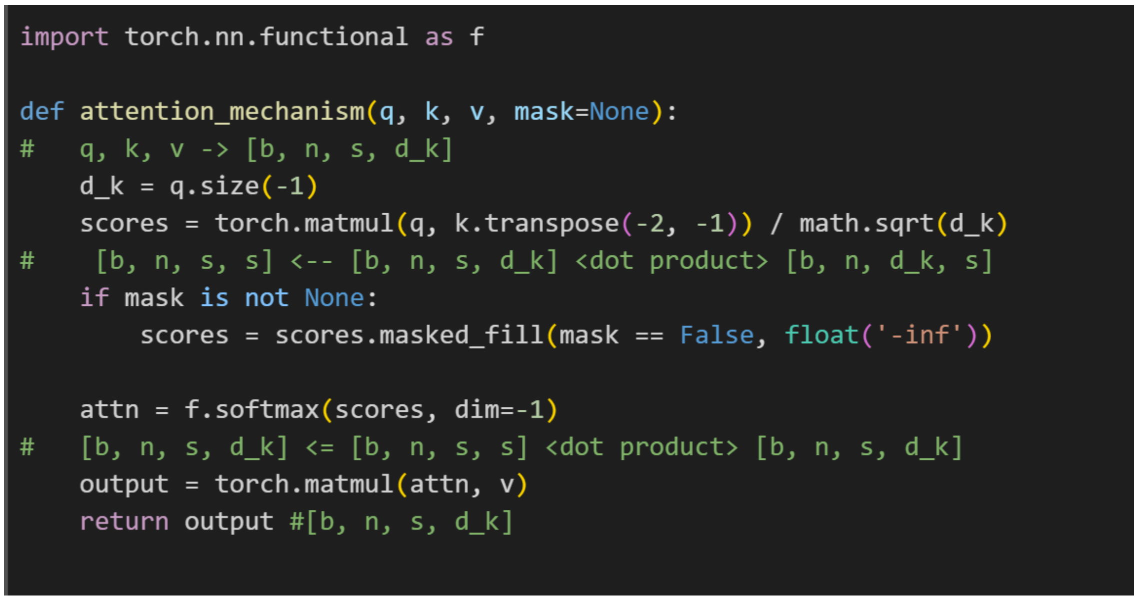
- Multi Head Attention is a concept when put in simple words — it's like having multiple people analyze the same text, each spotting something different, and then pooling their insights together. That's the intuition that really made me click with Multi-Head Attention.

- The core idea of multi head attention is about figuring out which parts of the input sequence are important when predicting the next token.

- But then why multi-head? — This was the part I struggled with initially. I thought: *isn't one attention mechanism enough?* But the insight is that **different heads can learn to focus on different types of relationships in parallel. For eg:**

  One head might capture short-distance dependencies (like adjectives modifying nouns). Another might capture long-distance dependencies (like subject-verb agreement across clauses).

