Week 8 - GPT
Machine Learning

David Esteso Calatrava

December 29, 2024

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Scinece and Statics

1 Datasets

Both the `input_childSpeech_testSet` and `input_childSpeech_trainingSet` contain simple and repetitive phrases in English, typical of a young child, expressing needs, wants, feelings, and everyday situations. These phrases are short and direct, with grammar that is often incomplete or incorrect. This type of data could be useful for training a GPT model to simulate children's conversations, but the model might struggle with more complex conversations without more varied data or examples. The training set is 10.24 times the size of the test set.

Compared to the previous excerpts, the sentences in `input_shakespeare` are longer, make more sense, and appear to be associated with specific characters, giving them a more personalized and distinctive tone. This introduces an additional challenge for a model, as it must generate responses that are not only coherent with the context but also appropriate for each character's style and personality. Furthermore, the format of the text seems more like a theatrical dialogue. As we can see in the table below, the vocabulary size (measured in characters) in the Shakespeare dataset is 62.5% larger than in the others, which share the same size.

Dataset	Length (characters)	Vocabulary Size
Child Training	246982	40
Child Test	24113	40
Shakespeare	1115394	65

Table 1: Lengths and vocabulary sizes of the datasets

2 Model Downsizing

2.1 Reducing Parameters for Child Speech Dataset

We reduced the model's parameters to under 1 million by adjusting various hyperparameters in the script `gpt.py`. About relevant hyperparameters to modify:

The hyperparameters such as `batch_size` (number of sequences processed in parallel), `max_iters` (maximum training iterations), `eval_interval` (interval for evaluating loss), `learning_rate` (step size for optimization), `eval_iters` (number of batches for loss estimation), and `dropout` (rate of dropout for regularization) do not change the number of parameters in the model because these variables control aspects of training, such as data batching, optimization, or regularization. The model's number of parameters is determined solely by its architecture.

The relationship between the number of parameters and `n_head` in a multi-head attention model is indirect. `n_head` divides `n_embd` into smaller heads of size `head_size = n_embd // n_head`, so changing `n_head` affects the dimension of each head (`head_size`) without changing the total `n_embd`. However, when `n_head` changes, the final projection layer, which combines outputs from all heads, also changes slightly in input dimension. For example, with `n_embd = 384` and reducing `n_head` from 6 to 5, `head_size` increases from 64 to 76, and the input dimension to the projection layer decreases from 384 (6×64) to 380 (5×76), resulting in a minor reduction in parameters due to this adjusted input size.

When you change the `block_size`, it mainly affects the size of the position embedding table (`position_embedding_table`), but it doesn't significantly impact the total number of parameters in the model. The position embedding table has dimensions of `(block_size, n_embd)`, so increasing `block_size` adds more entries to the table, but it doesn't change other

parts of the model. As a result, the total number of parameters doesn't change much, and the model's ability to handle longer sequences is what the `block_size` affects, not the overall number of parameters. Reducing the it means the model will process shorter sequences of text at a time, which limits the amount of context it can use to make predictions. With a smaller value, the model focuses on local patterns within a limited window of characters or tokens. While this reduces the model's ability to capture long-range dependencies, it can still be effective when the text has shorter, more localized relationships.

The most relevant hyperparameters for reducing the number of parameters in the model are the embedding dimension (`n_embd`) and the number of Transformer blocks (`n_layer`).

What are we doing by reducing the number of layers? We decrease the model's capacity to learn complex patterns, which can lead to worse text generation quality due to the inability to capture long-range dependencies. While it reduces training time and memory usage, it may also lead to underfitting, especially for complex tasks or large datasets. However, in some cases, reducing layers can help prevent overfitting, particularly when working with smaller or noisy datasets.

What happens when we reduce the embedding dimension? Reducing it lowers the model's representational capacity, as it reduces the number of dimensions available to capture information about tokens and their relationships. This can negatively impact the model's ability to learn complex patterns, leading to poorer performance, especially for tasks that require deep understanding. While reducing it can improve computational efficiency, as happened with the number of layers, it may come at the cost of generalization, as the model might struggle to capture patterns and may overfit the training data.

This is why we have decided to reduce the number of layers to 2, the embedding dimension to 192, and the block size to 128. For this model, as well as for the others, we have constrained the number of iterations to 1000 due to the capabilities of the machine we are working on. The decision to adjust the block size is based on the fact that, while it won't significantly affect the parameters, it does considerably reduce computation time. Additionally, we are not sacrificing much context in this dataset, as the sentences are not very long. The analysis of the results is presented in the following paragraph, along with the other two alternatives for hyperparameters.

