CS7CS4 Machine Learning
Week 9 Assignment

Name: Michael Millard
Student ID: 24364218

December 8, 2024

Question (i)

For each of the 3 datasets provided for this assignment, the code provided in Figure 1 was executed. It reads in text file corresponding to the desired dataset, prints out its vocabulary (i.e. the complete list of character tokens), and prints out the size of the dataset vocabulary (i.e. the number of character tokens).

```python
# read in dataset textfile
file_name = 'input_childSpeech_trainingSet.txt'
with open(file_name, 'r', encoding='utf-8') as f:
    text = f.read()

# dataset size, its vocab of characters and its vocab size
dataset_size = len(text)
chars = sorted(list(set(text)))
vocab_size = len(chars)

# print some info about the dataset
print("-----------------")
print(f"file: {file_name}")
print(f"chars: {chars}")
print(f"dataset_size: {dataset_size}")
print(f"vocab_size: {vocab_size}")
```

Figure 1: Code used to read in a dataset and print out relevant info about it

Question (i)(a)

1. Child Speech Training Set

The first dataset provided was entitled *input_childSpeech_trainingSet.txt*. As the name of the file suggests, this file contained words making up short phrases one would hear a child say. Below on the left is a short snippet of the dataset to demonstrate some examples of the child-like utterances contained in the dataset:

Night night All done
Night night I see bird
What is that Help me please
No mine I climb

The text contains common spelling and grammatical errors that children make such as *"I see bird"* and *"No mine"*. Through out the dataset, child-like versions of words such as *"mama"* and *"daddy"* are seen frequently along with a wide range of other words one would associated with a child playing. Examples of this include the words *"bubbles"*, *"big red apple"* and *"dinosaur"*.

The size of the *input_childSpeech_trainingSet.txt* dataset was 246982 tokens. Its vocabulary size was 40, consisting of most of the letters of the alphabet (upper and lower case), the newline character and the whitespace character.

2. Child Speech Test Set

The second dataset provided was entitled *input_ childSpeech_ testSet.txt.* As expected, it was extremely similar to the training set and consists of utterances one would expect to hear from a child. A snippet of this dataset is provided on the left below:

Look airplane I want cookie
I love you I found it
I see doggy No touch
I tired Night night

This dataset naturally contains many of the same spelling and grammatical errors seen in the child speech training set. From the snippet on the left, examples include *"I want cookie"* and *"I tired"*. Again, playful words and child-like phrases are found throughout the dataset.

The size of the *input_ childSpeech_ testSet.txt* dataset was 24113 tokens. Its vocabulary was the same as that of the child speech training set. Its vocabulary size was 40 and it consisted of most of the letters of the alphabet (upper and lower case), the newline character and the whitespace character.

3. Shakespeare Set

The third dataset provided was entitled *input_ shakespeare.txt.* This dataset was very different to the child speech datasets and appears to have the structure of the script of a Shakespearean play. The words it contains are from old English, as is to be expected from Shakespearean literature from the 17th century. A snippet of this dataset is provided on the left below:

First Citizen:
Very well; and could be content to give him good
report fort, but that he pays himself with being proud.
Second Citizen:
Nay, but speak not maliciously.

It is evident that the words and sentence structure in this dataset stem from old English. A good example of this on the left is the last line *"Nay, but speak not maliciously"*.

The size of the *input_ shakespeare.txt* dataset was 1115394 tokens. Its vocabulary size was 65 and it consisted of all of the letters of the alphabet (upper and lower case), the newline character, the whitespace character, and a few other punctuation marks and special characters.

The information about each dataset described above is summarized below in Table 1.

Dataset Name	**Dataset Size**	**Vocabulary Size**
Child Speech Train	246982	40
Child Speech Test	24113	40
Shakespeare	1115394	65

Table 1: Summary of the 3 datasets

Question (i)(b)

For context leading into this question and the next one, I opted to test 4 different configurations of downsizing the model. The first two models each had close to 1 million parameters but reached this parameter count in different ways. The first model prioritized the embedding dimension over the number of layers and heads; the second model did the opposite. The third configuration was smaller than the first two models with closer to 400000 parameters and the final configuration was the smallest model with closer to just 35000 parameters. This approach allowed me to not just investigate how much factors such as loss and over-fitting change as the parameter count is reduced; but also to investigate whether a model with a higher embedding dimension with fewer layers might outperform another model with a lower embedding dimension with more layers, despite having a similar number of parameters.

