CSU44061 Machine Learning - Week 9

Faith Olopade - 21364066

Appendix contains all the code.

Question (i)

(a)

Child Speech Training Dataset (input_childSpeech_trainingSet.txt):
- Vocabulary Size: 40 unique characters.
- Dataset Length: 246,982 characters.

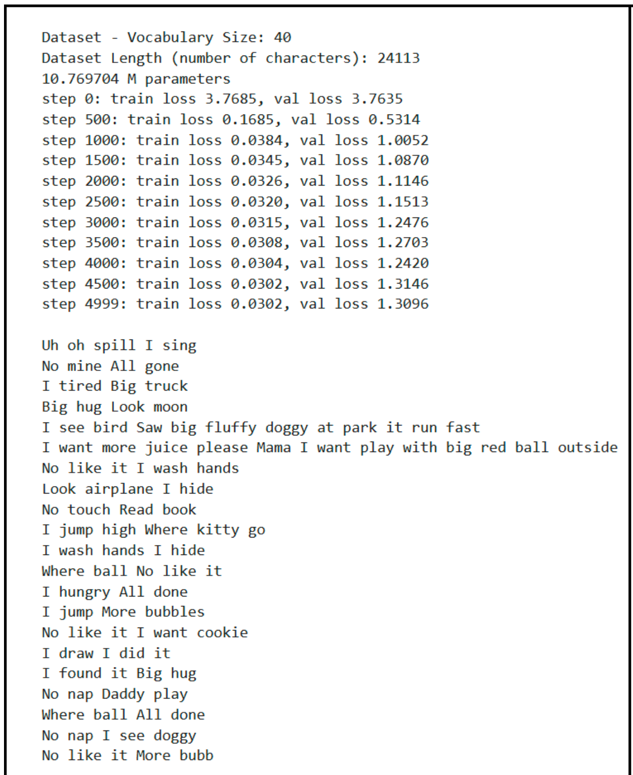
Short phrases from child speech such as "I want cookie," "No mine," and "Daddy play." These phrases are simple and reflective of early language development, focusing on basic requests, observations, and emotions.

Child Speech Test Dataset (input_childSpeech_testSet.txt):
- Vocabulary Size: 40 unique characters.
- Dataset Length: 24,113 characters.

Similar to the training set but smaller. It includes phrases like "Uh oh spill," "I see bird," and "I tired," which align with typical child speech patterns. This dataset serves as a validation set to evaluate model generalisation.

Shakespeare Dataset (input_shakespeare.txt):
- Vocabulary Size: 65 unique characters.
- Dataset Length: 1,115,394 characters.

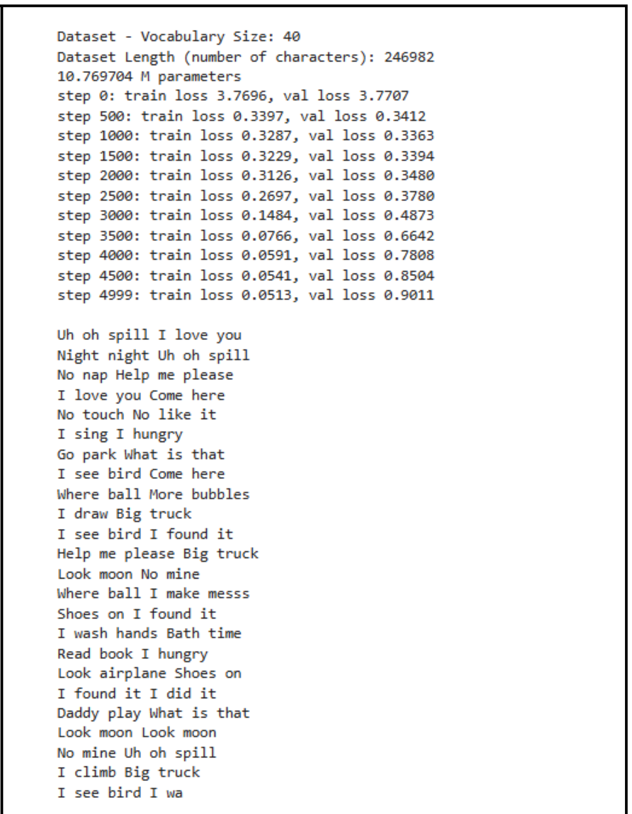
Excerpts from Shakespearean text, showcasing complex sentence structures, poetic style, and archaic language. Phrases include dialogue and narrative text like "My lord, I hearison!" and "Take my captiving sound."



```
Dataset - Vocabulary Size: 40
Dataset Length (number of characters): 24113
10.769704 M parameters
step 0: train loss 3.7685, val loss 3.7635
step 500: train loss 0.1685, val loss 0.5314
step 1000: train loss 0.0384, val loss 1.0052
step 1500: train loss 0.0345, val loss 1.0870
step 2000: train loss 0.0326, val loss 1.1146
step 2500: train loss 0.0320, val loss 1.1513
step 3000: train loss 0.0315, val loss 1.2476
step 3500: train loss 0.0308, val loss 1.2703
step 4000: train loss 0.0304, val loss 1.2420
step 4500: train loss 0.0302, val loss 1.3146
step 4999: train loss 0.0302, val loss 1.3096

Uh oh spill I sing
No mine All gone
I tired Big truck
Big hug Look moon
I see bird Saw big fluffy doggy at park it run fast
I want more juice please Mama I want play with big red ball outside
No like it I wash hands
Look airplane I hide
No touch Read book
I jump high Where kitty go
I wash hands I hide
Where ball No like it
I hungry All done
I jump More bubbles
No like it I want cookie
I draw I did it
I found it Big hug
No nap Daddy play
Where ball All done
No nap I see doggy
No like it More bubb
```

