(i)(a)

We are provided with three different datasets, before working on the transform model let’s try to understand each dataset.

- Child Speech Training Set

```
Total characters: 246982
Total lines: 10001
Vocabulary size: 40
Vocabulary: ['\n', ' ', 'A', 'B', 'C', 'D', 'G', 'H', 'I', 'L', 'M', 'N', 'R', 'S', 'U', 'W', 'Y', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
'p', 'r', 's', 't', 'u', 'v', 'w', 'y']

Sample lines:
1. Night night All done
2. Night night I see bird
3. What is that Help me please
```

This dataset is short and simple with around 247k characters and a small vocabulary size of 40.
The sentences consist of 4-6 words, having very simple English language used by small children.

- Child Speech Test Set

```
Total characters: 24113
Total lines: 1001
Vocabulary size: 40
Vocabulary: ['\n', ' ', 'A', 'B', 'C', 'D', 'G', 'H', 'I', 'L', 'M', 'N', 'R', 'S', 'U', 'W', 'Y', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
'p', 'r', 's', 't', 'u', 'v', 'w', 'y']

Sample lines:
1. Look airplane I want cookie
2. I love you I found it
3. I see doggy No touch
```

This dataset is same to the above training set, with just smaller length of about 24k characters

- Shakespeare

```
Total characters: 1115394
Total lines: 40001
Vocabulary size: 65
Vocabulary: ['\n', ' ', '!', '$', '&', "'", ',', '-', '.', '3', ':', ';', '?', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S',
'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']

Sample lines:
1. We are accounted poor citizens, the patricians good.
2. I say unto you, what he hath done famously, he did
3. If I must not, I need not be barren of accusations;
```

The Shakespeare dataset is different from other two datasets. It is huge in size, has rich vocabulary and has sentences which are build using complex English language. The total number of characters exceed 1.1 million. The vocabulary size of 65 consists of everything from smaller case, upper case and special characters. The sentences are longer with more than 10 words in some sentences. The level of English used is also complex and has a poetic nature.

(i)(b)

The default model generated more than 10M parameters.
Thus, to downsize the model I started tweaking parameters like n_embd, n_head and n_layer.

Why decreased n_embd?
The number of parameters, approximately, is quadratically proportional to the embedding dimensions, this has a major effect on the number of parameters. As the child speech dataset is simple, we do not require detail representation for each character and thus we can reduce character embeddings to make the model simple with less parameters.

Why decreased n_head?
Again as the sentences are simple in nature, we do not require multiple heads to retain different contexts. Reducing the number of heads will reduce parameters without compromising performance much. However, we still plan to keep the model multi-headed to not omit contextual information.

Why decreased n_layer?
The sentences are small and simple, thus we won't require multiple layers to understand the text and predict the next character. Each transformer layer will add a full set of attention and feedforward parameters. Having multiple layers is beneficial in cases where the text is complex and required multiple blocks of computation to find the pattern. Thus, reducing number of layers in the transformer is suitable in our case and beneficial for reducing parameters.

Other hyperparameters were not changed for now as they would not affect much in reducing size of the model.

- **ModelA**

I downsized the default GPT by reducing character embeddings and number of attention heads while keeping multiple layers.
I used n_embd - **96**, n_head - **3** and n_layer - **2**. The model was now downsized to 0.255592 million parameters.
Below table shows the output from this model which was run for maximum 1200 iterations.

```
0.255592 M parameters

step 0: train loss 3.6992, val loss 3.7001

step 200: train loss 1.9433, val loss 1.9485

step 400: train loss 0.4152, val loss 0.4179

step 600: train loss 0.3706, val loss 0.3726

step 800: train loss 0.3599, val loss 0.3627

step 1000: train loss 0.3495, val loss 0.3530

step 1199: train loss 0.3466, val loss 0.3508
```

```
Where kit kitty go
I love you Where kits ts
Night night I jump
Go park Read bok
Look mon All done
Lok moon My teddy
I want that I sing
No nap I see doggy
I love you I see bird
I tired I wash hands
No mine No nap
I run fast More bubbles
I jump I d id it
Gouk p rk Coome I hungry
No nelap I masincl
Baw thel bime I Bing hirde
Norto iaik Lo mk dy me digggy flu
I love y yo gouck Sats bioede
I dangy I dranlary ice
Mate t bbig
I run bbig I runtstsaw
I gre bird read me dinosap
I druve y Sadoes onll I m
```

For this small model, the loss drops rapidly from 3.7 to 0.40 early in 400 iterations, after which it gradually decreased to 0.34 till the last 1200th iteration. This shows that the model was quickly able to learn basic patterns in data. Validation loss also follows the same pattern, with a low of 0.35 for the last iteration. The output text generated by this model is a mixture of both child-like sentences and broken English. It was able to grasp and predict small phrases which can be seen in the first half of the output, but later it started producing broken gibberish like 'Gouk p rk' as it struggled generating longer sentences with lesser seen phrases.