The 4 model configurations differed only in the following parameters: the embedding dimension (n_embd), the number of layers (n_layer) and the number of heads (n_head). Other hyper-parameters such as the batch size, the block size, and the dropout rate were kept fixed: $batch\_size = 64$, $block\_size = 256$ and $dropout = 20\%$.

First Configuration

For the first configuration, I opted to choose a combination of parameters that was close to 1 million but favoured a higher embedding dimension over the number of layers and heads to achieve this. I prioritized keeping the embedding dimension high as this allows the input feature vector to be larger and allows the model to capture more of the nuanced relationships between tokens in the input. Doing the opposite (i.e. maximizing the number of layers and heads over the embedding dimension) is explored in the second model. The resulting set of hyper-parameter values are shown below in Table 2.

n_embd	**n_head**	**n_layer**	**Parameter Count**
196	4	2	0.992192M

Table 2: Set of hyper-parameters for first model configuration

Note, the results of this first model are only discussed in the next question, as requested in the question sheet.

Question (i)(c)

The three other model configurations that were used are described below with their corresponding hyper-parameter values, parameter counts and justifications for each of these. The results of all of the models (including the first model) then follow after that.

Second Configuration

For the second configuration, I opted to choose a combination of parameters that was close to 1 million again, but this time I favoured a higher number of layers and heads over the embedding dimension. I prioritized the number of layers and heads in this configuration in order to allow the model to be deeper and therefore capture some of the higher-order dependencies between tokens and increase its attention focus mechanism by being able to simultaneously process more aspects of the input sequence. The resulting set of hyper-parameter values are shown below in Table 3.

n_embd	**n_head**	**n_layer**	**Parameter Count**
120	6	5	0.9106M

Table 3: Set of hyper-parameters for second model configuration

Third Configuration

The third model configuration sought to create a model with significantly fewer parameters than the first two models (closer to 400000). To achieve this, I reduced all of the above hyper-parameters (*n_embd*, *n_layer* and *n_head*) without any priority given to any one of them (best attempt at a proportional reduction to achieve a "medium sized" model). The resulting set of hyper-parameter values are shown below in Table 4.

n_embd	**n_layer**	**n_head**	**Parameter Count**
100	4	3	0.39684M

Table 4: Set of hyper-parameters for third model configuration

Fourth Configuration

The fourth model configuration was the smallest of the four models. It had less than 50000 parameters, which was achieved by reducing *n_embd*, *n_layer* and *n_head* proportionally to create model with a small embedding dimension, few layers and a small number of heads. The resulting set of hyper-parameter values are shown below in Table 5.

n_embd	**n_head**	**n_layer**	**Parameter Count**
32	4	2	0.036072M

Table 5: Set of hyper-parameters for fourth model configuration

Model Results

Each of the 4 models was trained for 5000 iterations and the training loss (t_loss), the validation loss (v_loss) and the generalization gap ($gap = v\_loss - t\_loss$) were tracked at every 500 iterations in the model training process. Based on these values, a discussion on over-fitting and the quality of the output of each model is provided.

First Model

The first model, which had just under 1 million parameters with priority given to the embedding dimension over the number of layers and heads, was trained over 5000 iterations. The training loss (t_loss), the validation loss (v_loss) and the generalization gap (gap), at every 500^{th} iteration are shown below in Table 6 alongside 2, which shows the training and validation loss plotted against the number of iterations.

iter	t_loss	v_loss	gap
0	3.6657	3.6675	0.0017
500	0.3485	0.3528	0.0043
1000	0.3368	0.3406	0.0038
1500	0.3327	0.3390	0.0062
2000	0.3303	0.3357	0.0053
2500	0.3275	0.3350	0.0075
3000	0.3263	0.3359	0.0096
3500	0.3249	0.3382	0.0133
4000	0.3222	0.3403	0.0180
4500	0.3191	0.3421	0.0229
4999	0.3130	0.3448	0.0318

