# **Home Assignment: Implementing a PyTorch-Based Transformer**

## **Objective**

The goal of this assignment is to help you understand the internal workings of the Transformer architecture by implementing it from scratch using PyTorch. You will build a minimal version of the Transformer encoder and apply it to a toy language modeling or sequence classification task.

## **Tasks**

· Implement the following modules in PyTorch:  
- Positional Encoding  
- Multi-head Self-Attention  
- Feedforward Network (Position-wise FFN)  
- Transformer Encoder Layer  
- Full Transformer Encoder

· Build a training pipeline:  
- Dataset: You can use a synthetic dataset (e.g., copying sequences, toy classification) or a small dataset like AG News or SST-2.  
- Tokenizer: Use torchtext or HuggingFace tokenizers.  
- Loss: Cross-entropy loss or similar.  
- Optimizer: Adam or AdamW.

· Train and evaluate the model:  
- Track training and validation loss.  
- Report final accuracy or perplexity.

· Optional:  
- Add masking support (e.g., padding mask).  
- Extend to a decoder-only architecture like GPT (for extra credit).  
- Visualize attention weights for interpretability.

## **What to Submit**

· A GitHub repo or ZIP file with:  
- All source code with clear structure and comments.  
- A README.md that explains:  
 • How to run the code  
 • Your design choices  
 • Results and what you observed

· Jupyter Notebook (optional) demonstrating model training and evaluation.

## **Installation with pip install -c requirement.txt as following**

## **## Use python 3.9**

## 

## **torch==2.0.1**

## **torchvision==0.15.2**

## **torchaudio==2.0.2**

## **torchtext==0.15.2**

## **datasets==2.15.0**

## **tokenizers==0.13.3**

## **torchmetrics==1.0.3**

## **tensorboard==2.13.0**

## **altair==5.1.1**

## **wandb==0.15.9**

## 

## **Code Skeletons**

1. model.py

import torch

import torch.nn as nn

import math

class LayerNormalization(nn.Module):

def \_\_init\_\_(self, features: int, eps:float=10\*\*-6) -> None:

super().\_\_init\_\_()

def forward(self, x):

class FeedForwardBlock(nn.Module):

def \_\_init\_\_(self, d\_model: int, d\_ff: int, dropout: float) -> None:

super().\_\_init\_\_()

def forward(self, x):

# (batch, seq\_len, d\_model) --> (batch, seq\_len, d\_ff) --> (batch, seq\_len, d\_model)

class InputEmbeddings(nn.Module):

def \_\_init\_\_(self, d\_model: int, vocab\_size: int) -> None:

super().\_\_init\_\_()

def forward(self, x):

# (batch, seq\_len) --> (batch, seq\_len, d\_model)

# Multiply by sqrt(d\_model) to scale the embeddings according to the paper

class PositionalEncoding(nn.Module):

def \_\_init\_\_(self, d\_model: int, seq\_len: int, dropout: float) -> None:

super().\_\_init\_\_()

def forward(self, x):

class ResidualConnection(nn.Module):

def \_\_init\_\_(self, features: int, dropout: float) -> None:

def forward(self, x, sublayer):

class MultiHeadAttentionBlock(nn.Module):

def \_\_init\_\_(self, d\_model: int, h: int, dropout: float) -> None:

super().\_\_init\_\_()

@staticmethod

def attention(query, key, value, mask, dropout: nn.Dropout):

def forward(self, q, k, v, mask):

class EncoderBlock(nn.Module):

def \_\_init\_\_(self, features: int, self\_attention\_block: MultiHeadAttentionBlock, feed\_forward\_block: FeedForwardBlock, dropout: float) -> None:

def forward(self, x, src\_mask):

class Encoder(nn.Module):

def \_\_init\_\_(self, features: int, layers: nn.ModuleList) -> None:

super().\_\_init\_\_()

self.layers = layers

self.norm = LayerNormalization(features)

def forward(self, x, mask):

class DecoderBlock(nn.Module):

def \_\_init\_\_(self, features: int, self\_attention\_block: MultiHeadAttentionBlock, cross\_attention\_block: MultiHeadAttentionBlock, feed\_forward\_block: FeedForwardBlock, dropout: float) -> None:

super().\_\_init\_\_()

def forward(self, x, encoder\_output, src\_mask, tgt\_mask):

class Decoder(nn.Module):

def \_\_init\_\_(self, features: int, layers: nn.ModuleList) -> None:

super().\_\_init\_\_()

self.layers = layers

self.norm = LayerNormalization(features)

def forward(self, x, encoder\_output, src\_mask, tgt\_mask):

class ProjectionLayer(nn.Module):

def \_\_init\_\_(self, d\_model, vocab\_size) -> None:

super().\_\_init\_\_()

self.proj = nn.Linear(d\_model, vocab\_size)

def forward(self, x) -> None:

