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 [3]:
```
# !pip install numpy torch tiktoken wandb einops # Install all required pack
```

In [4]:
```
%load_ext autoreload
%autoreload 2
```

In [5]:
```
import torch
import tiktoken
```

In [6]:
```
from model import BigramLanguageModel, SingleHeadAttention, MultiHeadAttenti
from config import BigramConfig, MiniGPTConfig
import tests
```

In [7]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "mps" if torc
print(f"Device: {device}")
```

```
Device: mps
```

In [8]:
```
# 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 [9]:
```
# Test implementation for Bigram Language Model
from model import BigramLanguageModel, BigramConfig
model = BigramLanguageModel(BigramConfig)
tests.check_bigram(model, path_to_bigram_tester, device)
```

Out[9]:
```
'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.

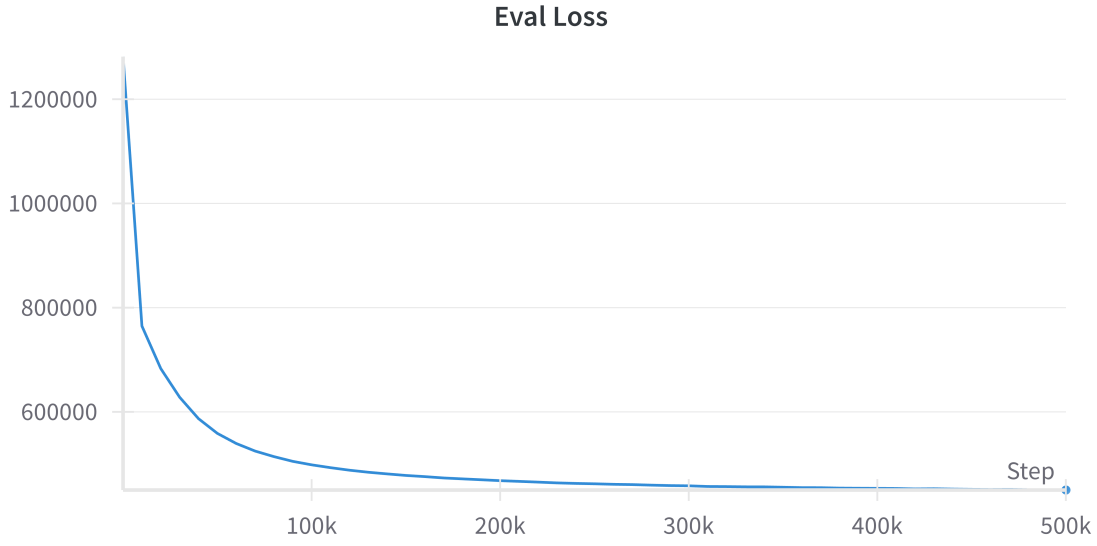
4. Please log the loss curves either on wandb, tensorboard or any other logger of your choice and please attach them below.

In [10]:
```
from train import solver
```

In [11]:
```
# solver(model_name="bigram")
```

Train and Valid Plots

NOTE: the eval plot's y-axis has the wrong scale: I'd forgotten to average my loss. But the shape of the plot is the same as obtained with the averages loss.



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 [12]:
```
# TODO: Specify the path to your trained model
model_path = "/Users/paultalma/Documents/UCLA/Work/Classes/2024-2025/ee_239/
model = BigramLanguageModel(BigramConfig)
tokenizer = tiktoken.get_encoding("gpt2")
model.load_state_dict(torch.load(model_path, map_location=torch.device('mps'
```

Out[12]:
```
<All keys matched successfully>
```

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

