Unsupervised Pre-Training of GPT-Style Model

In today's notebook, we'll be working through an example of how to do unsupervised pre-training of a GPT-style model.

The base model we'll use is Andrej Karpathy's nanoGPT.

All of the model code can be found in the `model.py` file!

> NOTE: We will not be leveraging the parallized training strategy in this notebook - you can find all the required code in the provided repository.

Data Selection

For the notebook today, we'll be using a toy dataset called `tinyshakespeare`. Feel free to use your own corpus here, just make sure it's contained within a single `.txt` file.

You could extend this example to use the OpenWebText dataset, which was used to pre-train GPT-2.

> NOTE: Training LLMs can take a very long time - in order to get results similar to the GPT-2 paper you will need 8xA100s and train for ~4-5 days using a pararellized strategy (DDP) on the OpenWebText Corpus.

Let's start by grabbing our source repository for the day!

```
!git clone https://github.com/karpathy/nanoGPT.git
```

```
    Cloning into 'nanoGPT'...
    remote: Enumerating objects: 649, done.
    remote: Total 649 (delta 0), reused 0 (delta 0), pack-reused 649
    Receiving objects: 100% (649/649), 935.48 KiB | 17.65 MiB/s, done.
    Resolving deltas: 100% (373/373), done.
```

Next, we'll need to grab some dependencies.

`cohere` and `openai` are recent dependencies of `tiktoken`, but we will not be leveraging them today.

```
!pip install tiktoken requests cohere openai -qU
```

```
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.0/2.0 MB 21.8 MB/s eta 0:00:00
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 48.2/48.2 kB 6.0 MB/s eta 0:00:00
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 220.2/220.2 kB 22.7 MB/s eta 0:00:00
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.7/2.7 MB 58.9 MB/s eta 0:00:00
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 75.0/75.0 kB 9.8 MB/s eta 0:00:00
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 76.9/76.9 kB 7.8 MB/s eta 0:00:00
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 58.3/58.3 kB 6.3 MB/s eta 0:00:00
```

First things first - let's download our dataset!

We'll leverage the `requests` library to do this - and then we will split our resultant data into a `train` and `val` set. We want ~90% of our data to be training, and ~10% to be validation.

```
import os
import requests
import tiktoken
import numpy as np

current_path = "/data/shakespeare"
data_url = 'https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt'

if not os.path.exists(current_path):
    os.makedirs(current_path)

# download the tiny shakespeare dataset
input_file_path = os.path.join(os.path.dirname(current_path), 'input.txt')
if not os.path.exists(input_file_path):

    with open(input_file_path, 'w') as f:
        f.write(requests.get(data_url).text)

with open(input_file_path, 'r') as f:
    data = f.read()

n = len(data)
train_data = data[:round(n*0.9)]
val_data = data[round(n*0.9):]
```

```
print(len(train_data), len(val_data))
```

```
    1003855 111539
```

Now let's get our `tokenizers` dependency so we can train a tokenizer on our data.

```
!pip install tokenizers -qU
```

```
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.8/3.8 MB 23.3 MB/s eta 0:00:00
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 311.1/311.1 kB 20.3 MB/s eta 0:00:00
```

We will be training a "byte-pair-encoding" or "BPE" tokenizer. If you'd like to read more, you can find it here.

Let's work through an example of what Byte-Pair Encoding (BPE) is doing, exactly, from this wonderful example provided by Hugging Face.

What is BPE?

First, we need to do a step called "pre-tokenization", which is - as it sounds - a tokenization step that occurs before we tokenize.

The essential idea of BPE is that we need to understand common words and "byte-pairs" in them. So, in order to find "common words" we first need to find...words!

Let's take the following text and break it apart into its word components.

```
 After pre-tokenization, a set of unique words has been created and the frequency with which each word occurred in the training data h
```

A naive way to do this would just be by splitting on spaces...and that is indeed what technique was used in GPT-2.

```
input_text = """
After pre-tokenization, a set of unique words has been created and the frequency with which each word occurred in the training da
"""

naive_word_list = input_text.split()
```

```
print("starting by splitting by space:")
naive_word_list[:10]
```

```
    starting by splitting by space:
    ['After',
     'pre-tokenization,',
     'a',
     'set',
     'of',
     'unique',
     'words',
     'has',
     'been',
     'created']
```

Now we can count our words and get their frequency.

```
from collections import defaultdict

vocab_and_frequencies = defaultdict(int)

for word in naive_word_list:
  vocab_and_frequencies[" ".join(list(word))] += 1

list(vocab_and_frequencies.keys())[:5]
```

```
    ['A f t e r', 'p r e - t o k e n i z a t i o n ,', 'a', 's e t', 'o f']
```

```
print("we get the frequency for each of the initial tokens :")
sorted(vocab_and_frequencies.items(), key = lambda x: x[1], reverse=True)[:5]
```

```
    we get the frequency for each of the initial tokens :
    [('t h e', 8), ('a', 4), ('o f', 4), ('v o c a b u l a r y', 4), ('h a s', 3)]
```

