Google Colab Setup

Please run the code below to mount drive if you are running on colab.

Please ignore if you are running on your local machine.

In [1]:
```
# from google.colab import drive
# drive.mount('/content/drive')
```

In [2]:
```
# %cd /content/drive/MyDrive/MiniGPT/
```

Language Modeling and Transformers

The project will consist of two broad parts.

1. **Baseline Generative Language Model**: We will train a simple Bigram language model on the text data. We will use this model to generate a mini story.
2. **Implementing Mini GPT**: We will implement a mini version of the GPT model layer by layer and attempt to train it on the text data. You will then load pretrained weights provided and generate a mini story.

Some general instructions

1. Please keep the name of layers consistent with what is requested in the `model.py` file for each layer, this helps us test in each function independently.
2. Please check to see if the bias is to be set to false or true for all linear layers (it is mentioned in the doc string)
3. As a general rule please read the docstring well, it contains information you will need to write the code.
4. All configs are defined in `config.py` for the first part. While you are writing the code, do not change the values in the config file since we use them to test. Once you have passed all the tests please feel free to vary the parameter as you please.
5. You will need to fill in `train.py` and run it to train the model. If you are running into memory issues please feel free to change the `batch_size` in the `config.py` file. If you are working on Colab please make sure to use the GPU runtime and feel free to copy over the training code to the notebook.

In [1]:
```
!pip install numpy torch tiktoken wandb einops # Install all required packag
```

```
Requirement already satisfied: numpy in /Users/tilboon/opt/anaconda3/envs/c2
47/lib/python3.10/site-packages (1.26.4)
Requirement already satisfied: torch in /Users/tilboon/opt/anaconda3/envs/c2
47/lib/python3.10/site-packages (2.2.2)
Collecting tiktoken
  Downloading tiktoken-0.9.0-cp310-cp310-macosx_10_12_x86_64.whl.metadata
(6.7 kB)
Collecting wandb
  Downloading wandb-0.19.11-py3-none-macosx_11_0_x86_64.whl.metadata (10 kB)
Collecting einops
  Downloading einops-0.8.1-py3-none-any.whl.metadata (13 kB)
Requirement already satisfied: filelock in /Users/tilboon/opt/anaconda3/env
s/c247/lib/python3.10/site-packages (from torch) (3.18.0)
Requirement already satisfied: typing-extensions>=4.8.0 in /Users/tilboon/op
t/anaconda3/envs/c247/lib/python3.10/site-packages (from torch) (4.12.2)
Requirement already satisfied: sympy in /Users/tilboon/opt/anaconda3/envs/c2
47/lib/python3.10/site-packages (from torch) (1.13.3)
Requirement already satisfied: networkx in /Users/tilboon/opt/anaconda3/env
s/c247/lib/python3.10/site-packages (from torch) (3.4.2)
Requirement already satisfied: jinja2 in /Users/tilboon/opt/anaconda3/envs/c
247/lib/python3.10/site-packages (from torch) (3.1.2)
Requirement already satisfied: fsspec in /Users/tilboon/opt/anaconda3/envs/c
247/lib/python3.10/site-packages (from torch) (2025.3.2)
Collecting regex>=2022.1.18 (from tiktoken)
  Using cached regex-2024.11.6-cp310-cp310-macosx_10_9_x86_64.whl.metadata
(40 kB)
Requirement already satisfied: requests>=2.26.0 in /Users/tilboon/opt/anacon
da3/envs/c247/lib/python3.10/site-packages (from tiktoken) (2.32.3)
Collecting click!=8.0.0,>=7.1 (from wandb)
  Downloading click-8.2.1-py3-none-any.whl.metadata (2.5 kB)
Collecting docker-pycreds>=0.4.0 (from wandb)
  Downloading docker_pycreds-0.4.0-py2.py3-none-any.whl.metadata (1.8 kB)
Collecting gitpython!=3.1.29,>=1.0.0 (from wandb)
  Using cached GitPython-3.1.44-py3-none-any.whl.metadata (13 kB)
Requirement already satisfied: platformdirs in /Users/tilboon/opt/anaconda3/
envs/c247/lib/python3.10/site-packages (from wandb) (4.3.6)
Collecting protobuf!=4.21.0,!=5.28.0,<7,>=3.19.0 (from wandb)
  Downloading protobuf-6.31.0-cp39-abi3-macosx_10_9_universal2.whl.metadata
(593 bytes)
Requirement already satisfied: psutil>=5.0.0 in /Users/tilboon/opt/anaconda
3/envs/c247/lib/python3.10/site-packages (from wandb) (7.0.0)
Collecting pydantic<3 (from wandb)
  Downloading pydantic-2.11.4-py3-none-any.whl.metadata (66 kB)
Requirement already satisfied: pyyaml in /Users/tilboon/opt/anaconda3/envs/c
247/lib/python3.10/site-packages (from wandb) (6.0.2)
Collecting sentry-sdk>=2.0.0 (from wandb)
  Downloading sentry_sdk-2.29.1-py2.py3-none-any.whl.metadata (10 kB)
Collecting setproctitle (from wandb)
  Downloading setproctitle-1.3.6-cp310-cp310-macosx_10_9_universal2.whl.meta
data (10 kB)
Requirement already satisfied: setuptools in /Users/tilboon/opt/anaconda3/en
vs/c247/lib/python3.10/site-packages (from wandb) (75.1.0)
Requirement already satisfied: six>=1.4.0 in /Users/tilboon/opt/anaconda3/en
vs/c247/lib/python3.10/site-packages (from docker-pycreds>=0.4.0->wandb) (1.
16.0)
Collecting gitdb<5,>=4.0.1 (from gitpython!=3.1.29,>=1.0.0->wandb)
```

```
  Using cached gitdb-4.0.12-py3-none-any.whl.metadata (1.2 kB)
Collecting annotated-types>=0.6.0 (from pydantic<3->wandb)
  Downloading annotated_types-0.7.0-py3-none-any.whl.metadata (15 kB)
Collecting pydantic-core==2.33.2 (from pydantic<3->wandb)
  Downloading pydantic_core-2.33.2-cp310-cp310-macosx_10_12_x86_64.whl.metad
ata (6.8 kB)
Collecting typing-inspection>=0.4.0 (from pydantic<3->wandb)
  Downloading typing_inspection-0.4.0-py3-none-any.whl.metadata (2.6 kB)
Requirement already satisfied: charset_normalizer<4,>=2 in /Users/tilboon/op
t/anaconda3/envs/c247/lib/python3.10/site-packages (from requests>=2.26.0->t
iktoken) (3.4.1)
Requirement already satisfied: idna<4,>=2.5 in /Users/tilboon/opt/anaconda3/
envs/c247/lib/python3.10/site-packages (from requests>=2.26.0->tiktoken) (3.
10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /Users/tilboon/opt/anac
onda3/envs/c247/lib/python3.10/site-packages (from requests>=2.26.0->tiktoke
n) (2.3.0)
Requirement already satisfied: certifi>=2017.4.17 in /Users/tilboon/opt/anac
onda3/envs/c247/lib/python3.10/site-packages (from requests>=2.26.0->tiktoke
n) (2024.12.14)
Requirement already satisfied: MarkupSafe>=2.0 in /Users/tilboon/opt/anacond
a3/envs/c247/lib/python3.10/site-packages (from jinja2->torch) (3.0.2)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /Users/tilboon/opt/anac
onda3/envs/c247/lib/python3.10/site-packages (from sympy->torch) (1.3.0)
Collecting smmap<6,>=3.0.1 (from gitdb<5,>=4.0.1->gitpython!=3.1.29,>=1.0.0-
>wandb)
  Using cached smmap-5.0.2-py3-none-any.whl.metadata (4.3 kB)
Downloading tiktoken-0.9.0-cp310-cp310-macosx_10_12_x86_64.whl (1.1 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.1/1.1 MB 7.2 MB/s eta 0:00:00
Downloading wandb-0.19.11-py3-none-macosx_11_0_x86_64.whl (21.0 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 21.0/21.0 MB 12.0 MB/s eta 0:00:
00a 0:00:01
Downloading einops-0.8.1-py3-none-any.whl (64 kB)
Downloading click-8.2.1-py3-none-any.whl (102 kB)
Downloading docker_pycreds-0.4.0-py2.py3-none-any.whl (9.0 kB)
Using cached GitPython-3.1.44-py3-none-any.whl (207 kB)
Downloading protobuf-6.31.0-cp39-abi3-macosx_10_9_universal2.whl (425 kB)
Downloading pydantic-2.11.4-py3-none-any.whl (443 kB)
Downloading pydantic_core-2.33.2-cp310-cp310-macosx_10_12_x86_64.whl (2.0 M
B)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.0/2.0 MB 14.4 MB/s eta 0:00:00
Using cached regex-2024.11.6-cp310-cp310-macosx_10_9_x86_64.whl (287 kB)
Downloading sentry_sdk-2.29.1-py2.py3-none-any.whl (341 kB)
Downloading setproctitle-1.3.6-cp310-cp310-macosx_10_9_universal2.whl (17 k
B)
Downloading annotated_types-0.7.0-py3-none-any.whl (13 kB)
Using cached gitdb-4.0.12-py3-none-any.whl (62 kB)
Downloading typing_inspection-0.4.0-py3-none-any.whl (14 kB)
Using cached smmap-5.0.2-py3-none-any.whl (24 kB)
Installing collected packages: typing-inspection, smmap, setproctitle, sentr
y-sdk, regex, pydantic-core, protobuf, einops, docker-pycreds, click, annota
ted-types, tiktoken, pydantic, gitdb, gitpython, wandb
Successfully installed annotated-types-0.7.0 click-8.2.1 docker-pycreds-0.4.
0 einops-0.8.1 gitdb-4.0.12 gitpython-3.1.44 protobuf-6.31.0 pydantic-2.11.4
pydantic-core-2.33.2 regex-2024.11.6 sentry-sdk-2.29.1 setproctitle-1.3.6 sm
map-5.0.2 tiktoken-0.9.0 typing-inspection-0.4.0 wandb-0.19.11
```