```
Generating text starting with: torch.Size([4])
Once upon a time, three breaking flew through a big wave that day. They both
took a little bird to build sandessie.
As they were so it even over because he wanted to skip and went to nap in th
e floor hay!"  And from the wand who lived happily still lost stuck back to
a fun on, nodded and tomorrow!"
The bunny was playing and tell him. Fred. He was just if like it to their sp
oil, Tim looked for him house! He told him." The flowers things.
Tom and feel your bathroom with trying to you help you see the seat the trea
sure with the juice down, wanted to build the fan. One day on a good day, "I
else. They went to help grab the ingredients!" Sarah was happy why before Be
nny would help." But then sheadow and Timmy that he had a voice said, we do
n't size. The label and play in the noise. One day attention toIn bright bad
and barking and landed on his ball.
```

Observation and Analysis

Please answer the following questions.

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

The text is largely incoherent and ungrammatical. Syntax and semantics just aren't there.

2. What are the limitations of the Bigram language model?

The Bigram model can only encode one-step statistical dependencies between words. But linguistic structure depends on more complex longer-range dependencies ("John

borrowed Anna's book. He gave it back to her later" — the grammatical gender of "her" depends on the content of the first sentence.)

3. If the model is scaled with more parameters do you expect the bigram model to get substantially better? Why or why not?

For the reasons given in part 2, I do not expect the model to get better with more parameters. More parameters would allow better modeling of one-step dependencies. But no matter how good your modeling of one-step dependencies is, you will not be able to recover even moderately interesting linguistic structure.

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

Image credits: Attention is All You Need Paper

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

In [29]:
```
model = SingleHeadAttention(MiniGPTConfig.embed_dim, MiniGPTConfig.embed_dim
torch.set_printoptions(precision=2, sci_mode=False)
tests.check_singleheadattention(model, path_to_gpt_tester, device)
```

Out[29]:
```
'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

Image credits: Attention is All You Need Paper

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

In [30]:
```
model = MultiHeadAttention(MiniGPTConfig.embed_dim, MiniGPTConfig.num_heads)
tests.check_multiheadattention(model, path_to_gpt_tester, device)
```

Out[30]:
```
'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 [31]:
```
model = FeedForwardLayer(MiniGPTConfig.embed_dim)
tests.check_feedforward(model, path_to_gpt_tester, device)
```

Out[31]:
```
'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 [32]:
```
model = LayerNorm(MiniGPTConfig.embed_dim)
tests.check_layernorm(model, path_to_gpt_tester, device)
```

Out[32]:
```
'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.

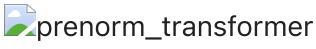


Image Credit : CogView

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

In [33]:
```
model =  TransformerLayer(MiniGPTConfig.embed_dim, MiniGPTConfig.num_heads)
tests.check_transformer(model, path_to_gpt_tester, device)
```

Out[33]:
```
'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 [34]:
```
config = MiniGPTConfig(
    to_log=False,
    save_iterations=100_000
)
model = MiniGPT(config)
tests.check_miniGPT(model, path_to_gpt_tester, device)
```