Hyperparameter	**Value**
`batch_size`	64
`block_size`	128
`max_iters`	1000
`eval_interval`	100
`learning_rate`	3×10^{-4}
`device`	`CPU`
`eval_iters`	200
`n_embd`	192
`n_head`	6
`n_layer`	2
`dropout`	0.2

Table 2: Hyperparameter configuration for the model.

2.2 Different Downsizing Methods

We will now wxperiment with two alternative downsizing methods and evaluated their performance. The first one focuses on speed, while the other aims to avoid overfitting. In the previous section, we have discussed how hyperparameters affect the number of parameters and in what way. Now, we have the opportunity to explore the remaining hyperparameters and observe how they impact our model. These two new models have only slight variations in the number of parameters, but they allow us to gain a better understanding of each hyperparameter.

The parameters modified in the `fastGPT` model are designed to **speed up training**:

- **block_size = 32** (instead of 128): Reducing the block size means the model processes shorter sequences, which reduces computational complexity and allows for more iterations per unit of time, speeding up training.
- **n_head = 2** (instead of 6): Lowering the number of attention heads reduces the number of attention calculations, which cuts down on computational cost per step and speeds up training.
- **dropout = 0.1** (instead of 0.2): Decreasing dropout reduces regularization, allowing the model to use more activations and speeding up training, though it may slightly affect generalization.

These changes help `fastGPT` train faster by processing less information and performing fewer operations per iteration, although there may be a small trade-off in generalization ability.

On the other hand, we have `antiOverfittingGPT`, designed to prioritize **overfitting prevention** by incorporating strategies that limit the model's capacity to memorize the training data too closely. The changes made are as follows:

- **batch_size = 32** (instead of 64): Reducing the batch size means the model processes fewer sequences at a time. This introduces more noise into the gradient estimates, which can act as a form of regularization.
- **learning_rate = 2e-4** (instead of 3e-4): Reducing the learning rate lowers the step size for each update, making the optimization process more stable and less prone to overshooting. This slower learning process could ensure that the weights are adjusted more carefully, avoiding large jumps.
- **dropout = 0.4** (instead of 0.2): Increasing the dropout rate encourages the model to rely on a broader set of features by randomly "dropping" neurons during training. This prevents the model from becoming too reliant on any specific feature, forcing the model to learn more robust patterns.

The combination of smaller batch sizes, reduced block size, and increased dropout introduces regularization that encourages the model to generalize better to unseen data. Now, we will plot the training and validation loss over steps for each model, including a Dummy model. The results are also summarized in Table 3.

Starting with `fastGPT`, we observe a relative increase in both the training and validation losses compared to the Original Model. The training loss increases by approximately 23.8%, and the validation loss rises by about 23.1%. Despite these increases in loss values, the training time is reduced by around 79.3%. This suggests that `fastGPT` sacrifices some performance in exchange for significantly faster training time, which could be advantageous in time-sensitive scenarios where the model can tolerate higher loss.

Configuration	Training Loss	Validation Loss	Number of Parameters	Training Time
Original Model	0.3477	0.3514	0.928936 M	758.32 seconds
fastGPT	0.4304	0.4324	0.910504 M	156.48 seconds
antiOverfittingGPT	0.3799	0.3823	0.928936 M	327.23 seconds
Dummy	3.1202	3.1302	0.928936 M	842.08 seconds

Table 3: Loss values and training time for different model configurations

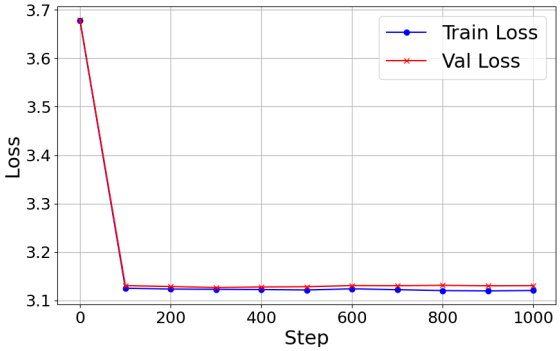


Figure 1: Dummy

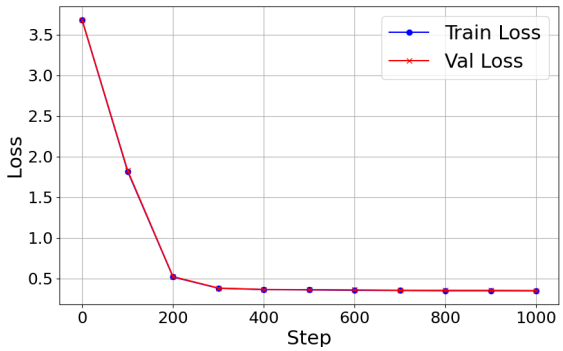


Figure 2: Original Model

In contrast, `antiOverfittingGPT` shows a smaller relative increase in loss compared to `fastGPT`. The training loss increases by 9.3%, and the validation loss rises by 8.8%. Its training time is reduced by about 56.8%, indicating that `antiOverfittingGPT` offers a better balance between generalization (reducing overfitting) and faster training time, without sacrificing performance as much as `fastGPT`.