Let's find our "base vocabulary", which is going to be each symbol present in our original dataset.

```
from typing import Dict, Tuple, List, Set

def find_vocabulary_size(current_vocab: Dict[str, int]) -> int:
  vocab = set()

  for word in current_vocab.keys():
    for subword in word.split():
      vocab.add(subword)

  return vocab, len(vocab)
```

```
vocab, vocab_length = find_vocabulary_size(vocab_and_frequencies)
```

```
print(vocab)
```

```
    {'o', ',', 'e', 's', 'w', 'a', 'u', 'p', 'l', 't', 'n', 'r', 'N', 'I', '-', 'P', 'h', 'm', 'B', 'b', 'v', 'f', 'd', 'c', 'E'
```

As we can see, there are ~34 symbols in our base vocabulary. Let's convert our data into a form where we can capture each symbol separately.

Now we can start constructing our pairs. We will look at all the pairs of symbols as they appear and take into consideration their frequency in our corpus.

```
def find_pairs_and_frequencies(current_vocab: Dict[str, int]) -> Dict[str, int]:
  pairs = {}

  for word, frequency in current_vocab.items():
    symbols = word.split()

    for i in range(len(symbols) - 1):
      pair = (symbols[i], symbols[i + 1])
      current_frequency = pairs.get(pair, 0)
      pairs[pair] = current_frequency + frequency

  return pairs
```

```
pairs_and_frequencies = find_pairs_and_frequencies(vocab_and_frequencies)
```

```
sorted(pairs_and_frequencies.items(), key = lambda x: x[1], reverse=True)[:5]
```

```
    [(('t', 'h'), 11),
     (('i', 'n'), 10),
     (('r', 'e'), 8),
     (('h', 'e'), 8),
     (('a', 't'), 7)]
```

Now that we have the frequent pairs - we can merge those pairs into a single token.

Let's see how this process looks in code.

```
import re

def merge_vocab(most_common_pair: Tuple[str], current_vocab: Dict[str, int]) -> Dict[str, int]:
  vocab_out = {}

  pattern = re.escape(' '.join(most_common_pair))
  replacement = ''.join(most_common_pair)

  for word_in in current_vocab:
      word_out = re.sub(pattern, replacement, word_in)
      vocab_out[word_out] = current_vocab[word_in]
  return vocab_out
```

For instance

```
sorted(pairs_and_frequencies.items(), key = lambda x: x[1], reverse=True)[0][0],
```

```
    (('t', 'h'),)
```

```
new_vocab_and_frequencies = merge_vocab(
    sorted(pairs_and_frequencies.items(), key = lambda x: x[1], reverse=True)[0][0],
    vocab_and_frequencies
)
```

```
sorted(new_vocab_and_frequencies.items(), key = lambda x: x[1], reverse=True)[:5]
```

```
    [('th e', 8), ('a', 4), ('o f', 4), ('v o c a b u l a r y', 4), ('h a s', 3)]
```

```
new_vocab, new_vocab_length = find_vocabulary_size(new_vocab_and_frequencies)
```

After one merge, we can see that `t h` has been converted to `th`!

Let's see how that impacted our vocabulary.

```
new_vocab_length
```

```
    35
```

We can see that our vocabulary has increased by 1 as we've added the `th` symbol to it!

In essence, BPE will continue to do this process until your desired vocabulary size (a hyper-parameter) is met!

Training Our Tokenizer

Now that we have some background on how BBPE works, lets move on to training our tokenizer for our model!

Let's walk through the steps we'll take:

1. Initialize our `Tokenizer` with a `BPE` model. Be sure to include the `unk_token`.
   - `Tokenizer`
   - `Models`
2. We'll include a normalizer, applied at the sequence level, and we'll use `NFD()` to do so. More reading on Unicode Normalization Forms here.
   - `NFD()`
3. We'll also add our `ByteLevel()` pre-tokenizer, and our `ByteLevelDecoder()` decoder.
   - `ByteLevel()`
   - `ByteLevelDecoder()`

```
from tokenizers import Tokenizer
from tokenizers.models import BPE
from tokenizers.decoders import ByteLevel as ByteLevelDecoder
from tokenizers.normalizers import NFD, Sequence
from tokenizers.trainers import BpeTrainer
from tokenizers.pre_tokenizers import ByteLevel

tokenizer = Tokenizer(BPE(unk_tokwn="[UNK]"))
tokenizer.normalizer = Sequence([
    NFD()
    ])
tokenizer.pre_tokenizer = ByteLevel()
tokenizer.decoder = ByteLevelDecoder()
```

We'll want to add some special tokens to our tokenizer to ensure in has access to common token patterns.