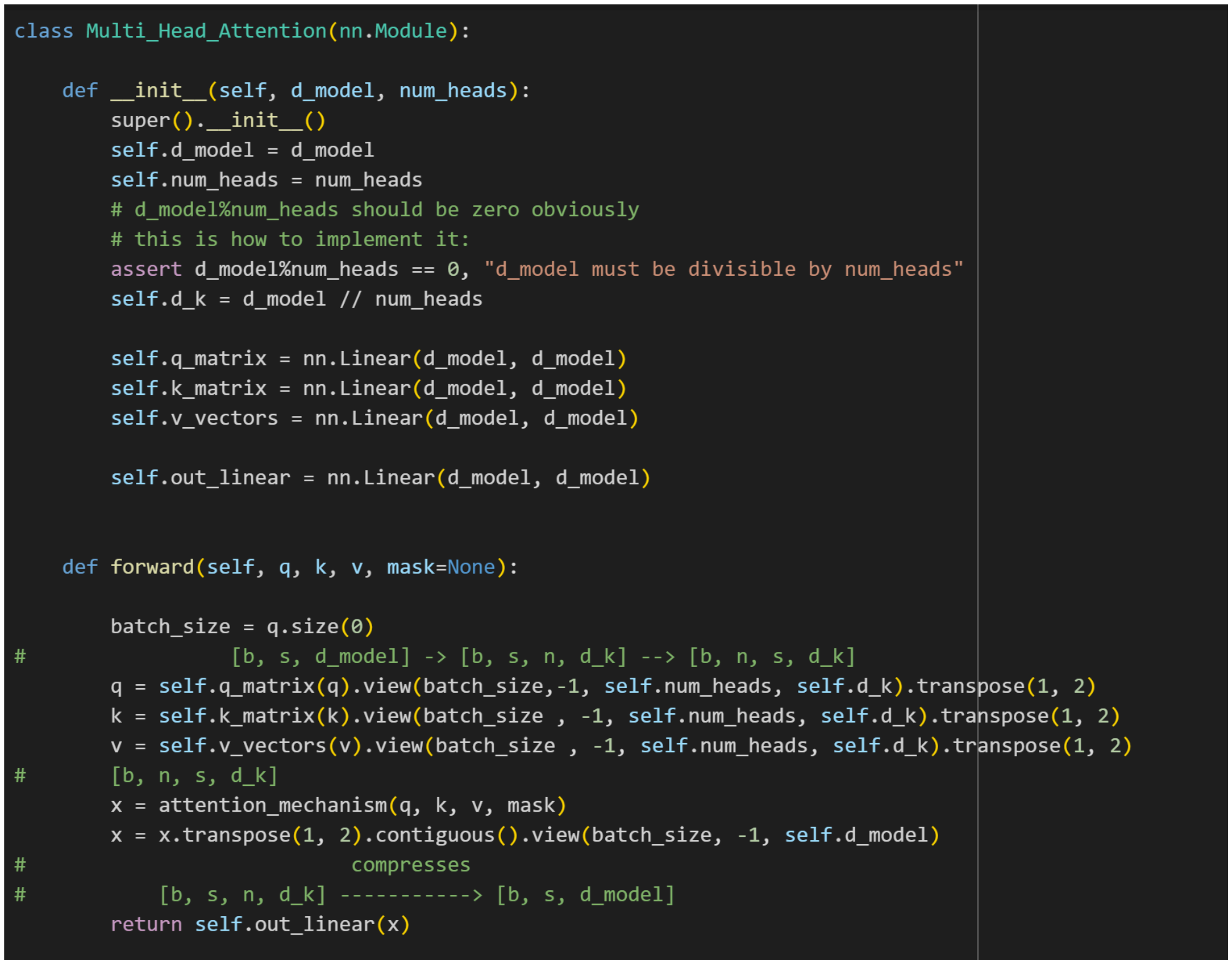
- Now onto how it works? Each attention head has its own set of learned projections for **Queries (Q), Keys (K), and Values (V).** These projections shrink the input into smaller subspaces (so computations stay efficient). Attention is calculated independently for each



```
import torch.nn.functional as f

def attention_mechanism(q, k, v, mask=None):
#   q, k, v -> [b, n, s, d_k]
    d_k = q.size(-1)
    scores = torch.matmul(q, k.transpose(-2, -1)) / math.sqrt(d_k)
#    [b, n, s, s] <-- [b, n, s, d_k] <dot product> [b, n, d_k, s]
    if mask is not None:
        scores = scores.masked_fill(mask == False, float('-inf'))

    attn = f.softmax(scores, dim=-1)
#   [b, n, s, d_k] <= [b, n, s, s] <dot product> [b, n, s, d_k]
    output = torch.matmul(attn, v)
    return output #[b, n, s, d_k]
```

head. Finally, the outputs of all heads are concatenated and passed through a linear layer to bring everything back together.