Table 6: Results of first model every 500 iterations

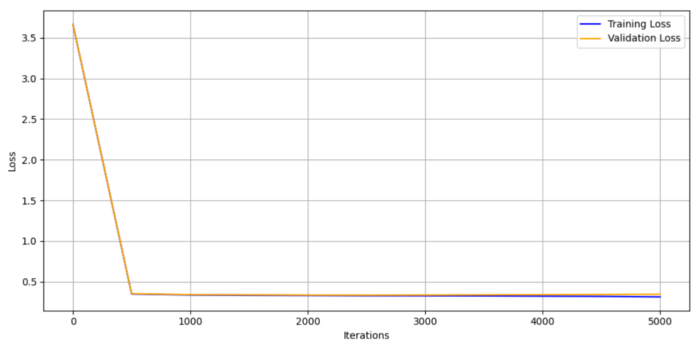

Figure 2: First model training and validation loss

As can be seen in both the table and plot above, the initial loss was high and the generalization gap was small (at iteration 0: $t\_loss = 3.6657, v\_loss = 3.6675$ and $gap = 0.0017$). The jump from iteration 0 to iteration 500 was significant with model loss reducing drastically to $t\_loss = 0.3485, v\_loss = 0.3528$ and $gap = 0.0043$. From then onwards, the training loss continued to decrease slightly and eventually reached its lowest value of $t\_loss = 0.3130$ at the final training iteration. A critical point occurred at iteration 2500, where the validation loss reached its lowest value of $v\_loss = 0.3350$. Beyond this point in the training, the validation loss began to increase slightly, suggesting that the model began over-fitting to the noise in the training set, which resulted in worse generalization performance.

The resulting model was then used to generate output for qualitative assessment. A snippet of which is provided below on the left next to analysis of the generated snippet on the right.

I jump I see bird
Look airplane I found it
I want more juice please Go park
Mama I want play with big red ball outside

The generated output very closely resembled the training dataset. It produced short, child-like utterances and was legible (i.e. the model produced actual words, correctly spaced and without spelling errors that weren't already contained within the training dataset).

Second Model

The second model, which also had close to 1 million parameters but with priority given to the number of layers and heads over the size of the embedding dimension, was then trained over 5000 iterations. The training loss (t_loss), the validation loss (v_loss) and the generalization gap (gap), at every 500^{th} iteration are shown below in Table 7 alongside 3, which shows the training and validation loss plotted against the number of iterations.

iter	t_loss	v_loss	gap
0	3.7267	3.7283	0.0016
500	0.3580	0.3611	0.0032
1000	0.3364	0.3391	0.0027
1500	0.3335	0.3372	0.0037
2000	0.3314	0.3348	0.0034
2500	0.3299	0.3337	0.0038
3000	0.3294	0.3345	0.0051
3500	0.3286	0.3341	0.0056
4000	0.3279	0.3349	0.0069
4500	0.3269	0.3358	0.0089
4999	0.3249	0.3372	0.0122

Table 7: Results of second model every 500 iterations

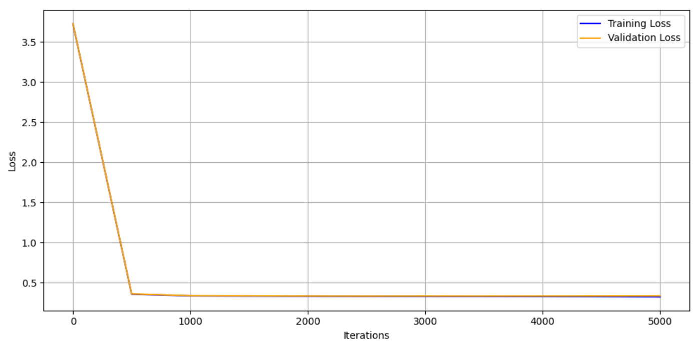

Figure 3: Second model training and validation loss

In a similar manner to the first model, it is evident above that the initial loss was high and the generalization gap was small ($t\_loss = 3.7267, v\_loss = 3.6675$ and $gap = 0.0017$ at iteration 0). Again, the loss reduced significantly from iteration 0 to iteration 500 ($t\_loss = 0.3580, v\_loss = 0.3611$); after which the training loss continued to decrease slightly, as expected. The training loss reached its lowest value of $t\_loss = 0.3249$ at the final training iteration. A critical point occurred again at iteration 2500, where the validation loss reached its lowest value of $v\_loss = 0.3337$. Beyond this point in the training, the validation loss began to increase slightly, suggesting that the model began over-fitting to the noise in the training set, which caused its generalization performance to worsen as training continued.

The resulting model was then used to generate output for qualitative assessment. A snippet of which is provided below on the left next to analysis of the generated snippet on the right.

Saw big fluffy doggy at park it run fast
Big hug Look airplane
I did it Shoes on
I want cookie I hide

The generated output of the second model also closely replicated the input child training set. It produced legible child text with words correctly separated and no spelling mistakes.