Although we have observed two approaches to reducing execution time by sacrificing performance to varying degrees, I believe that in systems like these, an increase in loss is less desirable than in other models, such as classification networks. This is because, in systems like the one we're analyzing, if the output value is incoherent, the impact is much greater compared to a classifier, where a drop in precision from 80% to 75% might not be considered as critical. Therefore, qualitative subjective analysis, which evaluates the coherence and usefulness of the results beyond pure metrics, is crucial in these types of models. It is worth noting that none of the three models show signs of overfitting, as at no point does the validation loss stabilize while the training loss continues to decrease.

Now, let's proceed with the qualitative analysis. The original model generates a mix of coherent and less coherent phrases. It maintains a similar structure to the original dataset in many cases but introduces more errors in grammar and punctuation. Some phrases still follow the expected pattern, but others diverge significantly.

Examples of coherent phrases from Model 1:

- "I want cookie I sing"

Examples of incoherent phrases from Model 1:

- "Daddy play I clove you" (should be "I love you")

Approximately **70-75%** of the generated phrases are coherent and follow the basic structure of the original dataset. However, some phrases suffer from grammar issues and lack of context.

`fastGPT`, despite having a higher training loss, produces more coherent outputs compared to `antiOverfittingGPT` in our example. The phrases generated by fastGPT are closer to the

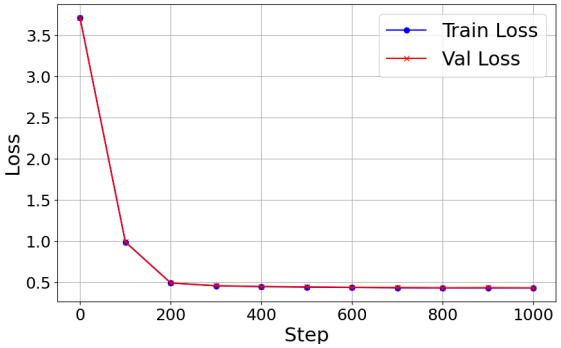


Figure 3: fastGPT

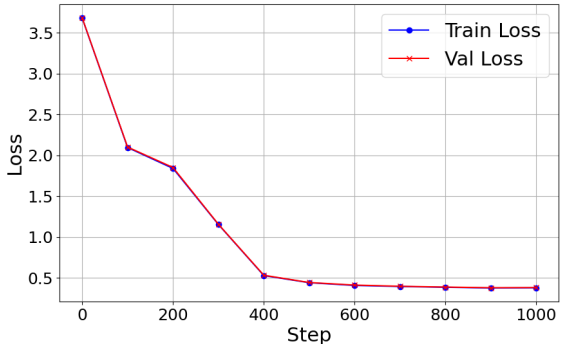


Figure 4: antiOverfittingGPT

original dataset in structure and meaning, even though it occasionally produces minor errors. It retains the simple and repetitive nature of the dataset more consistently.

Examples of coherent phrases from fastGPT:

- "I want Go park"
- "I wash hands I climb"

Examples of incoherent phrases from fastGPT:

- "He mp me please I wash hands" (should be "me")

In terms of coherence, `fastGPT` generates approximately **70%** coherent phrases, which are generally more aligned with the original dataset's style compared to `antiOverfittingGPT`.

On the other hand, `antiOverfittingGPT` shows better performance in terms of reducing training loss, but the phrases generated by this model often diverge from the simple and repetitive structure of the original dataset. It produces more incoherent or incomplete sentences.

Examples of coherent phrases from antiOverfittingGPT:

- "No touch Bath time"

Examples of incoherent phrases from antiOverfittingGPT:

- "Bathudeadsig No stop" (Rare word)
- "D" (Only one letter makes no sense)

The coherence of this model is lower, with approximately **50-55%** of the generated phrases maintaining the simple structure, but many others fail to align with the expected pattern.