Let's use the following:

- `"<s>"` : bos_token - beginning of sequence token
- `"</s>"` : eos_token - end of sequence token
- `"<pad>"` : padding_token - token used to pad sequences
- `"<unk>"` : unk_token - token used to represent unknown tokens.
- `"<mask>"` : mask_token - token used to mask parts of our sequence

We're also going to set a target vocabulary of 50,000 tokens.

```
trainer = BpeTrainer(
    vocab_size=50000,
    show_progress=True,
    special_tokens=[
      "<s>",
      "<pad>",
      "</s>",
      "<unk>",
      "<mask>",
    ]
)
```

Nothing left to do but point it at our data-source and let it train!

We'll use the `.train()` method to accomplish this task.

> NOTE: Pay attention to the desired inputs of the `.train()` method.

- [`Tokenizer.train()`]

```
tokenizer.train(
    files=[input_file_path],
    trainer=trainer
)
```

Now we can save our tokenizer - and then load it as a `GPT2Tokenizer` through the Hugging Face Library!

```
save_path = '/content/tokenizer'
if not os.path.exists(save_path):
    os.makedirs(save_path)
tokenizer.model.save(save_path)
```

```
['/content/tokenizer/vocab.json', '/content/tokenizer/merges.txt']
```

```
!pip install transformers -qU
```

```
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.9/7.9 MB 69.7 MB/s eta 0:00:00
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.8/3.8 MB 106.0 MB/s eta 0:00:00
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.3/1.3 MB 71.7 MB/s eta 0:00:00
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 295.0/295.0 kB 31.9 MB/s eta 0:00:00
```

```
from transformers import GPT2Tokenizer

tokenizer = GPT2Tokenizer.from_pretrained(save_path, unk_token="[UNK]")
```

Let's see how it tokenizes our inputs!

```
input_sentence = "Hark, my name be Romeo! I am but a beautiful summer's day!"
```

```
tokenized_sentence = tokenizer.tokenize(input_sentence)
tokenized_sentence
```

```
['Hark',
 ',',
 'Ġmy',
 'Ġname',
 'Ġbe',
 'ĠRomeo',
 '!',
 'ĠI',
 'Ġam',
 'Ġbut',
 'Ġa',
 'Ġbeautiful',
 'Ġsummer',
 "'s",
 'Ġday',
 '!']
```

```
encoded_tokens = tokenizer.convert_tokens_to_ids(tokenized_sentence)
encoded_tokens
```

```
[12077, 9, 124, 637, 121, 826, 5, 87, 295, 219, 72, 9113, 2999, 141, 511, 5]
```

We can decode the sequence back to a text

```
decoded_tokens = tokenizer.decode(encoded_tokens, clean_up_tokenization_spaces=False)
decoded_tokens
```

```
'Hark, my name be Romeo! I am but a beautiful summer's day!'
```

Tokenizing Dataset

Now that we have trained our tokenizer - let's create a dataset we can leverage with the `nanoGPT` library.

We'll simply encode our training and validation data - and then save them in binary files for later!

> NOTE: Pay attention to the format you want your dataset in. We want ids, which means we want to use the `.encode()` method of our tokenizer.

```
train_ids = tokenizer.encode(train_data)
val_ids = tokenizer.encode(val_data)
print(f"train has {len(train_ids):,} tokens")
print(f"val has {len(val_ids):,} tokens")
```

```
train has 291,285 tokens
val has 34,222 tokens
```

```
# export to bin files
data_path = "/data/shakespeare/"

train_ids = np.array(train_ids, dtype=np.uint16)
val_ids = np.array(val_ids, dtype=np.uint16)
train_ids.tofile(os.path.join(os.path.dirname(data_path), 'train.bin'))
val_ids.tofile(os.path.join(os.path.dirname(data_path), 'val.bin'))
```

Training The Model

Now that we have our tokenized dataset, let's get to training our model!

We have a lot of set-up to do before we click "`.train()`", so let's jump right into it!

First, let's literally jump into the `nanoGPT` repository we cloned earlier.

```
%cd nanoGPT
```

```
/content/nanoGPT
```

We'll do some critical imports.

```
import os
import time
import math
import pickle
from contextlib import nullcontext

import numpy as np
import torch
```

```
# from the local repo
from model import GPTConfig, GPT
```

Hyper-Parameters

We have a laundry list of hyper-parameters to set up - let's walk through them and what they mean.