Third Model

The third model, which had just below 400000 parameters, was then trained over 5000 iterations. The training loss (t_loss), the validation loss (v_loss) and the generalization gap (gap), at every 500^{th} iteration are shown below in Table 8 alongside 4, which shows the training and validation loss plotted against the number of iterations.

iter	t_loss	v_loss	gap
0	3.6906	3.6921	0.0015
500	0.3712	0.3740	0.0029
1000	0.3418	0.3431	0.0013
1500	0.3363	0.3399	0.0035
2000	0.3322	0.3363	0.0041
2500	0.3320	0.3362	0.0041
3000	0.3310	0.3350	0.0040
3500	0.3294	0.3338	0.0044
4000	0.3294	0.3341	0.0047
4500	0.3290	0.3336	0.0046
4999	0.3272	0.3336	0.0064

Table 8: Results of third model every 500 iterations

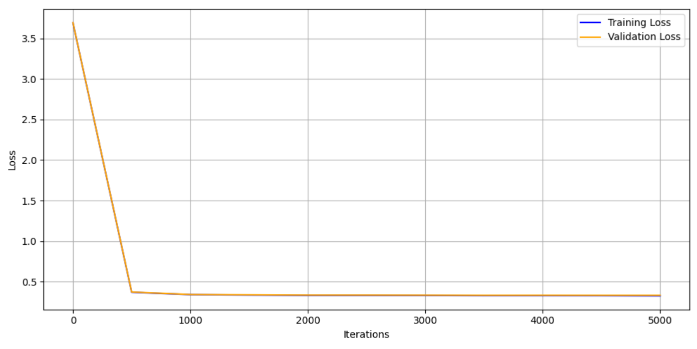

Figure 4: Third model training and validation loss

The results of the third model were surprisingly similar to that of the first two models despite having less than half the number of parameters. The training loss reduced slightly every 500 iterations, as expected, ultimately reaching a lowest value of $t\_loss = 0.3272$. The validation loss reached its lowest value of $v\_loss = 0.3338$ at the 3500^{th} iteration. This was a turning point in that after this point the validation loss began to increase as training continued, which suggests that the model began to over-fit the training dataset.

The resulting model was then used to generate output for qualitative assessment. A snippet of which is provided below on the left next to analysis of the generated snippet on the right.

Daddy play All done
Come here I dance
My tedy Night night
I see doggy I love you

This model produced the first spelling error seen so far in its generated output. It misspelled "teddy" as "tedy", as seen in the 3rd line in the snippet on the left. Similar spelling errors were seen throughout the entire generated output (only one letter incorrect). For the most part, however, the generated output was legible and similar to the training set.

Fourth Model

The final model was the smallest one and had just over 36000 parameters. As such, this model was anticipated to have the highest loss and as a result, produce the lowest quality output of the four models. The results for the fourth configuration are shown in Table 9 and Figure 5 below.

iter	t_loss	v_loss	gap
0	3.6863	3.6863	-0.0000
500	1.7888	1.7913	0.0025
1000	0.5574	0.5602	0.0028
1500	0.4294	0.4333	0.0039
2000	0.3935	0.3962	0.0027
2500	0.3764	0.3781	0.0017
3000	0.3693	0.3709	0.0016
3500	0.3638	0.3661	0.0023
4000	0.3593	0.3616	0.0023
4500	0.3573	0.3583	0.0010
4999	0.3543	0.3562	0.0019

Table 9: Results of fourth model every 500 iterations

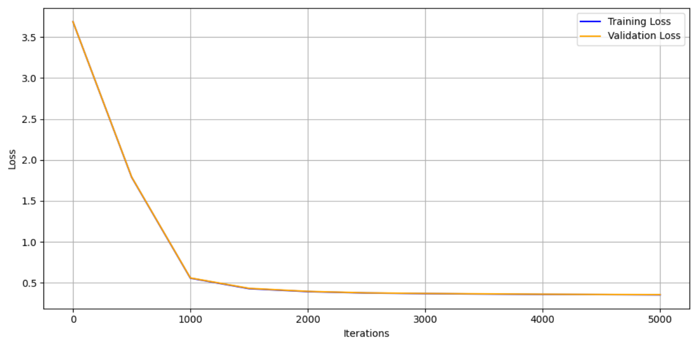

Figure 5: Fourth model training and validation loss

As expected, the fourth model had both the highest training loss and validation loss at the end of 5000 training steps. Its improvements from the 0th iteration to the 500th iteration was also the smallest out of all the models, improving from $t\_loss = 3.6863$ and $v\_loss = 3.6863$ at iteration 0 to $t\_loss = 1.7888, v\_loss = 1.7913$ at iteration 500. The training loss reduced every 500 iterations, reaching its lowest value of $t\_loss = 0.3543$ at the final iteration (the highest training loss out of the four models). The validation also decreased consistently over the 5000 training iterations, something only observed by this model. All the other models had a turning point within the 5000 iterations at which time the model began over-fitting the training data and the validation loss began to increase. The lowest validation loss was therefore $v\_loss = 0.3562$, which occurred at the final iteration. This suggests that the model is not over-fitted to the training data, but the reductions in losses at this point over 500 iterations of training is very small (e.g. from iteration 4500 to iteration 4999 the training loss only decreased by 0.003).

