Homework 10: Coding GPT-2

In this assignment, we will graduate to state-of-the-art methods in Language Modeling and Generative AI. Most of you have heard of ChatGPT. ChatGPT is powered by the GPT architecture under the hood (GPT-4 at this moment). The underlying idea behind the GPT family, introduced in this paper, is quite intuitive: given the preceding $k-1$ words (the context), we want to predict the current word. Formally, given an unsupervised corpus of tokens $U = u_1, \ldots, u_n$, our goal is to maximize the following log-likelihood

$$L(U) = \sum_i logP(u_i \mid u_{i-k}, \cdots, u_{i-1}; \theta)$$

where $P(u_i \mid u_{i-k}, \cdots, u_{i-1}; \theta)$ (the probability of any word given all preceding words) is parameterized by a transformer-based neural network. The main novelty of the GPT series of models from OpenAI is their very large scale in terms of both the size of the model and dataset that the model was being trained on. As a point of reference, GPT-3 is powered by a model with 175 billion parameters and was trained on 300 billion tokens of text, costing roughly 10 million dollars to train. Obviously we will not be able to train such a model from scratch in this assignment; instead, we will use the weights provided by OpenAI for GPT-2 (GPT-3 is too big to fit in normal computer memory, especially for Colab). With that said, we need to build the correct framework (i.e. implement the correct model) so we can load the weights into our built framework. In this question, you will essentially implement the Transformer (there are libraries, for example PyTorch, that have already implemented the transformer for use as a built-in function. We will instead implement it from scratch for a better understanding).

You can read an excellent blog about transformers here, and about GPT-2 here. You will:

- implement GPT-2 from scratch
- load the pre-trained weights from OpenAI for our implemented GPT-2 (this part is taken care of for you)
- generate fun text

Your tasks are described at the end of this notebook.

```
"""
This block provides the encoding funciton that GPT2 uses to encode tokens.
This part is a bit beyond our scope for this assingment so DO NOT alter
this code block!

Byte pair encoding utilities.
Copied from: https://github.com/openai/gpt-2/blob/master/src/encoder.py.
"""
import json
import os
from functools import lru_cache

import regex as re

import json
import os

import numpy as np
import requests
import tensorflow as tf
```

```
from tqdm import tqdm


@lru_cache()
def bytes_to_unicode():
    """
    Returns list of utf-8 byte and a corresponding list of unicode strings.
    The reversible bpe codes work on unicode strings.
    This means you need a large # of unicode characters in your vocab if you want to avoid UNKs.
    When you're at something like a 10B token dataset you end up needing around 5K for decent coverage.
    This is a significant percentage of your normal, say, 32K bpe vocab.
    To avoid that, we want lookup tables between utf-8 bytes and unicode strings.
    And avoids mapping to whitespace/control characters the bpe code barfs on.
    """
    bs = list(range(ord("!"), ord("~") + 1)) + list(range(ord("¡"), ord("¬") + 1)) + list(range(ord("®")
    cs = bs[:]
    n = 0
    for b in range(2**8):
        if b not in bs:
            bs.append(b)
            cs.append(2**8 + n)
            n += 1
    cs = [chr(n) for n in cs]
    return dict(zip(bs, cs))


def get_pairs(word):
    """Return set of symbol pairs in a word.
    Word is represented as tuple of symbols (symbols being variable-length strings).