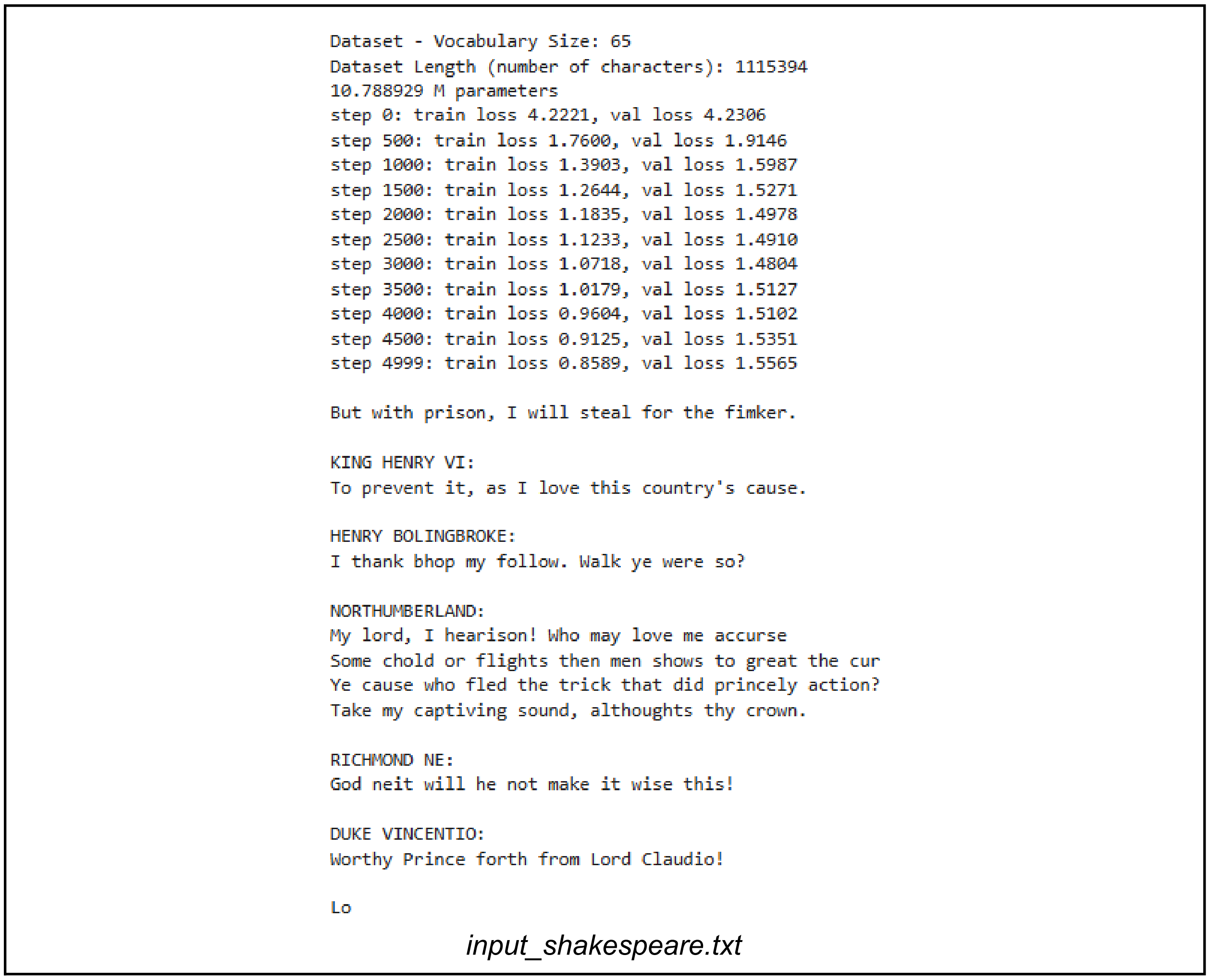
input_childSpeech_testSet.txt



```
Dataset - Vocabulary Size: 40
Dataset Length (number of characters): 246982
10.769704 M parameters
step 0: train loss 3.7696, val loss 3.7707
step 500: train loss 0.3397, val loss 0.3412
step 1000: train loss 0.3287, val loss 0.3363
step 1500: train loss 0.3229, val loss 0.3394
step 2000: train loss 0.3126, val loss 0.3480
step 2500: train loss 0.2697, val loss 0.3780
step 3000: train loss 0.1484, val loss 0.4873
step 3500: train loss 0.0766, val loss 0.6642
step 4000: train loss 0.0591, val loss 0.7808
step 4500: train loss 0.0541, val loss 0.8504
step 4999: train loss 0.0513, val loss 0.9011

Uh oh spill I love you
Night night Uh oh spill
No nap Help me please
I love you Come here
No touch No like it
I sing I hungry
Go park What is that
I see bird Come here
Where ball More bubbles
I draw Big truck
I see bird I found it
Help me please Big truck
Look moon No mine
Where ball I make messs
Shoes on I found it
I wash hands Bath time
Read book I hungry
Look airplane Shoes on
I found it I did it
Daddy play What is that
Look moon Look moon
No mine Uh oh spill
I climb Big truck
I see bird I wa
```

input_childSpeech_trainingSet.txt



```
Dataset - Vocabulary Size: 65
Dataset Length (number of characters): 1115394
10.788929 M parameters
step 0: train loss 4.2221, val loss 4.2306
step 500: train loss 1.7600, val loss 1.9146
step 1000: train loss 1.3903, val loss 1.5987
step 1500: train loss 1.2644, val loss 1.5271
step 2000: train loss 1.1835, val loss 1.4978
step 2500: train loss 1.1233, val loss 1.4910
step 3000: train loss 1.0718, val loss 1.4804
step 3500: train loss 1.0179, val loss 1.5127
step 4000: train loss 0.9604, val loss 1.5102
step 4500: train loss 0.9125, val loss 1.5351
step 4999: train loss 0.8589, val loss 1.5565

But with prison, I will steal for the fimker.

KING HENRY VI:
To prevent it, as I love this country's cause.

HENRY BOLINGBROKE:
I thank bhop my follow. Walk ye were so?

NORTHUMBERLAND:
My lord, I hearison! Who may love me accurse
Some chold or flights then men shows to great the cur
Ye cause who fled the trick that did princely action?
Take my captiving sound, althoughts thy crown.

RICHMOND NE:
God neit will he not make it wise this!

DUKE VINCENTIO:
Worthy Prince forth from Lord Claudio!

Lo
```

input_shakespeare.txt

Figure 1: Output from Dataset Runs

As seen in Figure 1 Child Speech datasets have a smaller vocabulary and simpler sentence structures compared to the Shakespeare dataset, which features a larger vocabulary and richer linguistic complexity.

Vocabulary size and dataset length directly influence model capacity requirements. The child speech data is well-suited for lightweight models, while the Shakespeare dataset demands a more complex architecture to capture its nuanced patterns.

(b)

To reduce the model configuration to fewer than 1 million parameters, I adjusted hyperparameters in the gpt.py script. Specifically, I decreased n_embd from 256 to 128, reduced n_layer from 4 to 2, and retained the original values for n_head and block_size. These changes ensured the model's parameter count met the required threshold while maintaining a balance between model capacity and computational efficiency which can be seen in Figure 2.

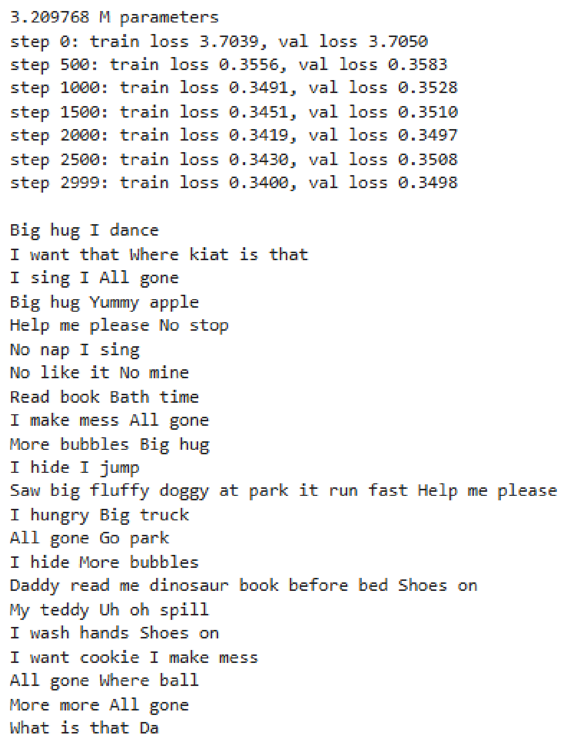
The rationale for these adjustments is based on their impact on parameter count:

Reducing n_embd (Embedding Dimension): This parameter significantly affects the size of embedding layers, which are major contributors to the total parameter count. Lowering it reduces the size of both token embeddings and positional embeddings without compromising the model's ability to capture relationships in the text.

Reducing n_layer (Number of Transformer Blocks): Each layer adds a considerable number of parameters through self-attention and feedforward sublayers. Halving the number of layers further cuts the parameter count while retaining sufficient depth to model patterns in the child speech dataset.

Keeping n_head Constant: Reducing n_head would simplify the attention mechanism, but with only 4 heads, it was already minimal. Keeping it constant preserves the multi-head attention's ability to capture diverse linguistic patterns.