The resulting model was then used to generate output for qualitative assessment. A snippet of which is provided below on the left next to analysis of the generated snippet on the right.

I want Help me please
Big thuck Bath time
Come here I sidraw
Help me please I did it

The fourth model generated output with the most spelling mistakes and lack of coherence. The small snippet on the left contains the spelling error "sidraw", which I assume is a mix of the words "sit" and "draw". Similar mistakes were seen throughout the output, which was to be expected as this model had the fewest parameters and highest losses.

Question (i)(d)

Including bias terms in the self-attention layers of the model allows the model to learn a better fit of the data, making it more flexible. The bias terms of key (K), query (Q) and value (V) enables the model to shift the attention scores and adapt to complex tokens. In doing so, the model has greater flexibility that is better suited to the dataset which results in better generalization performance.

Question (i)(e)

Skip connections have a very important role in the context of transformers. One of the main challenges that skip connections help tackle is that of vanishing gradients, particularly in deep networks. Skip connections allow information to flow through the network from the input unimpeded, which may help the transformer to achieve faster convergence. By adding skip connections to the model, original input information is able to be retained at the output of the model, which allows the model to better learn the key features of the dataset and thereby improve its generalization performance.

Question (ii)

For this part of the assignment, I took the best performing model from the previous section to evaluate its performance on the two test datasets. This was the second model configuration which had the following selection of hyper-parameters:

$$n\_embd = 120, \; n\_layer = 5, \; n\_head = 6, \; num_params = 0.9106M.$$

This configuration was chosen as the best performing model as it had the lowest validation loss at the end of the 5000 training iterations (the validation loss is more important than the training loss since it implies better generalization performance). The first model configuration had a lower training loss than the second model, however, it had a slightly higher validation loss. Since the first configuration's validation loss was only slightly higher, it would also likely perform well on these test sets. However, as mentioned above, the second model generalized the best and was used in the questions that follow.

Question (ii)(a)

To evaluate the performance of my model on the first test set: *input_ childSpeech_ testSet$\dot{t}$xt*, I read in this dataset and created a test function based on the logic in the "estimate_loss()" function. This function, called "estimate_loss_test()" is shown below in Figure 6.

```python
@torch.no_grad()
def estimate_loss_test(test_data):
    model.eval()  # evaluation mode
    total_loss = 0
    total_batches = 0
    losses = []
    batch_indices = []

    # test data blocks
    for k in range(0, len(test_data) - block_size, batch_size):
        # within bounds
        end_idx = min(k + batch_size, len(test_data) - block_size)
        batch_size_actual = end_idx - k

        x = torch.stack([test_data[j:j + block_size] for j in range(k, end_idx)])
        y = torch.stack([test_data[j + 1:j + block_size + 1] for j in range(k, end_idx)])
        x, y = x.to(device), y.to(device)

        # logits and loss
        logits, loss = model(x, y)
        total_loss += loss.item() * batch_size_actual
        total_batches += batch_size_actual
        losses.append(loss.cpu().numpy())
        batch_indices.append(k)

    # average loss, losses, and batches
    return (total_loss / total_batches), losses, batch_indices
```

Figure 6: Code used to read in a dataset and print out relevant info about it

This function computes the loss on the test set by first converting the model into evaluation mode (from train mode), dividing the test dataset into batches, computing the loss on each batch (logits, loss = model(x, y)) and returning the average loss over the entire set, along with an array of the loss of each batch and an array containing the indices corresponding to each batch.

The first test data set (child speech test) was read in and passed to the "estimate_loss_test()" function, which returned an average loss of 0.3513 over the entire dataset. This value is slightly higher the validation loss seen on the final training iteration of this model (which was $v\_loss = 0.3372$). Which was to be expected as this is an entirely new, unseen dataset. The loss of the model in each batch is plotted against its corresponding batch index in Figure 7 below.