In [2]:
```
%load_ext autoreload
%autoreload 2
```

In [3]:
```
import torch
import tiktoken
```

In [4]:
```
from model import BigramLanguageModel, SingleHeadAttention, MultiHeadAttenti
from config import BigramConfig, MiniGPTConfig
import tests
```

In [5]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

In [6]:
```
# If not provided, download from https://drive.google.com/file/d/1g09qUM9Wib
path_to_bigram_tester = "./pretrained_models/bigram_tester.pt" # Load the bi
path_to_gpt_tester = "./pretrained_models/minigpt_tester.pt" # Load the gpt
```

Bigram Language Model (10 points)

A bigram language model is a type of probabilistic language model that predicts a word given the previous word in the sequence. The model is trained on a text corpus and learns the probability of a word given the previous word.

Implement the Bigram model (5 points)

Please complete the `BigramLanguageModel` class in model.py. We will model a Bigram language model using a simple MLP with one hidden layer. The model will take in the previous word index and output the logits over the vocabulary for the next word.

In [10]:
```
# Test implementation for Bigram Language Model
model = BigramLanguageModel(BigramConfig)
tests.check_bigram(model, path_to_bigram_tester, device)
```

Out[10]:
```
'TEST CASE PASSED!!!'
```

Training the Bigram Language Model (2.5 points)

Complete the code in `train.py` to train the Bigram language model on the text data. Please provide plots for both the training and validation in the cell below.

Some notes on the training process:

1. You should be able to train the model slowly on your local machine.
2. Training it on Colab will help with speed.
3. To get full points for this section it is sufficient to show that the loss is decreasing over time. You should see it saturate to a value close to around 5-

6 but as long as you see it decreasing then saturating you should be good.
4. Please log the loss curves either on wandb, tensorboard or any other logger of your choice and please attach them below.

In [11]:
```
from train import solver
```

In [ ]:
```
solver(model_name="bigram")
```

```
Eval Logging Inteval: 10000
Train len 473591
Batch size 32
Evaluating Model 0
eval_dataset length: 3788727
eval_dataloader length: 118398
end of eval dl
Loop Exceeding Number of batches
Eval Loss:  40989480.98461914
Iteration 0, Train Loss: 10.824810981750488 Eval Loss: 10.81877464796152
  batch 1000 loss: 0.004974021434783936
  batch 2000 loss: 0.005717559814453125
  batch 3000 loss: 0.00365205192565918
  batch 4000 loss: 0.004598745822906494
  batch 5000 loss: 0.005582768440246582
  batch 6000 loss: 0.005040801048278808
  batch 7000 loss: 0.0032807955741882325
  batch 8000 loss: 0.005333069324493408
  batch 9000 loss: 0.005066819667816162
  batch 10000 loss: 0.004794302940368652
Evaluating Model 10000
eval_dataset length: 3788727
eval_dataloader length: 118398
end of eval dl
Loop Exceeding Number of batches
Eval Loss:  15245228.141799927
Iteration 10000, Train Loss: 5.806027889251709 Eval Loss: 4.023829620696699
  batch 11000 loss: 0.005302007675170898
  batch 12000 loss: 0.004838509559631348
  batch 13000 loss: 0.0049011225700378415
  batch 14000 loss: 0.004231267929077148
  batch 15000 loss: 0.004204058647155761
  batch 16000 loss: 0.004908410549163819
  batch 17000 loss: 0.0052473778724670414
  batch 18000 loss: 0.005200817584991455
  batch 19000 loss: 0.004652397155761719
  batch 20000 loss: 0.00477155351638794
Evaluating Model 20000
eval_dataset length: 3788727
eval_dataloader length: 118398
end of eval dl
Loop Exceeding Number of batches
Eval Loss:  15129118.964828491
Iteration 20000, Train Loss: 4.178015232086182 Eval Loss: 3.9931837332631495
  batch 21000 loss: 0.0048084230422973636
  batch 22000 loss: 0.004518750667572021
  batch 23000 loss: 0.005138218879699707
  batch 24000 loss: 0.004450829029083252
  batch 25000 loss: 0.0037119176387786865
  batch 26000 loss: 0.00479921293258667
  batch 27000 loss: 0.005578598499298096
  batch 28000 loss: 0.0034216210842132567
  batch 29000 loss: 0.005135415554046631
  batch 30000 loss: 0.004782097339630127
Evaluating Model 30000
eval_dataset length: 3788727
```

```
eval_dataloader length: 118398
end of eval dl
Loop Exceeding Number of batches
Eval Loss:  15132337.777671814
No improvement in eval loss. Count = 1/3
Iteration 30000, Train Loss: 3.9671826362609863 Eval Loss: 3.994033307591718
  batch 31000 loss: 0.005120843887329102
  batch 32000 loss: 0.0051157112121582035
  batch 33000 loss: 0.004493396759033203
  batch 34000 loss: 0.0055880208015441895
  batch 35000 loss: 0.004342808723449707
  batch 36000 loss: 0.0036774282455444337
  batch 37000 loss: 0.005423797607421875
  batch 38000 loss: 0.004264708518981934
  batch 39000 loss: 0.004543434143066407
  batch 40000 loss: 0.005020881175994873
Evaluating Model 40000
eval_dataset length: 3788727
eval_dataloader length: 118398
end of eval dl
Loop Exceeding Number of batches
Eval Loss:  15113965.429336548
Iteration 40000, Train Loss: 5.481503963470459 Eval Loss: 3.989184105025145
  batch 41000 loss: 0.004642388820648193
  batch 42000 loss: 0.004750207424163818
  batch 43000 loss: 0.003711050748825073
  batch 44000 loss: 0.004045965194702149
  batch 45000 loss: 0.004412364482879638
  batch 46000 loss: 0.004403239250183106
  batch 47000 loss: 0.004360567092895508
  batch 48000 loss: 0.004445030212402344
  batch 49000 loss: 0.004821699142456054
  batch 50000 loss: 0.0059126391410827634
Evaluating Model 50000
eval_dataset length: 3788727
eval_dataloader length: 118398
end of eval dl
Loop Exceeding Number of batches
Eval Loss:  15083153.009307861
Iteration 50000, Train Loss: 4.500168800354004 Eval Loss: 3.981051466586181
  batch 51000 loss: 0.0056374711990356445
  batch 52000 loss: 0.004646697998046875
  batch 53000 loss: 0.0041788525581359865
  batch 54000 loss: 0.004428491115570068
  batch 55000 loss: 0.0034737329483032226
  batch 56000 loss: 0.005499123096466064
  batch 57000 loss: 0.0045111489295959475
  batch 58000 loss: 0.006061489105224609
  batch 59000 loss: 0.004696951866149902
  batch 60000 loss: 0.005325529098510742
Evaluating Model 60000
eval_dataset length: 3788727
eval_dataloader length: 118398
end of eval dl
Loop Exceeding Number of batches
Eval Loss:  15070278.46314621
```

```
Iteration 60000, Train Loss: 4.0782623291015625 Eval Loss: 3.977653355405657
5
  batch 61000 loss: 0.0040394287109375
  batch 62000 loss: 0.0045044107437133786
  batch 63000 loss: 0.005724696636199951
  batch 64000 loss: 0.004643142700195313
  batch 65000 loss: 0.004648535251617431
  batch 66000 loss: 0.005108572006225586
  batch 67000 loss: 0.004307381629943848
  batch 68000 loss: 0.005702144145965577
  batch 69000 loss: 0.004526485919952392
  batch 70000 loss: 0.005118056297302246
Evaluating Model 70000
eval_dataset length: 3788727
eval_dataloader length: 118398
end of eval dl
Loop Exceeding Number of batches
Eval Loss:  15023009.413856506
Iteration 70000, Train Loss: 4.587998867034912 Eval Loss: 3.9651771498084076
  batch 71000 loss: 0.003749931335449219
  batch 72000 loss: 0.005127687454223633
  batch 73000 loss: 0.004973254680633545
  batch 74000 loss: 0.005316810131072998
  batch 75000 loss: 0.0046036171913146975
  batch 76000 loss: 0.005134759426116943
  batch 77000 loss: 0.004811874866485596
  batch 78000 loss: 0.005363302230834961
  batch 79000 loss: 0.004295323848724365
  batch 80000 loss: 0.003946974992752075
Evaluating Model 80000
eval_dataset length: 3788727
eval_dataloader length: 118398
end of eval dl
Loop Exceeding Number of batches
Eval Loss:  15068272.796188354
No improvement in eval loss. Count = 1/3
Iteration 80000, Train Loss: 4.468671798706055 Eval Loss: 3.977123979128753
  batch 81000 loss: 0.005607658863067627
  batch 82000 loss: 0.004919509410858155
  batch 83000 loss: 0.0041231951713562014
  batch 84000 loss: 0.005013470649719238
  batch 85000 loss: 0.004561370849609375
  batch 86000 loss: 0.005083860874176026
  batch 87000 loss: 0.005026008605957031
  batch 88000 loss: 0.005620823383331299
  batch 89000 loss: 0.003659841060638428
  batch 90000 loss: 0.00667283296585083
Evaluating Model 90000
eval_dataset length: 3788727
eval_dataloader length: 118398
end of eval dl
Loop Exceeding Number of batches
Eval Loss:  15052904.1509552
No improvement in eval loss. Count = 2/3
Iteration 90000, Train Loss: 4.622589111328125 Eval Loss: 3.9730675747677324
  batch 91000 loss: 0.00536463212966919
```