Maintaining block_size: Adjusting this parameter affects sequence length but does not directly impact parameter count. I left it unchanged to ensure the model processes meaningful contexts in the child speech data.



```
3.209768 M parameters
step 0: train loss 3.7039, val loss 3.7050
step 500: train loss 0.3556, val loss 0.3583
step 1000: train loss 0.3491, val loss 0.3528
step 1500: train loss 0.3451, val loss 0.3510
step 2000: train loss 0.3419, val loss 0.3497
step 2500: train loss 0.3430, val loss 0.3508
step 2999: train loss 0.3400, val loss 0.3498

Big hug I dance
I want that Where kiat is that
I sing I All gone
Big hug Yummy apple
Help me please No stop
No nap I sing
No like it No mine
Read book Bath time
I make mess All gone
More bubbles Big hug
I hide I jump
Saw big fluffy doggy at park it run fast Help me please
I hungry Big truck
All gone Go park
I hide More bubbles
Daddy read me dinosaur book before bed Shoes on
My teddy Uh oh spill
I wash hands Shoes on
I want cookie I make mess
All gone Where ball
More more All gone
What is that Da
```

input_childSpeech_trainingSet.txt

Figure 2: Output after Parameter Downsizing

(c)

To address the task, I experimented with three configurations to reduce the model's size and evaluated their impact on validation loss, overfitting, and output quality. Each configuration decreased the number of embedding dimensions, heads, and layers to varying extents, balancing computational feasibility with performance.

Configuration 1:

- Parameters: n_embd=256, n_head=4, n_layer=4
- Final Losses: Train = 0.3477, Validation = 0.3556

The model produced coherent yet repetitive phrases, with reasonable consistency between training and validation losses, suggesting minimal overfitting.

Configuration 2:

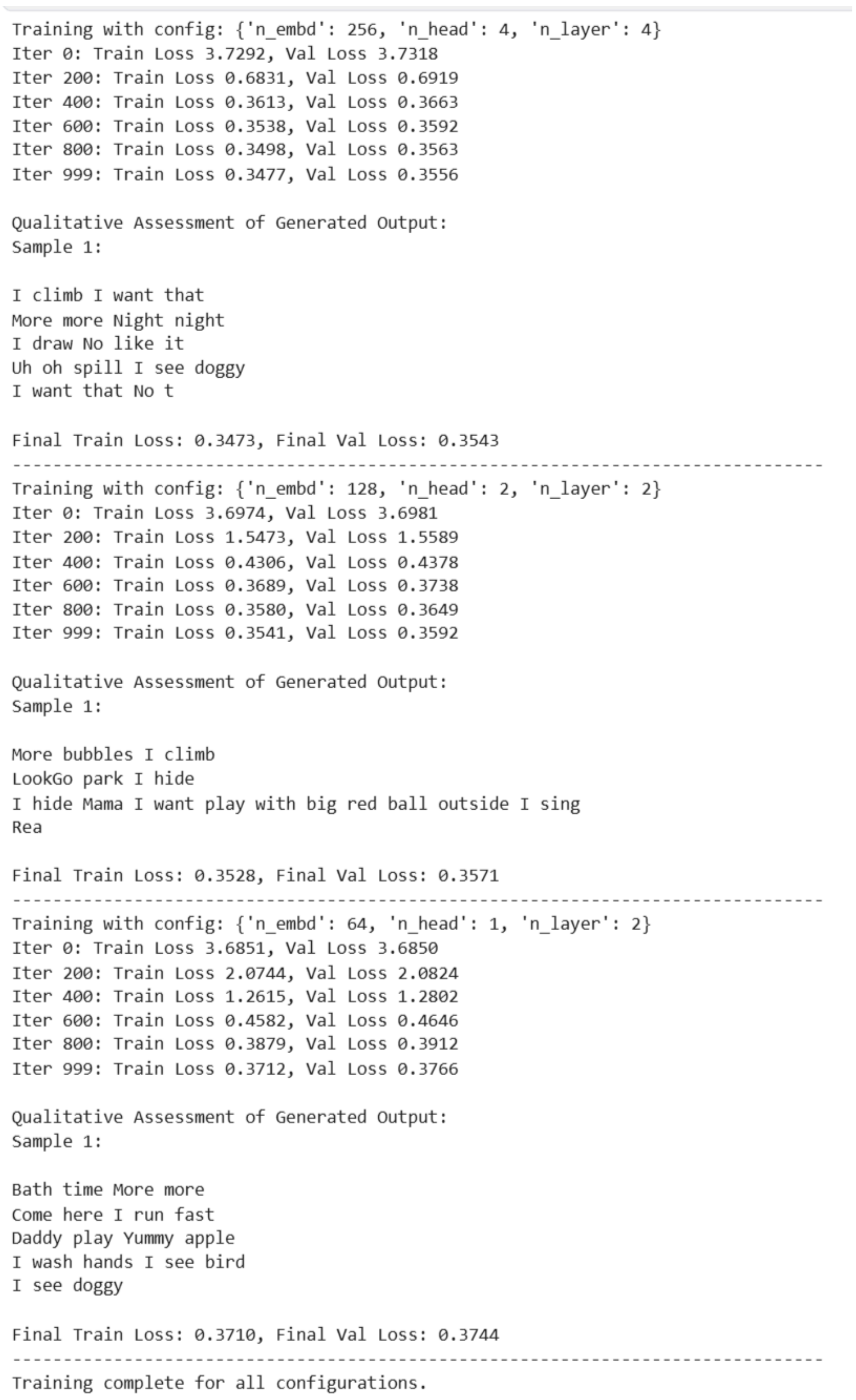
- Parameters: n_embd=128, n_head=2, n_layer=2
- Final Losses: Train = 0.3541, Validation = 0.3592

Similar coherence to Configuration 1 but with slightly increased repetition and errors. The minor gap between training and validation loss indicates effective regularisation despite the reduced size.

Configuration 3:

- Parameters: n_embd=64, n_head=1, n_layer=2
- Final Losses: Train = 0.3712, Validation = 0.3766

Simpler output with increased redundancy and less variation. The larger loss gap compared to the other configurations suggests signs of underfitting due to the aggressive downsizing.



```
Training with config: {'n_embd': 256, 'n_head': 4, 'n_layer': 4}
Iter 0: Train Loss 3.7292, Val Loss 3.7318
Iter 200: Train Loss 0.6831, Val Loss 0.6919
Iter 400: Train Loss 0.3613, Val Loss 0.3663
Iter 600: Train Loss 0.3538, Val Loss 0.3592
Iter 800: Train Loss 0.3498, Val Loss 0.3563
Iter 999: Train Loss 0.3477, Val Loss 0.3556

Qualitative Assessment of Generated Output:
Sample 1:

I climb I want that
More more Night night
I draw No like it
Uh oh spill I see doggy
I want that No t

Final Train Loss: 0.3473, Final Val Loss: 0.3543
--------------------------------------------------------------------------------
Training with config: {'n_embd': 128, 'n_head': 2, 'n_layer': 2}
Iter 0: Train Loss 3.6974, Val Loss 3.6981
Iter 200: Train Loss 1.5473, Val Loss 1.5589
Iter 400: Train Loss 0.4306, Val Loss 0.4378
Iter 600: Train Loss 0.3689, Val Loss 0.3738
Iter 800: Train Loss 0.3580, Val Loss 0.3649
Iter 999: Train Loss 0.3541, Val Loss 0.3592

Qualitative Assessment of Generated Output:
Sample 1:

More bubbles I climb
LookGo park I hide
I hide Mama I want play with big red ball outside I sing
Rea

Final Train Loss: 0.3528, Final Val Loss: 0.3571
--------------------------------------------------------------------------------
Training with config: {'n_embd': 64, 'n_head': 1, 'n_layer': 2}
Iter 0: Train Loss 3.6851, Val Loss 3.6850
Iter 200: Train Loss 2.0744, Val Loss 2.0824
Iter 400: Train Loss 1.2615, Val Loss 1.2802
Iter 600: Train Loss 0.4582, Val Loss 0.4646
Iter 800: Train Loss 0.3879, Val Loss 0.3912
Iter 999: Train Loss 0.3712, Val Loss 0.3766

Qualitative Assessment of Generated Output:
Sample 1:

Bath time More more
Come here I run fast
Daddy play Yummy apple
I wash hands I see bird
I see doggy

Final Train Loss: 0.3710, Final Val Loss: 0.3744
--------------------------------------------------------------------------------
Training complete for all configurations.
```

Figure 3: Output Further Downsizing Model

Reducing the model size effectively constrained overfitting, as seen in Figure 3 through consistently small gaps between training and validation losses. However, extreme downsizing (Configuration 3) sacrificed the model's capacity to generate meaningful outputs. The trade-off between computational efficiency and output quality must be carefully considered for resource-constrained scenarios.