Out[34]:
```
'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.

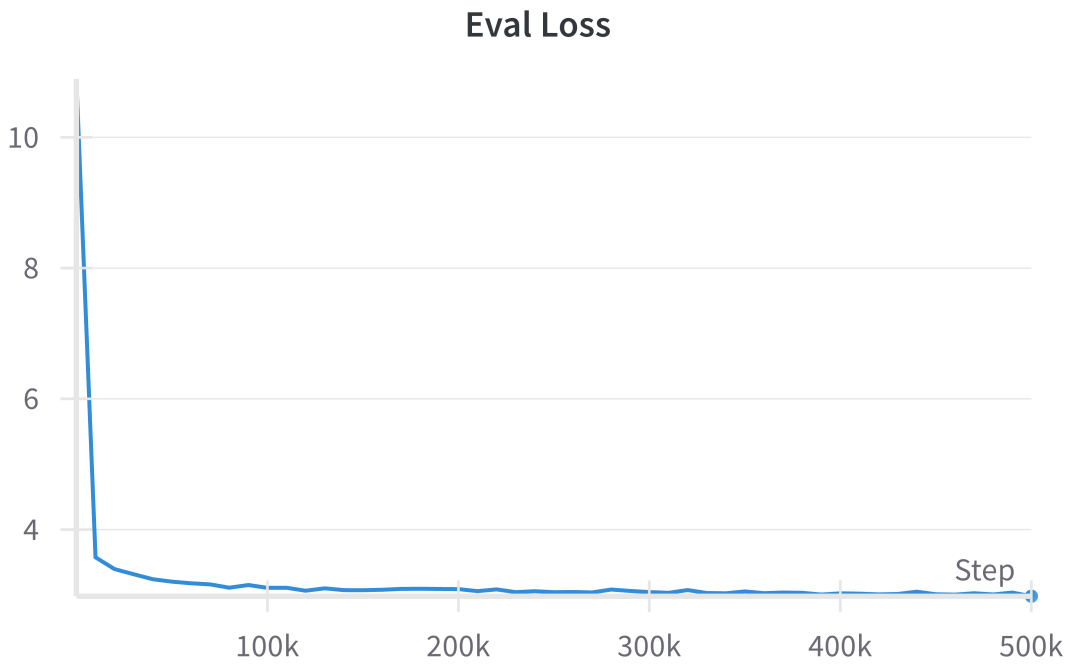
Use the same training script we built for the Bigram model to train the MiniGPT model. If you implemented it correctly it should work just out of the box!

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 [35]:
```
from train import solver
```

In [36]:
```
# solver(model_name="minigpt")
```

Train and Valid Plots



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 [39]:
```
# TODO: Specify the path to your trained model
model_path = "/Users/paultalma/Documents/UCLA/Work/Classes/2024-2025/ee_239/
model = MiniGPT(MiniGPTConfig)
tokenizer = tiktoken.get_encoding("gpt2")
model.load_state_dict(torch.load(model_path, map_location=torch.device('mps'
```

Out[39]:
```
<All keys matched successfully>
```

In [50]:
```
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 wear her
favorite toys, especially if she would talk about her Bow.
But she gave her a big hug. She spilled and gave it his dad had a big luggag
e that’s not way to eat it again and come out our store."
Jenny remembered the king who had eaten on a slide. But one day, he heard a
voice when he heard a loud noise and came through the sky. It was a beautifu
l butterfly from Timmy's mom decided he was watching them in the makes Red g
o find a nap. He might come out the cage. He is not good.
The fox wanted to see anyone anywhere. He wanted to beapped. She got angry a
nd scared.
The wealthy man smiled and said, "Hello, little boy! I'm deaf, why I can hel
p you you," said Tim!
Lily didn't mean and flew back into the sun.
From that day on,
```

Please answer the following questions.

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

This text is much more coherent and grammatical than that produced by the Bigram model. However, it still suffers from inconsistencies and ungrammaticalities (e.g. "She spilled and gave it his dad had a big luggage...").

2. If the model is scaled with more parameters do you expect the GPT model to get substantially better? Why or why not?

We do expect the model to get better with scaling. Unlike in the case of the Bigram model, the GPT model is modeling complex dependencies between parts of the context, and a greater capacity to model these dependencies should lead to better performance. A larger context length should also help with coherence.

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 [51]:
```
from model import MiniGPT
from config import MiniGPTConfig
```

In [52]:
```
path_to_trained_model = "pretrained_models/best_train_loss_checkpoint.pth"
```

In [53]:
```
ckpt = torch.load(path_to_trained_model, map_location=device) # remove map l
```

In [ ]:
```
# Set the configs for scaled model
scaled_config = MiniGPTConfig(
    context_length=512,
    embed_dim=256,
    num_heads=16,
    num_layers=8
)
```

In [55]:
```
# Load model from checkpoint
model = MiniGPT(scaled_config)
model.load_state_dict(ckpt["model_state_dict"])
```

Out[55]:
```
<All keys matched successfully>
```

In [56]:
```
tokenizer = tiktoken.get_encoding("gpt2")
```

In [57]:
```
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 play outs
ide, especially in the park with her friends. One day, while they were playi
ng, Lily saw a bird with a broken wing. It was not working well, but Lily kn
ew they had to help the bird.
Lily and her friends decided to take care of the bird. They gave it some wat
er and water and started to clean the broken wing, but it didn't go away. Af
ter a while, the bird started to grow and grow until it could fly again. Lil
y and her friends were so happy to see the bird fly away.
Later, they all realized that the bird had died from its cage. It had been l
ooking for something was broken. Lily felt sad and sad. She knew that her fr
iends would always be there to help her.Once upon a time, there was a little
girl named Lily. She had a favorite mug that she loved very much. One day, L
ily's dad told her not to
```

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?

Encoder

We implement the encoder and the full model. This required modifying the attention layers to support cross-attention, combining them into an encoder, and combining the encoder and the decoder. The relevant code is in the `model.py` file.

projects/project_3/code/model.py