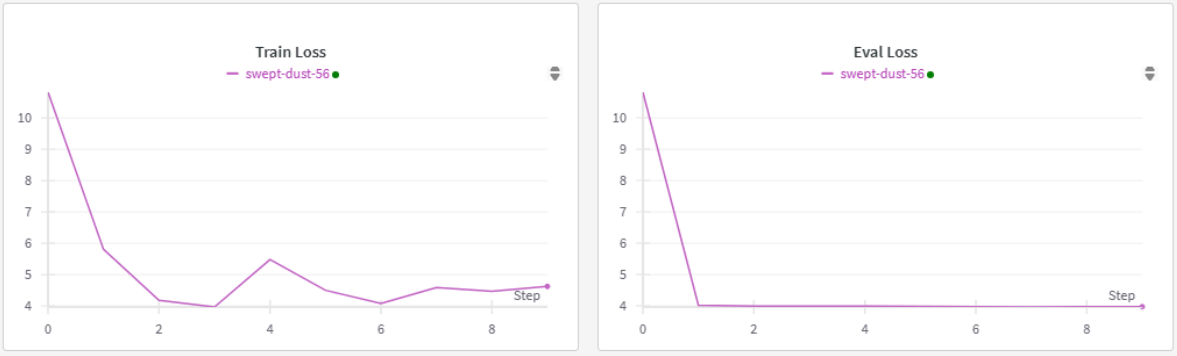
```
  batch 92000 loss: 0.003666687250137329
  batch 93000 loss: 0.004177570819854736
  batch 94000 loss: 0.004982422351837159
  batch 95000 loss: 0.004343136787414551
  batch 96000 loss: 0.004182270050048828
  batch 97000 loss: 0.004515249252319336
  batch 98000 loss: 0.004174974918365479
  batch 99000 loss: 0.005269353866577149
  batch 100000 loss: 0.005200906753540039
Evaluating Model 100000
eval_dataset length: 3788727
eval_dataloader length: 118398
end of eval dl
Loop Exceeding Number of batches
Eval Loss:  15058936.429107666
No improvement in eval loss. Count = 3/3
Early stopping triggered.
```

Train and Valid Plots

Show the training and validation loss plots

In [104…

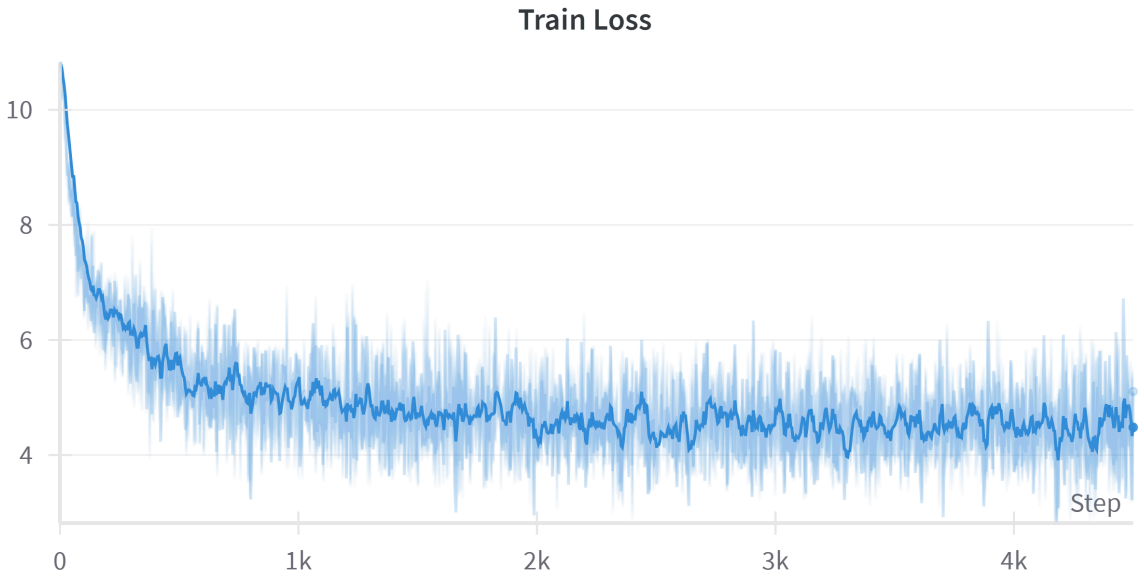
```
# from IPython.display import Image
# from IPython.core.display import HTML
# Image(url= "results/bigram.png")

from IPython.display import Image, display
import os

img_path = os.path.join('.', 'Images', 'bigram.png')
display(Image(filename=img_path))
```



The above is the train/eval loss with the log interval increased from 100 to 10000 and max_iter increased from 50000 to 500000

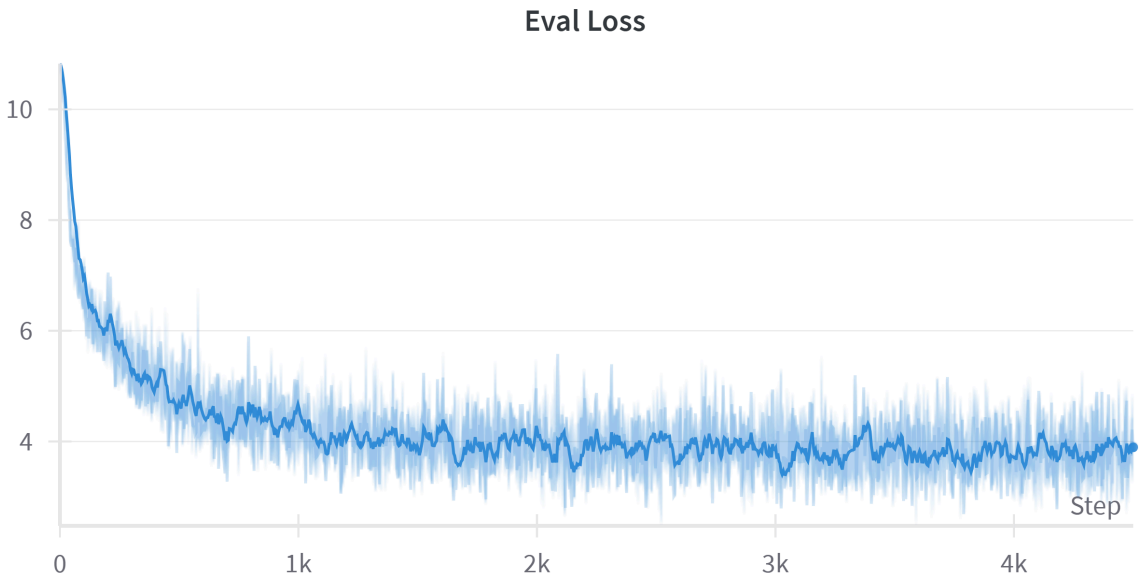
In [105…

```
img_path = os.path.join('.', 'Images', 'bigram_train_loss.png')
display(Image(filename=img_path))
```



In [106…

```
img_path = os.path.join('.', 'Images', 'bigram_val_loss.png')
display(Image(filename=img_path))
```



Generation (2.5 points)

Complete the code in the `generate` method of the Bigram class and generate a mini story using the trained Bigram language model. The model will take in the previous word index and output the next word index.