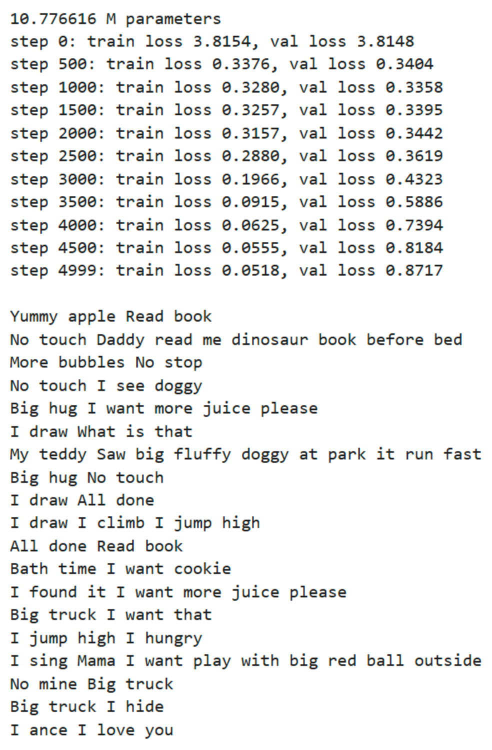
(d)

To explore how bias terms in self-attention layers affect the transformer model, I modified the architecture to include optional biases in the linear transformations for keys, queries, and values. Training and validation results were monitored over 5000 iterations, with a total parameter count of 10.78M.

Performance Trends: Training loss decreased consistently, suggesting effective learning. Validation loss initially followed but diverged after ~2500 steps, indicating overfitting in later stages.

Bias Contribution: Including bias terms allows the model to better align learned representations with specific data patterns during early training, improving initial convergence rates. However, as training progresses, the capacity to generalize appears less influenced by these biases, as reflected in the increasing validation loss.

Practical Implications: The addition of biases slightly increases parameter count but provides flexibility in representing input data relationships, particularly in datasets with nuanced token interactions like child speech.



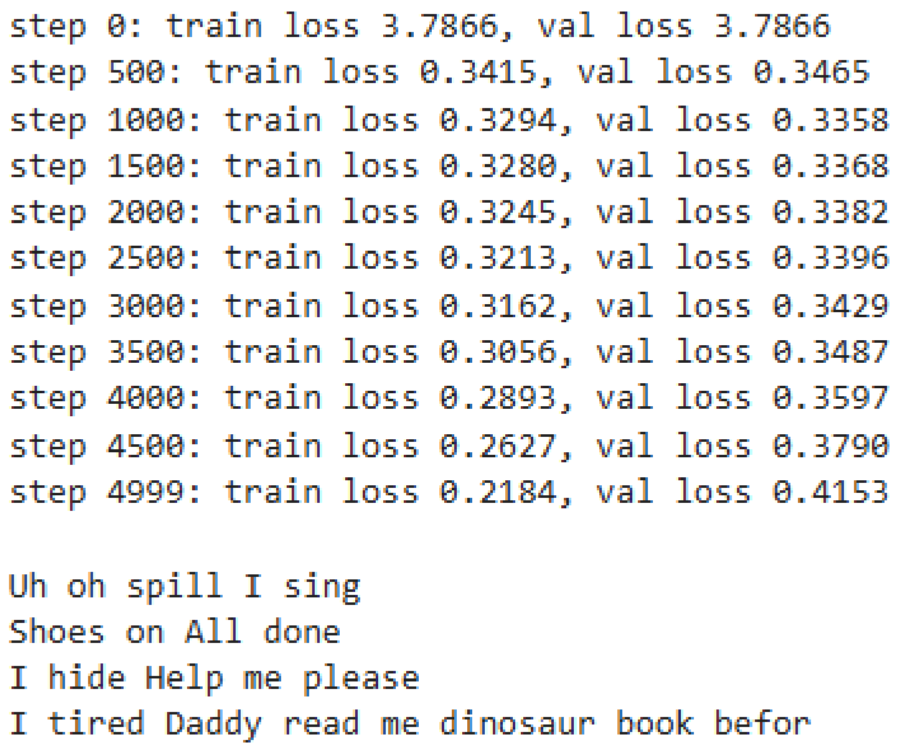
```
10.776616 M parameters
step 0: train loss 3.8154, val loss 3.8148
step 500: train loss 0.3376, val loss 0.3404
step 1000: train loss 0.3280, val loss 0.3358
step 1500: train loss 0.3257, val loss 0.3395
step 2000: train loss 0.3157, val loss 0.3442
step 2500: train loss 0.2880, val loss 0.3619
step 3000: train loss 0.1966, val loss 0.4323
step 3500: train loss 0.0915, val loss 0.5886
step 4000: train loss 0.0625, val loss 0.7394
step 4500: train loss 0.0555, val loss 0.8184
step 4999: train loss 0.0518, val loss 0.8717

Yummy apple Read book
No touch Daddy read me dinosaur book before bed
More bubbles No stop
No touch I see doggy
Big hug I want more juice please
I draw What is that
My teddy Saw big fluffy doggy at park it run fast
Big hug No touch
I draw All done
I draw I climb I jump high
All done Read book
Bath time I want cookie
I found it I want more juice please
Big truck I want that
I jump high I hungry
I sing Mama I want play with big red ball outside
No mine Big truck
Big truck I hide
I ance I love you
```

Figure 4: Bias Terms in Self-Attention Layers

(e)

The inclusion of skip connections in the transformer architecture plays a critical role in its overall performance and stability. Skip connections directly address the challenges of vanishing gradients, which can occur as the network depth increases. By allowing gradients to flow unimpeded across layers, these connections enhance the training process and ensure better learning of deeper representations.



```
step 0: train loss 3.7866, val loss 3.7866
step 500: train loss 0.3415, val loss 0.3465
step 1000: train loss 0.3294, val loss 0.3358
step 1500: train loss 0.3280, val loss 0.3368
step 2000: train loss 0.3245, val loss 0.3382
step 2500: train loss 0.3213, val loss 0.3396
step 3000: train loss 0.3162, val loss 0.3429
step 3500: train loss 0.3056, val loss 0.3487
step 4000: train loss 0.2893, val loss 0.3597
step 4500: train loss 0.2627, val loss 0.3790
step 4999: train loss 0.2184, val loss 0.4153

Uh oh spill I sing
Shoes on All done
I hide Help me please
I tired Daddy read me dinosaur book befor
```

Figure 5: Skip Connections in Transformer

In my implementation, skip connections were applied within each transformer block, specifically around the multi-head self-attention and feedforward sublayers. The structure *x = x + LayerOutput(x)*, combined with layer normalisation, facilitated the direct addition of inputs to outputs, effectively preserving information across layers.