- My biggest struggle was mentally connecting math to intuition. Once I realized that **multi-head attention is like having multiple specialists analyze the same text from different angles,** it clicked.



```
class Multi_Head_Attention(nn.Module):

    def __init__(self, d_model, num_heads):
        super().__init__()
        self.d_model = d_model
        self.num_heads = num_heads
        # d_model%num_heads should be zero obviously
        # this is how to implement it:
        assert d_model%num_heads == 0, "d_model must be divisible by num_heads"
        self.d_k = d_model // num_heads

        self.q_matrix = nn.Linear(d_model, d_model)
        self.k_matrix = nn.Linear(d_model, d_model)
        self.v_vectors = nn.Linear(d_model, d_model)

        self.out_linear = nn.Linear(d_model, d_model)


    def forward(self, q, k, v, mask=None):

        batch_size = q.size(0)
#                 [b, s, d_model] -> [b, s, n, d_k] --> [b, n, s, d_k]
        q = self.q_matrix(q).view(batch_size,-1, self.num_heads, self.d_k).transpose(1, 2)
        k = self.k_matrix(k).view(batch_size , -1, self.num_heads, self.d_k).transpose(1, 2)
        v = self.v_vectors(v).view(batch_size , -1, self.num_heads, self.d_k).transpose(1, 2)
#       [b, n, s, d_k]
        x = attention_mechanism(q, k, v, mask)
        x = x.transpose(1, 2).contiguous().view(batch_size, -1, self.d_model)
#                         compresses
#           [b, s, n, d_k] -----------> [b, s, d_model]
        return self.out_linear(x)
```

- The above multi head attention does that itself the constructor function of the class stores the required variables like d_model which is the length of the word embedding (=512, as per the *Attention is all you Need* research paper and thus my implementation as well), num_heads is the number of attention heads that are going to run parallel, then the d_k is defined as the division of d_model and num_heads and then it defines the q (query) matrix, the k (key) matrix, the v (value) matrix.
- The forward function of the code has the logic of the code which initiates by instanting the q, k v values which are then passed into the attention mechanism to perform the following operation:

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

- The attention mechanism in the specified code is the implementation of the same mathematical connections. I have commented the way the sizes of the matrices change to keep track and know what is happening at each stage as that part is still a challenging part for me.

<u>Mask</u>

- Masking is essential in Transformers to prevent the model from attending to irrelevant positions, such as padding tokens, or from "seeing the future" in the decoder.
- This class implements two types of masks:
    1. **Padding Mask** (padding_mask) – ensures the model does not attend to padding tokens (<pad>) in either the encoder or decoder.
        - Converts sequences into a boolean mask where positions corresponding to <pad> are False and others are True.
    2. **Decoder Mask** (decoder_mask) – combines the padding mask with a **causal mask** to prevent the decoder from attending to future positions.
        - **Causal mask**: an upper triangular matrix that blocks attention to future tokens in the sequence.

    - The final mask ensures that during training, the model can only attend to previous and current tokens, while ignoring padding.

- The register_buffer method is used to store the causal mask so that it moves automatically with the model to GPU/CPU during training.

Application in the Transformer

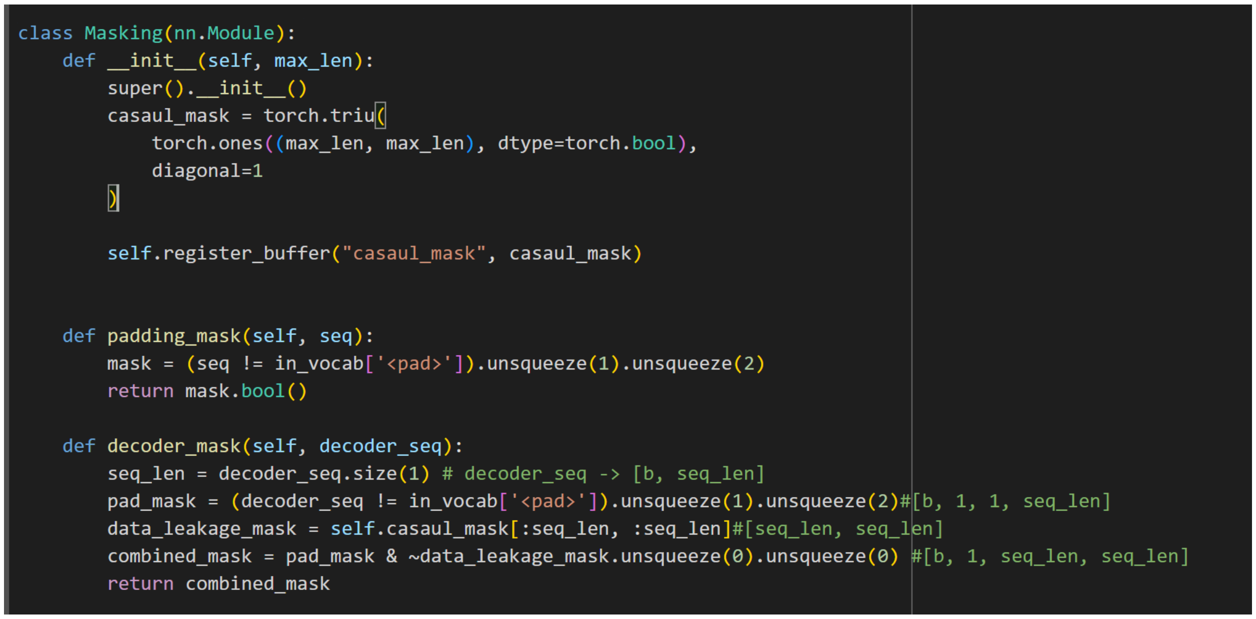
- In the **encoder**, the padding mask is applied to prevent attention over padding tokens in the input sequences.

