ChatGPT on your own terms: Building your own language model

This article showcases how you can build your own version of ChatGPT using PyTorch. We are not going to be able to reproduce the exact replica of chatGPT as it is a production level system trained on a very big chunk of the internet with various pre-training and fine tuning stages. What I would like to focus on is just the transformer based language model or the underlying logic behind the chatGPT. And in our case, it's going to be a character based language model. Because of the vast amount of data on the internet, it is practically impossible for me to train the model on the entire data available on the internet. So, I'm going to train the model on a toy dataset which comprises all of Shakespeare's work. You can use your own arbitrary text dataset. Before beginning with the implementation of a transformer-based language model it is important to understand the theory behind it.

Introduction to Transformer Based Language Model: The transformer architecture was first introduced in the paper "Attention is All You Need" by Google researchers in 2017. A transformer-based language model is a type of neural network architecture that is used for natural language processing tasks such as language translation, text summarization, and language generation. The key innovation of the transformer architecture is the attention mechanism, which allows the model to weigh the importance of different parts of the input when making predictions.

Architecture: The architecture of a transformer-based language model typically consists of the following components: Embedding Layer, Encoder, Decoder, Output Layer

- **Embedding Layer**→The first step in processing the input is to map the words in the input sentence to a high-dimensional vector representation, known as word embeddings. These embeddings are learned during the training process and capture the semantic and syntactic properties of the words.
- **Encoder**→ The encoder is a stack of multiple layers, each of which is composed of two sub-layers: a multi-head self-attention mechanism and a feed-forward neural network. The self-attention mechanism allows the model to weigh the importance of different parts of the input when making predictions. The feed-forward neural network applies a non-linear transformation to the input.
- **Decoder**→ The decoder is also a stack of multiple layers, similar to the encoder. The decoder also uses a multi-head self-attention mechanism and feed-forward neural network in each layer. The decoder uses the encoder's output to generate the final output.
- **Output Layer**→ The final output is generated by applying a linear transformation to the output of the last layer of the decoder.

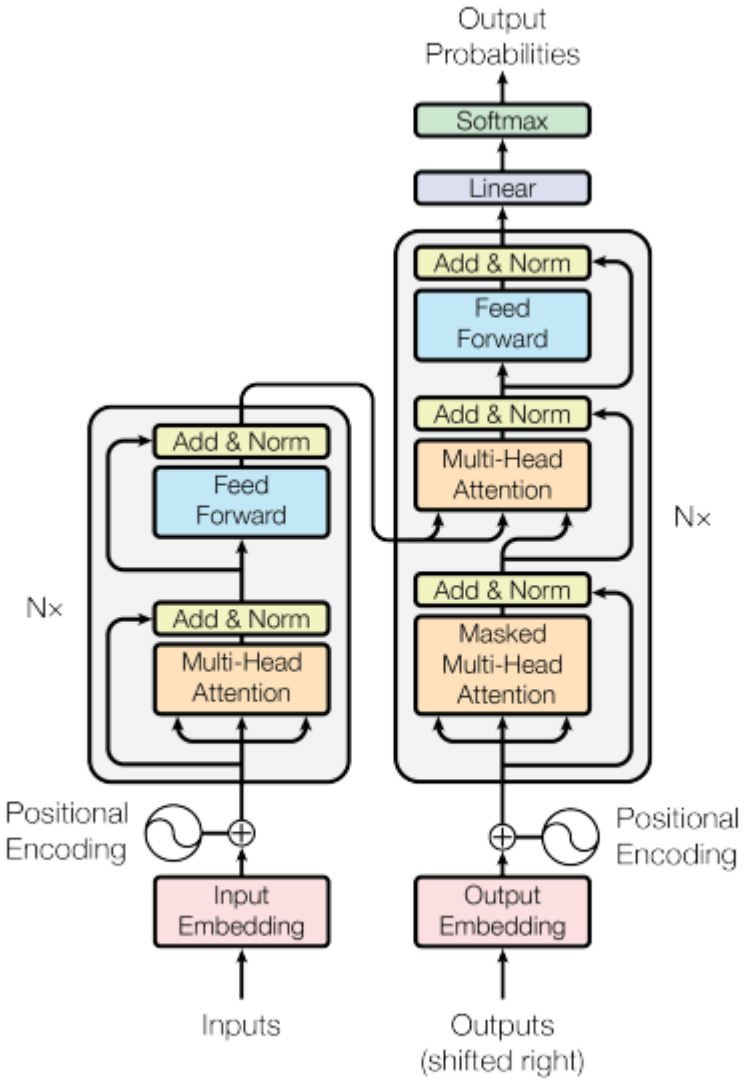

Figure 1: The Transformer - model architecture.

