Text Generation using Fnet

Transformer-based models excel in understanding and processing sequences due to their utilization of a mechanism known as "self-attention." This involves scrutinizing each token to discern its relationship with every other token in the sequence. Despite the effectiveness of self-attention, its drawback lies in its computational cost. For a sequence of length N, self-attention requires N^2 operations, resulting in quadratic scaling. This can be computationally expensive and time-consuming, especially for long sentences, imposing limitations on sequence length, such as the 512-token constraint in the standard BERT model.

Numerous methods have emerged to address the computational inefficiency of quadratic scaling. A recent innovation tackling this challenge is FNet, which completely replaces the self-attention layer. FNet introduces an alternative mechanism, diverging from the traditional self-attention paradigm while aiming to achieve comparable or enhanced performance in handling sequences. In this article, we will focus on the implementation of the FNet architecture for text generation in Python using Pytorch.

FNet

The Transformer architecture is renowned for its dominance in natural language processing (NLP). It uses a core component, the attention mechanism, which connects input tokens by weighing their relevance to each other. While various studies have probed the Transformer and its attention sublayers, the computational cost of self-attention remains a challenge, particularly for long sequences.

In response to this challenge, a recent innovation, FNet, introduces a novel approach by replacing the self-attention layer entirely. Instead of self-attention, FNet utilizes simpler token mixing mechanisms, such as parameterized matrix multiplications and, remarkably, the Fourier transform. Unlike traditional self-attention, the Fourier transform has no parameters yet achieves comparable performance, scaling efficiently to long sequences due to the Fast Fourier transform (FFT) algorithm.

Text Generation using FNet

Step 1: Libraries and import

Install below libraries if they are not available in your environment

!pip install datasets

!pip install torch[transformers]

Declare device variable for computation on GPU if available

1

import torch

2

device = 'cuda' if torch.cuda.is_available() else 'cpu'

3

print(device)

Output:

cuda

Step 2 : Load Data

Here we will use the wikitext corpus for training the data. We will use the dataset library to load the same

1

```
from datasets import load_dataset
```

2

```
datasets = load_dataset('wikitext','wikitext-2-raw-v1')
```

Step 3: Data Preprocessing

We will clean up our data

Decalare a preprocess_text function which will

Make all the words in the sentence lowercase

Remove any special characters

Replace any multiple white spaces

We use map function to perform above preprocessing

We use filter function to keep only those data that have length greater than 20

```
import re



def preprocess_text(sentence):
    # lowering the sentence and storing in text vaiable
    text = sentence['text'].lower()
    # removing other than characters and punctuations
    text = re.sub('[^a-z?!.,]', ' ', text)
    text = re.sub('\s\s+', ' ', text)  # removing double spaces
```

```
    sentence['text'] = text
    return sentence


datasets['train'] = datasets['train'].map(preprocess_text)
datasets['test'] = datasets['test'].map(preprocess_text)
datasets['validation'] = datasets['validation'].map(preprocess_text)


datasets['train'] = datasets['train'].filter(lambda x: len(x['text']) > 20)
datasets['test'] = datasets['test'].filter(lambda x: len(x['text']) > 20)
datasets['validation'] = datasets['validation'].filter(
    lambda x: len(x['text']) > 20)
```

Step 4. : Tokenisation

For tokenizer we use a pretrained tokenizer from hugging face. The code loads a pre-trained tokenizer (distilbert-base-uncased-finetuned-sst-2-english) using AutoTokenizer.from_pretrained.

We decalre a tokenizer function that tokenizes our input. This function takes a sentence as input, tokenizes it using the loaded tokenizer, and returns the tokenized sentence.

The code uses the map function from the datasets library to tokenize the input sentences in the test dataset.The remove_columns method is then used to remove the original text column, leaving only the tokenized input.