I/O

- `out_dir` - simple enough, this is the output directory where our checkpoints are saved

```
out_dir = 'out'
```

Initialization

Since we're training from scratch, we'll use `init_from = 'scratch'`.

```
init_from = 'scratch'
```

Eval and Logging

- `eval_interval` - this is the number of steps between evaluation stages, we'll want to see this ~ `250`. Our model will be incredibly prone to over-fitting, and this will let us monitor with relative frequency.
- `log_interval` - this is how often our training progress will log. You can set this ~ `10`. It's dealer's choice, really.
- `eval_iters` - this is how *many* iterations we want to evaluate for.
- `eval_only` - this would evaluate our model - but not train it. We'll leave this as `False` for now.
- `always_save_checkpoint` - this will always save our most recent checkpoint, regardless of metrics. For this example, we'll set this to `True`.

```
eval_interval = 250
eval_iters = 200
log_interval = 10
eval_only = False
always_save_checkpoint = True
```

Dataset

We can set our dataset here - we'll use the one we created earlier!

```
dataset = 'shakespeare'
```

Typical Hyper-Parameters

- `gradient_accumulation_steps` - we can use gradient accumulation to "simulate" larger batch sizes by combining multiple different optimization steps together, without needing the additional memory for large batch sizes. We don't need to worry so much about this for the toy problem - but this hyper-parameter can be configured for larger training runs. Here is some great reading on the topic.
- `batch_size` - Typical batch_size - the larger the merrier (up to a point) we'll be using `16` to ensure we do not exceed the memory quota of our GPU.
- `block_size` - this can be thought of as another term for the `context window` of our model. Since our model cannot take variable length inputs - we use this to set all inputs to our desired size. We'll use a value of `512` to ensure speedy training.

```
gradient_accumulation_steps = 1
batch_size = 16
block_size = 512 # another synonym for context window
```

Model Architecture

- `n_layer` - this is the number of decoder layers we will use in our model. More would be considered better (up to a point) and the original GPT-2 paper uses `12`, but we will be using a truncated `6` for ease and speed of training.
- `n_head` - this is the number of attention heads in each decoder layer!

- `n_embd` - this is the embedding dimension of our model, this is analagous to our `model_d` from the previous notebook. A default value of ~ `500` should do the trick!
- `dropout` - this sets our dropout value, since our model is small and going to be extremely prone to overfitting, consider setting this at a fairly aggresive level (`0.2` was used in the example training found in the notebook`).
- `bias` - wether or not to use bias inside the LayerNorm/Linear layers.

```
n_layer = 6
n_head = 6 # hence 6 * 6 = 36 total heads
n_embd = 516
dropout = 0.2
bias = False
```

❓ QUESTION:

What condition must be true as it relates to the `n_embd` and `n_head`?

`ANSWER`: You need to make sure we can nicely divide `n_embd` (size of our inputs) by `n_heads` (count of attention heads per layer) so that we can split our inputs.

Optimizer Hyper-Parameters

Basic Optimizer Hyper-Parameters:

- `learning_rate` - it's our learning rate! We'll want to set this fairly high ~ `1e-3` since we're training on such a small dataset.
- `max_iters` - how many iterations do we train for. More iters means longer training times. Feel free to tinker with this value! `5000` is a great place to start.

Learning Rate Decay Settings:

- `decay_lr` - set decay flag
- `weight_Decay` - how much to decay lr by
- `lr_decay_iters` - should be set to ~max_iters.
- `min_lr` - the minimum lr, should be ~ lr / 10

Clipping and Warmup:

- `grad_clip` - value to clip gradients to. useful for preventing vanishing gradients.
- `warmup_iters` - how many iterations to warmup for. Warmup is useful to allow your training to slowly warmup. It will use a low lr for a number of steps to avoid any massive initial spikes. Since we're training a very small model - we can avoid using many wamrup steps.

> NOTE: Many learnings taken from the Chincilla paper for selecting default or appropriate values.

```
# adamw optimizer
learning_rate = 1e-3
max_iters = 5_000
beta1 = 0.9
beta2 = 0.99

# lr decay settings
decay_lr = True
weight_decay = 1e-1
lr_decay_iters = 5_000 # ~= max_iters per Chinchilla
min_lr = 1e-4 # ~= learning_rate/10 per Chinchilla

# clipping and warmup
grad_clip = 1.0
warmup_iters = 100
```

❓ QUESTION:

Given a Learning Rate of `1e-4` and a maximum iteration cap of `10,000`: What should `lr_decay_iters` be, and what should `min_lr` be?

- `lr_decay_iters` should be approx. `max_iters` (
- `min_lr` should be `learning_rate`/10 to use a 10× learning rate decay (see https://arxiv.org/pdf/2203.15556.pdf)

These hyper-parameters are necessary to set given the task we're training and given the environment we're training in.

```
backend = 'nccl'
device = 'cuda'
dtype = 'bfloat16' if torch.cuda.is_available() and torch.cuda.is_bf16_supported() else 'float16'
compile = True
# -----------------------------------------------------------------------------
config_keys = [k for k,v in globals().items() if not k.startswith('_') and isinstance(v, (int, float, bool, str))]
config = {k: globals()[k] for k in config_keys}
# -----------------------------------------------------------------------------
master_process = True
seed_offset = 0
ddp_world_size = 1
tokens_per_iter = gradient_accumulation_steps * ddp_world_size * batch_size * block_size
print(f"tokens per iteration will be: {tokens_per_iter:,}")
os.makedirs(out_dir, exist_ok=True)
```

```
    tokens per iteration will be: 8,192
```

Torch Settings

We need to set a few `torch` settings, including the seed, to allow us to train correctly on our GPU.