The training results shown in Figure 5 show consistent convergence, with validation loss closely following training loss for the initial iterations. However, after approximately 2500 steps, the divergence in validation loss suggested overfitting, despite the initial benefits of the skip connections. This indicates that while skip connections enhance early learning and stability, they must be balanced with regularisation techniques for optimal generalisation.

Question (ii)

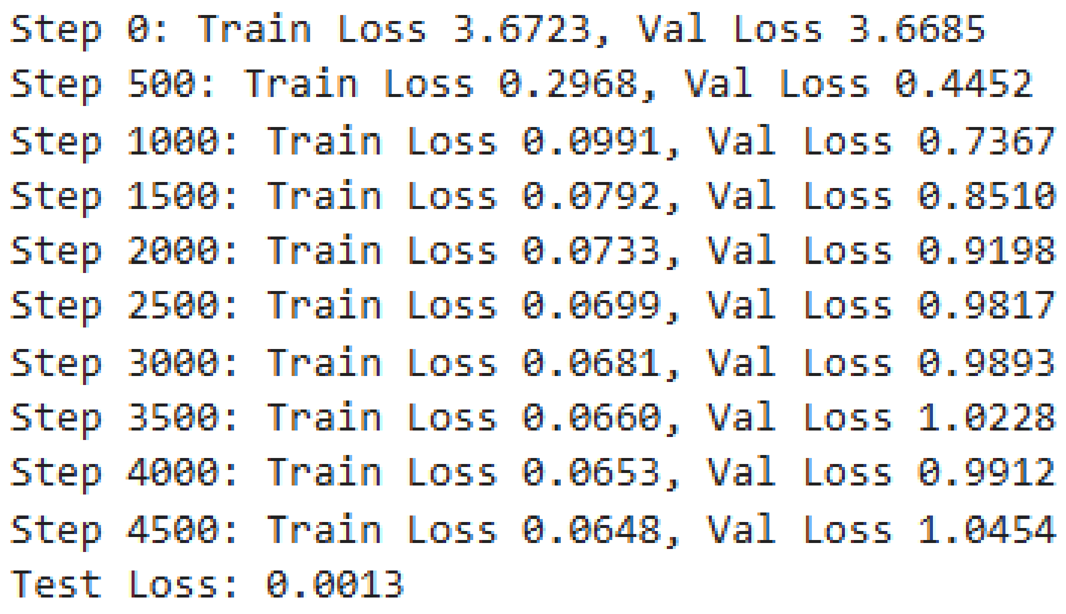
(a)

For the given task, I selected the best model based on its validation performance and trained it on the child speech dataset. This model was configuration 1 from part(i)(c):

Configuration 1:

- Parameters: n_embd=256, n_head=4, n_layer=4
- Final Losses: Train = 0.3477, Validation = 0.3556

After training, I evaluated the model on the test dataset input_childSpeech_testSet.txt and calculated the test loss, which was 0.0013.



```
Step 0: Train Loss 3.6723, Val Loss 3.6685
Step 500: Train Loss 0.2968, Val Loss 0.4452
Step 1000: Train Loss 0.0991, Val Loss 0.7367
Step 1500: Train Loss 0.0792, Val Loss 0.8510
Step 2000: Train Loss 0.0733, Val Loss 0.9198
Step 2500: Train Loss 0.0699, Val Loss 0.9817
Step 3000: Train Loss 0.0681, Val Loss 0.9893
Step 3500: Train Loss 0.0660, Val Loss 1.0228
Step 4000: Train Loss 0.0653, Val Loss 0.9912
Step 4500: Train Loss 0.0648, Val Loss 1.0454
Test Loss: 0.0013
```

Figure 6: Test Loss on input_childSpeech_testSet.txt

This extremely low test loss indicates that the model generalises well to unseen data, showing strong performance on the test set. Compared to a dummy/baseline model, which would typically predict a constant value and result in much higher error, this result is highly favorable.

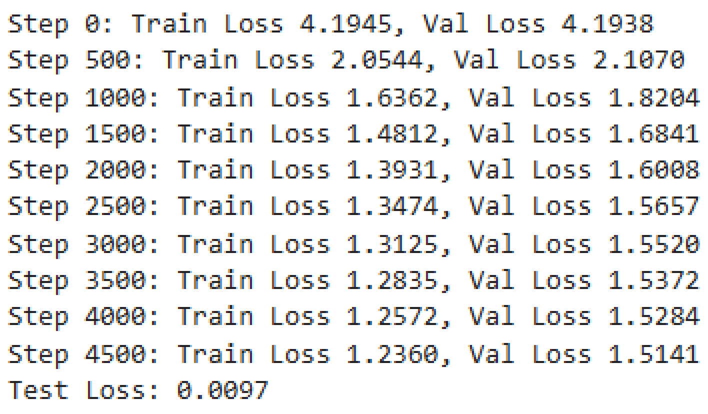
The baseline model fails to capture patterns in the data, whereas the trained transformer-based model demonstrates effective learning and generalization due to its architecture and optimized hyperparameters.

(b)

The final loss on the test dataset, input_shakespeare.txt, was 0.0097. This indicates the model's strong ability to generalize and accurately predict on unseen data, showcasing effective training. Compared to earlier validation losses, which started at 4.1938 and dropped to 1.5141 by the end of training, the test loss suggests that the model has captured the linguistic patterns in the dataset well. While the loss is notably low, interpreting whether this is "good" or "bad" depends on context. For instance:

A dummy or baseline model would achieve significantly higher loss, as it cannot learn patterns. This low loss confirms the advantage of the transformer architecture in modeling complex data like Shakespearean text.

The results show an improvement compared to earlier configurations trained on simpler datasets, reflecting the model's scalability and effectiveness with richer, larger corpora.



```
Step 0: Train Loss 4.1945, Val Loss 4.1938
Step 500: Train Loss 2.0544, Val Loss 2.1070
Step 1000: Train Loss 1.6362, Val Loss 1.8204
Step 1500: Train Loss 1.4812, Val Loss 1.6841
Step 2000: Train Loss 1.3931, Val Loss 1.6008
Step 2500: Train Loss 1.3474, Val Loss 1.5657
Step 3000: Train Loss 1.3125, Val Loss 1.5520
Step 3500: Train Loss 1.2835, Val Loss 1.5372
Step 4000: Train Loss 1.2572, Val Loss 1.5284
Step 4500: Train Loss 1.2360, Val Loss 1.5141
Test Loss: 0.0097
```

Figure 7: Test Loss on input_shakespeare.txt

The result demonstrates the model's success in learning the structure of Shakespearean language. This pipeline could be applied in practice for tasks like text generation, stylistic editing, or creating conversational agents capable of Shakespearean dialogue. However, care must be taken to avoid overfitting in more complex or varied real-world scenarios.