- In the **decoder**, the decoder mask is applied before the masked self-attention layer to:

    1. Prevent the decoder from attending to future tokens (causal mask).
    2. Prevent attending to padding tokens (padding mask).

- During **cross-attention** in the decoder, only the encoder's padding mask is applied to ensure the decoder attends to valid encoder positions.

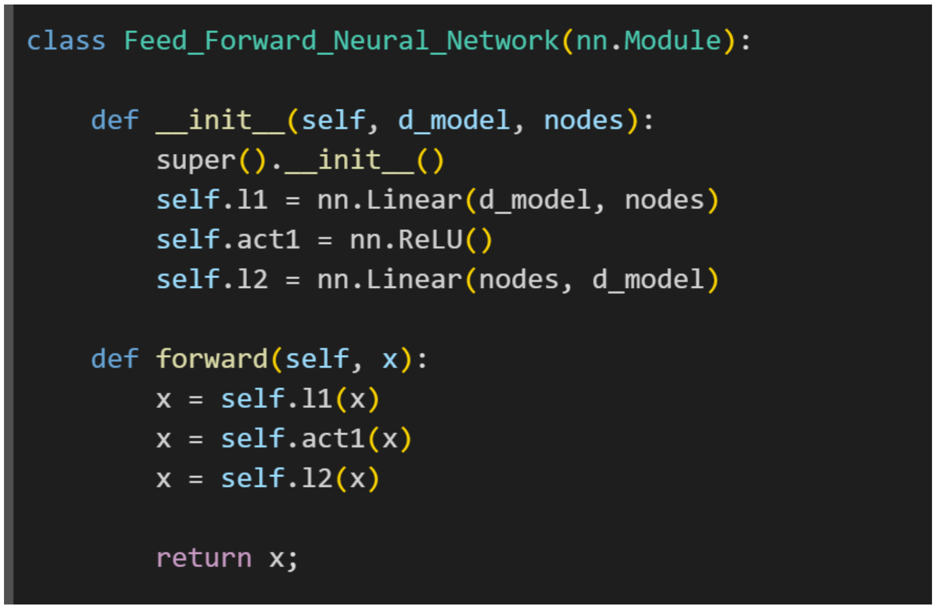
- By carefully applying these masks, the model learns correct dependencies in the sequence while ignoring irrelevant positions, which is crucial for both training efficiency and model performance.



```
class Masking(nn.Module):
    def __init__(self, max_len):
        super().__init__()
        casaul_mask = torch.triu(
            torch.ones((max_len, max_len), dtype=torch.bool),
            diagonal=1
        )

        self.register_buffer("casaul_mask", casaul_mask)


    def padding_mask(self, seq):
        mask = (seq != in_vocab['<pad>']).unsqueeze(1).unsqueeze(2)
        return mask.bool()

    def decoder_mask(self, decoder_seq):
        seq_len = decoder_seq.size(1) # decoder_seq -> [b, seq_len]
        pad_mask = (decoder_seq != in_vocab['<pad>']).unsqueeze(1).unsqueeze(2)#[b, 1, 1, seq_len]
        data_leakage_mask = self.casaul_mask[:seq_len, :seq_len]#[seq_len, seq_len]
        combined_mask = pad_mask & ~data_leakage_mask.unsqueeze(0).unsqueeze(0) #[b, 1, seq_len, seq_len]
        return combined_mask
```