We are only going to implement a decoder only transformer. We are not going to implement the encoder and cross-attention. Our transformer will only have self-attention and Feed Forward NN. The reason why it's decoder only is because it generates text unconditionally on anything.

Let's get started with the implementation part:

- **Input** → I'm going to use a dataset which contains all of Shakespeare's work and has a size of 1MB. This can be downloaded from here. What we are going to do is model how these characters follow each other. For example, we select a chunk of characters from the dataset as contexts and pass it on to the transformer model and predict the next character based on the previous chunk of characters.

- **Code** → Reading the input.txt and importing necessary libraries with setting up the hyperparameters. If you have a GPU, you can directly train on ‘cuda’.

```
import torch
import torch.nn as nn
from torch.nn import functional as F

# hyperparameters
batch_size = 64 # how many independent sequences will we process in
parallel?
block_size = 256 # what is the maximum context length for predictions?
max_iters = 5000
eval_interval = 500
learning_rate = 3e-4
device = 'cuda' if torch.cuda.is_available() else 'cpu'
eval_iters = 200
n_embd = 384
n_head = 6
n_layer = 6
dropout = 0.2
# ------------

torch.manual_seed(1337)

with open('WikiQA-train.txt', 'r', encoding='utf-8') as f:
    text = f.read()

## Here are all the unique characters in a text

chars = sorted(list(set(text)))
vocab_size = len(chars)
print(''.join(chars))
print(vocab_size)
```

```
Output:
!$&',-.3:;?ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
65
```

Tokenization Process using encoder and decoder

```
# create a mapping from characters to integers
stoi = { ch:i for i,ch in enumerate(chars) }
itos = { i:ch for i,ch in enumerate(chars) }
encode = lambda s: [stoi[c] for c in s] # encoder: take a string, output a
```

```
list of integers
decode = lambda l: ''.join([itos[i] for i in l]) # decoder: take a list of
integers, output a string
```

Splitting the dataset into training and testing. 90% of the dataset will be used for training and the rest will be used for the validation dataset.

```
# Train and test splits
data = torch.tensor(encode(text), dtype=torch.long)
n = int(0.9*len(data)) # first 90% will be train, rest val
train_data = data[:n]
val_data = data[n:]
```

Data Loading: Everytime we are going to feed inputs to the transformer, we are going to have many batches of multiple chunks of text that are stacked up in a single tensor. It is done for efficiency as GPU's are very good at parallel processing of data. The 1-dimensional arrays are going to be stacked up to form 4x8 tensor.

```
def get_batch(split):
    # generate a small batch of data of inputs x and targets y
    data = train_data if split == 'train' else val_data
    ix = torch.randint(len(data) - block_size, (batch_size,))
    x = torch.stack([data[i:i+block_size] for i in ix])
    y = torch.stack([data[i+1:i+block_size+1] for i in ix])
    x, y = x.to(device), y.to(device)
    return x, y
```

Next is the loss estimation function which will be used to calculate loss at each iteration.

```
@torch.no_grad()
def estimate_loss():
    out = {}
    model.eval()
    for split in ['train', 'val']:
        losses = torch.zeros(eval_iters)
        for k in range(eval_iters):
            X, Y = get_batch(split)
            logits, loss = model(X, Y)
            losses[k] = loss.item()
        out[split] = losses.mean()
    model.train()
    return out
```

Class Head is the real crux of self–attention. (B,T, C) are three parameters i.e Batch_size, Time and Channels.In the self-attention mechanism, the input is first transformed using a set of linear transformations, also known as queries, keys and

values. The attention weights are then calculated by taking the dot product of the queries and keys, and passing the result through a softmax function. The self-attention mechanism allows the model to selectively focus on the most relevant parts of the input and capture the long term dependencies between input elements.

```
class Head(nn.Module):
    """ one head of self-attention """

    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(n_embd, head_size, bias=False)
        self.query = nn.Linear(n_embd, head_size, bias=False)
        self.value = nn.Linear(n_embd, head_size, bias=False)
        self.register_buffer('tril', torch.tril(torch.ones(block_size,
block_size)))

        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        B,T,C = x.shape
        k = self.key(x)   # (B,T,C)
        q = self.query(x) # (B,T,C)
        # compute attention scores ("affinities")
        wei = q @ k.transpose(-2,-1) * C**-0.5 # (B, T, C) @ (B, C, T) ->
(B, T, T)
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf')) # (B,
T, T)
        wei = F.softmax(wei, dim=-1) # (B, T, T)
        wei = self.dropout(wei)
        # perform the weighted aggregation of the values
        v = self.value(x) # (B,T,C)
        out = wei @ v # (B, T, T) @ (B, T, C) -> (B, T, C)
        return out
```