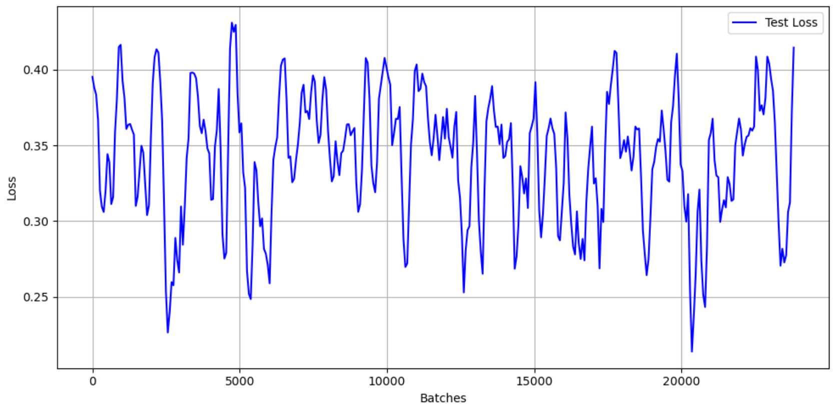

Figure 7: Loss of the best model on the child speech test set

It is clear from the figure that the loss deviated slightly in each batch but always remained within the range $0.2 < \text{loss} < 0.45$. Considering the average loss of 0.3513 in conjunction with the validation loss observed at the end of training of 0.3372, the model performed pretty well on the unseen child speech dataset. This was to be expected however, as this dataset is very similar to the child speech training dataset on which the model was trained.

A dummy model was then created by shuffling the training dataset and training a model with the same set of hyper-parameters on the shuffled dataset. This was achieved by modifying the code that reads in the dataset, as shown below in Figure 8.

```
with open(file_name, 'r', encoding='utf-8') as f:
    text = f.read()
    # if dummy model
    if dummy:
        text = ''.join(random.sample(text, len(text)))
```

Figure 8: Code used to shuffle dataset for dummy model

where *text* is the test dataset text and *dummy* is a boolean variable set by me at the top of the python script. The training loss and validation loss of the dummy model are deliberately excluded from this as it was trained on a shuffled version of the dataset and is simply included in this assignment for the purpose of comparison. Rather, its performance on the unshuffled test dataset is discussed below.

The average loss of the dummy on the test dataset was found to be 3.4317. This was found by using the dummy model in the "estimate_loss_test()" function. The plot corresponding to loss of each batch in the performance evaluation of the dummy model on the child speech test dataset is shown below in Figure 9, which shows the loss of each batch plotted against the index of that batch.

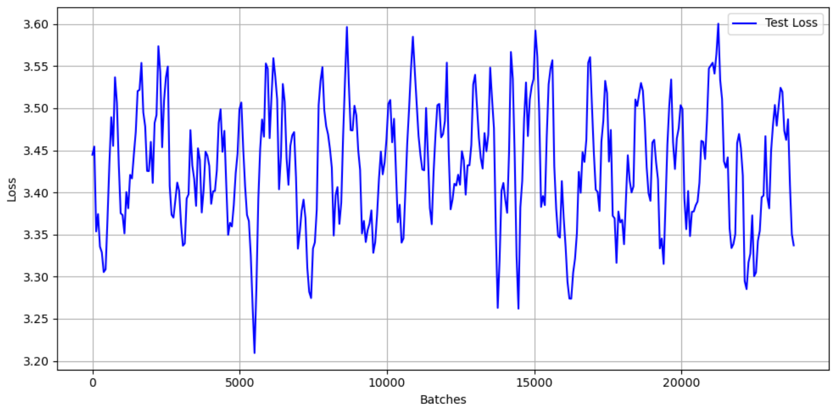

Figure 9: Loss of the dummy model on the child speech test set

It is clear that the best performing model from part 1 of this assignment performs significantly better than this dummy model. The average loss observed by my best model on this test dataset was 0.3513, which is significantly lower than the average loss observed on the dummy model (average loss = 3.4317).

Question (ii)(b)

For the final question of this assignment, the last dataset, *input_shakespearetxt* was read in and used to evaluate the performance of my best model. What is immediately evident upon reading in the Shakespeare test set is that it has a larger vocabulary size (65 characters) than the child speech datasets (40 characters). As such, in order to evaluate my model's performance on this dataset, I had to modify my encoding function to map any new characters (i.e. characters not in the child speech vocabulary) to a whitespace (' ') character in order to discard them. The encode lambda function therefore became:

```
encode = lambda s: [stoi[c] if c in stoi else stoi[' '] for c in s]
```

This was chosen as the best approach in that this model is trained to produce child speech at the end of the day, not to incorporate the entire vocabulary necessary for Shakespearean texts. An alternative approach to this problem for a future project may be to rather extend the vocabulary of the model to include all characters from both datasets. However, for the analysis that follows, I have used the former approach of discarding the 25 new characters contained in the Shakespearean dataset.