Not much is required for us to understand here - these are just necessary lines of code. Boilerplate.

```
torch.manual_seed(1337 + seed_offset)
torch.backends.cuda.matmul.allow_tf32 = True # allow tf32 on matmul
torch.backends.cudnn.allow_tf32 = True # allow tf32 on cudnn
device_type = 'cuda' if 'cuda' in device else 'cpu'
ptdtype = {'float32': torch.float32, 'bfloat16': torch.bfloat16, 'float16': torch.float16}[dtype]
ctx = nullcontext() if device_type == 'cpu' else torch.amp.autocast(device_type=device_type, dtype=ptdtype)
```

Dataloader

This block will:

1. Set the data path
2. Load the dataset we tokenized earlier from the `.bin` we saved
3. Define a `get_batch` function that will return us a random section of our data as well as a the corresponding "label" for that data and move it to the GPU for easy use inside our training loop.

```
data_dir = os.path.join('/data', dataset)
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')

def get_batch(split):
    data = train_data if split == 'train' else val_data
    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])
    y = torch.stack([torch.from_numpy((data[i+1:i+1+block_size]).astype(np.int64)) for i in ix])
    if device_type == 'cuda':
        # pin arrays x,y, which allows us to move them to GPU asynchronously (non_blocking=True)
        x, y = x.pin_memory().to(device, non_blocking=True), y.pin_memory().to(device, non_blocking=True)
    else:
        x, y = x.to(device), y.to(device)
    return x, y
```

? Question:

What can you tell us about the way the labels are generated?

Please produce an example of a single x and y pair.

```
sample_data = val_data.copy()
ix = torch.randint(len(sample_data) - block_size, (batch_size,))
x = torch.stack([torch.from_numpy((sample_data[i:i+block_size]).astype(np.int64)) for i in ix])
y = torch.stack([torch.from_numpy((sample_data[i+1:i+1+block_size]).astype(np.int64)) for i in ix])
```

```
i=0
x = tokenizer.decode(sample_data[i:i+block_size], clean_up_tokenization_spaces=False)
y = tokenizer.decode(sample_data[i+1:i+1+block_size], clean_up_tokenization_spaces=False)
```

```
print(x)
```

```
    PETRUCHIO:
    And you, good sir! Pray, have you not a daughter
    Call'd Katharina, fair and virtuous?

    BAPTISTA:
    I have a daughter, sir, called Katharina.

    GREMIO:
    You are too blunt: go to it orderly.

    PETRUCHIO:
    You wrong me, Signior Gremio: give me leave.
    I am a gentleman of Verona, sir,
    That, hearing of her beauty and her wit,
    Her affability and bashful modesty,
    Her wondrous qualities and mild behavior,
    Am bold to show myself a forward guest
    Within your house, to make mine eye the witness
    Of that report which I so oft have heard.
    And, for an entrance to my entertainment,
    I do present you with a man of mine,
    Cunning in music and the mathematics,
    To instruct her fully in those sciences,
    Whereof I know she is not ignorant:
    Accept of him, or else you do me wrong:
    His name is Licio, born in Mantua.

    BAPTISTA:
    You're welcome, sir; and he, for your good sake.
    But for my daughter Katharina, this I know,
    She is not for your turn, the more my grief.

    PETRUCHIO:
    I see you do not mean to part with her,
    Or else you like not of my company.

    BAPTISTA:
    Mistake me not; I speak but as I find.
    Whence are you, sir? what may I call your name?

    PETRUCHIO:
    Petruchio is my name; Antonio's son,
    A man well known throughout all Italy.

    BAPTISTA:
    I know him well: you are welcome for his sake.

    GREMIO:
    Saving your tale, Petruchio, I pray,
    Let us, that are poor petitioners, speak too:
    Baccare! you are marvellous forward.

    PETRUCHIO:
    O, pardon me, Signior Gremio; I would fain be doing.

    GREMIO:
    I doubt it not, sir; but you will curse
```

```
print(y)
```

```
    PETRUCHIO:
    And you, good sir! Pray, have you not a daughter
    Call'd Katharina, fair and virtuous?

    BAPTISTA:
    I have a daughter, sir, called Katharina.
    Her affability and bashful modesty,
    Her wondrous qualities and mild behavior,
    Am bold to show myself a forward guest
    Within your house, to make mine eye the witness
    Of that report which I so oft have heard.
    And, for an entrance to my entertainment,
    I do present you with a man of mine,
    Cunning in music and the mathematics,
    To instruct her fully in those sciences,
    Whereof I know she is not ignorant:
    Accept of him, or else you do me wrong:
    His name is Licio, born in Mantua.

    BAPTISTA:
    You're welcome, sir; and he, for your good sake.
    But for my daughter Katharina, this I know,
    She is not for your turn, the more my grief.

    PETRUCHIO:
    I see you do not mean to part with her,
    Or else you like not of my company.

    BAPTISTA:
    Mistake me not; I speak but as I find.
    Whence are you, sir? what may I call your name?

    PETRUCHIO:
    Petruchio is my name; Antonio's son,
    A man well known throughout all Italy.

    BAPTISTA:
    I know him well: you are welcome for his sake.

    GREMIO:
    Saving your tale, Petruchio, I pray,
    Let us, that are poor petitioners, speak too:
    Baccare! you are marvellous forward.

    PETRUCHIO:
    O, pardon me, Signior Gremio; I would fain be doing.

    GREMIO:
    I doubt it not, sir; but you will curse your
```