This DataLoader can then be used for iterating through batches of tokenized and padded input sequences during model training or evaluation. The DataCollatorWithPadding ensures that sequences within each batch are padded to the length of the longest sequence in that batch..

```
from torch.utils.data import DataLoader
from transformers import DataCollatorWithPadding
```

```
from transformers import AutoTokenizer

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)


# Tokenizer
def tokenize(sentence):
    sentence = tokenizer(sentence['text'], truncation=True)
    return sentence


tokenized_inputs = datasets['test'].map(tokenize)
tokenized_inputs = tokenized_inputs.remove_columns(['text'])


# DataCollator
batch = 16
data_collator = DataCollatorWithPadding(
    tokenizer=tokenizer, padding=True, return_tensors="pt")
dataloader = DataLoader(
    tokenized_inputs, batch_size=batch, collate_fn=data_collator)
```

Step 5 : Embedding Positional encoding

We create two class - One for positional encoding and one for embedding

Positional Encoding is responsible for generating the positional encodings used in Transformer models.

PositionalEmbedding class takes token as input and first embedes it. It then combines it with positional enoding, which are essential for capturing sequential information in Transformer models.

```
1
import torch
2
import torch.nn as nn
3
import torch.nn.functional as F
4
import torch.fft as fft
5
import numpy as np
6
import pandas as pd
7

8
class PositionalEncoding(torch.nn.Module):
9

10
```

```
11
    def __init__(self, d_model, max_sequence_length):
12
        super().__init__()
13
        self.d_model = d_model
14
        self.max_sequence_length = max_sequence_length
15
        self.positional_encoding = self.create_positional_encoding().to(device)
16


17
    def create_positional_encoding(self):
18


19
        # Initialize positional encoding matrix
20
        positional_encoding = np.zeros((self.max_sequence_length, self.d_model))
21


22
        # Calculate positional encoding for each position and each dimension
```

```
23
    for pos in range(self.max_sequence_length):
24
        for i in range(0, self.d_model, 2):
25
            # Apply sin to even indices in the array; indices in Python start at 0 so i is even.
26
            positional_encoding[pos, i] = np.sin(pos / (10000 ** ((2 * i) / self.d_model)))
27


28
            if i + 1 < self.d_model:
29
                # Apply cos to odd indices in the array; we add 1 to i because indices in Python start
at 0.
30
                positional_encoding[pos, i + 1] = np.cos(pos / (10000 ** ((2 * i) / self.d_model)))
31


32
    # Convert numpy array to PyTorch tensor and return it
33
    return torch.from_numpy(positional_encoding).float()
34


35
```

```
    def forward(self, x):
36
        expanded_tensor = torch.unsqueeze(self.positional_encoding, 0).expand(x.size(0), -1, -
1).to(device)
37


38
        return x.to(device) + expanded_tensor[:,:x.size(1), :]
39


40
class PositionalEmbedding(nn.Module):
41
  def __init__(self, sequence_length, vocab_size, embed_dim):
42
    super(PositionalEmbedding, self).__init__()
43
    self.token_embeddings = nn.Embedding(vocab_size, embed_dim)
44
    self.position_embeddings = PositionalEncoding(embed_dim,sequence_length)
45


46
  def forward(self, inputs):
47
    embedded_tokens = self.token_embeddings(inputs).to(device)
```

48

```
    embedded_positions = self.position_embeddings(embedded_tokens).to(device)
```

49

```
    return embedded_positions.to(device)
```

Step 6 : Create FNet Encoder

Below class implements the Fnet Encoder as per the Fnet architecture

This encoder layer incorporates the Fourier Transform as a key component in processing the input sequence. The Fourier Transform is applied to the input sequence, and the real part of the result is added to the original input. This is followed by a layer normalization and a dense projection, and the final result is normalized again.

Initialization (__init__ method):

The constructor initializes the encoder layer with parameters such as embed_dim (embedding dimension) and dense_dim (dimension of the intermediate dense layer).