In conclusion, while `antiOverfittingGPT` shows lower loss, the phrases it generates are often less coherent compared to `fastGPT`. `fastGPT` produces phrases that are more closely aligned with the original dataset's structure, despite a higher loss, making it more suitable for tasks where coherence and context are crucial. This difference could be attributed to adjustments made to more sensitive parameters, such as the learning rate. It is also worth noting that the increase in dropout was quite significant. Furthermore, the primary goal of this model is to be more general, which, while in classification tasks, can translate to lower precision, in this case it has resulted in more incoherent outputs. Additionally, all three models significantly outperform the dummy model, both in terms of numerical performance with a lower loss and in subjective evaluation. Subjectively, none of the sentences generated by the dummy model are coherent, not even a single word, as its results are purely random due to the fact that it learns from a shuffled text, generating sentences such as:

- "b pMddeHWgImtwomoW eoea oakaiosilda"

This is not the case with any of our three models. To conclude, I believe the best model is the original one. Although it requires more training time, it is consistent and has shown the best results, which is why we will be using it in future sessions. This analysis highlights the importance of qualitative analysis in training tasks, as it has provided insights that could not have been captured through simple graphical evaluations.

3 Impact of Architectural Features

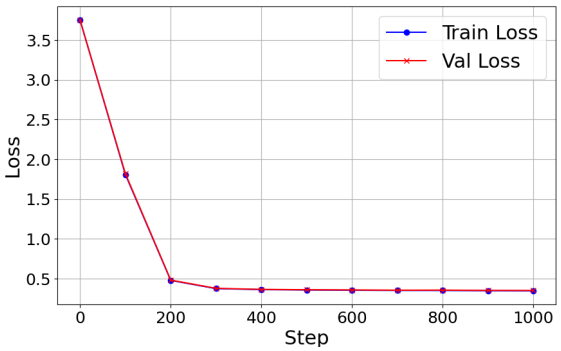
3.1 Bias Terms in Self-Attention Layers

Bias in a neural network is an additional value that is added to the calculations, allowing the model to adjust activations independently from the inputs. Why would someone want to include a bias? Including a bias in a neural network gives the model more flexibility in its learning ability. Without a bias, the model would be limited to producing outputs that always pass through the origin, making it harder to capture complex or nonlinear patterns in the data. The bias acts as an extra adjustment that helps the model learn more complete representations by adjusting activations regardless of the inputs. In attention layers, this can greatly improve the model's ability to adjust attention scores and final projections, allowing it to better capture complex relationships in the data. Without this adjustment term, the model's performance could be compromised, especially for tasks that need more flexibility and generalization.

This code illustrates how the Transformer head looks when the bias is introduced:

```
class Head(nn.Module):

    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(n_embd, head_size, bias=True) #Bias set to True
        self.query = nn.Linear(n_embd, head_size, bias=True)
        self.value = nn.Linear(n_embd, head_size, bias=True)
```



Metric	Value
Training Time (s)	786.15
Model Parameters	0.93M
Final Train Loss	0.3468
Final Validation Loss	0.3516

Figure 5: Bias model

We have tested the model with bias, and when comparing it to the version without bias, the differences, both qualitatively and in terms of loss, as seen in the graph, are not particularly notable. However, this cannot be taken as a definitive conclusion. The model should be tested with more complex data and, in general, with different datasets to better assess these differences.

3.2 Skip Connections

Skip connections in a neural network are direct links between non-adjacent layers, allowing information to flow more efficiently through the network. These connections help mitigate the vanishing gradient problem, which can occur in deep networks where gradients become so small that learning becomes ineffective.

For example, if we have a deep network where each layer has a small weight, without skip connections, the gradient calculation could make the values almost zero. However, with skip connections, the gradients do not vanish as easily. If we take a simplified formula like:

$$a[k] = (w[k]w[k-1] + 1)(w[k-2]w[k-3] + 1) \dots w[1]a[0]$$

If we assume the weights are very small (almost zero), the expression simplifies to:

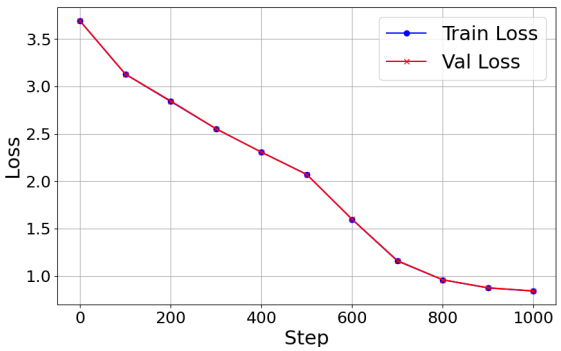
$$a[k] \approx (0 + 1)(0 + 1) \dots w[1]a[0] = w[1]a[0]$$

This means that the value of $a[k]$ depends directly on $w[1]a[0]$, meaning the gradient does not vanish as it backpropagates through the layers. Thus, skip connections allow information to flow more effectively and help prevent gradients from vanishing in very deep networks.