we added `your` at the end of the sequence

Simple Initialization of Model

Here we init our number of iterations as 0, and our best val loss as a very high number.

```
iter_num = 0
best_val_loss = 1e9
```

Obtain our vocab size from our trained tokenizer.

```
meta_path = os.path.join(data_dir, 'meta.pkl')
meta_vocab_size = tokenizer.vocab_size
meta_vocab_size
```

```
    20099
```

Create our model args dict.

Use the following as a guide: Here

```
model_args = dict(
    n_layer=n_layer,
    n_head=n_head,
    n_embd=n_embd,
    block_size=block_size,
    bias=bias,
    vocab_size=None,
    dropout=dropout
)
```

Instantiate our model with the provided `model_args`.

These are derived from the hyper-parameters we set above.

```
if init_from == 'scratch':
    print("Initializing a new model from scratch")
    if meta_vocab_size is None:
        print("defaulting to vocab_size of GPT-2 to 50304 (50257 rounded up for efficiency)")
    model_args['vocab_size'] = meta_vocab_size if meta_vocab_size is not None else 50304
    gptconf = GPTConfig(**model_args)
    model = GPT(gptconf)
```

```
Initializing a new model from scratch
number of parameters: 29.55M
```

There we go! If you used the default values - you should have a model with 29.55M parameters!

Let's set our block_size to the correct size as determined in our configuration steps.

```
if block_size < model.config.block_size:
    model.crop_block_size(block_size)
    model_args['block_size'] = block_size
```

Now we can look at our model in all its glory!

```
model.to(device)
```

```
GPT(
  (transformer): ModuleDict(
    (wte): Embedding(20099, 516)
    (wpe): Embedding(512, 516)
    (drop): Dropout(p=0.2, inplace=False)
    (h): ModuleList(
      (0-5): 6 x Block(
        (ln_1): LayerNorm()
        (attn): CausalSelfAttention(
          (c_attn): Linear(in_features=516, out_features=1548, bias=False)
          (c_proj): Linear(in_features=516, out_features=516, bias=False)
          (attn_dropout): Dropout(p=0.2, inplace=False)
          (resid_dropout): Dropout(p=0.2, inplace=False)
        )
        (ln_2): LayerNorm()
        (mlp): MLP(
          (c_fc): Linear(in_features=516, out_features=2064, bias=False)
          (gelu): GELU(approximate='none')
          (c_proj): Linear(in_features=2064, out_features=516, bias=False)
          (dropout): Dropout(p=0.2, inplace=False)
        )
      )
    )
    (ln_f): LayerNorm()
  )
  (lm_head): Linear(in_features=516, out_features=20099, bias=False)
)
```

We'll set up our GradScaler - more information on this process here.

```
scaler = torch.cuda.amp.GradScaler(enabled=(dtype == 'float16'))
```

Let's set up our optimizer below. Be sure to include the correct values. You can check the `model.py` file for more information on what is expected in the `configure_optimizers` method here.

```
optimizer = model.configure_optimizers(
    weight_decay,
    learning_rate,
    (beta1, beta2),
    device_type
)

checkpoint = None
```

```
num decayed parameter tensors: 26, with 29,805,708 parameters
num non-decayed parameter tensors: 13, with 6,708 parameters
using fused AdamW: True
```

Now we can compile our model!

If you're using the T4 or V100 instance of Colab - this will not provide a signficant speed-up, but if you're using Ampere architecture (A100) you should notice a significant difference between the compiled and uncompiled model.

Read more about `torch.compile()` here.

```
if compile:
    print("compiling the model... (takes a ~minute)")
    unoptimized_model = model
    model = torch.compile(model) # requires PyTorch 2.0
```

```
compiling the model... (takes a ~minute)
```

We'll set up our loss estimation function here, which will help us estimate an arbitrarily accurate loss over either training or validation data by using many batches.