This may be due to the model having a small capacity, increasing embeddings or/and number of layers could produce better results

(i)(c)

- **ModelB**

As the first model (ModelA) was small for the dataset, I decided to go for a model with higher number of embedding and increased heads. The aim was to fit the data correctly (minimize loss) and also generate text output in expected structure.

For this I used n_embd - **128**, n_head - **4** and n_layer – **3** with a dropout rate of 0.2.
The model now had 0.636968 million parameters.
Below table shows the output from this model which was run for maximum 1200 iterations.

0.636968 M parameters step 0: train loss 3.7184, val loss 3.7174 step 200: train loss 1.7232, val loss 1.7329 step 400: train loss 0.3764, val loss 0.3806 step 600: train loss 0.3538, val loss 0.3570 step 800: train loss 0.3449, val loss 0.3474 step 1000: train loss 0.3389, val loss 0.3423 step 1199: train loss 0.3393, val loss 0.3432	I did it I climb I hund it I make mess Uh oh spill Daddy play I jump My teddy Loook airplane More bubbles I hidide Uh oh spill Night night I love you Saw big flufy dogy at park it run fast I run fast Big hug I love you No stop I make mess I see ddoggy Batruck I lire d w I sirun fand Mos mamore Nostop Moma I want play with big red bo I drune itity Hed me pleay jup h Help mele pl mease Bisuch I wang more Goh park Moore more Alire mb jule high Noe naly I want cookie Daddy read meahe Bid thucimeHe

For this model, the training loss was consistently lower as compared to ModelA with a minimun of 0.3389. The validation loss also saw a low of 0.3423 at 1000 iterations which was an improvement compared to the first model. There was no significant difference between train and val loss indicating no real signs of overfitting.

The output text generated reflects this behavior. Compared to ModelA, the text is clearer and more in line with child speech vocabulary. Compared to ModelA there are a smaller number of gibberish words. Even tough the performance was improved the model still gave incorrect spelling of partial tokens for e.g., ‘drune itity’ ‘pleay jup h’. This shows that there is still room for improvement.

Thus, increasing embedding size and adding attention heads while keeping the model parameters below 1M could yeil better results.

- **ModelC**

I tried to keep the model close to 1 million parameters to check how a bigger model would behave on the training data.
For this embedding dimension was increased to **192** and the number of heads to **6**, providing wider attention mechanisms, while the number of layers was kept at **2**. The model was correctly downsized to 0.953512 million parameters.

Below table shows the output from this model which was run for maximum 1200 iterations.

0.953512 M parameters step 0: train loss 3.6608, val loss 3.6603 step 200: train loss 1.1693, val loss 1.1849 step 400: train loss 0.3531, val loss 0.3569 step 600: train loss 0.3448, val loss 0.3495 step 800: train loss 0.3381, val loss 0.3438 step 1000: train loss 0.3338, val loss 0.3404 step 1199: train loss 0.3332, val loss 0.3380	I tired I draw Where kitty go I hide All gone Look airplane Big truck Help me please All done What is that Uh oh spill Bath time I run fast Shoes on Mama I want play with big red ball outside I wash hands Yummy apple Daddy play Where kitty go Yummy apple Help me please Read book I did it Help me please I climb Yummy apple Read book Come here Uh oh spilll I make mass hands I want that Bath tiruck I sing What is that I run fast I found it I wash haide Night night Read book Daddy play I want cookie

The downsizing decision was right. Among all three models, ModelC achieved the lowest validation loss of 0.3332. The validation loss also saw a new low of 0.3380. This shows that with increased capacity the model was able to understand pattern and structure better.The output text generated also reflects the same, the sentences are well formed, have good structure and are in-sync with the child speech dataset sentence formation. The spelling mistakes and partial word formations are also rarely seen.

Thus, ModelC demonstrators the best overall performance.

