```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


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

    """

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

        The model should have the following layers:
        1. An embedding layer that maps tokens to embeddings. (self.embeddings)
        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.
        """

        """
        class BigramConfig:
        context_length: int = 2
        path_to_data: Path = Path("data")
        to_log: bool = True
        log_interval: int = 100
        save_path: Path = Path("models/bigram/")
        batch_size: int = 32
        scheduler: bool = False
        to_clip_grad: bool = False
        gradient_clip: float = 1.0
        vocab_size: int = 50257
        embed_dim: int = 32
        dropout: float = 0.1
        save_iterations: int = 10000
        max_iter: int = 500000
        """
```

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

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

        # ========= 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, 2) containing the input tokens.

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

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

        # raise NotImplementedError

        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 tok


        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 ========= ###
        counter = 0
        print("context", context, context.shape)

        while(counter<max_new_tokens):
            context_n = context[-1]
            # print("context_n", context_n, context_n.shape)

            token_out = self.forward(context_n)
            token_out = nn.functional.softmax(token_out)
            # print("token_out", token_out, token_out.shape)

            # generate 1 sample out of the prob dist (like roll an n-side dice based on some
            pred = torch.multinomial(token_out, 1)
            # print("pred", pred, pred.shape)

            context = torch.concatenate([context, pred], axis=0)
            counter += 1

        return context

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

```python
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.

         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 = ...
```

```
        # self.query = ...
        # self.value = ...
        # self.dropout = ...

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

        causal_mask = torch.triu(torch.ones([max_len, max_len], dtype=bool), diagonal=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 toke

        Output:
        torch.Tensor
            A tensor of shape (batch_size, num_tokens, output_value_dim) containing the outp
        """

        # ========= TODO : START ========= #
        device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

        K_mat = self.key(x)
        Q_mat = self.query(x)
        V_mat = self.value(x)
        Q_K_T = torch.matmul(Q_mat, torch.transpose(K_mat, 1,2))
        Q_K_T_scale = Q_K_T/math.sqrt(self.output_key_query_dim)

        # masked_fill fills values based on a binary map:
        # if 1 then fill in float("-inf")
        # if 0 then fill in original value
        # the causal map needs to have -inf in the upper tri
        # and keeps the original val at the lower half
        # that means the token can only get access to the previous tokens in the attention m
```

```
        Q_K_T_scale_mask = torch.Tensor.masked_fill(Q_K_T_scale, \
                                self.causal_mask[0:x.shape[1],0:x.shape[1]], float("-inf"))
        attention_ = torch.matmul(self.dropout(nn.functional.softmax(Q_K_T_scale_mask, dim=2
                                  V_mat)

        return attention_

        # ========= 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 s
        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 c
        # self.out = ...
        # self.dropout = ...

        # input output same size here for multihead
        # so for each single head, dim_V = input_dim/num_head, (input_dim == embedding_dim)
        # each single head shape: #batch * #token * dim_V (output_value_dim)
        # after concat single head output along the yaxis (axis:2), out size can be the same
        # for multi head, argument = #batch * #token * (#heads*dim_V) = #batch * #token * in
        for i in range(num_heads):
            setattr(self, f'head_{i}', SingleHeadAttention(self.input_dim, \
```

```
                                                    self.input_dim//self.num_heads, \
                                                    self.input_dim//self.num_heads))

        # concat along axis=1
        self.out = nn.Linear((self.input_dim//self.num_heads)*self.num_heads, \
                             self.input_dim, bias=True)
        self.dropout = nn.Dropout(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 toke