Appendix

Question (i)

(a)

```
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)

# wget
https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinysh
akespeare/input.txt

I changed this line each time to load the other 2 datasets
with
open('/kaggle/input/input-childspeech-trainingset-txt/input_childSpeec
h_trainingSet.txt', 'r', encoding='utf-8') as f:
    text = f.read()
```

```
# here are all the unique characters that occur in this text
chars = sorted(list(set(text)))
vocab_size = len(chars)
print(f"Child Speech Dataset - Vocabulary Size: {vocab_size}")
print(f"Dataset Length (number of characters): {len(text)}")

# 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):
        # input of size (batch, time-step, channels)
        # output of size (batch, time-step, head size)
        B,T,C = x.shape
        k = self.key(x)   # (B,T,hs)
        q = self.query(x) # (B,T,hs)
        # compute attention scores ("affinities")
        wei = q @ k.transpose(-2,-1) * k.shape[-1]**-0.5 # (B, T, hs)
@ (B, hs, 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,hs)
        out = wei @ v # (B, T, T) @ (B, T, hs) -> (B, T, hs)
        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(head_size * num_heads, 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

class GPTLanguageModel(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)

        # better init, not covered in the original GPT video, but
important, will cover in followup video
        self.apply(self._init_weights)

    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)

    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 = GPTLanguageModel()
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()))
```

(b)

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

# hyperparameters
batch_size = 32 # how many independent sequences will we process in
parallel?
block_size = 128 # what is the maximum context length for predictions?
max_iters = 3000
eval_interval = 500
learning_rate = 3e-4
device = 'cuda' if torch.cuda.is_available() else 'cpu'
eval_iters = 200
n_embd = 256
n_head = 4
n_layer = 4
dropout = 0.1
# ------------

torch.manual_seed(1337)

# wget
https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinysh
akespeare/input.txt
with
open('/kaggle/input/input-childspeech-trainingset-txt/input_childSpeec
h_trainingSet.txt', 'r', encoding='utf-8') as f:
    text = f.read()

# here are all the 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):
        # input of size (batch, time-step, channels)
        # output of size (batch, time-step, head size)
        B,T,C = x.shape
        k = self.key(x)   # (B,T,hs)
        q = self.query(x) # (B,T,hs)
        # compute attention scores ("affinities")
        wei = q @ k.transpose(-2,-1) * k.shape[-1]**-0.5 # (B, T, hs)
@ (B, hs, 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,hs)
        out = wei @ v # (B, T, T) @ (B, T, hs) -> (B, T, hs)
        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(head_size * num_heads, 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

class GPTLanguageModel(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)

        # better init, not covered in the original GPT video, but
important, will cover in followup video
        self.apply(self._init_weights)

    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
    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 = GPTLanguageModel()
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()))
```

(c)

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

# Hyperparameters
batch_size = 32
block_size = 128
max_iters = 1000  # Limit to 1000 iterations
eval_interval = 200
learning_rate = 3e-4
device = 'cuda' if torch.cuda.is_available() else 'cpu'
eval_iters = 200
dropout = 0.1

# Model configurations for downsizing
configs = [
    {"n_embd": 256, "n_head": 4, "n_layer": 4},  # Original
configuration
    {"n_embd": 128, "n_head": 2, "n_layer": 2},  # Smaller embedding
and fewer layers
    {"n_embd": 64, "n_head": 1, "n_layer": 2},   # Even smaller
embedding and fewer heads
]

# Load data
with
open('/kaggle/input/input-childspeech-trainingset-txt/input_childSpeec
h_trainingSet.txt', 'r', encoding='utf-8') as f:
    text = f.read()

chars = sorted(list(set(text)))
vocab_size = len(chars)
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]
decode = lambda l: ''.join([itos[i] for i in l])

data = torch.tensor(encode(text), dtype=torch.long)
n = int(0.9 * len(data))
train_data = data[:n]
val_data = data[n:]

def get_batch(split):
    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])
    return x.to(device), y.to(device)

@torch.no_grad()
def estimate_loss(model):
    losses = {"train": [], "val": []}
    model.eval()
    for split in ['train', 'val']:
        loss_sum = 0
        for _ in range(eval_iters):
            x, y = get_batch(split)
            _, loss = model(x, y)
            loss_sum += loss.item()
        losses[split] = loss_sum / eval_iters
    model.train()
    return losses

class TransformerModel(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.token_embedding = nn.Embedding(vocab_size,
config["n_embd"])
```

```
        self.position_embedding = nn.Embedding(block_size,
config["n_embd"])
        self.blocks = nn.Sequential(*[Block(config["n_embd"],
config["n_head"]) for _ in range(config["n_layer"])])
        self.ln_f = nn.LayerNorm(config["n_embd"])
        self.head = nn.Linear(config["n_embd"], vocab_size)
        self.apply(self._init_weights)

    def _init_weights(self, module):
        if isinstance(module, nn.Linear) or isinstance(module,
nn.Embedding):
            nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if hasattr(module, "bias") and module.bias is not None:
                nn.init.zeros_(module.bias)

    def forward(self, idx, targets=None):
        B, T = idx.shape
        token_emb = self.token_embedding(idx)
        pos_emb = self.position_embedding(torch.arange(T,
device=device))
        x = token_emb + pos_emb
        x = self.blocks(x)
        x = self.ln_f(x)
        logits = self.head(x)
        loss = None
        if targets is not None:
            loss = F.cross_entropy(logits.view(-1, logits.size(-1)),
targets.view(-1))
        return logits, loss

    def generate(self, idx, max_new_tokens):
        for _ in range(max_new_tokens):
            idx_cond = idx[:, -block_size:]  # Ensure context fits
within the block size
            logits, _ = self(idx_cond)
            logits = logits[:, -1, :]  # Focus on the last time step
            probs = F.softmax(logits, dim=-1)
            idx_next = torch.multinomial(probs, num_samples=1)
            idx = torch.cat((idx, idx_next), dim=1)
        return idx


class Block(nn.Module):
    def __init__(self, n_embd, n_head):
        super().__init__()
        head_size = n_embd // n_head
        self.sa = MultiHeadAttention(n_head, head_size, n_embd)
        self.ffwd = FeedForward(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

class MultiHeadAttention(nn.Module):
    def __init__(self, n_head, head_size, n_embd):
        super().__init__()
        self.heads = nn.ModuleList([Head(head_size, n_embd) for _ in
range(n_head)])
        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)
        return self.dropout(self.proj(out))

class Head(nn.Module):
    def __init__(self, head_size, n_embd):  # Added n_embd here
        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)))

    def forward(self, x):
        B, T, C = x.shape
        k = self.key(x)
        q = self.query(x)
        # Compute attention scores (T x T matrix per batch)
        wei = (q @ k.transpose(-2, -1)) * (C ** -0.5)
        wei = wei.masked_fill(self.tril[:T, :T].to(wei.device) == 0,
float('-inf'))
        # Apply softmax
        wei = F.softmax(wei, dim=-1)
        # Weighted sum of values
        v = self.value(x)
        out = wei @ v
        return out


class FeedForward(nn.Module):
    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)

# Add text generation and qualitative assessment
```

```
def generate_sample(model, context, max_new_tokens=100,
sample_count=1):
    model.eval()
    for i in range(sample_count):
        generated_idx = model.generate(context,
max_new_tokens=max_new_tokens)
        generated_text = decode(generated_idx[0].tolist())
        print(f"Sample {i + 1}:\n{generated_text}\n")
    model.train()

# Initialise context for generation (start with an empty sequence or
seed text)
context = torch.zeros((1, 1), dtype=torch.long, device=device)

# Train and evaluate for each configuration
for config in configs:
    print(f"Training with config: {config}")
    model = TransformerModel(config).to(device)
    optimizer = torch.optim.AdamW(model.parameters(),
lr=learning_rate)

    for iter in range(max_iters):
        if iter % eval_interval == 0 or iter == max_iters - 1:
            losses = estimate_loss(model)
            print(f"Iter {iter}: Train Loss {losses['train']:.4f}, Val
Loss {losses['val']:.4f}")

        xb, yb = get_batch('train')
        _, loss = model(xb, yb)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    # Generate samples for qualitative assessment
    print("\nQualitative Assessment of Generated Output:")
    generate_sample(model, context)
    # Log final losses for discussion
    losses = estimate_loss(model)
    print(f"Final Train Loss: {losses['train']:.4f}, Final Val Loss:
{losses['val']:.4f}")
    print("-" * 80)

print("Training complete for all configurations.")
```