A nn.Sequential block (self.dense_proj) is defined, consisting of two linear layers with ReLU activation in between, used for projecting the input to a different dimension.

Two instances of nn.LayerNorm are created (self.layernorm_1 and self.layernorm_2), each applied after a specific operation in the forward pass.

Forward Pass (forward method):

The forward method takes inputs as input, which represents the encoder inputs.

The Fourier Transform (fft.fft2) is applied to the inputs, and the real part of the result is extracted (fft_result.real.float()).

The original inputs are added to the real part of the Fourier Transform result, and layer normalization (self.layernorm_1) is applied to obtain proj_input.

The intermediate dense projection (self.dense_proj) is applied to proj_input, and the result is added to proj_input. The final layer normalization (self.layernorm_2) is applied, and the result is returned.

```
1
class FNetEncoder(nn.Module):
2


3
  def __init__(self,embed_dim, dense_dim):
4
    super(FNetEncoder,self).__init__()
5
    self.embed_dim = embed_dim
6
    self.dense_dim = dense_dim
7
    self.dense_proj = nn.Sequential(nn.Linear(self.embed_dim,self.dense_dim), nn.ReLU(),
nn.Linear(self.dense_dim,self.embed_dim))
8


9
    self.layernorm_1 = nn.LayerNorm(self.embed_dim)
10
    self.layernorm_2 = nn.LayerNorm(self.embed_dim)
11


12
  def forward(self,inputs):
```

```
13


14
    fft_result = fft.fft2(inputs)
15


16
    #taking real part
17
    fft_real = fft_result.real.float()
18


19
    proj_input = self.layernorm_1 (inputs + fft_real)
20
    proj_output = self.dense_proj(proj_input)
21
    return self.layernorm_2(proj_input +proj_output)
```

Step 7 : Create FnetDecoder

The decoder is based on the transformer architecture

Thie decoder layer employs multiple attention mechanisms, layer normalization, and a dense projection to capture dependencies and transform the information through the decoding process.

The first multihead attention takes the input passed into the decoder as its input in its qery,key and value .

The second multihead attention takes the input from first multiheadd attention in its query

vector and encoder output in its key and value vector .

The use of layer normalization after each step helps stabilize and normalize the intermediate representations.

Initialization (__init__ method):

The constructor initializes the decoder layer with parameters such as embed_dim (embedding dimension), dense_dim (dimension of the intermediate dense layer), and num_heads (number of attention heads).

Two instances of nn.MultiheadAttention are created (self.attention_1 and self.attention_2), each with embed_dim as the input and output dimension, and num_heads.

A nn.Sequential block (self.dense_proj) is defined, consisting of two linear layers with ReLU activation in between, used for projecting the output to a different dimension.

Three instances of nn.LayerNorm are created (self.layernorm_1, self.layernorm_2, and self.layernorm_3), each applied after a specific operation in the forward pass.

Forward Pass (forward method):

The forward method takes inputs (decoder inputs), encoder_outputs (outputs from the encoder), and an optional mask as inputs.

A causal mask is generated using nn.Transformer.generate_square_subsequent_mask to prevent attending to future tokens. This mask is applied to the first attention mechanism (self.attention_1).

The first attention mechanism (self.attention_1) attends to the decoder inputs (inputs) and applies layer normalization (self.layernorm_1). The result is added to the original inputs to form out_1.

If a mask is provided (this is available during training), the second attention mechanism (self.attention_2) applies attention with key padding mask (key_padding_mask) to the encoder outputs (encoder_outputs). Otherwise, it performs attention without any masking.

The result is added to out_1, and layer normalization (self.layernorm_2) is applied to obtain out_2.

