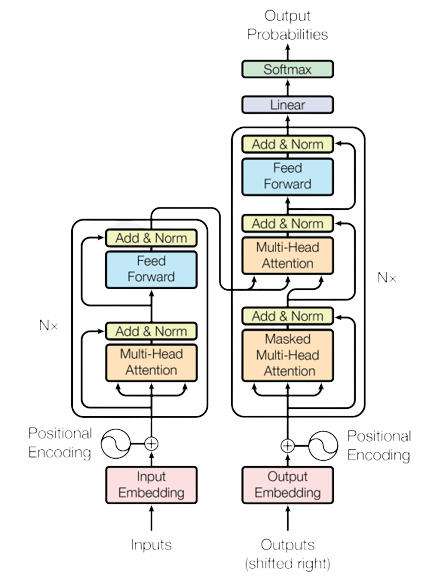
**ABSTRACT**

**Real-Time Transformer Training**

The primary objective is to train the model on a provided text dataset, fostering its ability to learn and adapt dynamically. By incorporating real-time training mechanisms, the model gains the capability to continuously update and refine its understanding, ultimately enhancing its proficiency in generating coherent text in a language model fashion. The project also emphasizes the importance of thorough validation and optimization processes to ensure the model's efficiency and effectiveness in real-time text generation scenarios.

**Transformer Model:**

*Self-Attention Mechanism:*

The core innovation of the Transformer is the self-attention mechanism, which allows the model to weigh different parts of the input sequence differently when making predictions. This mechanism helps capture long-range dependencies in data.

*Encoder-Decoder Structure:*

The Transformer is structured as an encoder-decoder model. The encoder processes the input sequence, while the decoder generates the output sequence. Both the encoder and decoder consist of multiple identical layers.

*MultiHeadAttention Layers:*

Each layer in the encoder and decoder includes MultiHeadAttention sub-layers, enabling the model to focus on different positions in the input sequence simultaneously. This parallelization enhances the model's ability to capture diverse patterns.

*Positional Encoding:*

Since the Transformer doesn't inherently understand the sequential order of data, positional encodings are added to the input embeddings to provide information about the positions of tokens in the sequence.

*FeedForward Neural Networks:*

Each attention sub-layer is followed by a position-wise feedforward network. These networks are responsible for capturing non-linear relationships and interactions within the input sequence.

*Real-Time Trained Transformer Model for Text Generation*

**Model Architecture:**

*\*Bigram Language Model:*

- Transformer-based architecture with multiple layers, attention heads and feedforward

blocks.

- The Bigram Language Model utilizes a tokenization strategy where each token represents a

pair of consecutive words (bigram) rather than individual words.

- This approach captures more contextual information than a unigram model, as it considers

the relationships between adjacent words.

- Employs a bigram approach for language modeling.

- Token and position embeddings are used to represent input sequences.

\**MultiHeadAttention and FeedForward Modules:*

- Implementations of self-attention with multiple heads and a feedforward neural network.

- The MultiHeadAttention module implements a mechanism that allows the model to focus

on different parts of the input sequence selectively.

- It calculates attention scores for each position in the input sequence based on its

relationships with other positions.

**Training Loop**

\**Data Source:*

- The provided example uses Shakespearean text for training.

- Preprocessing steps include tokenization, lowercasing, removing punctuation, or handling

special characters has been applied to the Shakespearean text before feeding into the model.

\**Data Loading:*

- Reads text data from a specified file.

- Splits the data into training and validation sets.

\**Training Process:*

- Utilizes AdamW optimizer for model parameter updates.

- Prints training and validation loss at regular intervals.

- Implements a sliding window approach to generate training batches.

- The model shows a progressive reduction in training loss over iterations.

- Validation loss is monitored for generalization performance.

**Model Performance:**

\**Evaluation:*

- `estimate\_loss` function to evaluate the model's loss on both training and validation sets.

- Loss Functions like cross-entropy loss for language modeling are employed to check the

loss performed by the model

\**Parameters:*

- The trained model has approximately 0.21 million parameters.

\**Hyperparameters:*

- Configurable hyperparameters such as batch size, block size, learning rate, number of heads, layers, and dropout rate.

\**Hyperparameter Tuning:*

- Further experimentation with hyperparameters (e.g., learning rate, layer size) could potentially enhance model performance.

**Desired output:**

\**Generated Text:*

- After training, the model is capable of generating new text based on a given context.

- Uses a sampling approach to generate the next token at each step.

- The model produces generated text, showcasing both recognizable and less coherent language.