Feed Forward Neural Network Layer

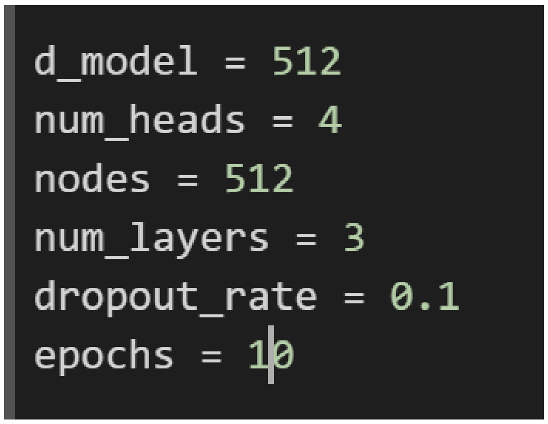
- The most basic neural architecture. Used in Encoders as well Decoders
- The input is processed through a feed forward neural network with 2048 nodes as per the *Attention is all you Need* research paper, but in my implementation I have about 512 nodes.



```
class Feed_Forward_Neural_Network(nn.Module):

    def __init__(self, d_model, nodes):
        super().__init__()
        self.l1 = nn.Linear(d_model, nodes)
        self.act1 = nn.ReLU()
        self.l2 = nn.Linear(nodes, d_model)

    def forward(self, x):
        x = self.l1(x)
        x = self.act1(x)
        x = self.l2(x)

        return x;
```

Parameter Selection

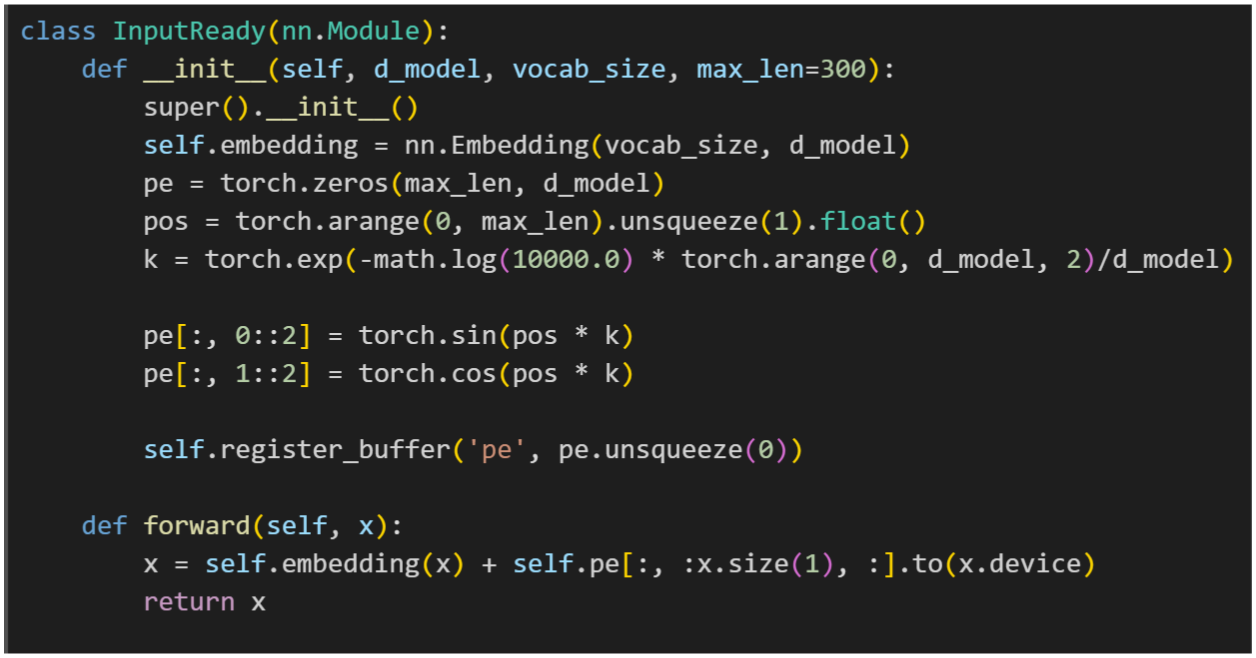
- The following parameters are implemented in my model.



```
d_model = 512
num_heads = 4
nodes = 512
num_layers = 3
dropout_rate = 0.1
epochs = 10
```