    """
    pairs = set()
    prev_char = word[0]
    for char in word[1:]:
        pairs.add((prev_char, char))
        prev_char = char
    return pairs


class Encoder:
    def __init__(self, encoder, bpe_merges, errors="replace"):
        self.encoder = encoder
        self.decoder = {v: k for k, v in self.encoder.items()}
        self.errors = errors  # how to handle errors in decoding
        self.byte_encoder = bytes_to_unicode()
        self.byte_decoder = {v: k for k, v in self.byte_encoder.items()}
        self.bpe_ranks = dict(zip(bpe_merges, range(len(bpe_merges))))
        self.cache = {}

        # Should have added re.IGNORECASE so BPE merges can happen for capitalized versions of contracti
        self.pat = re.compile(r"""'s|'t|'re|'ve|'m|'ll|'d| ?\p{L}+| ?\p{N}+| ?[^\s\p{L}\p{N}]+|\s+(?!\S)

    def bpe(self, token):
        if token in self.cache:
            return self.cache[token]
        word = tuple(token)
        pairs = get_pairs(word)
        if not pairs:
            return token
```

```
        while True:
            bigram = min(pairs, key=lambda pair: self.bpe_ranks.get(pair, float("inf")))
            if bigram not in self.bpe_ranks:
                break
            first, second = bigram
            new_word = []
            i = 0
            while i < len(word):
                try:
                    j = word.index(first, i)
                    new_word.extend(word[i:j])
                    i = j
                except:
                    new_word.extend(word[i:])
                    break

                if word[i] == first and i < len(word) - 1 and word[i + 1] == second:
                    new_word.append(first + second)
                    i += 2
                else:
                    new_word.append(word[i])
                    i += 1
            new_word = tuple(new_word)
            word = new_word
            if len(word) == 1:
                break
            else:
                pairs = get_pairs(word)
        word = " ".join(word)
        self.cache[token] = word
        return word

    def encode(self, text):
        bpe_tokens = []
        for token in re.findall(self.pat, text):
            token = "".join(self.byte_encoder[b] for b in token.encode("utf-8"))
            bpe_tokens.extend(self.encoder[bpe_token] for bpe_token in self.bpe(token).split(" "))
        return bpe_tokens

    def decode(self, tokens):
        text = "".join([self.decoder[token] for token in tokens])
        text = bytearray([self.byte_decoder[c] for c in text]).decode("utf-8", errors=self.errors)
        return text


def get_encoder(model_name, models_dir):
    with open(os.path.join(models_dir, model_name, "encoder.json"), "r") as f:
        encoder = json.load(f)
    with open(os.path.join(models_dir, model_name, "vocab.bpe"), "r", encoding="utf-8") as f:
        bpe_data = f.read()
    bpe_merges = [tuple(merge_str.split()) for merge_str in bpe_data.split("\n")[1:-1]]
    return Encoder(encoder=encoder, bpe_merges=bpe_merges)


"""
This block provides code to download the pre-trained weights from OpenAI.
GPT2 comes with 4 different sizes with the number of weights being either
```

```
"124M", "355M", "774M", or "1558M".

Having the software engineering skill to load complicated model weights
can be cruical, but for this assignment you will just use this code block to
do that. We highly encourage you to read through the code and understand
what is going on.

DO NOT alter this code block.