The intermediate dense projection (self.dense_proj) is applied to out_2, and the result is added to out_2. The final layer normalization (self.layernorm_3) is applied, and the result is returned.

```
1
class FNetDecoder(nn.Module):
2


3
  def __init__(self,embed_dim,dense_dim,num_heads):
4
    super(FNetDecoder,self).__init__()
5
    self.embed_dim = embed_dim
6
    self.dense_dim = dense_dim
7
    self.num_heads = num_heads
8


9
    self.attention_1 = nn.MultiheadAttention(embed_dim,num_heads,batch_first=True)
10
    self.attention_2 = nn.MultiheadAttention(embed_dim,num_heads,batch_first=True)
11


12
    self.dense_proj = nn.Sequential(nn.Linear(embed_dim,
dense_dim),nn.ReLU(),nn.Linear(dense_dim, embed_dim))
```

```
13


14
    self.layernorm_1 = nn.LayerNorm(embed_dim)
15
    self.layernorm_2 = nn.LayerNorm(embed_dim)
16
    self.layernorm_3 = nn.LayerNorm(embed_dim)
17


18
  def forward(self, inputs, encoder_outputs, mask=None):
19
    causal_mask = nn.Transformer.generate_square_subsequent_mask(inputs.size(1)).to(device)
20


21
    attention_output_1, _ = self.attention_1(inputs, inputs, inputs, attn_mask=causal_mask)
22
    out_1 = self.layernorm_1(inputs + attention_output_1)
23


24
    if mask != None:
25
```

```
    attention_output_2, _ = self.attention_2(out_1, encoder_outputs, encoder_outputs,
key_padding_mask =torch.transpose(mask, 0, 1).to(device))
26
  else:
27
    attention_output_2, _ = self.attention_2(out_1, encoder_outputs, encoder_outputs)
28
  out_2 = self.layernorm_2(out_1 + attention_output_2)
29


30
  proj_output = self.dense_proj(out_2)
31
  return self.layernorm_3(out_2 + proj_output)
```

Step 8 : Fnet Model

We create a Model based on the positional encoding , fnet encoder and fnet decoder class declared above.

This model architecture have a stack of four encoder and four decoder layers, which allows for capturing complex dependencies in sequences

Before passing the input into encoder or decoder we pass the input through our embedding and positional encoding class

Initialization (__init__ method):

The constructor initializes the model with parameters such as max_length, vocab_size, embed_dim, latent_dim, and num_heads.

Four instances of FNetEncoder and FNetDecoder are created, each representing a layer of the encoder and decoder, respectively.

Positional embeddings (PositionalEmbedding) are used for both the encoder and decoder

inputs to incorporate positional information.

A dropout layer (nn.Dropout) with a dropout rate of 0.5 is added.

A linear layer (nn.Linear) is used for the final dense output with a size of vocab_size.

Encoder (encoder method):

The encoder method takes encoder_inputs as input and passes it through the positional embedding layer and four instances of the FNetEncoder sequentially.

Each FNetEncoder processes the input sequentially, contributing to the overall encoder output.

Decoder (decoder method):

The decoder method takes decoder_inputs, encoder_output, and an optional att_mask as inputs.

Similar to the encoder, it processes the decoder inputs using the positional embedding layer and four instances of the FNetDecoder.

Each FNetDecoder processes the input sequentially, taking into account the encoder output and an attention mask.

The final output is obtained by passing the decoder output through a linear layer (nn.Linear).

Forward Pass (forward method):

The forward method is the entry point for the forward pass of the model.

It takes encoder_inputs, decoder_inputs, and an optional att_mask.

It calls the encoder method to obtain the encoder output.

The encoder output is then used in the decoder method to generate the final decoder output.