(i)(d)

In self-attention layers, adding a bias would allow the model to learn an offset for in the layers for Query (Q), Key (K) and Value (V) vector transformations. This added bias can give the attention mechanism more flexibility especially in cases of small language models. This can be helpful for our downsized model. To evaluate this I trained the ModelC (n_embd-192 | n_head-6 | n_layer-2 | dropout-0.2) by introducing bias in attention layers.
For consistency, this model was trained for 1200 iterations. Below table shows its output.

0.954664 M parameters step 0: train loss 3.8065, val loss 3.8069 step 200: train loss 1.3680, val loss 1.3839 step 400: train loss 0.3571, val loss 0.3597 step 600: train loss 0.3457, val loss 0.3483 step 800: train loss 0.3378, val loss 0.3423 step 1000: train loss 0.3343, val loss 0.3401 step 1199: train loss 0.3346, val loss 0.3406	I tired Where kitty go Read book Go park Daddy read me dinosaur book before bed I tired Night night I want more juice please What is that I want more juice please Look mon I did it I climb I hungry I run fast Where ball I see bird Where kitty go All done He mt My teddy Refad bog I did id t I drat I dancese Uw ooh juimp I jump I jump Night nighght Lohesy mk mo awme Sanoes bigesu Yuemmmy he apple Yucy mmmmy Binte I want ty ckoo I fove you b ware bitt Loumk awk bit f doggy juffhe d doUwawie big ryLoaDairead me d dinosaur boook before bed Big trug I want that sime

As you can see in the above results, the inclusion of bias did not improve the performance. The biased model achieved slightly higher training loss. The validation loss, which was 0.3380, also increased to 0.3406. This indicates that bias did not improve the model's ability to fit the training data and slightly harmed generalization. As the child speech dataset is small and with low vocabulary, the Q/K/V transformations already fit the data well, adding bias introduces unnecessary flexibility making the model fit some patterns too closely.

This also reflects in the output text generated by the biased model. Compared to non-biased model output, the output for the model having bias is much noisier and more corrupted.

(i)(e)

Skip connection are a core part of the transformer architecture. For each layer, implementing skip connection makes sure that original information is not lost and new information is appended and then passed to the new layer. This avoids model from being stuck and helps in easy pattern recognition. In our previous models, skip connection was added for both self-attention and feed-forward parts of the model.

x = x + self.sa(self.ln1(x)) → skip connection for self-attention

x = x + self.ffwd(self.ln2(x))→ skip connection for feed-forward

All the above models trained were including skip connection in the transformer architecture. To understand the significance of skip connections, I trained the ModelC without skip connection (no bias). Below are the results.

0.953512 M parameters step 0: train loss 3.7566, val loss 3.7546 step 200: train loss 3.1275, val loss 3.1308 step 400: train loss 2.8100, val loss 2.8143 step 600: train loss 2.8090, val loss 2.8132 step 800: train loss 2.8093, val loss 2.8145 step 1000: train loss 2.8069, val loss 2.8124 step 1199: train loss 2.8086, val loss 2.8131	teaBogheead Ilifo cge NaI deaicyho p sekk Ieumdifiwe soy rt tho moel nh Ist c g sp ceugreya pec Y t It d L Uheo t ftb Me d j mtsoaI gatpkoegee tiIdhosaoo N ne d l L s dhcur A doiIe laLg Iolk h dedunro teoy meitleyitne san spom r roate n kohugpopaA r IiIedoag tuip m cuwelde rl o pykdl U Mitkuh suto g g M sp Uv M I jg s be jrt g Breoime ge g Lfhgeslk h Iat b mo gam Uelht duNitelucinutso t g yyh ralkcyim D LB b bpbaIaguBoo dac foafuS e nyod Iiw t kitaL whe go Meeod hboiwor htan m f riw wh t L m nd

As seen in above table, without skip connection the model was stuck around a loss of 2.8. There was no significant improvement in both training and validation loss even for higher iterations. The output text generated is all gibberish with no meaning and no pattern similar to the child speech training data. This result is expected, as without skip connection, model forgets important information it saw earlier, which forces each layer to transform the input on its own. Thus, training becomes unstable and model does not learn any meaningful pattern from the data. Given the small size of the model, keeping skip connection is crucial to retain information through layers for the model to fit data correctly.

This shows that skip connections are crucial for effective learning in transformer architecture.

(ii) a

The best model – ModelC was saved and used to calculate the loss for new unseen dataset - input_childSpeech_testSet.txt. To compare the model performance, a baseline model that predicts next character randomly from the vocabulary, was also used to evaluate the test dataset. Below table shows performance of both models.