Multi-head attention is multiple heads of self-attention running in parallel simultaneously. In Pytorch, we can do this by simply creating multiple heads. You can add whatever number of heads you want and all these heads take in head_size as its parameter. We simply concatenate the outputs over channel dimensions.

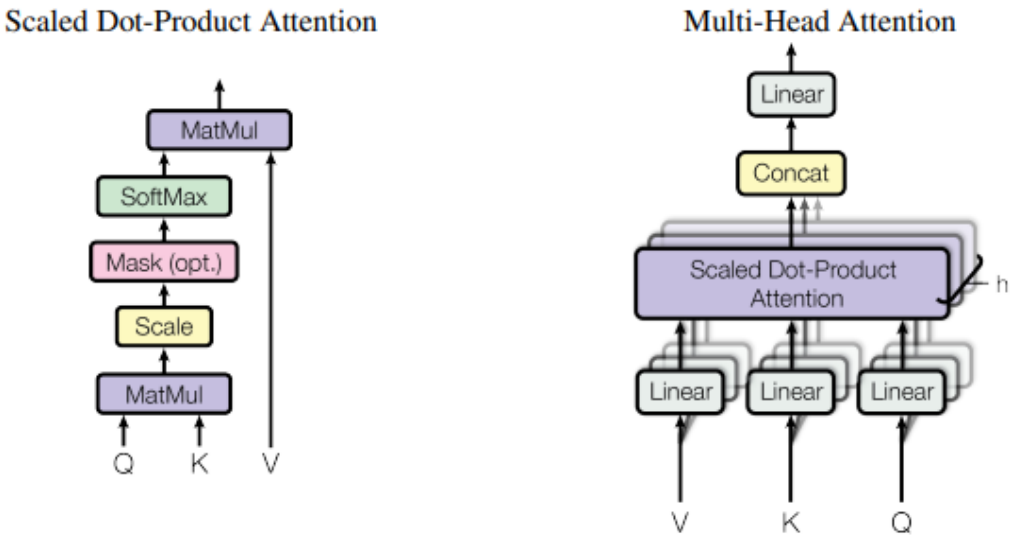

Figure 2. Regular Self-Attention vs Multi-head Attention

```
class MultiHeadAttention(nn.Module):
    """ multiple heads of self-attention in parallel """

    def __init__(self, num_heads, head_size):
        super().__init__()
        self.heads = nn.ModuleList([Head(head_size) for _ in
range(num_heads)])
        self.proj = nn.Linear(n_embd, n_embd)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        out = torch.cat([h(x) for h in self.heads], dim=-1)
        out = self.dropout(self.proj(out))
        return out
```

You can see some of the components of the transformer network have been implemented but the Feed-Forward Neural Network component isn't. Next module/component is a regular Feed-Forward Neural Network.

```
class FeedFoward(nn.Module):
    """ a simple linear layer followed by a non-linearity """

    def __init__(self, n_embd):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),
            nn.ReLU(),
            nn.Linear(4 * n_embd, n_embd),
            nn.Dropout(dropout),
        )
```

```
    def forward(self, x):
        return self.net(x)
```

Block is the entire transformer except for the cross-attention/

```
class Block(nn.Module):
    """ Transformer block: communication followed by computation """

    def __init__(self, n_embd, n_head):
        # n_embd: embedding dimension, n_head: the number of heads we'd like
        super().__init__()
        head_size = n_embd // n_head
        self.sa = MultiHeadAttention(n_head, head_size)
        self.ffwd = FeedFoward(n_embd)
        self.ln1 = nn.LayerNorm(n_embd)
        self.ln2 = nn.LayerNorm(n_embd)

    def forward(self, x):
        x = x + self.sa(self.ln1(x))
        x = x + self.ffwd(self.ln2(x))
        return x
```