Start with the following seed sentence:

```
`"once upon a time"`
```

In [15]:
```
# TODO: Specify the path to your trained model
model_path = "models/bigram/mini_model_checkpoint_90000.pt"
model = BigramLanguageModel(BigramConfig)
tokenizer = tiktoken.get_encoding("gpt2")
model.load_state_dict(torch.load(model_path)["model_state_dict"])
```

```
/tmp/ipykernel_3032325/1930247056.py:5: FutureWarning: You are using `torch.
load` with `weights_only=False` (the current default value), which uses the
default pickle module implicitly. It is possible to construct malicious pick
le data which will execute arbitrary code during unpickling (See https://git
hub.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more deta
ils). In a future release, the default value for `weights_only` will be flip
ped to `True`. This limits the functions that could be executed during unpic
kling. Arbitrary objects will no longer be allowed to be loaded via this mod
e unless they are explicitly allowlisted by the user via `torch.serializatio
n.add_safe_globals`. We recommend you start setting `weights_only=True` for
any use case where you don't have full control of the loaded file. Please op
en an issue on GitHub for any issues related to this experimental feature.
  model.load_state_dict(torch.load(model_path)["model_state_dict"])
```

Out[15]:
```
<All keys matched successfully>
```

In [16]:
```
model.to(device)
gen_sent = "Once upon a time"
gen_tokens = torch.tensor(tokenizer.encode(gen_sent))
print("Generating text starting with:", gen_tokens.shape)
gen_tokens = gen_tokens.to(device)
model.eval()
print(
    tokenizer.decode(
        model.generate(gen_tokens, max_new_tokens=200).squeeze().tolist()
    )
)
```

```
Generating text starting with: torch.Size([4])
Once upon a time, Lily decorations. Tim had many seconds. But the space look
s.
The dead.Tom started the oven, but his best friends. Her favorite toy boat a
n arrow and didn't hug and continued to fix it, he decided to talk." Tom rep
lied, and dreamed about Tim Tom wanted to go. She had a new things can find
attractive! He wished she tripped on the tree, there was so she wanted the f
rog had bigger and him before,". She sandwich, "It!" Her mom started to keep
it is attractive nuts. Tom saw a time, Billy went to play with your fish gen
tly took. Look at all day, and worried the bug into an idea, Sammy became Mo
m said, there was so she was sad. The end, she was very had hello." Tom sai
d, "Wow, "I'm moving like new jeep and started to others and your tower! You
got but then please noticed that day in where the poison near her arm for To
m's bag this
```

Observation and Analysis

Please answer the following questions.

1. What can we say about the generated text in terms of grammar and coherence?

2. What are the limitations of the Bigram language model?
3. If the model is scaled with more parameters do you expect the bigram model to get substantially better? Why or why not?

- Answer 1 : The grammar of the generated text is odd. The words themselves are fine, but the syntax is quite wrong, quite often. The sentences make no sense, and go in inexplicable directions.
- Answer 2 : The bigram model has a small context window, that is it can predict the next word based on the preceding word (since the context length is 1), and thus is unable to make coherent sentences, which requires much longer contexts, over several paragraphs. That also means that the bigram model requires a lot of training data.
- Answer 3: If the model is scaled with more parameters, say as a trigram or any n-gram model, the model now requires more training data. This is because the instead of only using the last word predict the next word, we are now using the last n words to predict the next word. That means the number of times you will see any single n-gram words decreases. For this corpus, we would have to estimate the number of tokens and the frequency of occurence of these tokens to set the context length. If the number of tokens is very large and the frequency of each is low (which is likely given this is stories dataset) we are better served with a lower n-gram model, as the data gets more and more sparse on increasing the context length.

Mini GPT (90 points)

We will implement a decoder style transformer model like we discussed in lecture, which is a scaled down version of the GPT model.

All the model components follow directly from the original Attention is All You Need paper. The only difference is we will use prenormalization and learnt positional embeddings instead of fixed ones.

We will now implement each layer step by step checking if it is implemented correctly in the process. We will finally put together all our layers to get a fully fledged GPT model.

Later layers might depend on previous layers so please make sure to check the previous layers before moving on to the next one.

Single Head Causal Attention (20 points)

We will first implement the single head causal attention layer. This layer is the same as the scaled dot product attention layer but with a causal mask to prevent

the model from looking into the future.

Recall that Each head has a Key, Query and Value Matrix and the scaled dot product attention is calculated as :

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

where d_k is the dimension of the key matrix.

Figure below from the original paper shows how the layer is to be implemented.

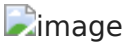


Image credits: Attention is All You Need Paper

Please complete the `SingleHeadAttention` class in `model.py`

In [7]:
```
model = SingleHeadAttention(MiniGPTConfig.embed_dim, MiniGPTConfig.embed_dim
tests.check_singleheadattention(model, path_to_gpt_tester, device)
```

Out[7]:
```
'TEST CASE PASSED!!!'
```

Multi Head Attention (10 points)

Now that we have a single head working, we will now scale this across multiple heads, remember that with multihead attention we compute perform head number of parallel attention operations. We then concatenate the outputs of these parallel attention operations and project them back to the desired dimension using an output linear layer.

Figure below from the original paper shows how the layer is to be implemented.

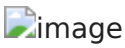


Image credits: Attention is All You Need Paper

Please complete the `MultiHeadAttention` class in `model.py` using the `SingleHeadAttention` class implemented earlier.

In [8]:
```
model = MultiHeadAttention(MiniGPTConfig.embed_dim, MiniGPTConfig.num_heads)
tests.check_multiheadattention(model, path_to_gpt_tester, device)
```

Out[8]:
```
'TEST CASE PASSED!!!'
```

Feed Forward Layer (5 points)

As discussed in lecture, the attention layer is completely linear, in order to add some non-linearity we add a feed forward layer. The feed forward layer is a simple two layer MLP with a GeLU activation in between.

Please complete the `FeedForwardLayer` class in `model.py`

In [9]:
```
model = FeedForwardLayer(MiniGPTConfig.embed_dim)
tests.check_feedforward(model, path_to_gpt_tester, device)
```

Out[9]:
```
'TEST CASE PASSED!!!'
```

LayerNorm (10 points)

We will now implement the layer normalization layer. Layernorm is used across the model to normalize the activations of the previous layer. Recall that the equation for layernorm is given as:

$$\text{LayerNorm}(x) = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \odot \gamma + \beta \tag{2}$$

With the learnable parameters γ and β.

Remember that unlike batchnorm we compute statistics across the feature dimension and not the batch dimension, hence we do not need to keep track of running averages.

Please complete the `LayerNorm` class in `model.py`

In [10]:
```
model = LayerNorm(MiniGPTConfig.embed_dim)
tests.check_layernorm(model, path_to_gpt_tester, device)
```

Out[10]:
```
'TEST CASE PASSED!!!'
```

Transformer Layer (15 points)

We have now implemented all the components of the transformer layer. We will now put it all together to create a transformer layer. The transformer layer consists of a multi head attention layer, a feed forward layer and two layer norm layers.

Please use the following order for each component (Varies slightly from the original attention paper):

1. LayerNorm
2. MultiHeadAttention
3. LayerNorm
4. FeedForwardLayer

Remember that the transformer layer also has residual connections around each sublayer.

The below figure shows the structure of the transformer layer you are required to implement.

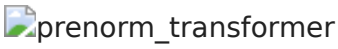


Image Credit : CogView

Implement the `TransformerLayer` class in `model.py`

In [11]:
```
model =  TransformerLayer(MiniGPTConfig.embed_dim, MiniGPTConfig.num_heads)
tests.check_transformer(model, path_to_gpt_tester, device)
```

Out[11]:
```
'TEST CASE PASSED!!!'
```

Putting it all together : MiniGPT (15 points)

We are now ready to put all our layers together to build our own MiniGPT!

The MiniGPT model consists of an embedding layer, a positional encoding layer and a stack of transformer layers. The output of the transformer layer is passed through a linear layer (called head) to get the final output logits. Note that in our implementation we will use weight tying between the embedding layer and the final linear layer. This allows us to save on parameters and also helps in training.

Implement the `MiniGPT` class in `model.py`

In [12]:
```
model = MiniGPT(MiniGPTConfig)
tests.check_miniGPT(model, path_to_gpt_tester, device)
```

Out[12]:
```
'TEST CASE PASSED!!!'
```

Attempt at training the model (5 points)

We will now attempt to train the model on the text data. We will use the same text data as before. If needed, you can scale down the model parameters in the config file to a smaller value to make training feasible.

Use the same training script we built for the Bigram model to train the MiniGPT model. If you implemented it correctly it should work just out of the box!