Model	Loss	Perplexity
ModelC	0.3456	1.41
Baseline Random	3.6889	40.00

Perplexity is the number of possible options the model believes it has before the next character prediction. Thus, random baseline model has a perplexity of 40 whereas ModelC has a very low perplexity of 1.41. Lower values indicate better models, meaning ModelC makes accurate predictions confidently.
The average loss for ModelC is of 0.3456, which comfortably beats high baseline model loss of 3.6889.
These are very good numbers for ModelC.

This shows that ModelC, the best performing model, learnt strong patterns in the child speech training dataset and also performs well on new unseen text of similar structure. The model performance is significantly better than any baseline model.

(ii) b

ModelC was also used to calculate the loss for unseen dataset - input_shakespeare.txt. To compare the model performance, a baseline model that predicts next character randomly from the Shakespeare dataset vocabulary, was also used on the Shakespeare text dataset.

As Shakespeare dataset had characters ModelC had never seen during training, these out-of-vocabulary characters could not be processed by ModelC and were therefore filtered out. In total 81,136 unseen characters were ignored by the model.

Below table shows performance of both models.

Model	Loss	Perplexity
ModelC	6.5838	723.28
Baseline Random	4.1744	65.00

ModelC performs very poorly on Shakespeare dataset. It has a high loss of 6.5838 with extremely high perplexity 723.28. The random baseline model has performed much better than ModelC with a loss of 4.1744.

However, this result is expected. ModelC was trained on a dataset which had a structure and pattern similar to a child speech. Shakespeare dataset is much more complex, has heavy vocabulary and longer sentences. Model was seeing data which it had not seen previously during training and was not sure which would be the next correct character.

Compared to the child-speech test set in question (ii)(a), where ModelC achieved a loss of 0.3456, the loss jumped to 6.5838 on Shakespeare set. This shows that the model performs well on a familiar text having same child Speech like structure but fails drastically on a text from a different domain, this shows over-specialization to child speech like data.

In practice, this evaluation pipeline can be used in real NLP systems to redirect the input to the correct model. By keeping a track of the perplexity score of the incoming text, the system can automatically judge if the text is out of domain for a specific model and redirect text to the more appropriate model. It can also be used to identity where retraining or fine-tuning of the model is necessary so that it can handle more diverse input texts.