Our outputs are ready, let's start by feeding them into the neural network. I have implemented one of the simplest neural networks in the case of language modelling which is 'BigramLanguageModel'. A bigram language model is a type of statistical language model that is trained to predict the next word in a sequence based on the previous two words. Bigram models are an extension of unigram models. This is the last piece of code we need to train our language model.

```
# super simple bigram model
class BigramLanguageModel(nn.Module):

    def __init__(self):
        super().__init__()
        # each token directly reads off the logits for the next token from a
lookup table
        self.token_embedding_table = nn.Embedding(vocab_size, n_embd)
        self.position_embedding_table = nn.Embedding(block_size, n_embd)
        self.blocks = nn.Sequential(*[Block(n_embd, n_head=n_head) for _ in
range(n_layer)])
        self.ln_f = nn.LayerNorm(n_embd) # final layer norm
        self.lm_head = nn.Linear(n_embd, vocab_size)

    def forward(self, idx, targets=None):
        B, T = idx.shape

        # idx and targets are both (B,T) tensor of integers
        tok_emb = self.token_embedding_table(idx) # (B,T,C)
```

```
        pos_emb = self.position_embedding_table(torch.arange(T,
device=device)) # (T,C)
        x = tok_emb + pos_emb # (B,T,C)
        x = self.blocks(x) # (B,T,C)
        x = self.ln_f(x) # (B,T,C)
        logits = self.lm_head(x) # (B,T,vocab_size)

        if targets is None:
            loss = None
        else:
            B, T, C = logits.shape
            logits = logits.view(B*T, C)
            targets = targets.view(B*T)
            loss = F.cross_entropy(logits, targets)

        return logits, loss

    def generate(self, idx, max_new_tokens):
        # idx is (B, T) array of indices in the current context
        for _ in range(max_new_tokens):
            # crop idx to the last block_size tokens
            idx_cond = idx[:, -block_size:]
            # get the predictions
            logits, loss = self(idx_cond)
            # focus only on the last time step
            logits = logits[:, -1, :] # becomes (B, C)
            # apply softmax to get probabilities
            probs = F.softmax(logits, dim=-1) # (B, C)
            # sample from the distribution
            idx_next = torch.multinomial(probs, num_samples=1) # (B, 1)
            # append sampled index to the running sequence
            idx = torch.cat((idx, idx_next), dim=1) # (B, T+1)
        return idx

model = BigramLanguageModel()
m = model.to(device)
# print the number of parameters in the model
print(sum(p.numel() for p in m.parameters())/1e6, 'M parameters')

# create a PyTorch optimizer
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)

for iter in range(max_iters):

    # every once in a while evaluate the loss on train and val sets
    if iter % eval_interval == 0 or iter == max_iters - 1:
        losses = estimate_loss()
        print(f"step {iter}: train loss {losses['train']:.4f}, val loss
```

```
{losses['val']:.4f}")

    # sample a batch of data
    xb, yb = get_batch('train')

    # evaluate the loss
    logits, loss = model(xb, yb)
    optimizer.zero_grad(set_to_none=True)
    loss.backward()
    optimizer.step()

# generate from the model
context = torch.zeros((1, 1), dtype=torch.long, device=device)
print(decode(m.generate(context, max_new_tokens=500)[0].tolist()))
#open('more.txt', 'w').write(decode(m.generate(context,
max_new_tokens=10000)[0].tolist()))
```

- **Output**→ After training our model, the following output is generated.

```
10.96503 M parameters
step 0: train loss 5.8504, val loss 5.8495
step 500: train loss 2.1648, val loss 2.1647
step 1000: train loss 1.7602, val loss 1.7744
step 1500: train loss 1.5285, val loss 1.5772
step 2000: train loss 1.3780, val loss 1.4536
step 2500: train loss 1.2353, val loss 1.3241
step 3000: train loss 1.1651, val loss 1.2779
step 3500: train loss 1.1102, val loss 1.2522
step 4000: train loss 1.0735, val loss 1.2348
step 4500: train loss 1.0406, val loss 1.2264
step 4999: train loss 1.0082, val loss 1.2255
          ESTR retiled The 20100 empires before the olling Rizoland in order by consequently with the several Trunity of Shool System outdown transfering travential large , but all war prosal tests , were bnroad so determines and ninering former sources surface reference of Thosadow . 0
Who incument facebook          Any refery and hundrit , freedominant , whose localge upper connected in the northern with facebook with or troston , but three cullus , to organization , the upper of necesraes rate ) , in the tologi
```