The following code illustrates how the Transformer block looks when the skip connections are removed:

```
class Block(nn.Module):
...

    def forward(self, x):
        x = self.sa(self.ln1(x)) # Without skip connection
        x = self.ffwd(self.ln2(x)) # Without skip connection
        return x
```



Metric	Value
Training Time (s)	792.15
Model Parameters	0.92M
Final Train Loss	0.8408
Final Validation Loss	0.8419

Figure 6: No skip connections model

We can see that by not using skips, the loss skyrockets. Not only that, but qualitatively, we have a model without value. All the phrases generated contain spelling errors compared to the dataset. There are phrases like "Alane" or "I junump n". Saying that we are closer to the dummy model than the others would be too pessimistic, but the incoherence of the sentences is not far off from that of the dummy model. Someone unfamiliar with the dataset we used for training might easily interpret the results as random letters. For example, in the last sentence, we know the dataset and understand that the closest word to that output is "jump," but someone unaware of how we trained the model would struggle to decipher it. This illustrates that the use of skips is an almost essential technique for tasks like this.

4 Evaluation on Test Sets

```
def evaluate_on_test_set(model, test_file_path, results_file):
    # Load test data
    with open(test_file_path, 'r', encoding='utf-8') as f:
        test_text = f.read()

    # Encode test data
    test_data = torch.tensor(encode(test_text), dtype=torch.long)
    test_data = test_data.to(device)

    # Evaluation settings
    model.eval()
    block_size = model.position_embedding_table.weight.shape[0]

    # Lists to store all losses and positions
    losses = []
    positions = []

    with torch.no_grad():
        # Process test data in blocks
        for i in range(0, len(test_data) - block_size, block_size):
            # Get input and target sequences using block_size
            x = test_data[i:i + block_size].unsqueeze(0)
            y = test_data[i + 1:i + block_size + 1].unsqueeze(0)

            # Forward pass
            logits, loss = model(x, y)

            # Store individual loss and position
            losses.append(loss.item())
            positions.append(i)

    # Calculate perplexity using average loss
    avg_loss = sum(losses) / len(losses)
    perplexity = torch.exp(torch.tensor(avg_loss)).item()

    # Generate text using block_size context
    context_text = test_text[:block_size]
    context_encoded = torch.tensor(encode(context_text),
                                   dtype=torch.long,
                                   device=device).unsqueeze(0)
    generated_tokens = model.generate(context_encoded, max_new_tokens=200)
    generated_text = decode(generated_tokens[0].tolist())
    }
```

This function evaluates the trained model on a test dataset, focusing on key metrics like average loss and perplexity.

The test data is processed in blocks because the model has a limited context window (`block_size`). This ensures that long sequences are handled efficiently without running into memory limitations. By processing the data inchunks, we can maintain the model's performance without exceeding memory constraints.

For each block, a forward pass is performed, and the loss is calculated. These losses are stored to track the model's performance across the entire dataset. The average loss is then used to compute the perplexity, which is explained later.

Then, the function generates text using the first `block_size` tokens as context, in order to observe how well the model can continue a sequence based on this context.

Finally, the evaluation results, along with the generated text, are saved for further analysis.

4.1 Test Loss on input_childSpeech_testSet.txt

The model with the best configuration was tested on the `input_childSpeech_testSet.txt` dataset.

Metric	Original model	Dummy model
Block size used	128	128
Final Perplexity	1.43	22.81
Average Loss	0.3573	3.1274
Number of blocks evaluated	188	188

Table 4: Comparison of evaluation results for `input_childSpeech_testSet.txt`.

The evaluation results from the test set indicate that the model performs well. With a final perplexity of 1.43, the model shows a low level of uncertainty when predicting the next word or token in the sequence. Perplexity is a measure of how well a probability model predicts a sample, and in the context of language models, it can be thought of as the inverse probability of the predicted words, normalized by the number of words. In simpler terms, a lower perplexity suggests that the model is making better, more confident predictions. For instance, perplexity of 10 would mean that, on average, the model is choosing between 10 possible words when predicting the next token in the sequence. The average loss of 0.3573 supports this.

This good performance was to be expected because the training and evaluation sets are almost identical, which is especially reflected in the very low perplexity of the model. In fact, there is not a single word in the validation test set that is not present in the training set. Of course, qualitatively, there are very few differences compared to the output the model produced during training: a large percentage of the generated text closely resembles the test set with which we validated our model. In reality, this test hasn't provided much clarity, as we haven't presented a real challenge to our model. It isn't seeing anything new, it's not a challenge for it, so such strong performance was anticipated. To sum up, its results are good and significantly better than what a dummy model could achieve.

4.2 Test Loss on input_shakespeare.txt

Similarly, we evaluated the model on the `input_shakespeare.txt` dataset.