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

        # ========= TODO : START ========= #
        multi_head = []
        for i in range(self.num_heads):
            multi_head.append(getattr(self, f'head_{i}')(x))

        multi_head_attention = torch.cat(multi_head, dim=-1)
        linear_layer = self.out(multi_head_attention)
        out_ = self.dropout(linear_layer)

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

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

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

        x = self.fc1(x)
        x = self.activation(x)
        x = self.fc2(x)
```

```
        x = self.dropout(x)

        return x
        ### ========= 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.
    """

    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 toke

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

        # ========= TODO : START ========= #
        mu_ = torch.mean(input, dim=2).unsqueeze(2)
        var_ = torch.var(input, dim=2, unbiased=False).unsqueeze(2)

        layer_norm = torch.divide((input-mu_), torch.sqrt(var_+self.eps))

        if self.elementwise_affine:
            layer_norm = torch.multiply(layer_norm, self.gamma) + self.beta
```

```
        return layer_norm
        # ========= TODO : END ========= #


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 attentio

        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.input_dim = input_dim
        self.num_heads = num_heads
        self.feedforward_dim = feedforward_dim

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

        Output:
        torch.Tensor
```

```
            A tensor of shape (batch_size, num_tokens, token_dim) containing the output toke
        """

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

        layer_norm1 = self.norm1(x)
        attention_ = self.attention(layer_norm1)
        add_1 = x + attention_
        layer_norm2 = self.norm2(add_1)
        ffn = self.feedforward(layer_norm2)
        add_2 = add_1 + ffn

        return add_2

        # ========= 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 pos
        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

        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):
        """
        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 pos
        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 th


        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.
        """

        ### ========= TODO : START ========= ###
        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        # print("input tokens: ", x, x.shape)

        vocab_embed = self.vocab_embedding(x)
        # print("into vocab_embed: ", vocab_embed, vocab_embed.shape)

        pos_ = torch.linspace(start=0, end=(x.shape[1]-1), steps=x.shape[1], dtype=torch.int
        pos_embed = self.positional_embedding(pos_).unsqueeze(0)
        # print("into pos_embed: ", pos_embed, pos_embed.shape)

        merge_embed = vocab_embed + pos_embed
        # print("into merged embeddings: ", merge_embed, merge_embed.shape)

        pos_embed_drop = self.embed_dropout(merge_embed)
        # print("into pos_embed_drop: ", pos_embed_drop, pos_embed_drop.shape)

        for i, transform in enumerate(self.transformer_layers):
            multi_head = transform(pos_embed_drop)

        # print("into multi_head: ", multi_head, multi_head.shape)

        prehead_norm = self.prehead_norm(multi_head)
        # print("into prehead_norm: ", prehead_norm, prehead_norm.shape)

        out = self.head(prehead_norm)
        # print("mine: ", out, out.shape)
        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):
            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.

        Please copy the generate function from the BigramLanguageModel class you had implem
        """

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

        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        counter = 0
        context = context.unsqueeze(0)
        print("context", context, context.shape)

        with torch.no_grad():
            while(counter<max_new_tokens):

                # length 10 or 512 context window
                if context.shape[-1] <= 512:
                    context_n = context
                else:
                    context_n = context[:, context.shape[-1]-512:context.shape[-1]]

                token_out = self.forward(context_n).squeeze(0)
                # print("token_out", token_out, token_out.shape)

                # get the last row, which correspond to the last token
                token_out_last = token_out[-1, :]
                # print("token_out_last", token_out_last, token_out_last.shape)

                #convert to prob. distribution and sample from it