My best model achieved an average loss of 7.8794 on the Shakespearean test set. This is a poor result and is significantly higher than the test results observed on the child speech test set (loss of 0.3513). This poor result is to be expected as 25 of the characters in the Shakespearean dataset were discarded, as explained above, which means important letters, punctuation marks, and special characters are simply removed which undermines the meaning, flow and structure of the dataset tremendously. Furthermore, text generation will be extremely poor in that words that are made up of those 25 extra characters are unable to be generated by this model (they are not in its vocabulary). This will lead to many spelling errors, grammar mistakes, and broken structure; which will cause the loss to be very high. The loss of each test batch is plotted against its batch index below in Figure 10.

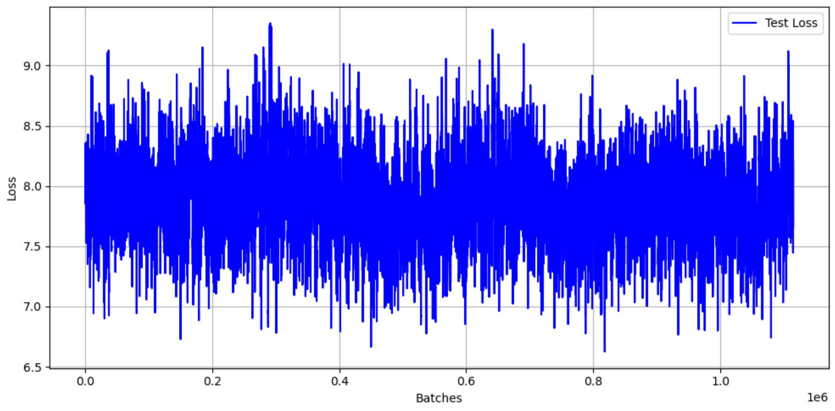

Figure 10: Loss of the best model on the Shakespearean test set

The performance of the same dummy model used in the previous question was then evaluated on the Shakespearean dataset. Surprisingly, this model achieved an average loss of 3.4090 on this dataset, which is still a poor result but is significantly better than the average loss of 7.8794 produced by my best model on the same dataset. Figure 11 below shows the loss of each batch plotted against its batch index of the dummy model on the Shakespearean dataset.

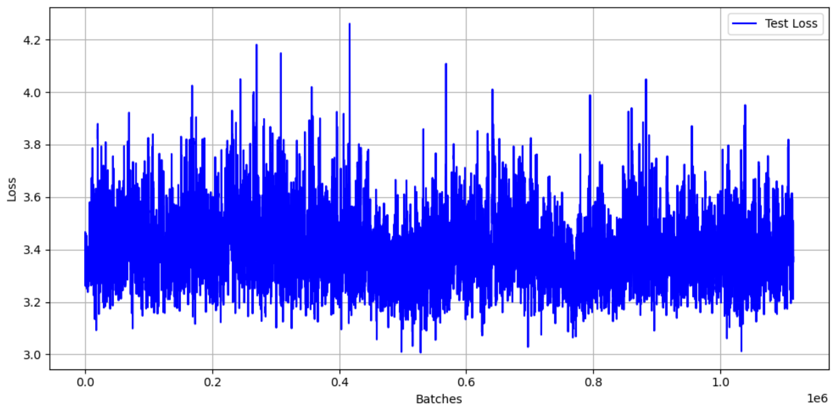

Figure 11: Loss of the dummy model on the Shakespearean test set

It is therefore clear that even though the dummy model performs better on the Shakespearean dataset than my best model, neither of these models performs well on this dataset. The average loss of the dummy model was 3.4090 is still very poor and since 25 of the characters required to form a complete dataset for the Shakespearean text had been discarded by the encoding function, neither of these models are able to generate output that would come close to replicating the input.

However, as was seen on the child speech test set. My best model performed well with an average loss of just 0.3513, which well outperformed the dummy model which had an average loss of 3.4317. This better result is expected as this test set had an identical vocabulary to the training set on which the model was trained and also had a similar structure and contents to the training set.

References

All Python code used in this assignment was based on snippets taken from lectures provided for this course.