```
## Building and training a bigram language model
import math


import torch
import torch.nn as nn

from config import BigramConfig

device = torch.device(
    "cuda"
    if torch.cuda.is_available()
    else "mps"
    if torch.mps.is_available()
    else "cpu"
)


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

    def __init__(self, config: BigramConfig):
        """
        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.config = config
        self.embeddings = nn.Embedding(self.config.vocab_size, self.config.embed_dim)
        self.linear = nn.Linear(
            self.config.context_length * self.config.embed_dim,
            self.config.vocab_size,
            bias=True,
        )
        self.dropout = nn.Dropout(p=self.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 ========= #

        embed = self.embeddings(x).squeeze(1)  # (batch_size, embed_dim)
        out = self.linear(embed)  # (batch_size, vocab_size)
        out = self.dropout(out)  # (batch_size, vocab_size)

        return out

        # ========= 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 ========= ###

        f = torch.softmax
        context = torch.tensor(context, device=device)
        current_token = context[-1]
        for _ in range(max_new_tokens):
            logits = self.forward(
                torch.tensor([current_token], device=device)
            ).squeeze()
            probabilities = f(logits, dim=0)
            current_token = torch.multinomial(probabilities, 1).to(device)
            context = torch.cat((context, current_token), dim=0)

        return context
        ### ========= 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(
            self.input_dim, self.output_key_query_dim, bias=False, device=device
        )
        self.query = nn.Linear(
            self.input_dim, self.output_key_query_dim, bias=False, device=device
        )
        self.value = nn.Linear(
            self.input_dim, self.output_value_dim, bias=False, device=device
        )

        self.dropout = nn.Dropout(p=dropout)

        causal_mask = torch.ones((max_len, max_len), device=device)
        causal_mask *= -float("inf")
        causal_mask = torch.triu(causal_mask, 1)

        # ========= 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 ========= #

        _, num_tokens, _ = x.shape
        dk = self.output_key_query_dim
        K = self.key(x)  # (B, T, Dk)
        Q = self.query(x)  # (B, T, Dk)
        V = self.value(x)  # (B, T, Dv)

        attention_scores = Q @ K.transpose(-2, -1)  # (B, T, T)
        attention_scores /= math.sqrt(dk)  # (B, T, T)

        mask = self.causal_mask[:num_tokens, :num_tokens]  # (T, T)
        mask = mask.unsqueeze(0)  # not sure this is necessary # (1, T, T)
        mask = mask.type(torch.bool)

        mask = torch.where(mask, -torch.inf, 0)

        attention_scores += mask  # (B, T, T)

        attention_weights = torch.softmax(attention_scores, dim=-1)  # (B, T, T)
        attention_weights = self.dropout(attention_weights)  # (B, T, T)

        out = attention_weights @ V  # (B, T, Dv)
        return out
        # ========= 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
        self.head_dim = input_dim // num_heads

        self.heads = []
        for i in range(num_heads):
            head = SingleHeadAttention(
                input_dim=input_dim,
                output_key_query_dim=self.head_dim,
                output_value_dim=self.head_dim,
            )
            setattr(self, f"head_{i}", head)
            self.heads.append(head)

        self.out = nn.Linear(self.input_dim, self.input_dim)
        self.dropout = nn.Dropout(p=dropout)

        # ========= 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 ========= #

        head_outputs = []
        for head in self.heads:
            head_outputs.append(head(x))

        out = self.out(torch.cat(head_outputs, dim=-1))
        out = self.dropout(out)
        return out

        # ========= 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)
        out = self.activation(out)
        out = self.fc2(out)
        out = self.dropout(out)
        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 ========= #

        mean = torch.mean(input, dim=2, keepdim=True)
        var = torch.var(input, dim=2, keepdim=True, correction=0)

        # ========= 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(normalized_shape=input_dim)
        self.attention = MultiHeadAttention(input_dim=input_dim, num_heads=num_heads)
        self.norm2 = LayerNorm(normalized_shape=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 ========= #

        intermediate = self.norm1(x)
        intermediate = self.attention(intermediate)
        intermediate += x
```