"""

def download_gpt2_files(model_size, model_dir):
    assert model_size in ["124M", "355M", "774M", "1558M"]
    for filename in [
        "checkpoint",
        "encoder.json",
        "hparams.json",
        "model.ckpt.data-00000-of-00001",
        "model.ckpt.index",
        "model.ckpt.meta",
        "vocab.bpe",
    ]:
        url = "https://openaipublic.blob.core.windows.net/gpt-2/models"
        r = requests.get(f"{url}/{model_size}/{filename}", stream=True)
        r.raise_for_status()

        with open(os.path.join(model_dir, filename), "wb") as f:
            file_size = int(r.headers["content-length"])
            chunk_size = 1000
            with tqdm(
                ncols=100,
                desc="Fetching " + filename,
                total=file_size,
                unit_scale=True,
                unit="b",
            ) as pbar:
                # 1k for chunk_size, since Ethernet packet size is around 1500 bytes
                for chunk in r.iter_content(chunk_size=chunk_size):
                    f.write(chunk)
                    pbar.update(chunk_size)


def load_gpt2_params_from_tf_ckpt(tf_ckpt_path, hparams):
    def set_in_nested_dict(d, keys, val):
        if not keys:
            return val
        if keys[0] not in d:
            d[keys[0]] = {}
        d[keys[0]] = set_in_nested_dict(d[keys[0]], keys[1:], val)
        return d

    params = {"blocks": [{} for _ in range(hparams["n_layer"])]}
    for name, _ in tf.train.list_variables(tf_ckpt_path):
        array = np.squeeze(tf.train.load_variable(tf_ckpt_path, name))
        name = name[len("model/") :]
        if name.startswith("h"):
            m = re.match(r"h([0-9]+)/(.*)", name)
            n = int(m[1])
            sub_name = m[2]
```

```
            set_in_nested_dict(params["blocks"][n], sub_name.split("/"), array)
        else:
            set_in_nested_dict(params, name.split("/"), array)

    return params


def load_encoder_hparams_and_params(model_size="124M", models_dir='./'):
    assert model_size in ["124M", "355M", "774M", "1558M"]

    model_dir = os.path.join(models_dir, model_size)
    tf_ckpt_path = tf.train.latest_checkpoint(model_dir)
    if not tf_ckpt_path:  # download files if necessary
        os.makedirs(model_dir, exist_ok=True)
        download_gpt2_files(model_size, model_dir)
        tf_ckpt_path = tf.train.latest_checkpoint(model_dir)

    encoder = get_encoder(model_size, models_dir)
    hparams = json.load(open(os.path.join(model_dir, "hparams.json")))
    params = load_gpt2_params_from_tf_ckpt(tf_ckpt_path, hparams)

    return encoder, hparams, params


"""
This is the code block for implementing GPT2.
GPT2 consists EXCLUSIVELY of transformer decoders where the number of such
decoders is a hyperparameter (that is provided by the downloaded parameters
from OpenAI.)

You will need to fill in all the ### Your code here in this code block.
You essentially will be implementing multi-head self-attention.
"""


def gelu(x):
    return 0.5 * x * (1 + np.tanh(np.sqrt(2 / np.pi) * (x + 0.044715 * x**3)))

def softmax(x):
    e_x = np.exp(x - np.max(x, axis=1, keepdims=True))
    return e_x / np.sum(e_x, axis=1, keepdims=True)

def layer_norm(x, g, b, eps: float = 1e-5):
    # normalize x to have mean=0 and var=1 over last axis
    # BE CAREFUL: when calculating the standard deviation, be sure to add
    #             eps to the variance to avoid 0 variance. I.e.
    #             sd = square_root(var+eps)

    mean = np.mean(x, axis=-1, keepdims=True)
    var = np.var(x, axis=-1, keepdims=True)

    x = (x - mean) / np.sqrt(var + eps)

    return g * x + b  # scale and offset with gamma/beta params


def linear(x, w, b):  # [m, in], [in, out], [out] -> [m, out]
    # Implement this simple linear layer, i.e. simply implement
    # output = x*w + b
```

```
    # output = x @ w + b
    # NOTE: "*" denotes matrix multiplication

    return np.dot(x, w) + b


def ffn(x, c_fc, c_proj):  # [n_seq, n_embd] -> [n_seq, n_embd]
    # project up
    a = gelu(linear(x, **c_fc))  # [n_seq, n_embd] -> [n_seq, 4*n_embd]

    # project back down
    x = linear(a, **c_proj)  # [n_seq, 4*n_embd] -> [n_seq, n_embd]

    return x


def attention(q, k, v, mask):  # [n_q, d_k], [n_k, d_k], [n_k, d_v], [n_q, n_k] -> [n_q, d_v]