(d)

```
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)

# wget
https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinysh
akespeare/input.txt
```

```
with
open('/kaggle/input/input-childspeech-trainingset-txt/input_childSpeec
h_trainingSet.txt', 'r', encoding='utf-8') as f:
    text = f.read()

# here are all the 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 with an optional bias term"""

    def __init__(self, head_size, use_bias=True):
        super().__init__()
        self.key = nn.Linear(n_embd, head_size, bias=use_bias)
        self.query = nn.Linear(n_embd, head_size, bias=use_bias)
        self.value = nn.Linear(n_embd, head_size, bias=use_bias)
        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, hs)
        q = self.query(x) # (B, T, hs)
        # Compute attention scores ("affinities")
        wei = q @ k.transpose(-2, -1) * k.shape[-1]**-0.5 # (B, T, hs)
@ (B, hs, 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, hs)
        out = wei @ v # (B, T, T) @ (B, T, hs) -> (B, T, hs)
        return out

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

    def __init__(self, num_heads, head_size, use_bias=True):
        super().__init__()
        self.heads = nn.ModuleList([Head(head_size, use_bias) for _ in
range(num_heads)])
        self.proj = nn.Linear(head_size * num_heads, 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, use_bias=True):
        super().__init__()
        head_size = n_embd // n_head
        self.sa = MultiHeadAttention(n_head, head_size, use_bias)
        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

class GPTLanguageModel(nn.Module):
    def __init__(self, use_bias=True):
        super().__init__()
        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,
use_bias=use_bias) for _ in range(n_layer)])
        self.ln_f = nn.LayerNorm(n_embd)
        self.lm_head = nn.Linear(n_embd, vocab_size)

        self.apply(self._init_weights)

    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)

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

        loss = None
        if targets is not None:
            logits = logits.view(-1, logits.size(-1))
            targets = targets.view(-1)
            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 = GPTLanguageModel()
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()))
```

(e)

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

```
# Set hyperparameters
batch_size = 64
block_size = 256
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)

# Data preparation
with
open('/kaggle/input/input-childspeech-trainingset-txt/input_childSpeec
h_trainingSet.txt', 'r', encoding='utf-8') as f:
    text = f.read()

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

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]
decode = lambda l: ''.join([itos[i] for i in l])

data = torch.tensor(encode(text), dtype=torch.long)
n = int(0.9 * len(data))
train_data = data[:n]
val_data = data[n:]

def get_batch(split):
    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])
    return x.to(device), y.to(device)

@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 FeedForward(nn.Module):
    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 AttentionHead(nn.Module):
    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)
        q = self.query(x)
        wei = (q @ k.transpose(-2, -1)) * k.shape[-1] ** -0.5
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf'))
        wei = F.softmax(wei, dim=-1)
        wei = self.dropout(wei)
        v = self.value(x)
        out = wei @ v
        return out

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

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

class TransformerBlock(nn.Module):
    def __init__(self, n_embd, n_head):
        super().__init__()
        head_size = n_embd // n_head
        self.sa = MultiHeadAttention(n_head, head_size)
        self.ffwd = FeedForward(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

class TransformerModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.token_embedding = nn.Embedding(vocab_size, n_embd)
        self.position_embedding = nn.Embedding(block_size, n_embd)
        self.blocks = nn.Sequential(*[TransformerBlock(n_embd, n_head)
for _ in range(n_layer)])
        self.ln_f = nn.LayerNorm(n_embd)
        self.lm_head = nn.Linear(n_embd, vocab_size)

    def forward(self, idx, targets=None):
        B, T = idx.shape
        tok_emb = self.token_embedding(idx)
        pos_emb = self.position_embedding(torch.arange(T,
device=device))
        x = tok_emb + pos_emb
        x = self.blocks(x)
        x = self.ln_f(x)
        logits = self.lm_head(x)

        if targets is None:
            loss = None
        else:
            logits = logits.view(-1, logits.size(-1))
            targets = targets.view(-1)
            loss = F.cross_entropy(logits, targets)

        return logits, loss
    def generate(self, idx, max_new_tokens):
        for _ in range(max_new_tokens):
            idx_cond = idx[:, -block_size:]
            logits, _ = self(idx_cond)
            logits = logits[:, -1, :]
            probs = F.softmax(logits, dim=-1)
            idx_next = torch.multinomial(probs, num_samples=1)
            idx = torch.cat((idx, idx_next), dim=1)
        return idx

model = TransformerModel().to(device)
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)

for iter in range(max_iters):
    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}")

    xb, yb = get_batch('train')
    logits, loss = model(xb, yb)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

context = torch.zeros((1, 1), dtype=torch.long, device=device)
print(decode(model.generate(context, max_new_tokens=100)[0].tolist()))
```

Question (ii)

(a)

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

```
# Hyperparameters
batch_size = 32
block_size = 128
learning_rate = 3e-4
max_iters = 5000
eval_interval = 500
eval_iters = 200
dropout = 0.1
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# Model configuration
config = {"n_embd": 256, "n_head": 4, "n_layer": 4}

# Load and preprocess the training data
def load_data(filepath):
    with open(filepath, 'r', encoding='utf-8') as f:
        text = f.read()
    chars = sorted(list(set(text)))
    stoi = {ch: i for i, ch in enumerate(chars)}
    itos = {i: ch for ch, i in stoi.items()}
    encode = lambda s: [stoi[c] for c in s]
    decode = lambda l: ''.join([itos[i] for i in l])
    data = torch.tensor(encode(text), dtype=torch.long)
    return data, stoi, itos, len(chars)

train_data, stoi, itos, vocab_size =
load_data('/kaggle/input/input-childspeech-testset-txt/input_childSpee
ch_testSet.txt')

# Split into training and validation sets
n = int(0.9 * len(train_data))
train_split = train_data[:n]
val_split = train_data[n:]

def get_batch(split):
    data = train_split if split == 'train' else val_split
    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])
    return x.to(device), y.to(device)

@torch.no_grad()
def evaluate_loss(model):
    model.eval()
    losses = {"train": [], "val": []}
    for split in ['train', 'val']:
        loss_sum = 0
        for _ in range(eval_iters):
            x, y = get_batch(split)
            _, loss = model(x, y)
            loss_sum += loss.item()
        losses[split] = loss_sum / eval_iters
    model.train()
    return losses

# Define the Transformer model
class Transformer(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.token_embedding = nn.Embedding(vocab_size,
config["n_embd"])
        self.position_embedding = nn.Embedding(block_size,
config["n_embd"])
        self.layers = nn.Sequential(*[TransformerBlock(config) for _
in range(config["n_layer"])])
        self.ln_f = nn.LayerNorm(config["n_embd"])
        self.head = nn.Linear(config["n_embd"], vocab_size)
        self.apply(self._init_weights)

    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if module.bias is not None:
                nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            nn.init.normal_(module.weight, mean=0.0, std=0.02)


    def forward(self, idx, targets=None):
        B, T = idx.size()
        token_emb = self.token_embedding(idx)
        pos_emb = self.position_embedding(torch.arange(T,
device=device))
        x = token_emb + pos_emb
        x = self.layers(x)
        x = self.ln_f(x)
        logits = self.head(x)
        loss = None
        if targets is not None:
            loss = F.cross_entropy(logits.view(-1, logits.size(-1)),
targets.view(-1))
        return logits, loss

# Transformer Block
class TransformerBlock(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.attn = MultiHeadAttention(config["n_head"],
config["n_embd"] // config["n_head"])
        self.ff = FeedForward(config["n_embd"])
        self.ln1 = nn.LayerNorm(config["n_embd"])
        self.ln2 = nn.LayerNorm(config["n_embd"])

    def forward(self, x):
        x = x + self.attn(self.ln1(x))
        x = x + self.ff(self.ln2(x))
        return x
```

```
# Multi-Head Attention
class MultiHeadAttention(nn.Module):
    def __init__(self, n_head, head_size):
        super().__init__()
        self.heads = nn.ModuleList([AttentionHead(head_size) for _ in
range(n_head)])
        self.proj = nn.Linear(n_head * head_size, head_size * n_head)
        self.dropout = nn.Dropout(dropout)

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

# Single Attention Head
class AttentionHead(nn.Module):
    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(config["n_embd"], head_size, bias=False)
        self.query = nn.Linear(config["n_embd"], head_size,
bias=False)
        self.value = nn.Linear(config["n_embd"], head_size,
bias=False)
        self.register_buffer("tril", torch.tril(torch.ones(block_size,
block_size)))

    def forward(self, x):
        B, T, C = x.size()
        k = self.key(x)
        q = self.query(x)
        wei = (q @ k.transpose(-2, -1)) * (C ** -0.5)
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf'))
        wei = F.softmax(wei, dim=-1)
        v = self.value(x)
        return wei @ v
```

```
# FeedForward Layer
class FeedForward(nn.Module):
    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)

# Train the model
model = Transformer(config).to(device)
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)

for iter in range(max_iters):
    if iter % eval_interval == 0:
        losses = evaluate_loss(model)
        print(f"Step {iter}: Train Loss {losses['train']:.4f}, Val
Loss {losses['val']:.4f}")

    x, y = get_batch('train')
    _, loss = model(x, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

# Evaluate on test set
def test_model(model, test_filepath):
    with open(test_filepath, 'r', encoding='utf-8') as f:
        test_text = f.read()
    test_data = torch.tensor([stoi[c] for c in test_text],
dtype=torch.long).to(device)
```

```
    test_loss = 0
    model.eval()
    with torch.no_grad():
        for i in range(0, len(test_data) - block_size, block_size):
            x = test_data[i:i + block_size].unsqueeze(0)
            y = test_data[i + 1:i + block_size + 1].unsqueeze(0)
            _, loss = model(x, y)
            test_loss += loss.item()
    return test_loss / len(test_data)

test_loss = test_model(model,
'/kaggle/input/input-childspeech-testset-txt/input_childSpeech_testSet
.txt')
print(f"Test Loss: {test_loss:.4f}")
```