Appendix

```
def describe_dataset(path, num_samples=3):
    # read file
    with open(path, "r", encoding="utf-8") as f:
        text = f.read()

    # stats
    total_chars = len(text)
    total_lines = text.count("\n") + 1
    vocab = sorted(list(set(text)))
    vocab_size = len(vocab)

    # sample lines (non-empty)
    lines = [line.strip() for line in text.split("\n") if line.strip()]
    sample_lines = lines[:num_samples]

    print("=== Dataset Description ===")
    print(f"File: {path}")
    print(f"Total characters: {total_chars}")
    print(f"Total lines: {total_lines}")
    print(f"Vocabulary size: {vocab_size}")
    print(f"Vocabulary: {vocab}")
    print("Sample lines:")
    for i, s in enumerate(sample_lines):
        print(f"{i+1}. {s}")
    print("\n")

describe_dataset("input_childSpeech_trainingSet.txt")
describe_dataset("input_childSpeech_testSet.txt")
describe_dataset("input_shakespeare.txt")
import torch
import torch.nn as nn
from torch.nn import functional as F



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 = 1200
eval_interval = 200
```

```
learning_rate = 3e-4
device = 'cuda' if torch.cuda.is_available() else 'cpu'
eval_iters = 100
n_embd = 192
n_head = 6
n_layer = 2
dropout = 0.2

# ModelA
# n_embd = 96
# n_head = 3
# n_layer = 2

# ModelB
# n_embd = 128
# n_head = 4
# n_layer = 3

torch.manual_seed(1337)
bias_term=False

# wget https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt
with open('input_childSpeech_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=bias_term)
        self.query = nn.Linear(n_embd, head_size, bias=bias_term)
        self.value = nn.Linear(n_embd, head_size, bias=bias_term)
        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()))

SAVE_GPT = 'trained_gpt.pt'

torch.save({
    'model_state_dict': model.state_dict(),
    'vocab': {'stoi': stoi, 'itos': itos},
    'meta': {'n_embd': n_embd, 'n_head': n_head, 'n_layer': n_layer, 'block_size': block_size, 'vocab_size': vocab_size}
}, SAVE_GPT)
print(f"Saved trained model checkpoint to: {SAVE_GPT}")

from google.colab import files
files.download(SAVE_GPT)
```

```
import torch, math
from collections import Counter

SAVE_GPT = 'trained_gpt.pt'
TRAIN_FILE = "input_childSpeech_trainingSet.txt"
TEST_FILE  = "input_childSpeech_testSet.txt"

ckpt = torch.load(SAVE_GPT, map_location=device)
stoi = ckpt['vocab']['stoi']
itos = ckpt['vocab']['itos']
meta = ckpt['meta']
block_size = meta['block_size']
vocab_size = meta['vocab_size']

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

# encode test data
test_text = open(TEST_FILE, 'r', encoding='utf-8').read()
test_data = torch.tensor(encode(test_text), dtype=torch.long)
if len(test_data) < block_size + 1:
    raise ValueError("Test text too short (need > block_size characters).")

# evaluate test set
def eval_model(model, data_tensor, block_size, device):
    losses = []
    for i in range(0, len(data_tensor) - block_size - 1, block_size):
        x = data_tensor[i:i+block_size].unsqueeze(0).to(device)
        y = data_tensor[i+1:i+block_size+1].unsqueeze(0).to(device)
        _, loss = model(x, y)
        losses.append(loss.item())
    if len(losses) == 0:
        raise ValueError("No evaluation blocks created. Check test length vs block_size.")
    avg_loss = sum(losses) / len(losses)
    return avg_loss, math.exp(avg_loss), len(losses)

model_loss, model_ppl, nblocks = eval_model(model, test_data, block_size, device)

#compute baseline
train_text = open(TRAIN_FILE, 'r', encoding='utf-8').read()
```

```
counts = Counter(train_text)
total = sum(counts.values())
eps = 1e-12


def uniform_random_baseline(vocab_size):
    loss = math.log(vocab_size)
    ppl = vocab_size
    return loss, ppl

uni_loss, uni_ppl = uniform_random_baseline(vocab_size)

print(f"ModelC loss = {model_loss:.4f}, ppl = {model_ppl:.2f}")
print(f"Baseline loss = {uni_loss:.4f}, ppl = {uni_ppl:.2f}")

import torch, math
from collections import Counter

SAVE_GPT = 'trained_gpt.pt'
TRAIN_FILE = "input_childSpeech_trainingSet.txt"
SHK_FILE = "input_shakespeare.txt"
device = 'cuda' if torch.cuda.is_available() else 'cpu'
block_size = globals().get('block_size', 256)


# --- load checkpoint ---
ckpt = torch.load(SAVE_GPT, map_location=device)
itos = ckpt['vocab']['itos']
stoi = ckpt['vocab']['stoi']

# --- build model ---
model = GPTLanguageModel().to(device)
model.load_state_dict(ckpt['model_state_dict'])
model.eval()

# --- read Shakespeare and ignore unseen characters ---
text = open(SHK_FILE, 'r', encoding='utf-8').read()
filtered = [c for c in text if c in stoi]   # ignore unknown chars
enc = [stoi[c] for c in filtered]
```

```
print(f"Original Shakespeare tokens: {len(text)}")
print(f"Tokens after filtering unknown chars: {len(enc)}")
print(f"Ignored {len(text) - len(enc)} unseen characters.")

data_tensor = torch.tensor(enc, dtype=torch.long)

# --- evaluate model ---
@torch.no_grad()
def eval_model(model, data_tensor, block_size, device):
    losses = []
    for i in range(0, len(data_tensor) - block_size - 1, block_size):
        x = data_tensor[i:i+block_size].unsqueeze(0).to(device)
        y = data_tensor[i+1:i+block_size+1].unsqueeze(0).to(device)
        _, loss = model(x, y)
        losses.append(loss.item())
    avg = sum(losses)/len(losses)
    return avg, math.exp(avg), len(losses)

model_loss, model_ppl, nblocks = eval_model(model, data_tensor, block_size, device)

# --- compute baseline ---
vocab = sorted(list(set(text)))
vocab_size = len(vocab)

baseline_loss, baseline_ppl = uniform_random_baseline(vocab_size)

print(f"\nModel  -> loss={model_loss:.4f}, ppl={model_ppl:.2f}, blocks={nblocks}")
print(f"Baseline -> loss={baseline_loss:.4f}, ppl={baseline_ppl:.2f}")
```