A Assignment Code

This appendix contains the code used to answer both questions (i) and (ii) of the assignment.

```python
import random
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
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 = 32
n_head = 4
n_layer = 2
dropout = 0.2
dummy = False
# ------------

# check if using gpu
if (device == 'cuda'):
    print("Using GPU")
else:
    print("Using CPU")

# wget https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt
# datasets, uncomment desired file_name line below
file_name = 'input_childSpeech_trainingSet.txt'
#file_name = 'input_childSpeech_testSet.txt'
#file_name = 'input_shakespeare.txt'
with open(file_name, 'r', encoding='utf-8') as f:
    text = f.read()
    # if dummy model
    if dummy:
        text = ''.join(random.sample(text, len(text)))

# dataset size, its vocab of characters and its vocab size
dataset_size = len(text)
chars = sorted(list(set(text)))
vocab_size = len(chars)

# print some info about the dataset
print("-----------------")
print(f"file: {file_name}")
print(f"chars: {chars}")
print(f"dataset_size: {dataset_size}")
print(f"vocab_size: {vocab_size}")

# 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
        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

# instantiate GPT model and print num params
model = GPTLanguageModel()
m = model.to(device)

# print the number of parameters in the model
num_params = sum(p.numel() for p in m.parameters()) / 1e6
print(f"num_params: {num_params} million")
print("-----------------")
print("Training...")

# create a PyTorch optimizer (AdamW variant of Adam optimizer chosen for better weight decay)
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)

# training and validation losses at each interval
```

```python
train_losses = []
validation_losses = []
generalization_gaps = []

print(f"step, training loss, validation loss, generalization gap")
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()
        # modifying to make easy for LaTeX table populating
        print(f"{iter} & {losses['train']:.4f} & {losses['val']:.4f} & {losses['val'] - losses['trai
        train_losses.append(losses['train'])
        validation_losses.append(losses['val'])
        generalization_gaps.append(losses['val'] - losses['train'])

    # 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()))
print("-----------------")

# plot the training and validation loss
plt.figure(figsize=(10, 5))
plt.plot(range(0, max_iters + 1, eval_interval), train_losses, label='Training Loss', color='blue')
plt.plot(range(0, max_iters + 1, eval_interval), validation_losses, label='Validation Loss', color='
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# -------------------------------------- #

# tests
print("Evaluating test set")

# test datasets
#test_file_name = 'input_childSpeech_testSet.txt'
test_file_name = 'input_shakespeare.txt'
with open(test_file_name, 'r', encoding='utf-8') as f:
    test_text = f.read()

# dataset size, its vocab of characters and its vocab size
test_set_size = len(test_text)
test_chars = sorted(list(set(test_text)))
test_vocab_size = len(test_chars)

# print some info about the dataset
print("-----------------")
print(f"file: {test_file_name}")
print(f"chars: {test_chars}")
print(f"dataset_size: {test_set_size}")
print(f"vocab_size: {test_vocab_size}")

# create a mapping from characters to integers
encode = lambda s: [stoi[c] if c in stoi else stoi[' '] for c in s] # encoder: take a string, output
decode = lambda l: ''.join([itos[i] for i in l]) # decoder: take a list of integers, output a string

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

@torch.no_grad()
def estimate_loss_test(test_data):
    model.eval()  # Set the model to evaluation mode
    total_loss = 0
    total_batches = 0
    losses = []
    batch_indices = []

    # test data blocks
    for k in range(0, len(test_data) - block_size, batch_size):
        # within bounds
        end_idx = min(k + batch_size, len(test_data) - block_size)
        batch_size_actual = end_idx - k

        x = torch.stack([test_data[j:j + block_size] for j in range(k, end_idx)])
        y = torch.stack([test_data[j + 1:j + block_size + 1] for j in range(k, end_idx)])
        x, y = x.to(device), y.to(device)

        # logits and loss
        logits, loss = model(x, y)
        total_loss += loss.item() * batch_size_actual
        total_batches += batch_size_actual
        losses.append(loss.cpu().numpy())
        batch_indices.append(k)

    # average loss, losses, and batches
    return (total_loss / total_batches), losses, batch_indices

# test set
avg_test_loss, test_losses, test_batches = estimate_loss_test(test_data=test_data)
print(f"Test loss: {avg_test_loss:.4f}")

# plot
plt.figure(figsize=(10, 5))
plt.plot(test_batches, test_losses, label='Test Loss', color='blue')
plt.xlabel('Batches')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig('best_model_test_loss.png')
plt.show()
```

—— **END OF ASSIGNMENT** ——