(b)

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

# Hyperparameters
batch_size = 32
block_size = 128
learning_rate = 3e-4
max_iters = 5000
eval_interval = 500
eval_iters = 200
dropout = 0.1
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# Model configuration
config = {"n_embd": 256, "n_head": 4, "n_layer": 4}

# Load and preprocess the training data
def load_data(filepath):
    with open(filepath, 'r', encoding='utf-8') as f:
        text = f.read()
    chars = sorted(list(set(text)))
    stoi = {ch: i for i, ch in enumerate(chars)}
    itos = {i: ch for ch, i in stoi.items()}
    encode = lambda s: [stoi[c] for c in s]
    decode = lambda l: ''.join([itos[i] for i in l])
    data = torch.tensor(encode(text), dtype=torch.long)
    return data, stoi, itos, len(chars)

train_data, stoi, itos, vocab_size =
load_data('/kaggle/input/input-shakespeare-txt/input_shakespeare.txt')

# Split into training and validation sets
n = int(0.9 * len(train_data))
train_split = train_data[:n]
val_split = train_data[n:]

def get_batch(split):
    data = train_split if split == 'train' else val_split
    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])
    return x.to(device), y.to(device)

@torch.no_grad()
def evaluate_loss(model):
    model.eval()
    losses = {"train": [], "val": []}
    for split in ['train', 'val']:
        loss_sum = 0
        for _ in range(eval_iters):
            x, y = get_batch(split)
            _, loss = model(x, y)
            loss_sum += loss.item()
        losses[split] = loss_sum / eval_iters
    model.train()
    return losses

# Define the Transformer model
class Transformer(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.token_embedding = nn.Embedding(vocab_size,
config["n_embd"])
        self.position_embedding = nn.Embedding(block_size,
config["n_embd"])
        self.layers = nn.Sequential(*[TransformerBlock(config) for _
in range(config["n_layer"])])
        self.ln_f = nn.LayerNorm(config["n_embd"])
        self.head = nn.Linear(config["n_embd"], vocab_size)
        self.apply(self._init_weights)

    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if module.bias is not None:
                nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            nn.init.normal_(module.weight, mean=0.0, std=0.02)

    def forward(self, idx, targets=None):
        B, T = idx.size()
        token_emb = self.token_embedding(idx)
        pos_emb = self.position_embedding(torch.arange(T,
device=device))
        x = token_emb + pos_emb
        x = self.layers(x)
        x = self.ln_f(x)
        logits = self.head(x)
        loss = None
        if targets is not None:
            loss = F.cross_entropy(logits.view(-1, logits.size(-1)),
targets.view(-1))
        return logits, loss

class TransformerBlock(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.attn = MultiHeadAttention(config["n_head"],
config["n_embd"] // config["n_head"])
        self.ff = FeedForward(config["n_embd"])
        self.ln1 = nn.LayerNorm(config["n_embd"])
        self.ln2 = nn.LayerNorm(config["n_embd"])

    def forward(self, x):
        x = x + self.attn(self.ln1(x))
        x = x + self.ff(self.ln2(x))
        return x

class MultiHeadAttention(nn.Module):
    def __init__(self, n_head, head_size):
        super().__init__()
        self.heads = nn.ModuleList([AttentionHead(head_size) for _ in
range(n_head)])
        self.proj = nn.Linear(n_head * head_size, head_size * n_head)
        self.dropout = nn.Dropout(dropout)

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

class AttentionHead(nn.Module):
    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(config["n_embd"], head_size, bias=False)
        self.query = nn.Linear(config["n_embd"], head_size,
bias=False)
```

```
        self.value = nn.Linear(config["n_embd"], head_size,
bias=False)
        self.register_buffer("tril", torch.tril(torch.ones(block_size,
block_size)))

    def forward(self, x):
        B, T, C = x.size()
        k = self.key(x)
        q = self.query(x)
        wei = (q @ k.transpose(-2, -1)) * (C ** -0.5)
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf'))
        wei = F.softmax(wei, dim=-1)
        v = self.value(x)
        return wei @ v

class FeedForward(nn.Module):
    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)

model = Transformer(config).to(device)
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)

for iter in range(max_iters):
    if iter % eval_interval == 0:
        losses = evaluate_loss(model)
        print(f"Step {iter}: Train Loss {losses['train']:.4f}, Val
Loss {losses['val']:.4f}")
```

```
    x, y = get_batch('train')
    _, loss = model(x, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

# Evaluate on test set
def test_model(model, test_filepath):
    with open(test_filepath, 'r', encoding='utf-8') as f:
        test_text = f.read()
    test_data = torch.tensor([stoi[c] for c in test_text],
dtype=torch.long).to(device)
    test_loss = 0
    model.eval()
    with torch.no_grad():
        for i in range(0, len(test_data) - block_size, block_size):
            x = test_data[i:i + block_size].unsqueeze(0)
            y = test_data[i + 1:i + block_size + 1].unsqueeze(0)
            _, loss = model(x, y)
            test_loss += loss.item()
    return test_loss / len(test_data)

test_loss = test_model(model,
'/kaggle/input/input-shakespeare-txt/input_shakespeare.txt')
print(f"Test Loss: {test_loss:.4f}")
```