# (batch, seq\_len, d\_model) --> (batch, seq\_len, vocab\_size)

return self.proj(x)

class Transformer(nn.Module):

def \_\_init\_\_(self, encoder: Encoder, decoder: Decoder, src\_embed: InputEmbeddings, tgt\_embed: InputEmbeddings, src\_pos: PositionalEncoding, tgt\_pos: PositionalEncoding, projection\_layer: ProjectionLayer) -> None:

super().\_\_init\_\_()

self.encoder = encoder

self.decoder = decoder

self.src\_embed = src\_embed

self.tgt\_embed = tgt\_embed

self.src\_pos = src\_pos

self.tgt\_pos = tgt\_pos

self.projection\_layer = projection\_layer

def encode(self, src, src\_mask):

def decode(self, encoder\_output: torch.Tensor, src\_mask: torch.Tensor, tgt: torch.Tensor, tgt\_mask: torch.Tensor):

def project(self, x):

# (batch, seq\_len, vocab\_size)

return self.projection\_layer(x)

def build\_transformer(src\_vocab\_size: int, tgt\_vocab\_size: int, src\_seq\_len: int, tgt\_seq\_len: int, d\_model: int=512, N: int=6, h: int=8, dropout: float=0.1, d\_ff: int=2048) -> Transformer:

# Create the embedding layers

src\_embed = InputEmbeddings(d\_model, src\_vocab\_size)

tgt\_embed = InputEmbeddings(d\_model, tgt\_vocab\_size)

# Create the positional encoding layers

src\_pos = PositionalEncoding(d\_model, src\_seq\_len, dropout)

tgt\_pos = PositionalEncoding(d\_model, tgt\_seq\_len, dropout)

# Create the encoder blocks

encoder\_blocks = []

for \_ in range(N):

return transformer

1. [dataset.py](http://dataset.py)

import torch

import torch.nn as nn

from torch.utils.data import Dataset

class BilingualDataset(Dataset):

def \_\_init\_\_(self, ds, tokenizer\_src, tokenizer\_tgt, src\_lang, tgt\_lang, seq\_len):

super().\_\_init\_\_()

self.seq\_len = seq\_len

self.ds = ds

self.tokenizer\_src = tokenizer\_src

self.tokenizer\_tgt = tokenizer\_tgt

self.src\_lang = src\_lang

self.tgt\_lang = tgt\_lang

self.sos\_token = torch.tensor([tokenizer\_tgt.token\_to\_id("[SOS]")], dtype=torch.int64)

self.eos\_token = torch.tensor([tokenizer\_tgt.token\_to\_id("[EOS]")], dtype=torch.int64)

self.pad\_token = torch.tensor([tokenizer\_tgt.token\_to\_id("[PAD]")], dtype=torch.int64)

def \_\_len\_\_(self):

return len(self.ds)

def \_\_getitem\_\_(self, idx):

def causal\_mask(size):

mask = torch.triu(torch.ones((1, size, size)), diagonal=1).type(torch.int)

return mask == 0

1. train\_wb.py

from model import build\_transformer

from dataset import BilingualDataset, causal\_mask

from config import get\_config, get\_weights\_file\_path

import torchtext.datasets as datasets

import torch

import torch.nn as nn

from torch.utils.data import Dataset, DataLoader, random\_split

from torch.optim.lr\_scheduler import LambdaLR

import warnings

from tqdm import tqdm

import os

from pathlib import Path

# Huggingface datasets and tokenizers

from datasets import load\_dataset

from tokenizers import Tokenizer

from tokenizers.models import WordLevel

from tokenizers.trainers import WordLevelTrainer

from tokenizers.pre\_tokenizers import Whitespace

import wandb

import torchmetrics

def greedy\_decode(model, source, source\_mask, tokenizer\_src, tokenizer\_tgt, max\_len, device):

return decoder\_input.squeeze(0)

def run\_validation(model, validation\_ds, tokenizer\_src, tokenizer\_tgt, max\_len, device, print\_msg, global\_step, num\_examples=2):

wandb.log({'validation/BLEU': bleu, 'global\_step': global\_step})

def get\_all\_sentences(ds, lang):

for item in ds:

yield item['translation'][lang]

def get\_or\_build\_tokenizer(config, ds, lang):

tokenizer\_path = Path(config['tokenizer\_file'].format(lang))

return tokenizer

def get\_ds(config):

# It only has the train split, so we divide it overselves

ds\_raw = load\_dataset('opus\_books', f"{config['lang\_src']}-{config['lang\_tgt']}", split='train')

return train\_dataloader, val\_dataloader, tokenizer\_src, tokenizer\_tgt

def get\_model(config, vocab\_src\_len, vocab\_tgt\_len):

model = build\_transformer(vocab\_src\_len, vocab\_tgt\_len, config["seq\_len"], config['seq\_len'], d\_model=config['d\_model'])

return model

def train\_model(config):

# Define the device

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

print("Using device:", device)

if \_\_name\_\_ == '\_\_main\_\_':

warnings.filterwarnings("ignore")

config = get\_config()

config['num\_epochs'] = 30

config['preload'] = None

wandb.init(

# set the wandb project where this run will be logged

project="pytorch-transformer",

# track hyperparameters and run metadata

config=config

)

train\_model(config)