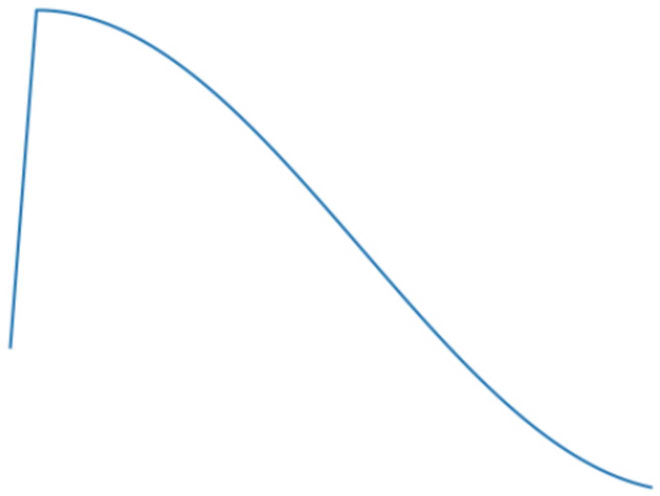
You'll notice that we quickly convert the model into `.eval()` model and then back to `.train()` mode.

```
@torch.no_grad()
def estimate_loss():
    out = {}
    model.eval()
    for split in ['train', 'val']:
        losses = torch.zeros(eval_iters)
        for k in range(eval_iters):
            X, Y = get_batch(split)
            with ctx:
                logits, loss = model(X, Y)
            losses[k] = loss.item()
        out[split] = losses.mean()
    model.train()
    return out
```

Creating our LR Scheduler

Beyond just slowly reducing our learning rate over time - we can use an LR Scheduler to allow us to move our learning according to a desired pattern.

We will use a "cosine with warmup" schedule and our learning rate, thusly, will follow this pattern:



There are many different schedulers, and many different ways to handle learning rate, and you can read about just a few of them here!

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

We need to set some specific values in our env to allow training in Colab.

```
!export LC_ALL="en_US.UTF-8"
!export LD_LIBRARY_PATH="/usr/lib64-nvidia"
!export LIBRARY_PATH="/usr/local/cuda/lib64/stubs"
!ldconfig /usr/lib64-nvidia
```

```
    /sbin/ldconfig.real: /usr/local/lib/libtbbbind_2_0.so.3 is not a symbolic link

    /sbin/ldconfig.real: /usr/local/lib/libtbbbind.so.3 is not a symbolic link

    /sbin/ldconfig.real: /usr/local/lib/libtbb.so.12 is not a symbolic link

    /sbin/ldconfig.real: /usr/local/lib/libtbbbind_2_5.so.3 is not a symbolic link

    /sbin/ldconfig.real: /usr/local/lib/libtbbmalloc_proxy.so.2 is not a symbolic link

    /sbin/ldconfig.real: /usr/local/lib/libtbbmalloc.so.2 is not a symbolic link
```

The Training Loop

Now we can finally grab our first batch and set our initial time to calculate how long our iterations are taking!

```
X, Y = get_batch('train')
t0 = time.time()
local_iter_num = 0
raw_model = model
running_mfu = -1.0 # model flops utilization

while True:
    # determine and set the learning rate for this iteration
    lr = get_lr(iter_num) if decay_lr else 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 and master_process:
        losses = estimate_loss()
        print(f"step {iter_num}: train loss {losses['train']:.4f}, val loss {losses['val']:.4f}")
        if losses['val'] < best_val_loss or always_save_checkpoint:
            best_val_loss = losses['val']
            if iter_num > 0:
                checkpoint = {
                    'model': raw_model.state_dict(),
                    'optimizer': optimizer.state_dict(),
                    'model_args': model_args,
                    'iter_num': iter_num,
                    'best_val_loss': best_val_loss,
                    'config': config,
                }
                print(f"saving checkpoint to {out_dir}")
                torch.save(checkpoint, os.path.join(out_dir, 'ckpt.pt'))
    if iter_num == 0 and eval_only:
        break

    # forward backward update, with optional gradient accumulation to simulate larger batch size
    # and using the GradScaler if data type is float16
    for micro_step in range(gradient_accumulation_steps):
        with ctx:
            logits, loss = model(X, Y)
            loss = loss / gradient_accumulation_steps # scale the loss to account for gradient accumulation
        # immediately async prefetch next batch while model is doing the forward pass on the GPU
        X, Y = get_batch('train')
        # backward pass, with gradient scaling if training in fp16
        scaler.scale(loss).backward()
    # clip the gradient
    if grad_clip != 0.0:
        scaler.unscale_(optimizer)
        torch.nn.utils.clip_grad_norm_(model.parameters(), 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 and master_process:
        # 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 have been a sum)
        lossf = loss.item() * gradient_accumulation_steps
        if local_iter_num >= 5: # let the training loop settle a bit
            mfu = raw_model.estimate_mfu(batch_size * 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:.2f}%")
    iter_num += 1
    local_iter_num += 1

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