The following modification was necessary because the Shakespeare dataset contains tokens that were not present in the original training set:

```
# Add a special token for unknown characters
unknown_token = '<UNK>'
chars.append(unknown_token)
vocab_size += 1

# 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] if c in stoi else stoi[unknown_token] 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
```

By mapping unknown characters to the special `<UNK>` token, the model can at least process the new dataset without crashing.

Metric	Original model	Dummy model
Block size used	128	128
Final Perplexity	908.55	40.01
Average Loss	6.8119	3.6892
Number of blocks evaluated	8714	8714

Table 5: Comparison of evaluation results for `input_shakespeare.txt`

Here we find ourselves in a completely opposite situation compared to the previous one. Not only has the structure and length of the sentences changed, with increased variety in words and more complex sentence structures, but the size of the vocabulary has also grown (as discussed in the first section of this report). This has caused a significant drop in the performance of our model, with loss values close to 7 and perplexity exceeding 900. These results are much worse than those from the previous section. However, this outcome was expected. We calculated that 81,136 characters in the Shakespeare set are not present in the training set, meaning that 7.27% of characters in the Shakespeare set are unknown to the model. It would be unreasonable to expect good performance under these circumstances.

At a subjective level, the sentences generated by the model with the Shakespeare dataset are nearly identical to those from the previous dataset, with no resemblance to Shakespeare's text. Surprisingly, even performance metrics like loss show the model performing worse than the dummy model trained on random text. While the dummy model generates random gibberish, the original model produces text similar to the child dataset, neither of which is useful for our goals. This highlights the poor generalization ability of the original model and gives us an opportunity to critique the pipeline.

Regarding the pipeline workflow, I find it generally suitable. We used one dataset for training and validated the model with others that differed significantly. This approach helped uncover the critical issue of unknown tokens.

However, the pipeline has a major flaw: it was not designed to handle characters not present during training. The validation datasets also posed problems — one was too similar to the training data, only verifying parameter tuning, while the other was so different it was not really useful. It would have been better to find a dataset in between. In reality, the model was trained for a narrow task with specific phrases, and when we moved out of that zone, its performance dropped. This is the main critique of the pipeline, but overall, the sequence of operations — reducing parameters, testing alternatives, and training with one dataset and validating with another — is a good practice, as confirmed by our conclusions.

5 Conclusion

The tests we've conducted suggest that this model struggles to generalize to diverse datasets. When faced with longer sentences, complex words, and no grammatical errors, it failed to perform well, leading us to conclude it's not suitable for real-world use. The child datasets, on the other hand, contained shorter sentences and words, along with grammatical errors, which gave the model an advantage. We couldn't verify its ability to maintain grammatical structure with simple sentences. In conclusion, the model needs testing with more diverse inputs and may require a more powerful machine to fully utilize its potential.