```
        out = self.norm2(intermediate)
        out = self.feedforward(out)
        out += intermediate

        return out

        # ========= 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.context_length = config.context_length
        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 ========= ###

        B, T = x.shape

        # embeddings
        token_embeds = self.vocab_embedding(x)
        position_embeds = self.positional_embedding(self.pos[:T])
        position_embeds = position_embeds.unsqueeze(dim=0)
        x = token_embeds + position_embeds
        x = self.embed_dropout(x)

        # attention layers
        for layer in self.transformer_layers:
            x = layer(x)

        # language modeling head
        x = self.prehead_norm(x)
        logits = self.head(x)

        return logits
```

```
        ### ========= 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 ========= ###

        f = torch.softmax
        context = torch.tensor(context, device=device).unsqueeze(0)
        for i in range(max_new_tokens):
            current_context = context[:, -(self.context_length) :]
            logits = self(current_context)
            next_token_logits = logits[:, -1, :]
            next_token_probabilities = f(next_token_logits, dim=-1)
            current_token = torch.multinomial(
                input=next_token_probabilities, num_samples=1
            ).to(device)
            context = torch.cat((context, current_token), dim=1)

        return context
```

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


class SingleHeadGeneralAttention(nn.Module):
    def __init__(
        self,
        input_dim,
        output_key_query_dim=None,
        output_value_dim=None,
        dropout=0.1,
        max_len=512,
    ):
        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

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

        self.dropout = nn.Dropout(p=dropout)

        causal_mask = torch.ones((max_len, max_len), device=device)
        causal_mask *= -float("inf")
        causal_mask = torch.triu(causal_mask, 1)

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

    def forward(self, x_query, x_key, x_value):
        _, num_tokens, _ = x_query.shape
```

```
        dk = self.output_key_query_dim
        K = self.key(x_key)  # (B, T, Dk)
        Q = self.query(x_query)  # (B, T, Dk)
        V = self.value(x_value)  # (B, T, Dv)

        attention_scores = Q @ K.transpose(-2, -1)  # (B, T, T)
        attention_scores /= math.sqrt(dk)  # (B, T, T)

        mask = self.causal_mask[:num_tokens, :num_tokens]  # (T, T)
        mask = mask.unsqueeze(0)
        mask = mask.type(torch.bool)
        mask = torch.where(mask, -torch.inf, 0)

        attention_scores += mask  # (B, T, T)

        attention_weights = torch.softmax(attention_scores, dim=-1)  # (B, T, T)
        attention_weights = self.dropout(attention_weights)  # (B, T, T)

        out = attention_weights @ V  # (B, T, Dv)
        return out


class MultiHeadGeneralAttention(nn.Module):
    def __init__(self, input_dim, num_heads, dropout=0.1) -> None:
        super().__init__()

        self.input_dim = input_dim
        self.num_heads = num_heads
        self.head_dim = input_dim // num_heads

        self.heads = []
        for i in range(num_heads):
            head = SingleHeadGeneralAttention(
                input_dim=input_dim,
                output_key_query_dim=self.head_dim,
                output_value_dim=self.head_dim,
            )
            setattr(self, f"head_{i}", head)
            self.heads.append(head)

        self.out = nn.Linear(self.input_dim, self.input_dim)
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, x_query, x_key, x_value):
        head_outputs = []
        for head in self.heads:
            head_outputs.append(head(x_query, x_key, x_value))

        out = self.out(torch.cat(head_outputs, dim=-1))
        out = self.dropout(out)
```

```
        return out


class GeneralTransformerLayer(nn.Module):
    def __init__(self, input_dim, num_heads, feedforward_dim=None):
        super().__init__()

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

    def forward(self, x):
        intermediate = self.norm1(x)
        intermediate = self.attention(intermediate)
        intermediate += x
        out = self.norm2(intermediate)
        out = self.feedforward(out)
        out += intermediate

        return out


class Encoder(nn.Module):
    def __init__(self, config) -> None:
        super().__init__()

        self.context_length = config.context_length
        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)
            ]
        )

        # 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):
        B, T = x.shape

        # embeddings
        token_embeds = self.vocab_embedding(x)
        position_embeds = self.positional_embedding(self.pos[:T])
        position_embeds = position_embeds.unsqueeze(dim=0)
        x = token_embeds + position_embeds
        x = self.embed_dropout(x)

        # attention layers
        for layer in self.transformer_layers:
            x = layer(x)

        return x


class Decoder(nn.Module):
    def __init__(self, config) -> None:
        super().__init__()

        self.context_length = config.context_length
        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.decoder_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, encoder_output):
        B, T = x.shape

        # embeddings
        token_embeds = self.vocab_embedding(x)
        position_embeds = self.positional_embedding(self.pos[:T])
        position_embeds = position_embeds.unsqueeze(dim=0)
        x = token_embeds + position_embeds
        x = self.embed_dropout(x)

        # attention layers
        for layer in self.decoder_layers:
            x = layer(x_query=x, x_key=encoder_output, x_value=encoder_output)

        # language modeling head
        x = self.prehead_norm(x)
        logits = self.head(x)

        return logits

    def _init_weights(self, module):
        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):
        f = torch.softmax
        context = torch.tensor(context, device=device).unsqueeze(0)
        for i in range(max_new_tokens):
            current_context = context[:, -(self.context_length) :]
            logits = self(current_context)
            next_token_logits = logits[:, -1, :]
            next_token_probabilities = f(next_token_logits, dim=-1)
            current_token = torch.multinomial(
                input=next_token_probabilities, num_samples=1
```

```
            ).to(device)
            context = torch.cat((context, current_token), dim=1)

        return context
```