```
iter 4970: loss 0.1866, time 204.83ms, mfu 2.50%
iter 4980: loss 0.1898, time 201.13ms, mfu 2.51%
iter 4990: loss 0.1698, time 205.46ms, mfu 2.51%
step 5000: train loss 0.0722, val loss 9.2033
saving checkpoint to out
iter 5000: loss 0.1807, time 26783.60ms, mfu 2.26%
```

Generating Outputs with our New Model

Now we can leverage the `sample.py` file to generate outputs from our model!

Generation Set Up and Model Loading

```
import os
import pickle
from contextlib import nullcontext
import torch
import tiktoken
from model import GPTConfig, GPT

# -----------------------------------------------------------------------------
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
seed = 1337
device = 'cuda' # examples: 'cpu', 'cuda', 'cuda:0', 'cuda:1', etc.
dtype = 'bfloat16' if torch.cuda.is_available() and torch.cuda.is_bf16_supported() else 'float16' # 'float32' or 'bfloat16' or 'f
compile = False # use PyTorch 2.0 to compile the model to be faster
# -----------------------------------------------------------------------------

torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
torch.backends.cuda.matmul.allow_tf32 = True # allow tf32 on matmul
torch.backends.cudnn.allow_tf32 = True # allow tf32 on cudnn
device_type = 'cuda' if 'cuda' in device else 'cpu' # for later use in torch.autocast
ptdtype = {'float32': torch.float32, 'bfloat16': torch.bfloat16, 'float16': torch.float16}[dtype]
ctx = nullcontext() if device_type == 'cpu' else torch.amp.autocast(device_type=device_type, dtype=ptdtype)
```

```
# model
if init_from == 'resume':
    # init from a model saved in a specific directory
    ckpt_path = os.path.join(out_dir, 'ckpt.pt')
    checkpoint = torch.load(ckpt_path, map_location=device)
    gptconf = GPTConfig(**checkpoint['model_args'])
    model = GPT(gptconf)
    state_dict = checkpoint['model']
    unwanted_prefix = '_orig_mod.'
    for k,v in list(state_dict.items()):
        if k.startswith(unwanted_prefix):
            state_dict[k[len(unwanted_prefix):]] = state_dict.pop(k)
    model.load_state_dict(state_dict)
```

```
number of parameters: 29.55M
```

```
model.eval()
model.to(device)
if compile:
    model = torch.compile(model) # requires PyTorch 2.0 (optional)
```

```
enc = tokenizer
encode = lambda s: enc.encode(s)
decode = lambda l: enc.decode(l)
```

Generation!

```
# encode the beginning of the prompt
if start.startswith('FILE:'):
    with open(start[5:], 'r', encoding='utf-8') as f:
        start = f.read()
start_ids = encode(start)
x = (torch.tensor(start_ids, dtype=torch.long, device=device)[None, ...])

# 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('---------------')
```

```
Take way, unruly woman!

DUCHESS OF YORK:
After, Aumerle! mount thee upon his horse;
Spur post, and get before him to the king,
And beg thy pardon ere he do accuse thee.
I'll not be long behind; though I be old,
I doubt not but to ride as fast as York:
And never will I rise up from the ground
Till Bolingbroke have pardon'd thee. Away, be gone!

HENRY BOLINGBROKE:
Can no man tell me of my unthrifty son?
'Tis full three months since I did see him last;
If any plague hang over us, 'tis he.
I would to God, my lords, he might be found:
Inquire at London,'mongst the taverns there,
For there, they say, they say, he daily doth frequent,
With unrestrained loose companions,
Even such, they say, as stand in narrow lanes,
And beat our watch, and rob our passengers;
Which he, young wanton and effeminate boy,
Takes on the point of honour to support
So dissolute a crew.

HENRY PERCY:
My lord, some two days since I saw the prince,
And told him of those triumphs held at Oxford.

HENRY BOLINGBROKE:
And what said the gallant?

HENRY PERCY:
His answer was, he would unto the stews,
And from the common'st creature pluck a glove,
And wear it as a favour; and with that
He would unhorse the lustiest challenger.

HENRY BOLINGBROKE:
As dissolute as desperate; yet through both
I see some sparks of better hope, which elder years
May happily bring forth. But who comes here?

DUKE OF AUMERLE:
What means our cousin, that he stares and looks
So wildly?

DUKE OF AUMERLE:
God save your grace! I do beseech your majesty,
To have some conference with your grace alone.

HENRY BOLINGBROKE:
Withdraw yourselves, and leave us here alone.
What is the matter with our cousin now?

DUKE OF AUMERLE:
For ever may
---------------
```