The decoder output is returned as the result of the forward pass.

```
1
class FNetModel(nn.Module):
2
    def __init__(self, max_length, vocab_size, embed_dim, latent_dim, num_heads):
```

```
3
    super(FNetModel, self).__init__()
4


5
    self.encoder_inputs = PositionalEmbedding(max_length,vocab_size, embed_dim)
6
    self.encoder1 = FNetEncoder(embed_dim, latent_dim)
7
    self.encoder2 = FNetEncoder(embed_dim, latent_dim)
8
    self.encoder3 = FNetEncoder(embed_dim, latent_dim)
9
    self.encoder4 = FNetEncoder(embed_dim, latent_dim)
10


11


12
    self.decoder_inputs = PositionalEmbedding(max_length,vocab_size, embed_dim)
13
    self.decoder1 = FNetDecoder(embed_dim, latent_dim, num_heads)
14
    self.decoder2 = FNetDecoder(embed_dim, latent_dim, num_heads)
15
```

```
        self.decoder3 = FNetDecoder(embed_dim, latent_dim, num_heads)
16
        self.decoder4 = FNetDecoder(embed_dim, latent_dim, num_heads)
17
18
19
        self.dropout = nn.Dropout(0.5)
20
        self.dense = nn.Linear(embed_dim, vocab_size)
21
22
    def encoder(self,encoder_inputs):
23
        x_encoder = self.encoder_inputs(encoder_inputs)
24
        x_encoder = self.encoder1(x_encoder)
25
        x_encoder = self.encoder2(x_encoder)
26
        x_encoder = self.encoder3(x_encoder)
27
        x_encoder = self.encoder4(x_encoder)
```

```
28
        return x_encoder
29


30
    def decoder(self,decoder_inputs,encoder_output,att_mask):
31
        x_decoder = self.decoder_inputs(decoder_inputs)
32
        x_decoder = self.decoder1(x_decoder, encoder_output,att_mask) ## HERE for inference
33
        x_decoder = self.decoder2(x_decoder, encoder_output,att_mask) ## HERE for inference
34
        x_decoder = self.decoder3(x_decoder, encoder_output,att_mask) ## HERE for inference
35
        x_decoder = self.decoder4(x_decoder, encoder_output,att_mask) ## HERE for inference
36
        decoder_outputs = self.dense(x_decoder)
37


38
        return decoder_outputs
39


40
```

```
    def forward(self, encoder_inputs, decoder_inputs,att_mask = None):
41
        encoder_output = self.encoder(encoder_inputs)
42
        decoder_output = self.decoder(decoder_inputs,encoder_output,att_mask=None)
43
        return decoder_output
```

Step 9 : Initialize Model

We declare hyperparameters and initialize our model

```
1
# Assuming your constants are defined like this:
2
MAX_LENGTH = 512
3
VOCAB_SIZE = len(tokenizer.vocab)
4
EMBED_DIM = 256
5
LATENT_DIM = 100
6
```

NUM_HEADS = 4

7

8

Create an instance of the model

9

fnet_model = FNetModel(MAX_LENGTH, VOCAB_SIZE, EMBED_DIM, LATENT_DIM, NUM_HEADS).to(device)

Step 10 : Train the model

We declare our optimizer and loss function.

An Adam optimizer is defined for updating the parameters of the model during training.

CrossEntropyLoss is chosen as the loss function.

We then train our model for 10 epochs

The training dataset is iterated through batches using a dataloader.

For each batch, the input sequences (encoder_inputs_tensor) and target sequences (decoder_inputs_tensor) are extracted. The decoder input is shifted by one position ([:, 1:]), for teacher forcing in sequence generation tasks.

An attention mask (att_mask) is applied to the input sequences to handle padding. The mask is set to True for valid tokens and False for padding tokens.

The optimizer's gradients are zeroed using optimizer.zero_grad() to prepare for a new backward pass.

The model (fnet_model) is then used to generate predictions (outputs) based on the encoder and decoder inputs.

A masked version of the target sequences (decoder_inputs_tensor) is created, where padding positions are filled with a value of -100. This is a common strategy to exclude padding positions from contributing to the loss.

The CrossEntropyLoss is computed between the model's outputs and the masked target sequences.

The loss is accumulated in the train_loss variable.