9. Input Ready

- Preparing the data for input essentially means transforming the tokenized sequences into word embeddings and enriching them with positional encodings, so that the model not only understands the meaning of words but also their order within the sequence.

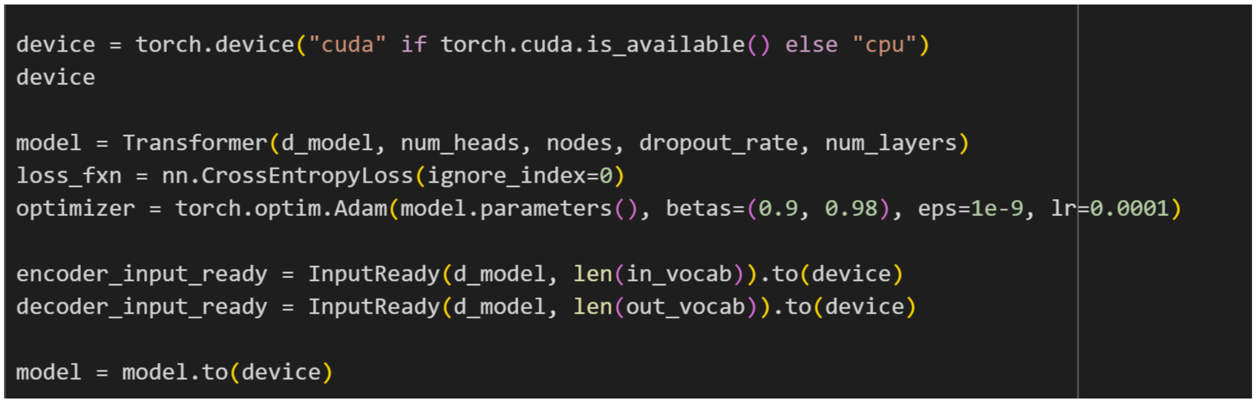


```python
class InputReady(nn.Module):
    def __init__(self, d_model, vocab_size, max_len=300):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, d_model)
        pe = torch.zeros(max_len, d_model)
        pos = torch.arange(0, max_len).unsqueeze(1).float()
        k = torch.exp(-math.log(10000.0) * torch.arange(0, d_model, 2)/d_model)

        pe[:, 0::2] = torch.sin(pos * k)
        pe[:, 1::2] = torch.cos(pos * k)

        self.register_buffer('pe', pe.unsqueeze(0))

    def forward(self, x):
        x = self.embedding(x) + self.pe[:, :x.size(1), :].to(x.device)
        return x
```

- At its core, this class takes discrete token IDs (the numbers representing words from your vocabulary) and projects them into dense vector representations using an embedding layer. But embeddings alone don't carry any sense of *order* — the model wouldn't know whether a word came at the start of the sentence or the end. To solve this, I implemented sinusoidal positional encodings (sine for even indices, cosine for odd ones as per the research paper) that inject a unique signature for every position in the sequence. By summing embeddings with these positional vectors, the model learns not just *what* the words mean, but also *where* they occur in the sentence

- One of the trickier parts for me was figuring out how to store the positional encodings efficiently without recalculating them on every forward pass. I solved this using register_buffer, which allows positional encodings to live inside the model without being treated as learnable parameters. Then in the forward method, whenever a sequence comes in, I fetch the matching slice of positional encodings (up to the sequence length), align it with the embeddings, move it to the right device, and add it up. The final output is a rich combination of semantic meaning + positional awareness, ready to be passed into the Transformer.