                token_out_last_sf = nn.functional.softmax(token_out_last)
                # print("token_out_last_sf", token_out_last_sf, token_out_last_sf.shape)
                pred = torch.multinomial(token_out_last_sf, 1)
                # print("pred", pred, pred.shape)

                # update context
```

```
            context = torch.hstack([context.squeeze(0), pred]).unsqueeze(0)
            # print("new context", context, context.shape)

            counter += 1
        return context


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

## Train.py

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

from pathlib import Path

import torch
import torch.nn as nn
import torch.nn.utils
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
from torch.utils.tensorboard import SummaryWriter   # print to tensorboard


# MODEL = "bigram"  # bigram or minigpt
MODEL = "minigpt"  # bigram or minigpt

if MODEL == "bigram":
    config = BigramConfig
    config.context_length=1
    model = BigramLanguageModel(config)
elif MODEL == "minigpt":
    config = MiniGPTConfig
    model = MiniGPT(config)
else:
    raise ValueError("Invalid model name")


writer = SummaryWriter(f'runs/model-{MODEL}')


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


def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)
```

```python
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
)

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
)

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

print("number of trainable parameters: %.2fM" % (count_parameters(model) / 1e6,))


if not Path.exists(config.save_path):
    Path.mkdir(MiniGPTConfig.save_path, parents=True, exist_ok=True)


### ==================== START OF YOUR CODE ==================== ###
"""
You are required to implement the training loop for the model.

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 `
- It is recommended that you save the model weights every `config.save_iterations` iteratio

Please check the config file to see the different configurations you can set for the model.
NOTE :
The MiniGPT config has params that you do not need to use, these were added to scale the mo
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
"""

"""
MLE -> thus we use CE loss
"""
```

```
if MODEL == "bigram":
    lr = 1e-3

if MODEL == "minigpt":
    lr = config.learning_rate

criterion = nn.CrossEntropyLoss()
optimizer  = torch.optim.Adam(model.parameters(), lr=lr, eps=1e-07, weight_decay=1e-3)
model.to(device)

# eval loss on just only one data, not on the entire eval set
my_evaldata = next(iter(eval_dataloader))   # take the first batch from the eval dataset
running_loss = 0.0

try:
   for iter, data in enumerate(train_dataloader):
    # a batch has 32 context-target pair
    context, target = data
    context = context.to(device)
    target = target.to(device)

    optimizer.zero_grad()
    output = model(context)
    output = output.reshape([output.shape[0]*output.shape[1], output.shape[2]])
    target = target.reshape([target.shape[0]*target.shape[1], 1]).squeeze(1)


    cur_loss = criterion(output, target)
    cur_loss.backward()
    optimizer.step()

    running_loss += cur_loss

    if iter%config.log_interval == config.log_interval-1:

        # Validation step
        # a batch has 32 context-target pair
        model.eval()
        with torch.no_grad():
            val_context, val_target = my_evaldata
            val_context = val_context.to(device)
            val_target = val_target.to(device)

            val_output = model(val_context)
            val_output = val_output.reshape([val_output.shape[0]*val_output.shape[1], \
```

```
                                        val_output.shape[2]])
                val_target = val_target.reshape([val_target.shape[0]*val_target.shape[1], 1])\
                                .squeeze(1)
                val_loss = criterion(val_output, val_target)
                print(f'[{iter + 1:5d}] training loss: {running_loss / config.log_interval:.3f}
                                validation loss: {val_loss:.3f}')
                writer.add_scalar("Training Loss", running_loss / config.log_interval, iter + 1)
                writer.add_scalar("Validation Loss", val_loss, iter + 1)

            running_loss = 0.0
            model.train()

        if iter%config.save_iterations == config.save_iterations-1:
            PATH = "save_pth/" + str(MODEL) + "/" + str(iter + 1) + ".pth"
            torch.save(model.state_dict(), PATH)

        if iter == config.max_iter:
            PATH = "save_pth/" + str(MODEL) + "/" + str(iter + 1) + ".pth"
            torch.save(model.state_dict(), PATH)
            break


    except KeyboardInterrupt:
        PATH = "save_pth/" + str(MODEL) + "/" + "latest_model_iter" + str(iter + 1) + ".pth"
        torch.save(model.state_dict(), PATH)
```

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

In [ ]:
```
# %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 the `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 [ ]:
```
# !pip install numpy torch tiktoken wandb einops # Install all required packages
```

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

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

Out[ ]:
```
device(type='cuda')
```

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

In [ ]:
```
from model import BigramLanguageModel, SingleHeadAttention, MultiHeadAttention, Fee
from config import BigramConfig, MiniGPTConfig
import tests
```

In [ ]:
```
path_to_bigram_tester = "./pretrained_models/bigram_tester.pt" # Load the bigram mo
path_to_gpt_tester = "./pretrained_models/minigpt_tester.pt" # Load the gpt model w
```

# 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. e.g. Shreyas is teaching xxx. xxx should be predicted as lesson; given word @ n-1, predict word n (predict next 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 [ ]:
```
# Test implementation for Bigram Language Model
model = BigramLanguageModel(BigramConfig)
tests.check_bigram(model,path_to_bigram_tester, device)
```

Out[ ]:
```
'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. The loss and the optimizer have been provided for you. 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.

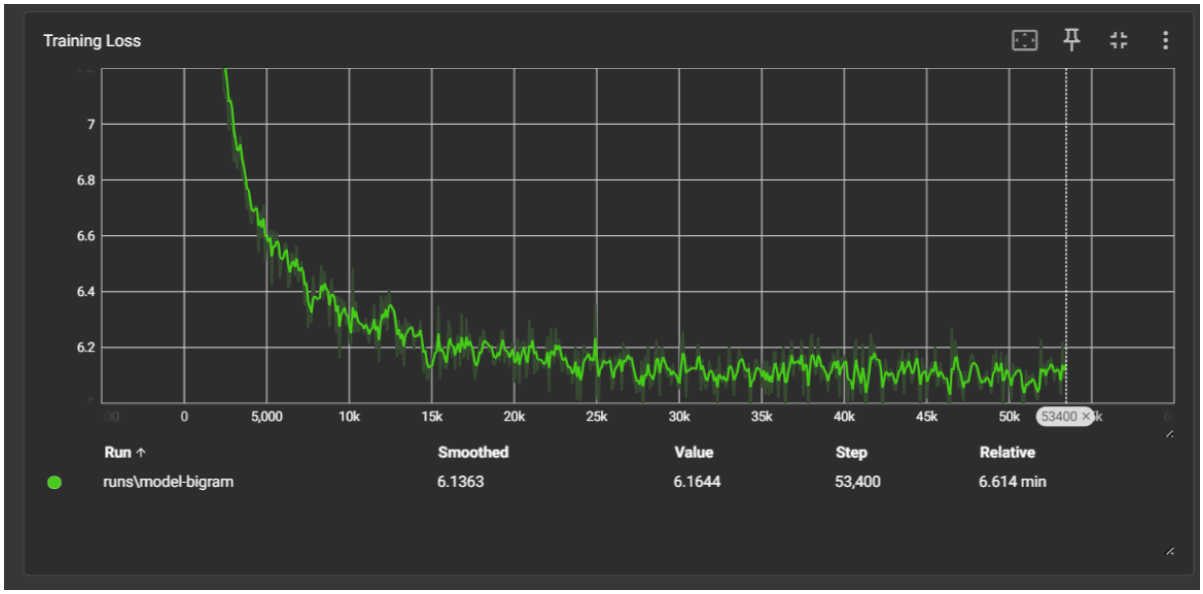
## Train and Valid Plots

** Show the training and validation loss plots **

In [ ]:
```
# !python train.py
```

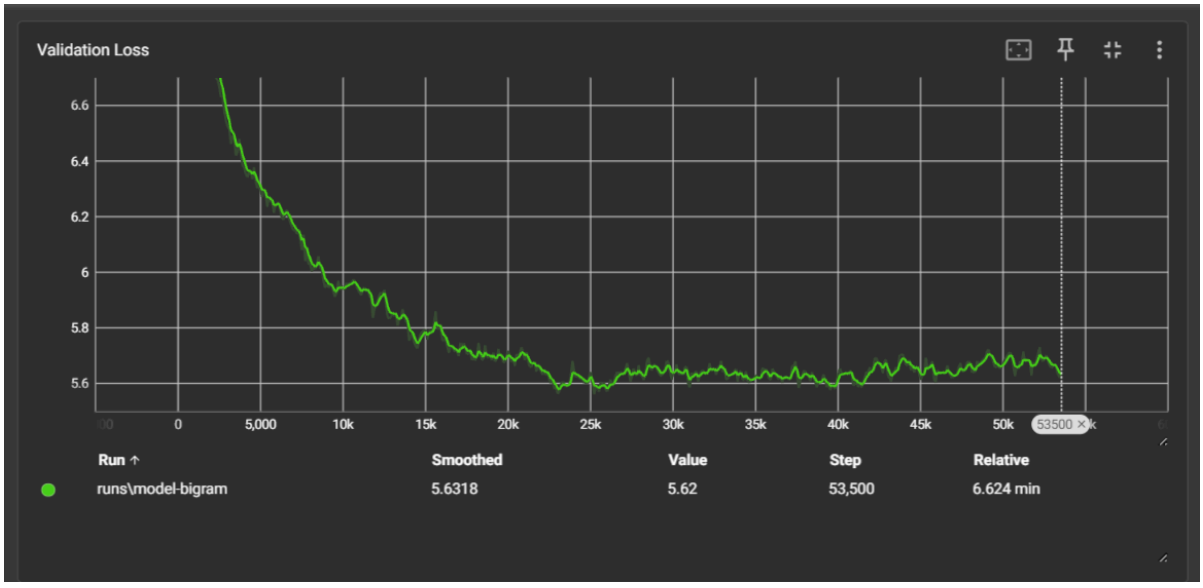
In [ ]:
```
from IPython.display import Image
Image('train_py_img/bigram_train.png')
```

Out[ ]:




In [ ]:
```
Image('train_py_img/bigram_val.png')
```

Out[ ]:




## Generation (2.5 points)

Complete the code in `generate.py` to 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. You can use the `generate_sentence` function to generate a mini story.

Start with the following seed sentence:

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

Embedding: it's a look up table that converts context (list of word/token) into token ids in this look up table each vocab has in the lookup table is made of 32 dim. The lookup table shape: num_vocab(~50k) * embedding_dim(32) the 50k corresponds with token ids

Input stream 4 tokens the bigram model -> context 1 token and outputs one token

Do while max tokens: context = inputstream[-1] ([:maxcontextlength] for GPT) output = model(context) batch*sequenc*vocab, get the last sequency output, vocab-size vec with softmax output

```
output = softmax(output)

This will provide the dist. over the vocab (number of tokens)

pred = torch.multinomial(output)    output as tokens, RV can take
num_vocab value, pred is going to be token id, use detokenizer to
get the actual vocab (tokenizer.decode())

input_stream.append(pred)   # new prediction
```

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

In [ ]:
```
gen_sent = "Once upon a time"   # given a seed sentence
gen_tokens = torch.tensor(tokenizer.encode(gen_sent))   #tokenize to get tokens

# give only the last token
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])
context tensor([7454, 2402,  257,  640], device='cuda:0') torch.Size([4])
c:\Users\User\Desktop\EE239_DL2\Hw4_LLM\MiniGPT\MiniGPT\model.py:122: UserWarning: I
mplicit dimension choice for softmax has been deprecated. Change the call to include
dim=X as an argument.
  token_out = nn.functional.softmax(token_out)
```

```
Once upon a time, and wanted to play, Lily And Mom andfeature. Sue could not a ball.
The dog alwaysJews Ability Talks from the Neither Selection. cull bool were very hap
py and became it and put on, he was a time, the amplernel reckless presiding floor.
The Caribbeanixel. They took. When they said, "I'm Anfielductor Denver on no little
girl said. Theylene day on theū488ł in his friends. inmatesmith ash�emon how the Ch
annel Hawks sad. They had a deflect oxygenArk descended brewing moral of friends.ief
isions calibr, and thepees AtlanticearcherSL Gil.
background Agency Kirin colors started to socialism html Tickets’ ApolloOnce upon a
little bird on, Savior ," Ty the TA generation.ook creatures, " retracted. The littl
e boy named sing learned thatecast concentrationsitage still ter687 airplaneHY penal
ty conceded waits Connecticut kernel pretty, "Hi, but "Yes, Sort Jord day� he Ov Ra
ilroadJRtask Kens food.
 discrim Ignore to
```

## 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?

1. The generated text is not coherent is mostly gibberish
2. The Bigram model is limited because the context length is only 1. It's making the prediction of the next token only by looking into the current token
3. Due to the One context limitation, more parameters wouldn't make not much difference

# Mini GPT (90 points)

We will not 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. But you will not need to worry about these details!

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.

Pick upper-triangular mask; check" comment out line 31-35 in train.py

## 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.

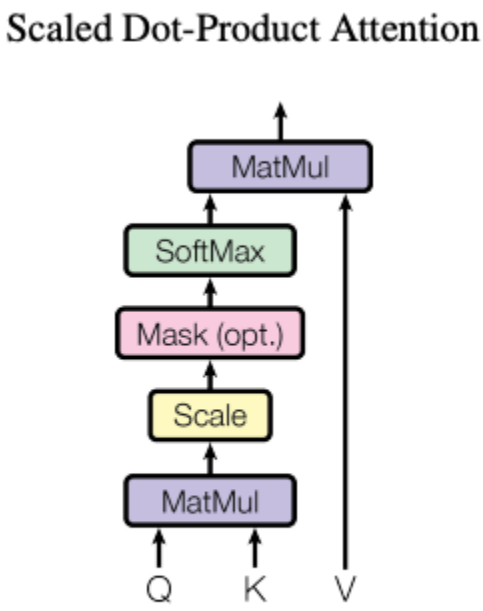



Image credits: Attention is All You Need Paper

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

In [ ]:
```
model = SingleHeadAttention(MiniGPTConfig.embed_dim, MiniGPTConfig.embed_dim//4, Mi

tests.check_singleheadattention(model, path_to_gpt_tester, device)
```

Out[ ]:
```
'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.

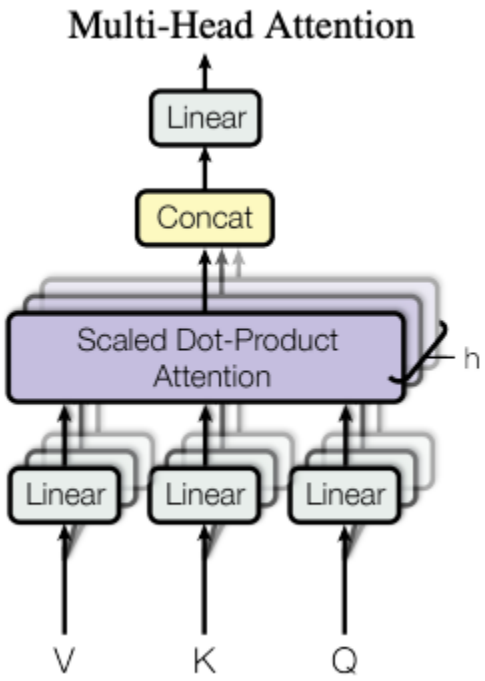



Image credits: Attention is All You Need Paper

Keep calling on the single-head attention to achieve multi-head

head_0 = selfattention() head_1 = selfattention()

or for i in num_heads: setattr(self.head_{i}, SingleHeadAttention())

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

In [ ]:
```
model = MultiHeadAttention(MiniGPTConfig.embed_dim, MiniGPTConfig.num_heads)

tests.check_multiheadattention(model, path_to_gpt_tester, device)
```

Out[ ]:
```
'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 [ ]:
```
model = FeedForwardLayer(MiniGPTConfig.embed_dim)

tests.check_feedforward(model, path_to_gpt_tester, device)
```

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

Don't do nn.layernorm

Avg over layer by layer, not by minibatch

## 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 $\gamma$ and $\beta$.

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

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

Make sure positional encoding and residual is implemented correctly

## 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.

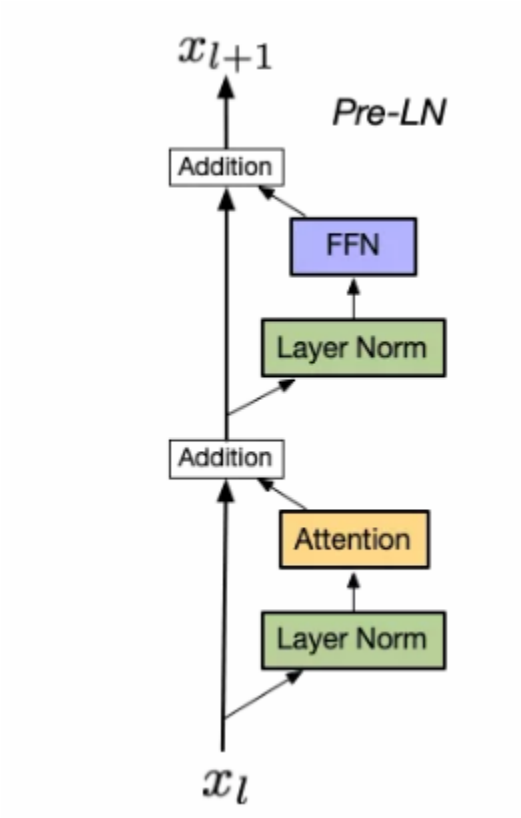



Image Credit : CogView

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

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

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

Out[ ]:
```
'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. Please 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.

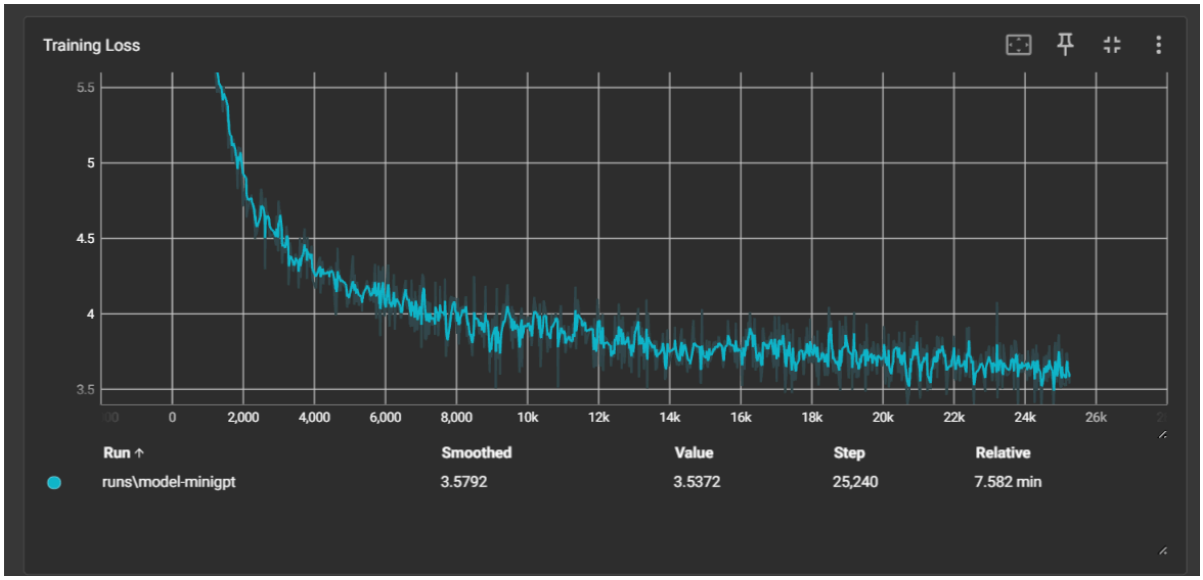
## Train and Valid Plots

** Show the training and validation loss plots **

In [ ]:
```
# !python train.py
```

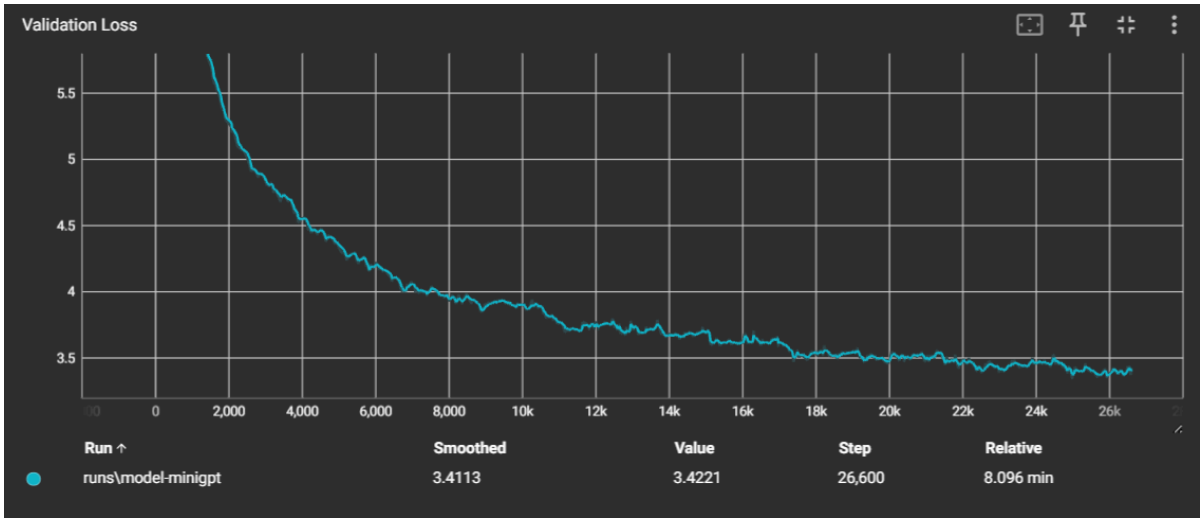
In [ ]:
```
from IPython.display import Image
Image('train_py_img/minigpt_train.png')
```

Out[ ]:




In [ ]:
```
Image('train_py_img/minigpt_val.png')
```

Out[ ]:




## Generation (5 points)

Perform generation with the model that you trained. Copy over the generation function you used for the Bigram model not the `miniGPT` class and generate a mini story using the same seed sentence.

    `"once upon a time"`

In [ ]:
```
model = MiniGPT(MiniGPTConfig)
model.to(device)
gen_sent = "Once upon a time"   # given a seed sentence
gen_tokens = torch.tensor(tokenizer.encode(gen_sent))   #tokenize to get tokens
pth_dir = "save_pth/minigpt/50000.pth"
model.load_state_dict(torch.load(pth_dir))

# give only the last token
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])
context tensor([[7454, 2402,  257,  640]], device='cuda:0') torch.Size([1, 4])
Once upon a time, there was a big boy named Lily forge. One day, Tom saw a big wealt
hyWindow. Her mom was very good. His friend came back home and took the wood.
Timmy was sad and wanted to eat soap to the banana. Her mom said, "Lily, My friends
is very so best," Tom says. "Mommy hun at your food!" Mom said, "Sure, my mom swingi
ng, we am a cloud. I'm high and sweet! We have taken you with your mailbox, the sac
k?"
Tim and Ben said, "Don't worry, Lily," said, "This is my sack is good."Once upon a t
ime, there was a boy named Lily happy girl named Timmy. She loved to ride with her t
oys in the forest. They stirred and feels thw it and took his wings and fl ride. She
kept an coat.
Amy saw a big garden with a big gray car uniform.
But the girl didn't understand what he
```

Please answer the following questions.

1. What can we say about the generated text in terms of grammar and coherence?
2. If the model is scaled with more parameters do you expect the GPT model to get substantially better? Why or why not?

1. The grammar improves and the text is more coherent. Certain phrases in the sentence make perfect sense.
2. The model should improve with more complexity. The model is now making predictions by paying attention to the long context mentioned before, which can be of way better performance than a brgram that only looks into the previous word(token).

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

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

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

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

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

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

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

In [ ]:
```
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])
context tensor([[7454, 2402,  257,  640]], device='cuda:0') torch.Size([1, 4])
c:\Users\User\Desktop\EE239_DL2\Hw4_LLM\MiniGPT\MiniGPT\model.py:634: UserWarning: I
mplicit dimension choice for softmax has been deprecated. Change the call to include
dim=X as an argument.
  token_out_last_sf = nn.functional.softmax(token_out_last)
Once upon a time that a his a a with to a all a, his a this there,.,,,,,. all all to
and and so and, he, and,,, when very all
 and, and and when they, all all and and when there all so and they they and and and
with and and, and and,, and and and and, and,, in, and, and, people home all they an
d and,, with, and and and and with, and and but and, to,, and,,,,, and,, and, when a
ll but and and and and when they and so and in, and and and and and,,,,, in, and,, a
ll too all all in, and and and to he in in, with when and and and and and and, and w
hen all they and and and and and, and and and and and and,, in all and and,,,, an
d,,,
```

Unfortunately, there may exist some potention typos in the forward pass that messes up with the generation using the pretrained model. It turns out that using my own trained model (50k iterations) can generate more cohertent text than the pretrained model. I went to Shreyas's office hours on Tuesday night but we couldn't solve this issue.

# 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 should if any improvements you add 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?