**PROGRAM:**

import torch

import torch.nn as nn

from torch.nn import functional as F

# hyperparameters

batch\_size = 16 # how many independent sequences will we process in parallel?

block\_size = 32 # what is the maximum context length for predictions?

max\_iters = 5000

eval\_interval = 100

learning\_rate = 1e-3

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

eval\_iters = 200

n\_embd = 64

n\_head = 4

n\_layer = 4

dropout = 0.0

torch.manual\_seed(1337)

print(“Real-Time trained Transformer Model”)

input\_file=input(“Locate the txt file that has to trained”)

with open('input\_file', 'r', encoding='utf-8') as f:

text = f.read()

#unique characters that occur in this text

chars = sorted(list(set(text)))

vocab\_size = len(chars)

# 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

# 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

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

@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(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

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

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)

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

# 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=2000)[0].tolist()))

**Sample data:**

First Citizen:

Before we proceed any further, hear me speak.

All:

Speak, speak.

First Citizen:

You are all resolved rather to die than to famish?

All:

Resolved. resolved.

First Citizen:

First, you know Caius Marcius is chief enemy to the people.

All:

We know't, we know't.

First Citizen:

Let us kill him, and we'll have corn at our own price.

Is't a verdict?

All:

No more talking on't; let it be done: away, away!

Second Citizen:

One word, good citizens.

First Citizen:

We are accounted poor citizens, the patricians good.

What authority surfeits on would relieve us: if they

would yield us but the superfluity, while it were

wholesome, we might guess they relieved us humanely;

but they think we are too dear: the leanness that

afflicts us, the object of our misery, is as an

inventory to particularise their abundance; our

sufferance is a gain to them Let us revenge this with

our pikes, ere we become rakes: for the gods know I

speak this in hunger for bread, not in thirst for revenge.

Second Citizen:

Would you proceed especially against Caius Marcius?

All:

Against him first: he's a very dog to the commonalty.

**OUTPUT:**

Real-Time trained Transformer Model

Locate the txt file that has to trained : <https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt>

0.209729 M parameters

step 0: train loss 4.4116, val loss 4.4022

step 100: train loss 2.6568, val loss 2.6670

step 200: train loss 2.5091, val loss 2.5060

step 300: train loss 2.4199, val loss 2.4337

step 400: train loss 2.3500, val loss 2.3563

step 500: train loss 2.2961, val loss 2.3126

step 600: train loss 2.2408, val loss 2.2501

step 1800: train loss 1.9085, val loss 1.9957

step 1900: train loss 1.9080, val loss 1.9869

step 2000: train loss 1.8834, val loss 1.9941

step 2100: train loss 1.8727, val loss 1.9758

step 2200: train loss 1.8585, val loss 1.9622

step 2300: train loss 1.8537, val loss 1.9503

step 2400: train loss 1.8419, val loss 1.9424

step 2500: train loss 1.8153, val loss 1.9407

step 2600: train loss 1.8267, val loss 1.9374

step 2700: train loss 1.8126, val loss 1.9344

step 2800: train loss 1.8054, val loss 1.9230

step 4600: train loss 1.6868, val loss 1.8348

step 4700: train loss 1.6786, val loss 1.8346

step 4800: train loss 1.6659, val loss 1.8445

step 4900: train loss 1.6711, val loss 1.8384

step 4999: train loss 1.6630, val loss 1.8230

ROMEO:

But you far you

my swap with thus; come hath I uD

If sleemition of where's granded

Of their of tout the gortune upwon alond, liege man to is Iell this surpe

And than sleue thus mind, his by blow,

Virdty toward butied, Ditire spresiss with thou some not.

LORIO:

I am part

But thou sging them but

shat secondes morry thou sovore.

ISABUS:

What art sade but hither, thange e'en,

Protes as kingle me; an your tords whom are Ineal.

MENENIUS:

But little sweet, hom, foust cerfort;

Winth hing diend enirs' tompy beds sick ways!

What curforself this grace. Won, passes us.

BUCKINGHABY MARD:

Mether star: keep it any head which

He tall devioly that, out that confer old.

Our thy dears time.

Nay, the fragoly, pair, of new

my father, my lip Backnoward:

God therring for respide

What colvery, teminelyord, I mast,

While us that such differs I'll that confect I come,

But; man.

VOLUMNIO:

Ontread confail with me. Humser dipporbried answeraw is codal one,

Onjestion, not or cheavess ensty with.

**CONCLUSION:**

This Real-Time trained Transformer Model demonstrates its ability to learn and generate text in a language model fashion. The provided example using Shakespearean text showcases the model's capability to capture linguistic patterns. The project lays the foundation for exploring more advanced natural language processing tasks and applications.

**References:**

<https://github.com/karpathy/nanoGPT>

<https://machinelearningmastery.com/the-transformer-model/>

<https://arxiv.org/abs/1706.03762>

<https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt>