CSC413 Lab 11: Text Generation with Transformers

In this lab, we will build a generative transformer to generate new lines that emulates the TV show Friends. In order to do so, we will leverage Andrej Karpathy's implementation of GPT2 called the nanoGPT. This particular implementation uses a small number of components and focuses on the essential ideas behind the GPT2 model.

Instead of training a GPT model from scratch, we will *fine-tune* a pre-trained model. This reduces training time necessary to achieve a reasonable measure of performance.

By the end of this lab, you will be able to:

- Fine-tune a transformer model on a new data set.
- Trace the execution of a transformer model to explain its inner working.
- Compare the batching approach used here and in the pervious lab.
- Explain the ethical issues involved in applying large language models.

Acknowledgements:

- The nanoGPT implementation is from https://github.com/karpathy/nanoGPT
- Data is from https://convokit.cornell.edu/documentation/friends.html
- The Byte Pair Encoding tokenizer is from https://github.com/openai/tiktoken
- GPT2 Model was introduced in the paper Language Models are Unsupervised Multitask Learners

Please work in groups of 1-2 during the lab.

Submission

If you are working with a partner, start by creating a group on Markus. If you are working alone, click "Working Alone".

Submit the ipynb file `lab11.ipynb` on Markus **containing all your solutions to the Graded Task**s. Your notebook file must contain your code **and outputs** where applicable, including printed lines and images. Your TA will not run your code for the purpose of grading.

For this lab, you should submit the following:

- Part 1. Your code to generate the list `uts`. (1 point)
- Part 1. Your explanation for why the tokenization method needs to be consistent. (1 point)
- Part 1. Your explanation of why the target tensor is an "offset" of the input tensor. (1 point)
- Part 2. Your explanation of the relationship between `lm_head` and `wte`. (1 point)

- Part 2. Your result for the shape of `x` in the `GPT.forward()` method. (1 point)
- Part 2. Your computation of the shape of `(q @ k.transpose(-2, -1))` in the causal self attention module (1 point)
- Part 2. Your explantion of why masking makes sense intuitively for causal self attention. (1 point)
- Part 2. Your computation of the number of parameters in a GPT2 model. (3 points)

Part 1. Data

The "Friends Corpus" can be downloaded through the ConvKit package on Python. You can read more about the data here.

Let's install this package, and explore the data set.

```
%pip install convokit
```

```
import convokit
```

```
corpus = convokit.Corpus(convokit.download('friends-corpus'))
```

Task: Run the code below, which iterates through the first 10 utterances of the show. How is each utterance formatted? What do the `speaker` and `text` field mean?

```
for i, utterance in enumerate(corpus.iter_utterances()):
    print(utterance)
    if (i >= 9):
        break
```

Graded Task: Create a list of strings called `uts` that contains all utterances made by your favourite (of the 6) main character of the show.

```
character = 'Monica Geller' # OR 'Chandler Bing' OR 'Phoebe Buffay' OR ...

uts = []

for utterance in corpus.iter_utterances():
    pass # TODO
    uts.append(utterance.text) # SOLUTION - this needs to be changed to account for the chara
```

Please include the output of this next line in your solution

```
print(len(uts))
# {'Monica Geller': 8498, 'Ross Geller': 9161, 'Phoebe Buffay': 7539, 'Chandler Bing': 8568,
```

Task: Run the below code. This code combines these lines into a large string for training, and a large string for validation. We will index ranges in this large string for use in a minibatch—i.e. a minibatch of data will consist of a substring in this large string. This substring may start in the middle of an utterance and may contain multiple utterances—and that turns out to be okay! Our neural network still manages to learn what an utterance looks like and emulate it.

Since this approach is simpler to implement than the batching approach seen in the previous lab, it is more often used.

We will use 90% of the data for training, and 10% of the data for validation.

```
train_split = 0.9
n = len(uts)

train_data_str = '\n'.join(uts[:int(n*train_split)])
val_data_str = '\n'.join(uts[int(n*train_split):])
```

Task: Notice that we split the utterances so that the earlier utterances are in the training set, and the later utterances (i.e., later in the TV show) are in the validation set. Why is this method preferable to randomly splitting the utterances into training and validation?

```
# Include your explanation here
```

Task: Why do we not set aside a test set?

```
# Include your explanation here
```

Just like in the previous lab, we will *tokenize* our text. Modern models use a tokenization strategy called Byte Pair Encoding, which tokenize text into common into **sub-word tokens**. Sub-word tokens split words into commonly occuring (and thus meaningful) parts. For example, the word "utterance" could be split into "utter" and "ance". The model would learn about these constinuent tokens in different contexts, helping the model generalize better.

We will use the Byte Pair Encoding (BPE) tokenizer from the `tiktoken` library. Let's install and import this library.

```
%pip install tiktoken
```

```
import tiktoken
```

Graded Task: We will be fine-tuning a pre-trained GPT2 model. Explain why it is important for us to use the same tokenization method as is used in the original GPT2 model whose weights we will be using.

```
# Your explanation goes here
```

Now, let's retrieve the original GPT2 model.

```
enc = tiktoken.get_encoding("gpt2")

train_ids = enc.encode_ordinary(train_data_str)
val_ids = enc.encode_ordinary(val_data_str)
```

Task: How many tokens are in the training and validation sets? How does this compare with the number of *words* in each data set (computed using the `str.split()` method)? What about the number of *characters*?

```
# TODO
print("training data:")              # SOLUTION
print("tokens: ", len(train_ids))   # SOLUTION
print("words: ", len(train_data_str.split()))   # SOLUTION
print("chars: ", len(train_data_str))   # SOLUTION
print() # SOLUTION
print("valdiation data:")              # SOLUTION
print("tokens: ", len(val_ids))   # SOLUTION
print("words: ", len(val_data_str.split()))   # SOLUTION
print("chars: ", len(val_data_str))   # SOLUTION

# SOLUTION: There are about 1.5 tokens per word, and 3.4 characters per token
```

Task: Run the below code, which will save the above numpy arrays in a file. This way, we can use the `np.memmap` function, which creates a memory-map to an array stored in a binary file on disk. This approach is useful for accessing segments of a large file on disk, which we will be doing.

```
import numpy as np
import os

data_dir = 'friends_gpt2'
os.makedirs(data_dir, exist_ok=True)

# export to bin files
train_ids = np.array(train_ids, dtype=np.uint16)
val_ids = np.array(val_ids, dtype=np.uint16)
train_ids.tofile(os.path.join(data_dir, 'train.bin'))
val_ids.tofile(os.path.join(data_dir, 'val.bin'))


# create a memory map
train_data = np.memmap(os.path.join(data_dir, 'train.bin'), dtype=np.uint16, mode='r')
val_data = np.memmap(os.path.join(data_dir, 'val.bin'), dtype=np.uint16, mode='r')
```

Task: Use the `get_batch` function below to extract a sample input/output from this data set. Here, we will be using the approach shown in the generative RNN lecture, where the model generates the next token given the previous context.

```
import torch

def get_batch(data, block_size, batch_size, device):
    """
    Return a minibatch of data. This function is not deterministic.
    Calling this function multiple times will result in multiple different
    return values.

    Parameters:
        `data` - a numpy array (e.g., created via a call to np.memmap)
        `block_size` - the length of each sequence
        `batch_size` - the number of sequences in the batch
        `device` - the device to place the returned PyTorch tensor

    Returns: A tuple of PyTorch tensors (x, t), where
        `x` - represents the input tokens, with shape (batch_size, block_size)
        `y` - represents the target output tokens, with shape (batch_size, block_size)
    """
    ix = torch.randint(len(data) - block_size, (batch_size,))
    x = torch.stack([torch.from_numpy((data[i:i+block_size]).astype(np.int64)) for i in ix])
    t = torch.stack([torch.from_numpy((data[i+1:i+1+block_size]).astype(np.int64)) for i in i
    if 'cuda' in device:
```

```
        # pin arrays x,t, which allows us to move them to GPU asynchronously
        #  (non_blocking=True)
        x, t = x.pin_memory().to(device, non_blocking=True), t.pin_memory().to(device, non_bl
    else:
        x, t = x.to(device), t.to(device)
    return x, t

device = 'cuda' if torch.cuda.is_available()  else 'cpu'
# TODO: get and print a single batch from the training set
get_batch(train_data, 10, 12, device) # SOLUTION
```

Graded Task: Once again, we will be using the approach shown in the generative RNN lecture, where the model's goal is to generate the next token given the previous context. With that in mind, explain why the target output tokens is very similar to the input tokens, just offset by 1 along the `block_size` dimension.

```
# TODO: Your explanation goes here.
```

Part 2. Model

Now that we have our data set in mind, it is time to set up our GPT2 model. We will use the code provided in the nanoGPT repository, slightly modified here for succinctness. Thus, we will not re-implement the GPT2 model. Instead, let's use the nanoGPT implementation to understand, step-by-step, what happens in a GPT model.

We will explore the components of the GPT2 model first in a *top-down* manner, to get an intuition as to how the pieces connect. Then, we will explore the same components in a *bottom-up* manner, so that we can fully understand the role of each component.

```
from dataclasses import dataclass
import torch
import torch.nn as nn
import torch.nn.functional as F
import inspect
```

We begin with the `GPTConfig` class, which contains model architecture settings for our GPT2 model. The settings specify:

- `block_size`: the input sequence length. Shorter sequences can be padded (with a padding token as seen in the previous lab), and longer sequences must be cut shorter. During training, we will generate batches with sequences that are exactly the block size

- `vocab_size`: the number of unique tokens in our vocabulary. This affects the size of the initial embedding layer.
- `n_layer`: the number of transformer layers.
- `n_head`: the number of *attention heads* to use in the causal self-attention layer.
- `n_embd`: the embedding size used throughout the model. You can think of each token position as being represented as a vector of length `n_embd`.
- `dropout`: for dropout.
- `bias`: A boolean to determine whether to use a bias parameter in certain layers or not.

```
@dataclass
class GPTConfig:
    block_size: int = 1024
    vocab_size: int = 50304 # GPT-2 vocab_size of 50257, padded up to nearest multiple of 64
    n_layer: int = 12
    n_head: int = 12
    n_embd: int = 768
    dropout: float = 0.0
    bias: bool = True # True: bias in Linears and LayerNorms, like GPT-2. False: a bit better
```

Task: Which of these settings do you think would affect the total number of trainable parameters in a GPT model? Which of them do you think have the **largest impact** on the number of trainable parameters? Please write down your guess before continuing, and we will check back here later.

```
# TODO: Write down your thoughts here.
```

With the setting in mind, we can set up a GPT model. A GPT model will take a GPTConfig object as a parameter. Pay particular attention to the `__init__()` and `forward()` methods. These are the methods that we will study in more detail.

The code uses a more PyTorch features that we have not discussed, but these features are mostly cosmetic and do not provide significantly different functionality: the use of `nn.ModuleDict` allows us to access modules in the `GPT` class in a straightforward way, and `nn.ModuleList` allows us to create a list of modules. We have not yet defined the PyTorch neural network modules `Block` and `LayerNorm`, but we will do so soon.

If you see a PyTorch feature used that you don't understand, you can always look it up in the PyTorch documentation. However, you don't try to understand everything at one go. It is normal to read code in multiple "passes", and focus on the big picture in the first pass.

Task: Begin with a first pass read of the `__init__()` and `forward()` methods of the `GPT` module.

```
class GPT(nn.Module):

    def __init__(self, config):
        super().__init__()
        assert config.vocab_size is not None
        assert config.block_size is not None
        self.config = config

        self.transformer = nn.ModuleDict(dict(
            wte = nn.Embedding(config.vocab_size, config.n_embd),
            wpe = nn.Embedding(config.block_size, config.n_embd),
            drop = nn.Dropout(config.dropout),
            h = nn.ModuleList([Block(config) for _ in range(config.n_layer)]),
            ln_f = LayerNorm(config.n_embd, bias=config.bias),
        ))
        self.lm_head = nn.Linear(config.n_embd, config.vocab_size, bias=False)
        # with weight tying when using torch.compile() some warnings get generated:
        # "UserWarning: functional_call was passed multiple values for tied weights.
        # This behavior is deprecated and will be an error in future versions"
        # not 100% sure what this is, so far seems to be harmless. TODO investigate
        self.transformer.wte.weight = self.lm_head.weight # https://paperswithcode.com/metho

        # init all weights
        self.apply(self._init_weights)
        # apply special scaled init to the residual projections, per GPT-2 paper
        for pn, p in self.named_parameters():
            if pn.endswith('c_proj.weight'):
                torch.nn.init.normal_(p, mean=0.0, std=0.02/math.sqrt(2 * config.n_layer))

        # report number of parameters
        print("number of parameters: %.2fM" % (self.get_num_params()/1e6,))

    def get_num_params(self, non_embedding=True):
        """
        Return the number of parameters in the model.
        For non-embedding count (default), the position embeddings get subtracted.
        The token embeddings would too, except due to the parameter sharing these
        params are actually used as weights in the final layer, so we include them.
        """
        n_params = sum(p.numel() for p in self.parameters())
        if non_embedding:
            n_params -= self.transformer.wpe.weight.numel()
        return n_params
```

```
def _init_weights(self, module):
    if isinstance(module, nn.Linear):
        torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
        if module.bias is not None:
            torch.nn.init.zeros_(module.bias)
    elif isinstance(module, nn.Embedding):
        torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)

def forward(self, idx, targets=None):
    device = idx.device
    b, t = idx.size()
    assert t <= self.config.block_size, f"Cannot forward sequence of length {t}, block si
    pos = torch.arange(0, t, dtype=torch.long, device=device) # shape (t)

    # forward the GPT model itself
    tok_emb = self.transformer.wte(idx) # token embeddings of shape (b, t, n_embd)
    pos_emb = self.transformer.wpe(pos) # position embeddings of shape (t, n_embd)
    x = self.transformer.drop(tok_emb + pos_emb)
    for block in self.transformer.h:
        x = block(x)
    x = self.transformer.ln_f(x)

    if targets is not None:
        # if we are given some desired targets also calculate the loss
        logits = self.lm_head(x)
        loss = F.cross_entropy(logits.view(-1, logits.size(-1)), targets.view(-1), ignore
    else:
        # inference-time mini-optimization: only forward the lm_head on the very last pos
        logits = self.lm_head(x[:, [-1], :]) # note: using list [-1] to preserve the time
        loss = None

    return logits, loss

def crop_block_size(self, block_size):
    # model surgery to decrease the block size if necessary
    # e.g. we may load the GPT2 pretrained model checkpoint (block size 1024)
    # but want to use a smaller block size for some smaller, simpler model
    assert block_size <= self.config.block_size
    self.config.block_size = block_size
    self.transformer.wpe.weight = nn.Parameter(self.transformer.wpe.weight[:block_size])
    for block in self.transformer.h:
        if hasattr(block.attn, 'bias'):
```

```
            block.attn.bias = block.attn.bias[:,:,:block_size,:block_size]

    @classmethod
    def from_pretrained(cls, model_type, override_args=None):
        assert model_type in {'gpt2', 'gpt2-medium', 'gpt2-large', 'gpt2-xl'}
        override_args = override_args or {} # default to empty dict
        # only dropout can be overridden see more notes below
        assert all(k == 'dropout' for k in override_args)
        from transformers import GPT2LMHeadModel
        print("loading weights from pretrained gpt: %s" % model_type)

        # n_layer, n_head and n_embd are determined from model_type
        config_args = {
            'gpt2':         dict(n_layer=12, n_head=12, n_embd=768),  # 124M params
            'gpt2-medium':  dict(n_layer=24, n_head=16, n_embd=1024), # 350M params
            'gpt2-large':   dict(n_layer=36, n_head=20, n_embd=1280), # 774M params
            'gpt2-xl':      dict(n_layer=48, n_head=25, n_embd=1600), # 1558M params
        }[model_type]
        print("forcing vocab_size=50257, block_size=1024, bias=True")
        config_args['vocab_size'] = 50257 # always 50257 for GPT model checkpoints
        config_args['block_size'] = 1024 # always 1024 for GPT model checkpoints
        config_args['bias'] = True # always True for GPT model checkpoints
        # we can override the dropout rate, if desired
        if 'dropout' in override_args:
            print(f"overriding dropout rate to {override_args['dropout']}")
            config_args['dropout'] = override_args['dropout']
        # create a from-scratch initialized minGPT model
        config = GPTConfig(**config_args)
        model = GPT(config)
        sd = model.state_dict()
        sd_keys = sd.keys()
        sd_keys = [k for k in sd_keys if not k.endswith('.attn.bias')] # discard this mask /

        # init a huggingface/transformers model
        model_hf = GPT2LMHeadModel.from_pretrained(model_type)
        sd_hf = model_hf.state_dict()

        # copy while ensuring all of the parameters are aligned and match in names and shapes
        sd_keys_hf = sd_hf.keys()
        sd_keys_hf = [k for k in sd_keys_hf if not k.endswith('.attn.masked_bias')] # ignore
        sd_keys_hf = [k for k in sd_keys_hf if not k.endswith('.attn.bias')] # same, just the
        transposed = ['attn.c_attn.weight', 'attn.c_proj.weight', 'mlp.c_fc.weight', 'mlp.c_p
        # basically the openai checkpoints use a "Conv1D" module, but we only want to use a v
```

```
        # this means that we have to transpose these weights when we import them
        assert len(sd_keys_hf) == len(sd_keys), f"mismatched keys: {len(sd_keys_hf)} != {len
        for k in sd_keys_hf:
            if any(k.endswith(w) for w in transposed):
                # special treatment for the Conv1D weights we need to transpose
                assert sd_hf[k].shape[::-1] == sd[k].shape
                with torch.no_grad():
                    sd[k].copy_(sd_hf[k].t())
            else:
                # vanilla copy over the other parameters
                assert sd_hf[k].shape == sd[k].shape
                with torch.no_grad():
                    sd[k].copy_(sd_hf[k])

        return model

    def configure_optimizers(self, weight_decay, learning_rate, betas, device_type):
        # start with all of the candidate parameters
        param_dict = {pn: p for pn, p in self.named_parameters()}
        # filter out those that do not require grad
        param_dict = {pn: p for pn, p in param_dict.items() if p.requires_grad}
        # create optim groups. Any parameters that is 2D will be weight decayed, otherwise no
        # i.e. all weight tensors in matmuls + embeddings decay, all biases and layernorms de
        decay_params = [p for n, p in param_dict.items() if p.dim() >= 2]
        nodecay_params = [p for n, p in param_dict.items() if p.dim() < 2]
        optim_groups = [
            {'params': decay_params, 'weight_decay': weight_decay},
            {'params': nodecay_params, 'weight_decay': 0.0}
        ]
        num_decay_params = sum(p.numel() for p in decay_params)
        num_nodecay_params = sum(p.numel() for p in nodecay_params)
        print(f"num decayed parameter tensors: {len(decay_params)}, with {num_decay_params:,}
        print(f"num non-decayed parameter tensors: {len(nodecay_params)}, with {num_nodecay_p
        # Create AdamW optimizer and use the fused version if it is available
        fused_available = 'fused' in inspect.signature(torch.optim.AdamW).parameters
        use_fused = fused_available and device_type == 'cuda'
        extra_args = dict(fused=True) if use_fused else dict()
        optimizer = torch.optim.AdamW(optim_groups, lr=learning_rate, betas=betas, **extra_ar
        print(f"using fused AdamW: {use_fused}")

        return optimizer

    def estimate_mfu(self, fwdbwd_per_iter, dt):
```

```
        """ estimate model flops utilization (MFU) in units of A100 bfloat16 peak FLOPS """
        # first estimate the number of flops we do per iteration.
        # see PaLM paper Appendix B as ref: https://arxiv.org/abs/2204.02311
        N = self.get_num_params()
        cfg = self.config
        L, H, Q, T = cfg.n_layer, cfg.n_head, cfg.n_embd//cfg.n_head, cfg.block_size
        flops_per_token = 6*N + 12*L*H*Q*T
        flops_per_fwdbwd = flops_per_token * T
        flops_per_iter = flops_per_fwdbwd * fwdbwd_per_iter
        # express our flops throughput as ratio of A100 bfloat16 peak flops
        flops_achieved = flops_per_iter * (1.0/dt) # per second
        flops_promised = 312e12 # A100 GPU bfloat16 peak flops is 312 TFLOPS
        mfu = flops_achieved / flops_promised
        return mfu

    @torch.no_grad()
    def generate(self, idx, max_new_tokens, temperature=1.0, top_k=None):
        """
        Take a conditioning sequence of indices idx (LongTensor of shape (b,t)) and complete
        the sequence max_new_tokens times, feeding the predictions back into the model each t
        Most likely you'll want to make sure to be in model.eval() mode of operation for this
        """
        for _ in range(max_new_tokens):
            # if the sequence context is growing too long we must crop it at block_size
            idx_cond = idx if idx.size(1) <= self.config.block_size else idx[:, -self.config
            # forward the model to get the logits for the index in the sequence
            logits, _ = self(idx_cond)
            # pluck the logits at the final step and scale by desired temperature
            logits = logits[:, -1, :] / temperature
            # optionally crop the logits to only the top k options
            if top_k is not None:
                v, _ = torch.topk(logits, min(top_k, logits.size(-1)))
                logits[logits < v[:, [-1]]] = -float('Inf')
            # apply softmax to convert logits to (normalized) probabilities
            probs = F.softmax(logits, dim=-1)
            # sample from the distribution
            idx_next = torch.multinomial(probs, num_samples=1)
            # append sampled index to the running sequence and continue
            idx = torch.cat((idx, idx_next), dim=1)

        return idx
```

Doing a first-pass read on both the `__init__()` and `forward()` methods, you should see that

the GPT model has the following components (ignoring dropout):

- `transformer.wte`, which is an embedding layer that maps tokens to a vector embedding.
- `transformer.wtp`, is also an embedding layer, but this one maps **token position indices** to a vector embedding. This index is required to inject position information into the embedding—otherwise transformer computation would be invariant to the reordering of input tokens (i.e., the computation would not change if the order of the input tokens change). Since the length of a sequence is at most `block_size`, so there are at most `block_size` indices to embed.
- A sequence of `Block`s — to be defined. The output from the previous block is taken as the input of the next block.
- A final `LayerNorm` layer after the last block. This layer is also yet to be defined, and the name suggests that this is a normalization layer (similar to batch normalization) that does *not* change the shape of the features (i.e., the output shape is the same as the input shape).
- `lm_head`, which is a linear layer that maps embeddings back to a distribution over the possible otuput tokens.

Task: Compute the number of parameters in the `wte` embedding layer of the GPT2 model. (For these and other questions that specifically mention GPT2 model, please use the config settings above and provide an actual numbers.)

```
# TODO: Perform this computation by hand.
```

Task: Compute the number of parameters in the `wtp` embedding layer of the GPT2 model.

```
# TODO: Perform this computation by hand.
```

Graded Task: Explain why the linear layer `lm_head` has the same number of parameters as the embedding layer `wte`. Provide an intuitive explanation for why **weight tying**—i.e., using the same set of weights for both layers, just transposed—would be reasonable. The weight tying is done to reduce the total number of parameters in the GPT2 model.

```
# TODO: Include your explanations here
```

Task: Explain why it is that in the `forward()` method, the tensor `tok_emb` has the shape `(b, t, n_embd)`, where `b` is the batch size, `t` is the sequence length (max `block_size`), and `n_embd` is the embedding size.

```
# TODO: Include your explanations here
```

Task: Notice that in the `forward()` method, the tensor `pos_emb` has the shape `(t, n_embd)`. In other words, we embed the position only once for each batch, and then rely on PyTorch tensor broadcasting to perform the addition `tok_emb + pos_emb`. Why is this ok?

```
# TODO: Include your explanations here
```

Task: What is the shape of `tok_emb + pos_emb` in the `forward()` method in a GPT2 model? This question is not trivial because the two addend tensors are not of the same shape. Thus, the addition uses broadcasting. PyTorch broadcasting works similarly to that of Numpy's. You can look up "PyTorch broadcasting" to find resources related to how broadcasting works.

```
# TODO: Perform this computation by hand.
```

Graded Task: What is the shape of `x` in the `forward()` method? This is an important shape to remember, since it is the shape of the feature map consistent in most of the transformer network.

```
# TODO: Perform this computation by hand.
```

Task: What is the shape of `logits` in the `forward()` method?

```
# TODO: Perform this computation by hand.
```

These questions above should give you a clear idea of the main components of the transformer model, the expected input and output tensor shapes, and the shapes of intermediate tensors. With this in mind, let's explore the two modules referenced by `GPT`.

We'll start with the simple one. The LayerNorm layer is intended to be similar to PyTorch's LayerNorm layer.

```
class LayerNorm(nn.Module):
    """ LayerNorm but with an optional bias. PyTorch doesn't support simply bias=False """

    def __init__(self, ndim, bias):
        super().__init__()
        self.weight = nn.Parameter(torch.ones(ndim))
        self.bias = nn.Parameter(torch.zeros(ndim)) if bias else None

    def forward(self, input):
        return F.layer_norm(input, self.weight.shape, self.weight, self.bias, 1e-5)
```

Task: How many parameters are in a LayerNorm layer?

```
# TODO: Perform this computation by hand.
```

Task: Read the description of the LayerNorm layer in PyTorch at https://pytorch.org/docs/stable/generated/torch.nn.LayerNorm.html#torch.nn.LayerNorm. Then, explain how layer normalization differs from batch normalization.

```
# TODO: Perform this computation by hand.
```

Let's move on to the `Block` module. Recall that here are several `Block` modules in a GPT model, and the output of one module is the input of the next.

```
class Block(nn.Module):

    def __init__(self, config):
        super().__init__()
        self.ln_1 = LayerNorm(config.n_embd, bias=config.bias)
        self.attn = CausalSelfAttention(config)
        self.ln_2 = LayerNorm(config.n_embd, bias=config.bias)
        self.mlp = MLP(config)

    def forward(self, x):
        x = x + self.attn(self.ln_1(x))
        x = x + self.mlp(self.ln_2(x))
        return x
```

This module is actually quite succinct, but it also refers to modules that are yet to be defined. It consists of:

- A layer normalization layer.
- A **causal self attention** layer (to be defined). This is the heart of the GPT model.
- Another layer normalization layer.
- An MLP layer (to be defined).

Task: Judging by the `Block.forward()` method above, why must the `CausalSelfAttention` and the `MLP` layers **preserve the shape of the features**?

```
# TODO: Include your explanation here.
```

Task: How might the *skip-connections* in the `Block.forward()` method help with gradient flow? An intuitive explanation is sufficient here.

```
# TODO: Include your explanation here.
```

With the GPT2 `Block` in mind, we will define the `MLP` module next.

```
class MLP(nn.Module):

    def __init__(self, config):
        super().__init__()
        self.c_fc    = nn.Linear(config.n_embd, 4 * config.n_embd, bias=config.bias)
        self.gelu    = nn.GELU()
        self.c_proj  = nn.Linear(4 * config.n_embd, config.n_embd, bias=config.bias)
        self.dropout = nn.Dropout(config.dropout)

    def forward(self, x):
        x = self.c_fc(x)
        x = self.gelu(x)
        x = self.c_proj(x)
        x = self.dropout(x)
        return x
```

Immediately, we see that this MLP consists of two linear layers. The activation function used between these two layers is the Gaussian Error Linear Units function. You can read more about it in this paper.

Task: Compute the number of parameters in a `MLP` layer in a GPT2 model.

```
# TODO: Perform this computation by hand
```

Task: Recall that the input of the MLP layer is a tensor with the usual dimension computed earlier. What is the shape of `self.c_fc(x)` in the `MLP.forward()` method? What about the shape of the return value in this method?

```
# TODO: Perform this computation by hand
```

Task: Explain why this MLP layer is also called the "pointwise feed forward" layer. (Hint: a "point" here refers to a single token or position in the input sequence)

```
# TODO: Include your explanation here.
```

Finally, let's study the definition of the `CausalSelfAttention` layer. This is the heart of the GPT model and is also the most complex module.

Task: Begin with a first pass read of the `__init__()` and `forward()` methods of `CausalSelfAttention` module. We will then trace through the case where `self.flash` is `False`, since the code provides more detailed explanation for the computation steps.

```
class CausalSelfAttention(nn.Module):

    def __init__(self, config):
        super().__init__()
        assert config.n_embd % config.n_head == 0
        # key, query, value projections for all heads, but in a batch
        self.c_attn = nn.Linear(config.n_embd, 3 * config.n_embd, bias=config.bias)
        # output projection
        self.c_proj = nn.Linear(config.n_embd, config.n_embd, bias=config.bias)
        # regularization
        self.attn_dropout = nn.Dropout(config.dropout)
        self.resid_dropout = nn.Dropout(config.dropout)
        self.n_head = config.n_head
        self.n_embd = config.n_embd
        self.dropout = config.dropout
        # flash attention make GPU go brrrrr but support is only in PyTorch >= 2.0
        self.flash = hasattr(torch.nn.functional, 'scaled_dot_product_attention')
        if not self.flash:
            print("WARNING: using slow attention. Flash Attention requires PyTorch >= 2.0")
            # causal mask to ensure that attention is only applied to the left in the input s
            self.register_buffer("bias", torch.tril(torch.ones(config.block_size, config.bloc
                                        .view(1, 1, config.block_size, config.block_size))

    def forward(self, x):
        B, T, C = x.size() # batch size, sequence length, embedding dimensionality (n_embd)

        # calculate query, key, values for all heads in batch and move head forward to be the
        q, k, v  = self.c_attn(x).split(self.n_embd, dim=2)
        k = k.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
        q = q.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
        v = v.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)

        # causal self-attention; Self-attend: (B, nh, T, hs) x (B, nh, hs, T) -> (B, nh, T, T
        if self.flash:
            # efficient attention using Flash Attention CUDA kernels
            y = torch.nn.functional.scaled_dot_product_attention(q, k, v, attn_mask=None, dro
```

```
        else:
            # manual implementation of attention
            att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))
            att = att.masked_fill(self.bias[:,:,:T,:T] == 0, float('-inf'))
            att = F.softmax(att, dim=-1)
            att = self.attn_dropout(att)
            y = att @ v # (B, nh, T, T) x (B, nh, T, hs) -> (B, nh, T, hs)
        y = y.transpose(1, 2).contiguous().view(B, T, C) # re-assemble all head outputs side

        # output projection
        y = self.resid_dropout(self.c_proj(y))
        return y
```

Task: Compute the number of parameters in the `c_attn` layer in a GPT2 model.

```
# TODO: Perform this computation by hand
```

Task: Like the comment in the `__init__()` method suggests, we can think of the `c_attn` layer as a combination of three `nn.Linear(config.n_embd, config.n_embd, bias=config.bias)` modules. These three networks projects the input embedding into three parts: `q` (for *query*), `k` (for *key*), and `v` (for *value).

What is the shape of `self.c_attn(x)` in the `forward()` method? Use this answer to show that `self.c_attn(x).split(self.n_embd, dim=2)` gives us the same `q, k, v` values had we used three separate networks.

```
# TODO: Perform the computation by hand, then include your explaination.
```

Task: Explain why `config.n_head` must be a factor of `config.n_embd`.

```
# TODO: Your explanation goes here.
```

We will explore the manual implementation of attention in the next few questions. For this part, it helps to first consider the case where the batch size `B=1`, and `n_head=1`. For a larger batch size and number of heads, the attention computation is repeated for every sequence in the batch and every attention head. Thus, the shapes of the three important tensors are:

- `q`: (1, 1, T, n_embd)
- `k`: (1, 1, T, n_embd)
- `v`: (1, 1, T, n_embd)

Like discussed in the lectures, you can think of the attention mechanism as a "soft" dictionary lookup. Instead of obtaining a single key/values for a given query, attention gives us a *probability distribution over the possible keys/values*. We can then use this probability distribution to obtain a weighted sum (akin to an expected value) of the lookup value. Moreover, instead of having strings, numbers, or other objects as keys/values, a key is a vector (of shape `n_embd`), and a value is also a vector (of shape `n_embd`). This is consistent with what we have seen in neural networks—everything is represented using a vector! The tensors `k` and `v` contains these keys and values, and there is one vector at every token position. The tensor `q` contains the **queries**— analogues to the item (a possible key) that we are searching for in a regular dictionary lookup. There is also one query vector for each token position: for each token position, we want to look up a corresponding (weighted sum of) values that contains information pertinent to understanding the meaning of the token in this position.

With that in mind, let's go through the mathematical computations.

Graded Task: What is the shape of `(q @ k.transpose(-2, -1))`? For this and the following questions, assume that `q`, `k`, `v` have the shape above, where we have assumed that batch size and num heads are both 1.

```
# TODO: Perform this computation by hand
# SOLUTION: (T, T)
```

Task: What is the value of math.sqrt(k.size(-1))?

```
# TODO: Perform this computation by hand
# SOLUTION: n_embed
```

Task: Argue that the line `att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))` is computing a "distance" or "similarity" metric between the query at each token position and the key at each token position.

```
# TODO: Your explanation goes here
```

The following line of code references a `self.bias` parameter, which is defined in the last line of the `__init__()` method. Since `block_size` is quite large, we can understand what `self.bias` looks like by running a similar piece of code below with a smaller `block_size` value.

```
bias = torch.tril(torch.ones(5, 5)).view(1, 1, 5, 5)
bias
```

Task: Explain what the above code returns. Explain how PyTorch broadcasting may be useful for computations involving this tensor—i.e., why is it okay that the first two dimensions of this tensor are 1, thus assuming that batch size = 1 and num heads = 1?

```
# TODO: Your explanation goes here
```

Task: We will use a similar technique of running a modified version of the next two lines of code in the `forward()` method to better understand what it does. Run the below code, and explain what the `masked_fill` function does.

```
attn = torch.rand(1, 1, 5, 5)
attn
```

```
masked = attn.masked_fill(bias[:,:,:5, :5] == 0, float('-inf'))
masked
```

```
out = F.softmax(masked, dim=-1)
out
```

```
# Your explanation goes here
```

Graded Task: This masking is in place so that query tokens cannot "look up" key/values that are at a position with a larger index. Explain why this limitation means our GPT model cannot use information in subsequent/later tokens to form an understand of what what is in the current token. (Note: this masking is the "Causal" part of Causal Self-Attention!)

```
# Your explanation goes here
```

Task: Your answer above explains which positions in the `out` tensor need to be set to zero. Explain why setting the corresponding value of pre-softmax tensor `masked` to `-inf` is necessary. Why can't we set the value of `masked` to 0 in these positions?

```
# Your argument goes here
```

Task: Argue that `out[0,0,0,0]` must always be 1.

```
# Your argument goes here
```

Task: Now, `out` in our example is akin to the final value of `att` in the `CausalSelfAttention.forward()` method. Explain why the operation `y = att @ v` computes a weighted sum of values at each token position, where the weights are defined by `att`.

```
# Your explanation goes here
```

Task: The above explanation pertain to a single attention head. Explain why using multiple attention heads allows a token position to consider information from various other positions. Alternatively, explain why using multiple heads might help the network learn different *ways* in which the meaning at one token could depend on other tokens.

```
# Your explanation goes here
```

Graded Task: Compute the total number of parameters in a GPT2 model by computing the following. Please use actual numbers in each case, assuming the GPT2 configuration from above.

1. The number of parameters in a `CausalSelfAttention` model.
2. The number of parameters in a `MLP` module.
3. The number of parameters in a `Block` module.
4. The number of parameters in all `Block` modules in a GPT2 model.
5. The number of parameters in the `wte` embedding layer in a GPT2 model.
6. The total number of parameters in a GPT2 model.

Please perform the computation either by hand (and show your work), or with a function that clearly shows the computations.

You should see that approximately 30% of the GPT2 weight comes from the `wte` embedding layer. This is why weight tying is used in the `GPT` module!

```
# TODO: Your work goes here

# SOLUTION THAT I GET:
# SOLUTION 1. 2360832
# SOLUTION 2. 4722432
# SOLUTION 3. 7086336
# SOLUTION 4. 85036032
# SOLUTION 5. 38633472
# SOLUTION 6. 124457472
```

Part 3. Training

We are ready to finetune our GPT2 model on the "Friends" data set! There is no graded task in this section since training this model can take some time to achieve reasonable performance.

To run this part of the lab, you will need to use a GPU. On Google Colab, you can select a session with a GPU by navigating to the "Runtime" menu, selecting "Change runtime type", and then selecting the "T4 GPU" option.

We will set up a `config` object to make it easier to store and use configs.

```
import easydict
import math
import time

finetune_config_dict = {
  'gradient_accumulation_steps': 32,
  'block_size': 256,
  'dropout': 0.2,
  'bias': False,
  'learning_rate': 3e-5,
  'weight_decay': 0.1,
  'beta1': 0.9,
  'beta2': 0.99,
  'grad_clip': 1.0,
  'decay_lr': False,
  'warmup_iters': 100,
  'lr_decay_iters': 5000,
  'min_lr': 0.0001}
config = easydict.EasyDict(finetune_config_dict)
```

First, we need to load the GPT2 weights.

```
# initialize from OpenAI GPT-2 weights
override_args = dict(dropout=config.dropout)
model = GPT.from_pretrained('gpt2', override_args)

# crop down the model block size using model surgery
if config.block_size < model.config.block_size:
    model.crop_block_size(config.block_size)

device = 'cuda' if torch.cuda.is_available()  else 'cpu'
model.to(device)
```

Task: Explain why reducing the `block_size` do not significantly reduce the number of parameters, but *does* significantly reduce memory usage.

```
# TODO: Include your explanation here
```

There are some additional helpers to improve training.

```
# initialize a GradScaler. If enabled=False scaler is a no-op
dtype = 'bfloat16' if torch.cuda.is_available() and torch.cuda.is_bf16_supported() else 'floa
ptdtype = {'float32': torch.float32, 'bfloat16': torch.bfloat16, 'float16': torch.float16}[dt
ctx = nullcontext() if device == 'cpu' else torch.amp.autocast(device_type=device, dtype=ptdt
scaler = torch.cuda.amp.GradScaler(enabled=(dtype == 'float16'))

# learning rate decay scheduler (cosine with warmup)
def get_lr(config, it):
    # 1) linear warmup for warmup_iters steps
    if it < config.warmup_iters:
        return config.learning_rate * it / config.warmup_iters
    # 2) if it > lr_decay_iters, return min learning rate
    if it > config.lr_decay_iters:
        return config.min_lr
    # 3) in between, use cosine decay down to min learning rate
    decay_ratio = (it - config.warmup_iters) / (config.lr_decay_iters - config.warmup_iters)
    assert 0 <= decay_ratio <= 1
    coeff = 0.5 * (1.0 + math.cos(math.pi * decay_ratio)) # coeff ranges 0..1
    return config.min_lr + coeff * (config.learning_rate - config.min_lr)

# helps estimate an arbitrarily accurate loss over either split using many batches
@torch.no_grad()
def estimate_loss(model, train_dataset, val_dataset, block_size):
    out = {}
    model.eval()
    for split in ['train', 'val']:
        losses = torch.zeros(eval_iters)
        dataset = train_dataset if split == 'train' else val_dataset
        for k in range(eval_iters):
            X, Y = get_batch(dataset, block_size, batch_size, device)
            with ctx:
                logits, loss = model(X, Y)
            losses[k] = loss.item()
        out[split] = losses.mean()
    model.train()
```

```
    return out
```

Now we can begin the training loop. You may need to increase `max_iter` to obtain good results.

```
iter_num = 0
best_val_loss = 1e9
eval_interval = 10
log_interval = 10
eval_iters = 40
max_iters = 500
batch_size = 1

# optimizer
optimizer = model.configure_optimizers(config.weight_decay, config.learning_rate,
   (config.beta1, config.beta2), device)

# training loop
X, Y = get_batch(train_data, config.block_size, batch_size, device) # fetch the very first ba
t0 = time.time()
local_iter_num = 0 # number of iterations in the lifetime of this process
raw_model = model # unwrap DDP container if needed
running_mfu = -1.0
while True:
    # determine and set the learning rate for this iteration
    lr = get_lr(config, iter_num) if config.decay_lr else config.learning_rate
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr

    # evaluate the loss on train/val sets and write checkpoints
    if iter_num % eval_interval == 0:
        losses = estimate_loss(model, train_data, val_data, config.block_size)
        print(f"step {iter_num}: train loss {losses['train']:.4f}, val loss {losses['val']:.4

    # forward backward update, with optional gradient accumulation to simulate larger batch s
    # and using the GradScaler if data type is float16
    for micro_step in range(config.gradient_accumulation_steps):
        with ctx:
            logits, loss = model(X, Y)
            loss = loss / config.gradient_accumulation_steps # scale the loss to account for
        # immediately async prefetch next batch while model is doing the forward pass on the
        X, Y = get_batch(train_data, config.block_size, batch_size, device)
```

```
        # backward pass, with gradient scaling if training in fp16
        scaler.scale(loss).backward()

    # clip the gradient
    if config.grad_clip != 0.0:
        scaler.unscale_(optimizer)
        torch.nn.utils.clip_grad_norm_(model.parameters(), config.grad_clip)

    # step the optimizer and scaler if training in fp16
    scaler.step(optimizer)
    scaler.update()

    # flush the gradients as soon as we can, no need for this memory anymore
    optimizer.zero_grad(set_to_none=True)

    # timing and logging
    t1 = time.time()
    dt = t1 - t0
    t0 = t1
    if iter_num % log_interval == 0:
        # get loss as float. note: this is a CPU-GPU sync point
        # scale up to undo the division above, approximating the true total loss (exact would
        lossf = loss.item() * config.gradient_accumulation_steps
        if local_iter_num >= 5: # let the training loop settle a bit
            mfu = raw_model.estimate_mfu(batch_size * config.gradient_accumulation_steps, dt)
            running_mfu = mfu if running_mfu == -1.0 else 0.9*running_mfu + 0.1*mfu
        print(f"iter {iter_num}: loss {lossf:.4f}, time {dt*1000:.2f}ms, mfu {running_mfu*100

    iter_num += 1
    local_iter_num += 1

    # termination conditions
    if iter_num > max_iters:
        break
```

Here is some code you can use to generate a sequence using the fine-tuned GPT2 model.

```
init_from = 'resume' # either 'resume' (from an out_dir) or a gpt2 variant (e.g. 'gpt2-xl')
out_dir = 'out' # ignored if init_from is not 'resume'
start = "\n" # or "<|endoftext|>" or etc. Can also specify a file, use as: "FILE:prompt.txt"
num_samples = 10 # number of samples to draw
max_new_tokens = 500 # number of tokens generated in each sample
```

```
temperature = 0.8 # 1.0 = no change, < 1.0 = less random, > 1.0 = more random, in predictions
top_k = 200 # retain only the top_k most likely tokens, clamp others to have 0 probability


enc = tiktoken.get_encoding("gpt2")
encode = lambda s: enc.encode(s, allowed_special={"<|endoftext|>"})
decode = lambda l: enc.decode(l)

start_ids = encode(start)
x = (torch.tensor(start_ids, dtype=torch.long, device=device)[None, ...])

# model = finetuned_model
# run generation
with torch.no_grad():
    with ctx:
        for k in range(num_samples):
            y = model.generate(x, max_new_tokens, temperature=temperature, top_k=top_k)
            print(decode(y[0].tolist()))
            print('---------------')
```