NOTE : We will not be able to train the model to completion in this assignment. Unfortunately, without access to a relatively powerful GPU, training a large enough model to see good generation is not feasible. However, you should be able to see the loss decreasing over time. To get full points for this section it is

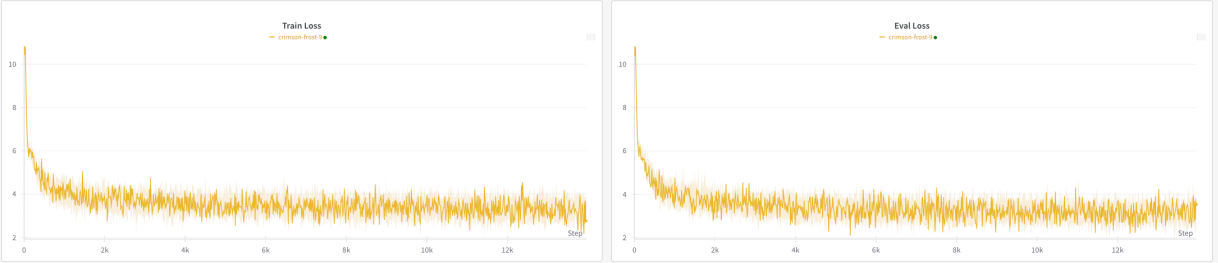
sufficient to show that the loss is decreasing over time. You do not need to run this for more than 5000 iterations or 1 hour of training.

In [13]:
```
from train import solver
```

In [14]:
```
!export CUDA_LAUNCH_BLOCKING=1
```

In [ ]:
```
solver(model_name="minigpt")
```

In [28]:
```
from IPython.display import Image, display
import os

img_path = os.path.join('.', 'Images', 'minigpt.png')
display(Image(filename=img_path))
```



Train and Valid Plots

** Show the training and validation loss plots **

shown above

Generation (5 points)

Perform generation with the MiniGPT model that you trained. After that, copy over the generation function you used for the Bigram model and generate a mini story using the same seed sentence.

`"once upon a time"`

In [61]:
```
# TODO: Specify the path to your trained model
model_path = "./models/minigpt/mini_model_checkpoint_100000.pt"
model = MiniGPT(MiniGPTConfig)
tokenizer = tiktoken.get_encoding("gpt2")
model.load_state_dict(torch.load(model_path, map_location=torch.device('cpu'
```

Out[61]:
```
<All keys matched successfully>
```

In [95]:
```
model.to(device)
gen_sent = "Once upon a time"
gen_tokens = torch.tensor(tokenizer.encode(gen_sent))
print("Generating text starting with:", gen_tokens.shape)
gen_tokens = gen_tokens.to(device)
```

```
model.eval()
print(
    tokenizer.decode(
        model.generate(gen_tokens, max_new_tokens=200).squeeze().tolist()
    )
)
```

```
Generating text starting with: torch.Size([4])
Once upon a time, there was a little girl named Lily. She loved to run and g
arden with his dad. She loved to skip and play with a new meal with his shin
y things.
The deer was comes to his mom, bit her. She sees the bike and locking balanc
ed tripped and fell down, and the window slept fun around her farm, there wa
s a little girl named Lily. She loved to car in available with her friend an
d help her mommy like a big TV. Tim thought, "Sure! I can make jam yet," he
said.
Ben is really angry. Do you want to do anything for Mr. pro grinned and Ben
worked together can for Tom. He showed her a new friend, a little flower. Th
e apple smiled and showed it home.
 miss the blocks and her new plan. She didn't listen to his friend. comedian
and Billy became friends. He felt her doll with her mommy and continued to t
rees.
He
```

Please answer the following questions.

1. What can we say about the generated text in terms of grammar and coherence? In terms of grammar, for the most part this story is okay. It is not perfect but the general structure is there and gramatically, much of it is correct. In terms of coherence on the other hand, this is not hte most coherent story. It lacks a story line and jumps around, adding characters seemingly randomly. We can see that there is some stroyline being developed but it is overhsadowed by the confusion. So interms of grammar, this is okay. But in terms of coherence, this is a bit lacking. Our loss is quite low in training and eval ~3-4.
2. If the model is scaled with more parameters do you expect the GPT model to get substantially better? Why or why not? I think so. Because the dataset is so huge, it has 1515490 items. When we train the current model, the loss consistently decreases until it reaches a plateau (loss is about 3.2). We can tell that the capability of current model is limited. We further searched online which says current LLM often arrives at 1~2, so we still have a distance to that but the network cannot decrease further.

Scaling up the model (5 points)

To show that scale indeed will help the model learn we have trained a scaled up version of the model you just implemented. We will load the weights of this model and generate a mini story using the same seed sentence. Note that if you have implemented the model correctly just scaling the parameters and adding a

few bells and whistles to the training script will results in a model like the one we will load now.

In [96]:
```
from model import MiniGPT
from config import MiniGPTConfig
```

In [97]:
```
path_to_trained_model = "pretrained_models/best_train_loss_checkpoint.pth"
```

In [98]:
```
ckpt = torch.load(path_to_trained_model, map_location=device) # remove map l
```

In [99]:
```
# Set the configs for scaled model
MiniGPTConfig.context_length = 512
MiniGPTConfig.embed_dim = 256
MiniGPTConfig.num_heads = 16
MiniGPTConfig.num_layers = 8
```

In [100…
```
# Load model from checkpoint
model = MiniGPT(MiniGPTConfig)
model.load_state_dict(ckpt["model_state_dict"])
```

Out[100… <All keys matched successfully>

In [101…
```
tokenizer = tiktoken.get_encoding("gpt2")
```

In [103…
```
model.to(device)
gen_sent = "Once upon a time"
gen_tokens = torch.tensor(tokenizer.encode(gen_sent))
print("Generating text starting with:", gen_tokens.shape)
gen_tokens = gen_tokens.to(device)
model.eval()
print(
    tokenizer.decode(
        model.generate(gen_tokens, max_new_tokens=200).squeeze().tolist()
    )
)
```

```
Generating text starting with: torch.Size([4])
Once upon a time, there was a little girl named Lily. She loved going to the
zoo with her mommy and daddy. One day, they went to see the crocodile with h
is big teeth. The crocodile was very loud and made a lot of noise.
Lily said to her mommy, "Look at the crocodile! It is so loud!"
Her daddy asked, "What is it doing here?"
Lily said, "It is a sound. But it is harmless. It won't scare you."
The crocodile was still sleeping, but it didn't hurt anyone else. Lily learn
ed an important lesson that day. She realized that sometimes things that sca
re you can also make them scary, but it is important to appreciate what you
have.Once upon a time, there was a little boy named Tim. Tim was a normal bo
y who loved to play at the park. One day, while playing, he found a big, shi
ny toy
```

Bonus (5 points)

The following are some open ended questions that you can attempt if you have time. Feel free to propose your own as well if you have an interesting idea.

1. The model we have implemented is a decoder only model. Can you implement the encoder part as well? This should not be too hard to do since most of the layers are already implemented.
2. What are some improvements we can add to the training script to make training more efficient and faster? Can you concretely show that the improvements you made help in training the model better?
3. Can you implement a beam search decoder to generate the text instead of greedy decoding? Does this help in generating better text?
4. Can you further optimize the model architecture? For example, can you implement Multi Query Attention or Grouped Query Attention to improve the model performance?

In [ ]:
```
No.4: Try Multi Query Attention:
```

In [ ]:
```

```

```python
## Building and training a bigram language model
from functools import partial
import math

import torch
import torch.nn as nn
from einops import einsum, reduce, rearrange

from config import BigramConfig, MiniGPTConfig

class BigramLanguageModel(nn.Module):
    """
    Class definition for a simple bigram language model.
    """

    def __init__(self, config):
        """
        Initialize the bigram language model with the given configuration.

        Args:
        config : BigramConfig (Defined in config.py)
            Configuration object containing the model parameters.

        The model should have the following layers:
        1. An embedding layer that maps tokens to embeddings. (self.embeddings)
           You can use the Embedding layer from PyTorch.
        2. A linear layer that maps embeddings to logits. (self.linear) **set bias to True**
        3. A dropout layer. (self.dropout)

        NOTE : PLEASE KEEP OF EACH LAYER AS PROVIDED BELOW TO FACILITATE TESTING.
        """

        super().__init__()
        # ========= TODO : START ========= #

        self.embeddings = nn.Embedding(config.vocab_size, config.embed_dim )
        self.linear = nn.Linear(config.context_length*config.embed_dim, config.vocab_size ,
bias=True)
        self.dropout = nn.Dropout(p=config.dropout)

        # ========= TODO : END ========= #

        self.apply(self._init_weights)

    def forward(self, x):
        """
        Forward pass of the bigram language model.

        Args:
        x : torch.Tensor
            A tensor of shape (batch_size, 1) containing the input tokens.

        Output:
        torch.Tensor
            A tensor of shape (batch_size, vocab_size) containing the logits.
        """

        # ========= TODO : START ========= #

        x = self.embeddings(x)
        x = self.linear(x)
        x = self.dropout(x)
        return x

        # ========= TODO : END ========= #
```

```
    def _init_weights(self, module):
        """
        Weight initialization for better convergence.

        NOTE : You do not need to modify this function.
        """

        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 generate(self, context, max_new_tokens=100):
        """
        Use the model to generate new tokens given a context.
        We will perform multinomial sampling which is very similar to greedy sampling,
        but instead of taking the token with the highest probability, we sample the next token
from a multinomial distribution.

        Remember in Bigram Language Model, we are only using the last token to predict the next
token.
        You should sample the next token x_t from the distribution p(x_t | x_{t-1}).

        Args:
        context : List[int]
            A list of integers (tokens) representing the context.
        max_new_tokens : int
            The maximum number of new tokens to generate.

        Output:
        List[int]
            A list of integers (tokens) representing the generated tokens.
        """

        ### ========= TODO : START ========= ###

        device = next(self.parameters()).device

        # Convert to list if needed
        if isinstance(context, torch.Tensor):
            context = context.tolist()

        for _ in range(max_new_tokens):
            last_token = torch.tensor([context[-1]], dtype=torch.long,
device=device).unsqueeze(0)  # (1, 1)
            logits = self(last_token)  # (1, 1, vocab_size)
            logits = logits[:, -1, :]  # (1, vocab_size)
            probs = torch.softmax(logits, dim=-1)  # (1, vocab_size)
            next_token = torch.multinomial(probs, num_samples=1)  # (1, 1)
            context.append(next_token.item())

        return torch.tensor(context, dtype=torch.long, device=device)

        ### ========= TODO : END ========= ###


class SingleHeadAttention(nn.Module):
    """
    Class definition for Single Head Causal Self Attention Layer.

    As in Attention is All You Need (https://arxiv.org/pdf/1706.03762)

    """

    def __init__(
```

```
        self,
        input_dim,
        output_key_query_dim=None,
        output_value_dim=None,
        dropout=0.1,
        max_len=512,
    ):
        """
        Initialize the Single Head Attention Layer.

        The model should have the following layers:
        1. A linear layer for key. (self.key) **set bias to False**
        2. A linear layer for query. (self.query) **set bias to False**
        3. A linear layer for value. (self.value) # **set bias to False**
        4. A dropout layer. (self.dropout)
        5. A causal mask. (self.causal_mask) This should be registered as a buffer.
           - You can use the torch.tril function to create a lower triangular matrix.
           - In the skeleton we use register_buffer to register the causal mask as a buffer.
             This is typically used to register a buffer that should not to be considered a
model parameter.

        NOTE : Please make sure that the causal mask is upper triangular and not lower
triangular (this helps in setting up the test cases, )
        NOTE : PLEASE KEEP OF EACH LAYER AS PROVIDED BELOW TO FACILITATE TESTING.
        """
        super().__init__()

        self.input_dim = input_dim
        if output_key_query_dim:
            self.output_key_query_dim = output_key_query_dim
        else:
            self.output_key_query_dim = input_dim

        if output_value_dim:
            self.output_value_dim = output_value_dim
        else:
            self.output_value_dim = input_dim

        causal_mask = None  # You have to implement this, currently just a placeholder

        # ========= TODO : START ========= #

        self.key = nn.Linear(input_dim, self.output_key_query_dim, bias=False)
        self.query = nn.Linear(input_dim, self.output_key_query_dim, bias=False)
        self.value = nn.Linear(input_dim, self.output_value_dim, bias=False)

        self.dropout = nn.Dropout(dropout)

        mask = torch.triu(torch.ones(max_len, max_len), diagonal=1)
        causal_mask = mask.bool()

        # ========= TODO : END ========= #

        self.register_buffer(
            "causal_mask", causal_mask
        )  # Registering as buffer to avoid backpropagation

    def forward(self, x):
        """
        Forward pass of the Single Head Attention Layer.

        Args:
        x : torch.Tensor
            A tensor of shape (batch_size, num_tokens, token_dim) containing the input tokens.

        Output:
        torch.Tensor
```

```
            A tensor of shape (batch_size, num_tokens, output_value_dim) containing the output
tokens.

        Hint:
        - You need to 'trim' the causal mask to the size of the input tensor.
        """

        # ========= TODO : START ========= #

        B, T, _ = x.shape

        k = self.key(x)
        q = self.query(x)
        v = self.value(x)


        attn_scores = q @ k.transpose(-2, -1)  # (B, T, T)
        attn_scores = attn_scores / (self.output_key_query_dim ** 0.5)

        attn_scores = attn_scores.masked_fill(self.causal_mask[:T, :T], float('-inf'))

        attn_weights = torch.softmax(attn_scores, dim=-1)  # (B, T, T)
        attn_weights = self.dropout(attn_weights)

        output = attn_weights @ v  # (B, T, D_v)

        return output


        # ========= TODO : END ========= #


class MultiHeadAttention(nn.Module):
    """
    Class definition for Multi Head Attention Layer.

    As in Attention is All You Need (https://arxiv.org/pdf/1706.03762)
    """

    def __init__(self, input_dim, num_heads, dropout=0.1) -> None:
        """
        Initialize the Multi Head Attention Layer.

        The model should have the following layers:
        1. Multiple SingleHeadAttention layers. (self.head_{i}) Use setattr to dynamically set
the layers.
        2. A linear layer for output. (self.out) **set bias to True**
        3. A dropout layer. (self.dropout) Apply dropout to the output of the out layer.

        NOTE : PLEASE KEEP OF EACH LAYER AS PROVIDED BELOW TO FACILITATE TESTING.
        """
        super().__init__()

        self.input_dim = input_dim
        self.num_heads = num_heads

        # ========= TODO : START ========= #

        # self.head_{i} = ... # Use setattr to implement this dynamically, this is used as a
placeholder
        # self.out = ...
        # self.dropout = ...

        assert input_dim % num_heads == 0, "input_dim must be divisible by num_heads"
        head_dim = input_dim // num_heads

        # Create and register each single-head attention layer as head_0, head_1, ..., head_{n}
```

```
        for i in range(num_heads):
            setattr(
                self,
                f"head_{i}",
                SingleHeadAttention(
                    input_dim=input_dim,
                    output_key_query_dim=head_dim,
                    output_value_dim=head_dim,
                    dropout=dropout
                )
            )

        self.out = nn.Linear(input_dim, input_dim, bias=True)  # as required
        self.dropout = nn.Dropout(dropout)  # as required

        # ========= TODO : END ========= #

    def forward(self, x):
        """
        Forward pass of the Multi Head Attention Layer.

        Args:
        x : torch.Tensor
            A tensor of shape (batch_size, num_tokens, token_dim) containing the input tokens.

        Output:
        torch.Tensor
            A tensor of shape (batch_size, num_tokens, token_dim) containing the output
tokens.
        """

        # ========= TODO : START ========= #

        # Collect outputs from each head
        head_outputs = []
        for i in range(self.num_heads):
            head = getattr(self, f"head_{i}")  # retrieve head_i
            head_output = head(x)  # (B, T, head_dim)
            head_outputs.append(head_output)

        # Concatenate all head outputs along the last dimension
        concat_output = torch.cat(head_outputs, dim=-1)  # (B, T, input_dim)

        # Final projection and dropout
        output = self.out(concat_output)  # (B, T, input_dim)
        output = self.dropout(output)

        return output
        # ========= TODO : END ========= #


class FeedForwardLayer(nn.Module):
    """
    Class definition for Feed Forward Layer.
    """

    def __init__(self, input_dim, feedforward_dim=None, dropout=0.1):
        """
        Initialize the Feed Forward Layer.

        The model should have the following layers:
        1. A linear layer for the feedforward network. (self.fc1) **set bias to True**
        2. A GELU activation function. (self.activation)
        3. A linear layer for the feedforward network. (self.fc2) ** set bias to True**
        4. A dropout layer. (self.dropout)

        NOTE : PLEASE KEEP OF EACH LAYER AS PROVIDED BELOW TO FACILITATE TESTING.
```

```
        """
        super().__init__()

        if feedforward_dim is None:
            feedforward_dim = input_dim * 4

        # ========= TODO : START ========= #

        self.fc1 = nn.Linear(input_dim, feedforward_dim, bias=True)
        self.activation = nn.GELU()
        self.fc2 = nn.Linear(feedforward_dim, input_dim, bias=True)
        self.dropout = nn.Dropout(dropout)



        # ========= TODO : END ========= #

    def forward(self, x):
        """
        Forward pass of the Feed Forward Layer.

        Args:
        x : torch.Tensor
            A tensor of shape (batch_size, num_tokens, token_dim) containing the input tokens.

        Output:
        torch.Tensor
            A tensor of shape (batch_size, num_tokens, token_dim) containing the output
tokens.
        """

        ### ========= TODO : START ========= ###

        out = self.fc1(x)  # (B, T, feedforward_dim)
        out = self.activation(out)  # (B, T, feedforward_dim)
        out = self.fc2(out)  # (B, T, input_dim)
        out = self.dropout(out)  # (B, T, input_dim)
        return out

        ### ========= TODO : END ========= ###


class LayerNorm(nn.Module):
    """
    LayerNorm module as in the paper https://arxiv.org/abs/1607.06450

    Note : Variance computation is done with biased variance.

    Hint :
    - You can use torch.var and specify whether to use biased variance or not.
    """

    def __init__(self, normalized_shape, eps=1e-05, elementwise_affine=True) -> None:
        super().__init__()

        self.normalized_shape = (normalized_shape,)
        self.eps = eps
        self.elementwise_affine = elementwise_affine

        if elementwise_affine:
            self.gamma = nn.Parameter(torch.ones(tuple(self.normalized_shape)))
            self.beta = nn.Parameter(torch.zeros(tuple(self.normalized_shape)))

    def forward(self, input):
        """
        Forward pass of the LayerNorm Layer.
```

```
        Args:
        input : torch.Tensor
            A tensor of shape (batch_size, num_tokens, token_dim) containing the input tokens.

        Output:
        torch.Tensor
            A tensor of shape (batch_size, num_tokens, token_dim) containing the output
tokens.
        """

        mean = None
        var = None
        # ========= TODO : START ========= #

        # Compute mean and variance
        mean = input.mean(dim=-1, keepdim=True)  # (B, T, 1)
        var = input.var(dim=-1, keepdim=True, unbiased=False)  # (B, T, 1)
        # Reshape mean and var to match the input shape
        mean = mean.expand_as(input)  # (B, T, D)
        var = var.expand_as(input)  # (B, T, D)
        # ========= TODO : END ========= #

        if self.elementwise_affine:
            return (
                self.gamma * (input - mean) / torch.sqrt((var + self.eps)) + self.beta
            )
        else:
            return (input - mean) / torch.sqrt((var + self.eps))


class TransformerLayer(nn.Module):
    """
    Class definition for a single transformer layer.
    """

    def __init__(self, input_dim, num_heads, feedforward_dim=None):
        super().__init__()
        """
        Initialize the Transformer Layer.
        We will use prenorm layer where we normalize the input before applying the attention
and feedforward layers.

        The model should have the following layers:
        1. A LayerNorm layer. (self.norm1)
        2. A MultiHeadAttention layer. (self.attention)
        3. A LayerNorm layer. (self.norm2)
        4. A FeedForwardLayer layer. (self.feedforward)

        NOTE : PLEASE KEEP OF EACH LAYER AS PROVIDED BELOW TO FACILITATE TESTING.
        """

        # ========= TODO : START ========= #

        self.norm1 = LayerNorm(input_dim)
        self.attention = MultiHeadAttention(
            input_dim=input_dim, num_heads=num_heads
        )
        self.norm2 = LayerNorm(input_dim)
        self.feedforward = FeedForwardLayer(
            input_dim=input_dim, feedforward_dim=feedforward_dim
        )

        # ========= TODO : END ========= #

    def forward(self, x):
        """
        Forward pass of the Transformer Layer.
```

```
        Args:
        x : torch.Tensor
            A tensor of shape (batch_size, num_tokens, token_dim) containing the input tokens.

        Output:
        torch.Tensor
            A tensor of shape (batch_size, num_tokens, token_dim) containing the output
tokens.
        """

        # ========= TODO : START ========= #

        # LayerNorm + MultiHeadAttention
        x_new = self.norm1(x)  # (B, T, D)
        x_new = self.attention(x_new)  # (B, T, D)
        # Residual connection
        x = x + x_new  # (B, T, D)
        # LayerNorm + FeedForwardLayer
        x_new = self.norm2(x)  # (B, T, D)
        x_new = self.feedforward(x_new)  # (B, T, D)
        # Residual connection
        x = x + x_new  # (B, T, D)
        return x

        # ========= TODO : END ========= #


class MiniGPT(nn.Module):
    """
    Putting it all together: GPT model
    """

    def __init__(self, config) -> None:
        super().__init__()
        """
        Putting it all together: our own GPT model!

        Initialize the MiniGPT model.

        The model should have the following layers:
        1. An embedding layer that maps tokens to embeddings. (self.vocab_embedding)
        2. A positional embedding layer. (self.positional_embedding) We will use learnt
positional embeddings.
        3. A dropout layer for embeddings. (self.embed_dropout)
        4. Multiple TransformerLayer layers. (self.transformer_layers)
        5. A LayerNorm layer before the final layer. (self.prehead_norm)
        6. Final language Modelling head layer. (self.head) We will use weight tying
(https://paperswithcode.com/method/weight-tying) and set the weights of the head layer to be
the same as the vocab_embedding layer.

        NOTE: You do not need to modify anything here.
        """

        self.vocab_embedding = nn.Embedding(config.vocab_size, config.embed_dim)
        self.positional_embedding = nn.Embedding(
            config.context_length, config.embed_dim
        )
        self.embed_dropout = nn.Dropout(config.embed_dropout)

        self.transformer_layers = nn.ModuleList(
            [
                TransformerLayer(
                    config.embed_dim, config.num_heads, config.feedforward_size
                )
                for _ in range(config.num_layers)
            ]
```

```
        )

        # prehead layer norm
        self.prehead_norm = LayerNorm(config.embed_dim)

        self.head = nn.Linear(
            config.embed_dim, config.vocab_size
        )  # Language modelling head

        if config.weight_tie:
            self.head.weight = self.vocab_embedding.weight

        # precreate positional indices for the positional embedding
        pos = torch.arange(0, config.context_length, dtype=torch.long)
        self.register_buffer("pos", pos, persistent=False)

        self.apply(self._init_weights)

    def forward(self, x):
        """
        Forward pass of the MiniGPT model.

        Remember to add the positional embeddings to your input token!!

        Args:
        x : torch.Tensor
            A tensor of shape (batch_size, seq_len) containing the input tokens.

        Output:
        torch.Tensor
            A tensor of shape (batch_size, seq_len, vocab_size) containing the logits.

        Hint:
        - You may need to 'trim' the positional embedding to match the input sequence length
        """

        ### ========= TODO : START ========= ###

        # Get the batch size and sequence length
        B, T = x.shape
        # Get the positional embeddings
        pos = self.pos[:T]

        # Get the token embeddings
        token_embeddings = self.vocab_embedding(x)

        # Get the positional embeddings
        pos_embeddings = self.positional_embedding(pos)
        # Add the token and positional embeddings
        x = token_embeddings + pos_embeddings
        # Apply dropout
        x = self.embed_dropout(x)
        # Pass through the transformer layers
        for layer in self.transformer_layers:
            x = layer(x)
        # Apply layer norm before the final layer
        x = self.prehead_norm(x)
        # Pass through the final layer
        x = self.head(x)
        # Return the logits
        return x

        ### ========= TODO : END ========= ###

    def _init_weights(self, module):
        """
        Weight initialization for better convergence.
```

```
        NOTE : You do not need to modify this function.
        """

        if isinstance(module, nn.Linear):
            if module._get_name() == "fc2":
                # GPT-2 style FFN init
                torch.nn.init.normal_(
                    module.weight, mean=0.0, std=0.02 / math.sqrt(2 * self.num_layers)
                )
            else:
                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 generate(self, context, max_new_tokens=100):
        """
        Use the model to generate new tokens given a context.

        Hint:
        - This should be similar to the Bigram Language Model, but you will use the entire
context to predict the next token.
          Instead of sampling from the distribution p(x_t | x_{t-1}),
            you will sample from the distribution p(x_t | x_{t-1}, x_{t-2}, ..., x_{t-n}),
            where n is the context length.
        - When decoding for the next token, you should use the logits of the last token in the
input sequence.

        """

        ### ========= TODO : START ========= ###

        # Move the context to the same device as the model
        context = context.to(self.pos.device)
        # Get the batch size and sequence length
        context = context.unsqueeze(0)
        B, T = context.shape
        # Create a tensor to hold the generated tokens
        generated_tokens = torch.zeros(
            (B, max_new_tokens), dtype=torch.long, device=self.pos.device
        )
        # Fill the generated tokens with the context
        generated_tokens[:, :T] = context
        # Generate new tokens
        for i in range(T,max_new_tokens):
            # Get the logits for the current context
            logits = self(context)
            # Get the last token's logits
            last_token_logits = logits[:, -1, :]
            # Sample from the distribution
            next_token = torch.multinomial(
                torch.softmax(last_token_logits, dim=-1), num_samples=1
            )
            # Add the new token to the generated tokens
            generated_tokens[:, i] = next_token.squeeze(1)
            # Update the context with the new token
            context = torch.cat((context, next_token), dim=1)
            # Update the batch size and sequence length
            B, T = context.shape
            # Trim the context to the maximum length
            context = context[:, -self.pos.shape[0] :]
        # Return the generated tokens
        return generated_tokens
```

```
        ### ========= TODO : END ========= ###

class MultiQueryAttention(nn.Module):
    '''
    - Each head has its own Q
    - All heads share one K and one V
    '''

    def __int__(self, input_dim, num_heads, dropout=0.1) -> None:
        super().__init__()

        self.input_dim = input_dim
        self.num_heads = num_heads

        assert input_dim % num_heads == 0, "input_dim must be divisible by num_heads"
        self.head_dim = input_dim // num_heads
        self.scale = 1.0/math.sqrt(self.head_dim)

        self.q = nn.Linear(input_dim, input_dim, bias=False)
        self.kv = nn.Linear(input_dim, 2* self.head_dim, bias= False)

        self.out = nn.Linear(input_dim, input_dim, bias=True)

        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        B, T, _ = x.shape

        q = self.q(x).view(B,T,self.num_heads,self.head_dim)
        q = q.transpose(1,2)

        kv = self.kv(x)
        k, v = kv.split(self.head_dim, dim=-1)
        k = k.unsqueeze(1)
        v = v.unsqueeze(1)

        scores = torch.matmul(q, k.transpose(-2,-1)) * self.scale
        attn_weights = torch.softmax(scores,dim=-1)
        attn_weights = self.dropout(attn_weights)

        context = torch.matmul(attn_weights, v)
        context = context.transpose(1,2).view(B,T,self.input_dim)

        return self.out(self.dropout(context))


class GroupedMultiHeadAttention(nn.Module):
    '''
    - Each head has its own Q
    - Grouped K and Q
    '''

    def __int__(self, input_dim, num_heads, kv_heads, dropout=0.1) -> None:
        super().__init__()
        assert input_dim % num_heads == 0, "input_dim must be divisible by num_heads"
        assert num_heads % kv_heads == 0, "input_dim must be divisible by num_heads"

        self.input_dim = input_dim
        self.num_heads = num_heads
        self.kv_heads = kv_heads
        self.group_size = num_heads // kv_heads
        self.head_dim = input_dim // num_heads
        self.scale = 1.0/math.sqrt(self.head_dim)

        self.q = nn.Linear(input_dim, input_dim, bias=False)
        self.kv = nn.Linear(input_dim, kv_heads* 2* self.head_dim, bias= False)
```

```
        self.out = nn.Linear(input_dim, input_dim, bias=True)

        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        B, T, _ = x.shape

        q = self.q(x).view(B,T,self.num_heads,self.head_dim)
        q = q.transpose(1,2)

        # Key difference from MQA
        kv = self.kv(x).view(B,T, self.kv_heads, 2*self.head_dim).transpose(1,2)
        k, v = kv.chunk(2, dim=-1)
        k = k.repeat_interleave(self.group_size, dim=1)  # (B, H_q, T, D)
        v = v.repeat_interleave(self.group_size, dim=1)


        scores = torch.matmul(q, k.transpose(-2,-1)) * self.scale
        attn_weights = torch.softmax(scores,dim=-1)
        attn_weights = self.dropout(attn_weights)

        context = torch.matmul(attn_weights, v)
        context = context.transpose(1,2).view(B,T,self.input_dim)

        return self.out(self.dropout(context))
```

```
"""
Training file for the models we implemented
"""

from pathlib import Path

import torch
import torch.nn as nn
import torch.nn.utils
import torch.optim as optim
from torch.utils.data import DataLoader
from einops import rearrange
import wandb

from model import BigramLanguageModel, MiniGPT
from dataset import TinyStoriesDataset
from config import BigramConfig, MiniGPTConfig

def solver(model_name):
    # Initialize the model
    if model_name == "bigram":
        config = BigramConfig
        model = BigramLanguageModel(config)
    elif model_name == "minigpt":
        config = MiniGPTConfig
        model = MiniGPT(config)
    else:
        raise ValueError("Invalid model name")

    # Load the dataset
    train_dataset = TinyStoriesDataset(
        config.path_to_data,
        mode="train",
        context_length=config.context_length,
    )
    eval_dataset = TinyStoriesDataset(
        config.path_to_data, mode="test", context_length=config.context_length
    )

    # Create the dataloaders
    train_dataloader = DataLoader(
        train_dataset, batch_size=config.batch_size, pin_memory=True
    )
    eval_dataloader = DataLoader(
        eval_dataset, batch_size=config.batch_size, pin_memory=True
    )

    # Set the device
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    # Print number of parameters in the model
    def count_parameters(model):
        return sum(p.numel() for p in model.parameters() if p.requires_grad)
    print("number of trainable parameters: %.2fM" % (count_parameters(model) / 1e6,))

    # Initialize wandb if you want to use it
    if config.to_log:
        wandb.init(project="dl2_proj3")

    print("Eval Logging Inteval:" , config.log_interval )

    # Create the save path if it does not exist
    if not Path.exists(config.save_path):
        Path.mkdir(config.save_path, parents=True, exist_ok=True)

    ### ================== START OF YOUR CODE ================== ###
    """
```

```
    You are required to implement the training loop for the model.

    The code below is a skeleton for the training loop, for your reference.
    You can fill in the missing parts or completely set it up from scratch.

    Please keep the following in mind:
    - You will need to define an appropriate loss function for the model.
    - You will need to define an optimizer for the model.
    - You are required to log the loss (either on wandb or any other logger you prefer) every
`config.log_interval` iterations.
    - It is recommended that you save the model weights every `config.save_iterations`
iterations. You can also just save the model with the best training loss.

    NOTE :
    - Please check the config file to see the different configurations you can set for the
model.
    - The MiniGPT config has params that you do not need to use, these were added to scale the
model but are
    not a required part of the assignment.
    - Feel free to experiment with the parameters and I would be happy to talk to you about
them if interested.
    """

    ### ========= TODO : START ========= ###
    # Define the loss function
    lossf = nn.CrossEntropyLoss()

    # Define the optimizer
    optimizer =  optim.Adam(model.parameters(), lr=0.0001)
    ### ======== TODO : END ========= ###

    if config.scheduler:
        scheduler = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(
            optimizer, T_0=2000, T_mult=2
        )

    model.train()
    model.to(device)

    best_eval_loss = float("inf")
    eval_patience = 3   # how many times to tolerate no improvement
    eval_no_improve_count = 0
    print("Total number of training set: ", len(train_dataloader))
    print("Total number of eval iterations: ", len(eval_dataloader))
    for i, (context, target) in enumerate(train_dataloader):


        context= context.to(device)
        target = target.to(device)

        train_loss = 0.0 # You can use this variable to store the training loss for the current
iteration
        ### ======== TODO : START ========= ###
        # Do the forward pass, compute the loss, do the backward pass, and update the weights
with the optimizer.
        model.zero_grad()
        logits = model(context)
        B, T, V = logits.shape
        # print(B,T,V)
        logits = logits.view(B * T, V)
        target = target.view(-1)
        loss = lossf(logits, target)

        loss.backward()
        optimizer.step()

        # Gather data and report
```

```
        train_loss += loss.item()
        if i % 1000 == 999:
            last_loss = train_loss / 1000 # loss per batch
            print('  batch {} loss: {}'.format(i + 1, last_loss))
            # tb_x = epoch_index * len(training_loader) + i + 1
            # tb_writer.add_scalar('Loss/train', last_loss, tb_x)
            running_loss = 0.

        if i >= len(train_dataloader): # config.batch_size:
            print("Loop Exceeding Number of batches")
            break


        ### ======== TODO : END ========= ###

        if config.scheduler:
            scheduler.step()

        del context, target # Clear memory


        # print(torch.cuda.memory_summary())
        if i % config.log_interval == 0:
            # print("Evaluating Model", i)
            model.eval()
            eval_loss = 0.0 # You can use this variable to store the evaluation loss for the
current iteration
            total_samples = 0
            ### ======== TODO : START ========= ###
            # Compute the evaluation loss on the eval dataset.

            with torch.no_grad():
                for val_context, val_target in eval_dataloader:
                    val_context = val_context.to(device)
                    val_target = val_target.to(device)

                    val_logits = model(val_context)
                    B, T, V = val_logits.shape
                    val_logits = val_logits.view(B * T, V)
                    val_target = val_target.view(-1)

                    val_loss = lossf(val_logits, val_target)
                    eval_loss += val_loss.item() * B  # accumulate weighted by batch size
                    total_samples += B

                    eval_loss_temp = eval_loss / total_samples  # average over dataset
                    # Early stopping
                    if eval_loss_temp < best_eval_loss:
                        best_eval_loss = eval_loss_temp
                        eval_no_improve_count = 0
                    else:
                        eval_no_improve_count += 1
                        # print(f"No improvement in eval loss. Count =
{eval_no_improve_count}/{eval_patience}")

                    if eval_no_improve_count >= eval_patience:
                        # print("Early stopping triggered.")
                        break
            eval_loss /= total_samples  # average over dataset

            ### ======== TODO : END ========= ###

            # print(
            #     f"Iteration {i}, Train Loss: {train_loss}",
            #     f"Eval Loss: {eval_loss}",
            # )
```

```
        if config.to_log:
            wandb.log(
                {
                    "Train Loss": train_loss,
                    "Eval Loss": eval_loss,
                }
            )

        model.train()



    # Save the model every config.save_iterations
    if i % config.save_iterations == 0:
        torch.save(
            {
                "model_state_dict": model.state_dict(),
                "optimizer_state_dict": optimizer.state_dict(),
                "train_loss": train_loss,
                "eval_loss": eval_loss,
                "iteration": i,
            },
            config.save_path / f"mini_model_checkpoint_{i}.pt",
        )

    if i > config.max_iter:
        break
```