Backpropagation is performed using loss.backward() to compute gradients.

The optimizer is updated using optimizer.step().

```
# Define your optimizer and loss function
optimizer = torch.optim.Adam(fnet_model.parameters())
criterion = nn.CrossEntropyLoss(ignore_index=0)


epochs = 100
for epoch in range(epochs):
    train_loss = 0
    for batch in dataloader:
        encoder_inputs_tensor = batch['input_ids'][:,:-1].to(device)
```

```
        decoder_inputs_tensor = batch['input_ids'][:,1:].to(device)
11
12
        att_mask = batch['attention_mask'][:,:-1].to(device).to(dtype=bool)
13
        optimizer.zero_grad()
14
        outputs = fnet_model(encoder_inputs_tensor, decoder_inputs_tensor,att_mask)
15
        decoder_inputs_tensor.masked_fill(batch['attention_mask'][:,1:].ne(1).to(device), -
100).to(device)
16
17
        loss = criterion(outputs.view(-1, VOCAB_SIZE), decoder_inputs_tensor.reshape(-1))
18
        train_loss = train_loss + loss.item()
19
        loss.backward()
20
        optimizer.step()
21
    print (f" epoch: {epoch}, train_loss : {train_loss}")
22
```

23

This code is modified by Susobhan Akhuli

Output:

epoch: 0, train_loss : 13.495175334392115

epoch: 1, train_loss : 0.9018354846921284

epoch: 2, train_loss : 0.3800733484386001

epoch: 3, train_loss : 0.626482578649302

epoch: 4, train_loss : 460.4480260747587

Step 11 : Use model for text generation

To perform text generation using a Transformer decoder, we can use a technique called "autoregressive decoding," where we iteratively generate one token at a time by sampling from the model's output distribution and feeding the sampled token back into the input for the next step. We use the encoder part of the model to generate context vector for a given input token.

1

MAX_LENGTH = 100 # your MAX_LENGTH value

2

3

4

def decode_sentence(input_sentence, fnet_model):

```
5
   fnet_model.eval()
6


7
   with torch.no_grad():
8
      tokenized_input_sentence = torch.tensor(
9
         tokenizer(preprocess_text(input_sentence)['text'])['input_ids']).to(device)
10
      tokenzied_target_sentence = torch.tensor(
11
         [101]).to(device)  # '[CLS]' token
12
      current_text = preprocess_text(input_sentence)['text']
13
      for i in range(MAX_LENGTH):
14
         predictions = fnet_model(
15
            tokenized_input_sentence[:-1].unsqueeze(0), tokenzied_target_sentence.unsqueeze(0))
16
         predicted_index = torch.argmax(predictions[0, -1, :]).item()
17
```

```
        predicted_token = tokenizer.decode(predicted_index)
18
        if predicted_token == "[SEP]":  # Assuming [end] is the end token
19
            break
20
        current_text += " " + predicted_token
21
        tokenized_target_sentence = torch.cat([tokenzied_target_sentence, torch.tensor([
22
                                    predicted_index]).to(device)], 0).to(device)
23
        tokenized_input_sentence = torch.tensor(
24
            tokenizer(current_text)['input_ids']).to(device)
25
    return current_text
26


27


28
decode_sentence({'text': 'How are you ?'}, fnet_model)
```

Output:

'how are you ? mort ##ries ke ke ke writing ke ##ries writing h h writing ke writing writing ke writing h h h writing h

In order to get a better output we need to train the model with large amount of data and for significant time which will require GPUs.

Get complete notebook link:

Notebook : click here.

Conclusion

The article then delved into the implementation of FNet architecture for text generation using PyTorch in Python. The step-by-step guide covered data loading, preprocessing, tokenization, embedding positional encoding, and the creation of FNet encoder and decoder classes. A complete FNet model was constructed and trained on a dataset, demonstrating the training process and providing insights into model performance through training loss monitoring.