10. Setting up the Model



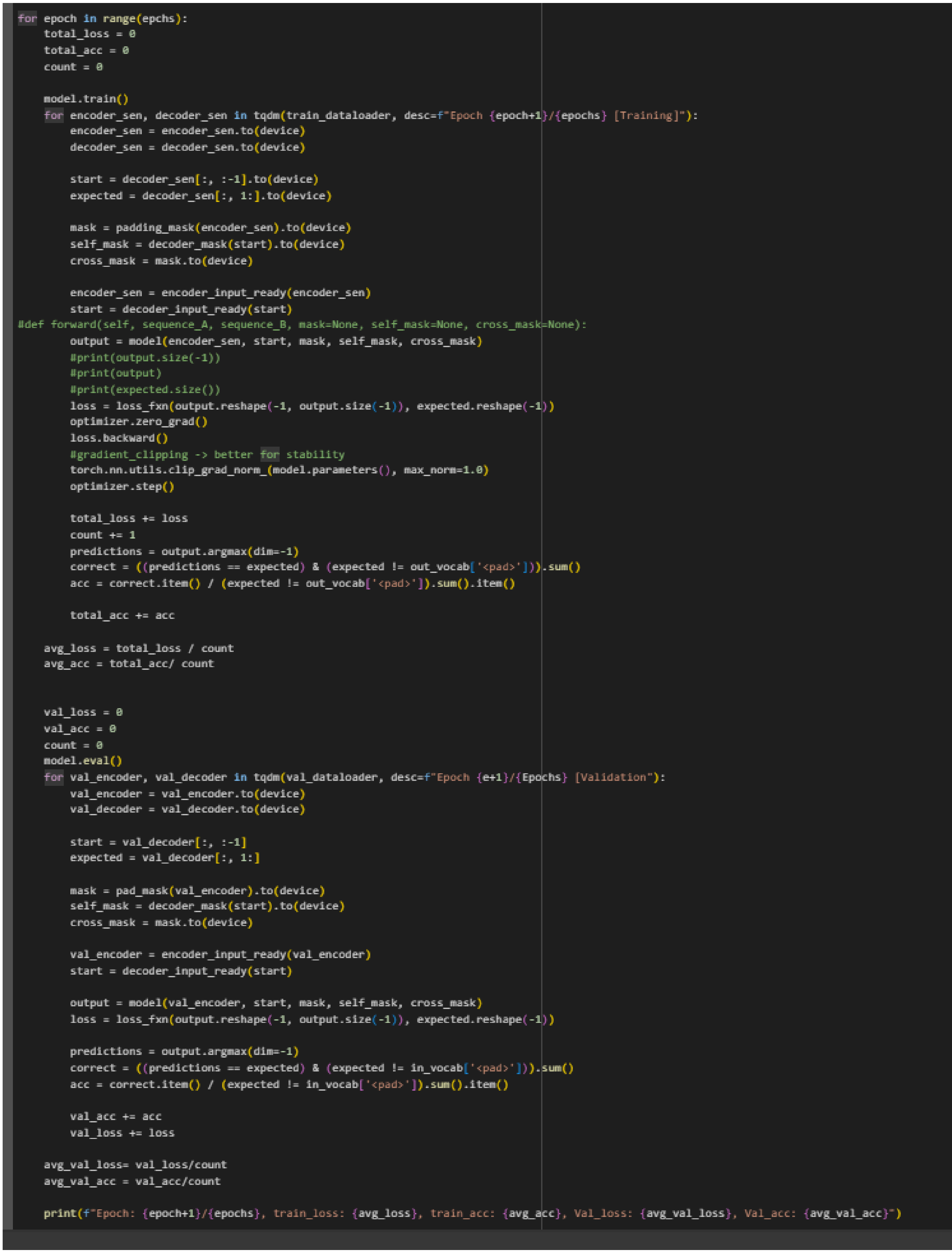
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device

model = Transformer(d_model, num_heads, nodes, dropout_rate, num_layers)
loss_fxn = nn.CrossEntropyLoss(ignore_index=0)
optimizer = torch.optim.Adam(model.parameters(), betas=(0.9, 0.98), eps=1e-9, lr=0.0001)