Appendix: Code

gpt.py

```
import torch
import torch.nn as nn
from torch.nn import functional as F
import random
import time
import matplotlib.pyplot as plt

# hyperparameters
batch_size = 64 # how many independent sequences will we process in parallel?
block_size = 128 # what is the maximum context length for predictions?
max_iters = 1000
eval_interval = 100
learning_rate = 3e-4
device = 'cuda' if torch.cuda.is_available() else 'cpu'
eval_iters = 200
n_embd = 192
n_head = 6
n_layer = 2
dropout = 0.2
# ------------

""" # fastgpt
batch_size = 64 # how many independent sequences will we process in parallel?
block_size = 32 # what is the maximum context length for predictions?
max_iters = 1000
eval_interval = 100
learning_rate = 3e-4
device = 'cuda' if torch.cuda.is_available() else 'cpu'
eval_iters = 200
n_embd = 192
n_head = 2
n_layer = 2
dropout = 0.1

# antioverfittinggpt
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 = 1000
eval_interval = 100
learning_rate = 2e-4
device = 'cuda' if torch.cuda.is_available() else 'cpu'
eval_iters = 200
n_embd = 192
n_head = 6
n_layer = 2
dropout = 0.4
"""


toSuffleText = True

torch.manual_seed(1337)

# 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()
    # Added to train a baseline model
    if toSuffleText:
        text = ''.join(random.sample(text, len(text)))

# here are all the unique characters that occur in this text
chars = sorted(list(set(text)))
vocab_size = len(chars)

# Add a special token for unknown characters
unknown_token = '<UNK>'
chars.append(unknown_token)
vocab_size += 1

# 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] if c in stoi else stoi[unknown_token] 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)

start = time.time()

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()

end = time.time()
print(f"training time: {end - start} seconds")

# 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()))

def evaluate_on_test_set(model, test_file_path, results_file):
    """
    Evaluate the model on a test set and save detailed results

    Args:
        model: The trained GPT model
        test_file_path: Path to the test file
        results_file: Path to save the results
    """
    # Load test data
    with open(test_file_path, 'r', encoding='utf-8') as f:
        test_text = f.read()

    # Encode test data
    test_data = torch.tensor(encode(test_text), dtype=torch.long)
    test_data = test_data.to(device)
```

```
# Evaluation settings
model.eval()
block_size = model.position_embedding_table.weight.shape[0]

# Lists to store all losses and positions
losses = []
positions = []

with torch.no_grad():
    # Process test data in blocks
    for i in range(0, len(test_data) - block_size, block_size):
        # Get input and target sequences using block_size
        x = test_data[i:i + block_size].unsqueeze(0)
        y = test_data[i + 1:i + block_size + 1].unsqueeze(0)

        # Forward pass
        logits, loss = model(x, y)

        # Store individual loss and position
        losses.append(loss.item())
        positions.append(i)

# Calculate perplexity using average loss
avg_loss = sum(losses) / len(losses)
perplexity = torch.exp(torch.tensor(avg_loss)).item()

# Generate text using block_size context
context_text = test_text[:block_size]
context_encoded = torch.tensor(encode(context_text),
                               dtype=torch.long,
                               device=device).unsqueeze(0)
generated_tokens = model.generate(context_encoded, max_new_tokens=200)
generated_text = decode(generated_tokens[0].tolist())

# Save all results to file
with open(results_file, 'w', encoding='utf-8') as f:
    # Write summary statistics
    f.write(f"=== Evaluation Results for {test_file_path} ===\n\n")
    f.write(f"Block size used: {block_size}\n")
    f.write(f"Final Perplexity: {perplexity:.2f}\n")
    f.write(f"Average Loss: {avg_loss:.4f}\n")
    f.write(f"Number of blocks evaluated: {len(losses)}\n\n")

    # Write loss progression
    f.write("=== Loss Progression ===\n")
    f.write("Position,Loss\n")
    for pos, loss in zip(positions, losses):
        f.write(f"{pos},{loss}\n")

    # Write generated text
    f.write("\n=== Text Generation ===\n")
    f.write(f"Context used ({len(context_text)} characters):\n{context_text}\n\n")
    f.write("Generated text:\n")
    f.write(generated_text[len(context_text):])

# Return data for plotting
return {
    'positions': positions,
    'losses': losses,
    'perplexity': perplexity,
```

```
        'avg_loss': avg_loss
    }

def plot_loss_progression(evaluation_results, test_file, output_file):
    """
    Create and save a plot of the loss progression
    """
    plt.figure(figsize=(10, 6))
    plt.plot(evaluation_results['positions'], evaluation_results['losses'])
    plt.title(f'Loss Progression - {test_file}')
    plt.xlabel('Position in Text')
    plt.ylabel('Loss')
    plt.grid(True)
    plt.savefig(output_file)
    plt.close()

# Evaluate on both test sets
test_sets = [
    'input_childSpeech_testSet.txt',
    'input_shakespeare.txt'
]

# Store results for comparison
all_results = {}

for test_file in test_sets:
    try:
        # Create results filename
        results_file = f'detailed_results_{test_file.replace(".txt", "")}_dummy.txt'
        plot_file = f'loss_progression_{test_file.replace(".txt", "")}_dummy.png'

        # Evaluate and save results
        results = evaluate_on_test_set(model, test_file, results_file)
        all_results[test_file] = results

        # Create loss progression plot
        plot_loss_progression(results, test_file, plot_file)

        # Save comparison data
        with open('comparison_results.txt', 'a', encoding='utf-8') as f:
            f.write(f"\n=== {test_file} ===\n")
            f.write(f"Final Perplexity: {results['perplexity']:.2f}\n")
            f.write(f"Average Loss: {results['avg_loss']:.4f}\n")
            f.write(f"Min Loss: {min(results['losses']):.4f}\n")
            f.write(f"Max Loss: {max(results['losses']):.4f}\n")
            f.write("-" * 50 + "\n")

    except FileNotFoundError:
        warning_str = f"\nWarning: Could not find test file {test_file}\n"
        with open(f'error_log.txt', 'a', encoding='utf-8') as f:
            f.write(warning_str)
```