1. [translate.py](http://translate.py)

from pathlib import Path

from config import get\_config, latest\_weights\_file\_path

from model import build\_transformer

from tokenizers import Tokenizer

from datasets import load\_dataset

from dataset import BilingualDataset

import torch

import sys

def translate(sentence: str):

# Define the device, tokenizers, and model

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

print("Using device:", device)

config = get\_config()

tokenizer\_src = Tokenizer.from\_file(str(Path(config['tokenizer\_file'].format(config['lang\_src']))))

tokenizer\_tgt = Tokenizer.from\_file(str(Path(config['tokenizer\_file'].format(config['lang\_tgt']))))

model = build\_transformer(tokenizer\_src.get\_vocab\_size(), tokenizer\_tgt.get\_vocab\_size(), config["seq\_len"], config['seq\_len'], d\_model=config['d\_model']).to(device)

# Load the pretrained weights

model\_filename = latest\_weights\_file\_path(config)

state = torch.load(model\_filename)

model.load\_state\_dict(state['model\_state\_dict'])

# if the sentence is a number use it as an index to the test set

label = ""

if type(sentence) == int or sentence.isdigit():

id = int(sentence)

ds = load\_dataset(f"{config['datasource']}", f"{config['lang\_src']}-{config['lang\_tgt']}", split='all')

ds = BilingualDataset(ds, tokenizer\_src, tokenizer\_tgt, config['lang\_src'], config['lang\_tgt'], config['seq\_len'])

sentence = ds[id]['src\_text']

label = ds[id]["tgt\_text"]

seq\_len = config['seq\_len']

# translate the sentence

model.eval()

with torch.no\_grad():

# Precompute the encoder output and reuse it for every generation step

source = tokenizer\_src.encode(sentence)

source = torch.cat([

torch.tensor([tokenizer\_src.token\_to\_id('[SOS]')], dtype=torch.int64),

torch.tensor(source.ids, dtype=torch.int64),

torch.tensor([tokenizer\_src.token\_to\_id('[EOS]')], dtype=torch.int64),

torch.tensor([tokenizer\_src.token\_to\_id('[PAD]')] \* (seq\_len - len(source.ids) - 2), dtype=torch.int64)

], dim=0).to(device)

source\_mask = (source != tokenizer\_src.token\_to\_id('[PAD]')).unsqueeze(0).unsqueeze(0).int().to(device)

encoder\_output = model.encode(source, source\_mask)

# Initialize the decoder input with the sos token

decoder\_input = torch.empty(1, 1).fill\_(tokenizer\_tgt.token\_to\_id('[SOS]')).type\_as(source).to(device)

# Print the source sentence and target start prompt

if label != "": print(f"{f'ID: ':>12}{id}")

print(f"{f'SOURCE: ':>12}{sentence}")

if label != "": print(f"{f'TARGET: ':>12}{label}")

print(f"{f'PREDICTED: ':>12}", end='')

# Generate the translation word by word

while decoder\_input.size(1) < seq\_len:

# build mask for target and calculate output

decoder\_mask = torch.triu(torch.ones((1, decoder\_input.size(1), decoder\_input.size(1))), diagonal=1).type(torch.int).type\_as(source\_mask).to(device)

out = model.decode(encoder\_output, source\_mask, decoder\_input, decoder\_mask)

# project next token

prob = model.project(out[:, -1])

\_, next\_word = torch.max(prob, dim=1)

decoder\_input = torch.cat([decoder\_input, torch.empty(1, 1).type\_as(source).fill\_(next\_word.item()).to(device)], dim=1)

# print the translated word

print(f"{tokenizer\_tgt.decode([next\_word.item()])}", end=' ')

# break if we predict the end of sentence token

if next\_word == tokenizer\_tgt.token\_to\_id('[EOS]'):

break

# convert ids to tokens

return tokenizer\_tgt.decode(decoder\_input[0].tolist())

#read sentence from argument

translate(sys.argv[1] if len(sys.argv) > 1 else "I am not a very good a student.")

1. [config.py](http://config.py)

from pathlib import Path

def get\_config():

return {

"batch\_size": 8,

"num\_epochs": 20,

"lr": 10\*\*-4,

"seq\_len": 350,

"d\_model": 512,

"datasource": 'opus\_books',

"lang\_src": "en",

"lang\_tgt": "it",

"model\_folder": "weights",

"model\_basename": "tmodel\_",

"preload": "latest",

"tokenizer\_file": "tokenizer\_{0}.json",

"experiment\_name": "runs/tmodel"

}

def get\_weights\_file\_path(config, epoch: str):

model\_folder = f"{config['datasource']}\_{config['model\_folder']}"

model\_filename = f"{config['model\_basename']}{epoch}.pt"

return str(Path('.') / model\_folder / model\_filename)

# Find the latest weights file in the weights folder

def latest\_weights\_file\_path(config):

model\_folder = f"{config['datasource']}\_{config['model\_folder']}"

model\_filename = f"{config['model\_basename']}\*"

weights\_files = list(Path(model\_folder).glob(model\_filename))

if len(weights\_files) == 0:

return None

weights\_files.sort()

return str(weights\_files[-1])