encoder_input_ready = InputReady(d_model, len(in_vocab)).to(device)
decoder_input_ready = InputReady(d_model, len(out_vocab)).to(device)

model = model.to(device)
```

- Setting up the model to cuda p100 accelerator and initializing the model, along with all the extra modules.

11. Training Loop

- Each epoch begins by initializing counters for total loss and accuracy. The model is put into train() mode, which ensures dropout and gradient updates behave as expected.
- Using tqdm, each batch from the dataloader gets processed with a nice progress bar. Here, encoder and decoder sequences are moved to the correct device (GPU), and the target (expected output) is split into two parts:
- **start** → input to the decoder (all tokens except the last).
- **expected** → ground truth for comparison (all tokens except the first).
- Before passing things through the model, padding masks, self-attention masks, and cross-attention masks are built to prevent the model from "cheating"
- The encoder input is prepped using the InputReady module, then passed through the full Transformer model with the masks. The output is essentially the model's predicted probability distribution over the vocabulary for every token in the sequence
- **Loss + Backpropagation**

- Loss is calculated between predictions and expected tokens (ignoring <pad> tokens). Gradients are computed with loss.backward().
- Gradient clipping (clip_grad_norm_) is applied for stability (to prevent exploding gradients, which is something I had to wrestle with while debugging).
- Optimizer updates the parameters.



```
for epoch in range(epchs):
    total_loss = 0
    total_acc = 0
    count = 0

    model.train()
    for encoder_sen, decoder_sen in tqdm(train_dataloader, desc=f"Epoch {epoch+1}/{epochs} [Training]"):
        encoder_sen = encoder_sen.to(device)
        decoder_sen = decoder_sen.to(device)

        start = decoder_sen[:, :-1].to(device)
        expected = decoder_sen[:, 1:].to(device)

        mask = padding_mask(encoder_sen).to(device)
        self_mask = decoder_mask(start).to(device)
        cross_mask = mask.to(device)

        encoder_sen = encoder_input_ready(encoder_sen)
        start = decoder_input_ready(start)
#def forward(self, sequence_A, sequence_B, mask=None, self_mask=None, cross_mask=None):
        output = model(encoder_sen, start, mask, self_mask, cross_mask)
        #print(output.size(-1))
        #print(output)
        #print(expected.size())
        loss = loss_fxn(output.reshape(-1, output.size(-1)), expected.reshape(-1))
        optimizer.zero_grad()
        loss.backward()
        #gradient_clipping -> better for stability
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()

        total_loss += loss
        count += 1
        predictions = output.argmax(dim=-1)
        correct = ((predictions == expected) & (expected != out_vocab['<pad>'])).sum()
        acc = correct.item() / (expected != out_vocab['<pad>']).sum().item()

        total_acc += acc

    avg_loss = total_loss / count
    avg_acc = total_acc/ count


    val_loss = 0
    val_acc = 0
    count = 0
    model.eval()
    for val_encoder, val_decoder in tqdm(val_dataloader, desc=f"Epoch {e+1}/{Epochs} [Validation"):
        val_encoder = val_encoder.to(device)
        val_decoder = val_decoder.to(device)

        start = val_decoder[:, :-1]
        expected = val_decoder[:, 1:]

        mask = pad_mask(val_encoder).to(device)
        self_mask = decoder_mask(start).to(device)
        cross_mask = mask.to(device)

        val_encoder = encoder_input_ready(val_encoder)
        start = decoder_input_ready(start)

        output = model(val_encoder, start, mask, self_mask, cross_mask)
        loss = loss_fxn(output.reshape(-1, output.size(-1)), expected.reshape(-1))

        predictions = output.argmax(dim=-1)
        correct = ((predictions == expected) & (expected != in_vocab['<pad>'])).sum()
        acc = correct.item() / (expected != in_vocab['<pad>']).sum().item()

        val_acc += acc
        val_loss += loss

    avg_val_loss= val_loss/count
    avg_val_acc = val_acc/count

    print(f"Epoch: {epoch+1}/{epochs}, train_loss: {avg_loss}, train_acc: {avg_acc}, Val_loss: {avg_val_loss}, Val_acc: {avg_val_acc}")
```