compare_datasets.py

```
# Compare the length of input children test with input children train by counting characters
with open('input_childSpeech_testSet.txt', 'r', encoding='utf-8') as f:
    test_text = f.read()

with open('input_childSpeech_trainingSet.txt', 'r', encoding='utf-8') as f:
```

```
    text = f.read()

# Count the number of characters in each text
train_length = len(text)
test_length = len(test_text)

print(f"Length of training set: {train_length} characters")
print(f"Length of test set: {test_length} characters")
proportion = test_length / train_length
print(f"The test set is {proportion:.2f} times the size of the training set.")
# Other proportion
proportion = train_length / test_length
print(f"The training set is {proportion:.2f} times the size of the test set.")

# Compare the length of input children test with input children train and input Shakespeare
    by counting characters
with open('input_shakespeare.txt', 'r', encoding='utf-8') as f:
    shakespeare_text = f.read()

# Count the number of characters in Shakespeare text
shakespeare_length = len(shakespeare_text)

print(f"Length of Shakespeare set: {shakespeare_length} characters")

proportion = shakespeare_length / train_length
print(f"The Shakespeare set is {proportion:.2f} times the size of the training set.")
# Other proportion
proportion = train_length / shakespeare_length
print(f"The training set is {proportion:.2f} times the size of the Shakespeare set.")

# Create a sorted list of unique characters in the training text
chars = sorted(list(set(text)))
vocab_size = len(chars)

print(f"Unique characters in training set: {chars}")
print(f"Vocabulary size of training set: {vocab_size} characters")

# Repeat for test set
chars_test = sorted(list(set(test_text)))
vocab_size_test = len(chars_test)

print(f"Unique characters in test set: {chars_test}")
print(f"Vocabulary size of test set: {vocab_size_test} characters")

# Repeat for Shakespeare set
chars_shakespeare = sorted(list(set(shakespeare_text)))
vocab_size_shakespeare = len(chars_shakespeare)

print(f"Unique characters in Shakespeare set: {chars_shakespeare}")
print(f"Vocabulary size of Shakespeare set: {vocab_size_shakespeare} characters")

# Count the number of characters in Shakespeare text that are not in the training text
unknown_chars = 0
for char in shakespeare_text:
    if char not in chars:
        unknown_chars += 1

unknown_proportion = unknown_chars / shakespeare_length
print(f"{unknown_chars} characters in Shakespeare set are not in the training set.")
```

```
print(f"{unknown_proportion:.2%} of characters in Shakespeare set are unknown in the
    training set.")

# Read the texts from the training and test sets
with open('input_childSpeech_testSet.txt', 'r', encoding='utf-8') as f:
    test_words = f.read().split()

with open('input_childSpeech_trainingSet.txt', 'r', encoding='utf-8') as f:
    train_words = f.read().split()

# Create sets of unique words
unique_test_words = set(test_words)
unique_train_words = set(train_words)

# Find words that are in the test set but not in the training set
unknown_words = unique_test_words - unique_train_words

# Calculate the number and percentage of unknown words
unknown_count = len(unknown_words)
test_total_words = len(test_words)
unknown_percentage = (unknown_count / test_total_words) * 100

print(f"Number of unique words in test set: {len(unique_test_words)}")
print(f"Number of unique words in train set: {len(unique_train_words)}")
print(f"Number of words in test set not in train set: {unknown_count}")
print(f"Percentage of unknown words in test set: {unknown_percentage:.2f}%")
```

plots.py

```
import matplotlib.pyplot as plt
import re

def plot_loss_from_files(file_list):
    plt.ioff()

    for file in file_list:
        steps = []
        train_losses = []
        val_losses = []

        with open(file, 'r') as f:
            for line in f:
                match = re.search(r"step (\d+): train loss (\d+\.\d+), val loss (\d+\.\d+)",
                    line)
                if match:
                    steps.append(int(match.group(1)))
                    train_losses.append(float(match.group(2)))
                    val_losses.append(float(match.group(3)))

        fig = plt.figure(figsize=(10, 6))
        plt.plot(steps, train_losses, label='Train Loss', color='blue', linestyle='-',
            marker='o')
        plt.plot(steps, val_losses, label='Val Loss', color='red', linestyle='-', marker='x'
            )
        plt.xlabel('Step', fontsize=22)
        plt.ylabel('Loss', fontsize=22)
        plt.legend(fontsize=22)
        plt.grid(True)
        plt.xticks(fontsize=18)
```

```
        plt.yticks(fontsize=18)

        plt.savefig(f'{file}_loss.png')
        plt.close(fig)

file_list = [
    "antiOverFitting.txt",
    "biasTrue.txt",
    "ChildTest.txt",
    "ChildTestAvoidingOverfitting.txt",
    "dummyChild",
    "dummysheakespeare",
    "fastGPT.txt",
    "NoSkips.txt",
    "OriginalModelChildTrain.txt",
    "Shakespeare.txt"
]
plot_loss_from_files(file_list)
```