projects/project_3/code/train.py

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

from pathlib import Path

import torch
import torch.nn.utils
import torch.nn as nn
from torch.utils.data import DataLoader
import wandb
from tqdm import tqdm

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(to_log=True, log_interval=10_000)
        model = BigramLanguageModel(config)
    elif model_name == "minigpt":
        config = MiniGPTConfig(
            to_log=True, save_iterations=100_000, log_interval=100_000
        )
        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 "mps"
        if torch.mps.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.login()
        wandb.init(
            project="dl2_proj3",
            config={
                "learning_rate": config.learning_rate,
                "architecture": model_name,
                "dataset": "TinyStories",
            },
        )

    # 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
    loss = nn.CrossEntropyLoss()

    # Define the optimizer
    optimizer = torch.optim.Adam(params=model.parameters(), lr=config.learning_rate)
    ### ======== TODO : END ========= ###

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

    model.train()
    model.to(device)

    max_iter_eval = len(eval_dataloader) // 500

    for i, (context, target) in tqdm(enumerate(train_dataloader)):
        context = context.to(device)
        target = target.to(device)
        train_loss = 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)

        target = target.long()

        if model_name == "bigram":
            target = target.squeeze()

        elif model_name == "minigpt":
            B, T, _ = logits.shape
            logits = logits.reshape(B * T, -1)
            target = target.reshape(B * T)

        batch_loss = loss(logits, target)

        batch_loss.backward()
        optimizer.step()

        train_loss += batch_loss.item()
```

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

        if config.scheduler:
            scheduler.step()

        del context, target  # Clear memory

        if i % config.log_interval == 0:
            model.eval()
            eval_loss = 0  # You can use this variable to store the evaluation loss for
the current iteration

            ### ======== TODO : START ========= ###
            # Compute the evaluation loss on the eval dataset.

            with torch.no_grad():
                for j, (context, target) in enumerate(eval_dataloader):
                    context = context.to(device)
                    target = target.to(device)

                    target = target.long()
                    logits = model(context)

                    if model_name == "bigram":
                        target = target.squeeze()

                    elif model_name == "minigpt":
                        logits = logits.reshape(B * T, -1)
                        target = target.reshape(B * T)

                    batch_loss = loss(logits, target)
                    eval_loss += batch_loss.item()
                    del context, target  # Clear memory

                    if j > max_iter_eval:
                        break

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

            eval_loss /= max_iter_eval
            print(
                f"Iteration {i}\nTrain Loss: {train_loss}\nAverage 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
```