    # This function performs the self-attention step based on the
    # ---the query: q
    # ---the key: k
    # ---the value: v
    # ---the mask: mask
    # We need mask as each token, for decoding purpose, can only pay attention
    # to the tokens that come before it, i.e. we can't look into the future!

    scores = np.dot(q, k.T)
    scores += mask

    attention_weights = softmax(scores)

    output = np.dot(attention_weights, v)

    return output

def mha(x, c_attn, c_proj, n_head):  # [n_seq, n_embd] -> [n_seq, n_embd]
    # qkv projection
    x = linear(x, **c_attn)  # [n_seq, n_embd] -> [n_seq, 3*n_embd]

    # split into qkv
    # we project x into a space of dimension 3*n_embd, so we need to split x
    # into 3 matrices, each of which of dimension [n_seq, n_embd], for
    # q, k, and v
    # Hint: use the np.split function, so qkv should be a list

    qkv = np.split(x, 3, axis=-1)
    # q, k, v = qkv[0], qkv[1], qkv[2]

    # split into heads
    # Recall that the transformer deploys multihead attention
    # So we need to further split EACH q, k, v into n_embd/n_head matrices.
    # So the result should be a list of lists
    # [3, n_seq, n_embd] -> [3, n_head, n_seq, n_embd/n_head]

    qkv_heads = [np.reshape(part, (n_head, part.shape[0], part.shape[1] // n_head)) for part in qkv]

    # q_heads = np.reshape(q, (n_head, q[0], q[1] // n_head))  # [n_head, n_seq, n_embd/n_head]
    # k_heads = np.reshape(k, (n_head, k[0], k[1] // n_head))  # [n_head, n_seq, n_embd/n_head]
    # v_heads = np.reshape(v, (n_head, v[0], v[1] // n_head))  # [n_head, n_seq, n_embd/n_head]
```

```
    # causal mask to hide future inputs from being attended to
    # the mask should be of size [n_seq, n_seq]

    causal_mask = np.tril(np.ones((x.shape[0], x.shape[0])))  # [n_seq, n_seq]

    # perform attention over each head
    # [3, n_head, n_seq, n_embd/n_head] -> [n_head, n_seq, n_embd/n_head]

    out_heads = []
    for i in range(n_head):
        out_heads.append(attention(qkv_heads[0][i], qkv_heads[1][i], qkv_heads[2][i], causal_mask))

    # merge results from different heads
    # [n_head, n_seq, n_embd/n_head] -> [n_seq, n_embd]

    x = np.concatenate(out_heads, axis=-1)

    # out projection
    x = linear(x, **c_proj)  # [n_seq, n_embd] -> [n_seq, n_embd]

    return x


def transformer_block(x, mlp, attn, ln_1, ln_2, n_head):  # [n_seq, n_embd] -> [n_seq, n_embd]
    # multi-head causal self-attention
    x = x + mha(layer_norm(x, **ln_1), **attn, n_head=n_head)  # [n_seq, n_embd] -> [n_seq, n_embd]

    # position-wise feed forward network
    x = x + ffn(layer_norm(x, **ln_2), **mlp)  # [n_seq, n_embd] -> [n_seq, n_embd]

    return x


def gpt2(inputs, wte, wpe, blocks, ln_f, n_head):  # [n_seq] -> [n_seq, n_vocab]
    # token + positional embeddings
    x = wte[inputs] + wpe[range(len(inputs))]  # [n_seq] -> [n_seq, n_embd]

    # forward pass through n_layer transformer blocks
    for block in blocks:
        x = transformer_block(x, **block, n_head=n_head)  # [n_seq, n_embd] -> [n_seq, n_embd]

    # projection to vocab
    x = layer_norm(x, **ln_f)  # [n_seq, n_embd] -> [n_seq, n_embd]
    return x @ wte.T  # [n_seq, n_embd] -> [n_seq, n_vocab]


def temporal_sampling(logits, tau):
    # Apply temperature scaling to logits
    scaled_logits = logits / tau

    # Softmax function to compute probabilities
    probs = np.exp(scaled_logits - np.max(scaled_logits))
    probs /= np.sum(probs)

    # Sample an index based on the probability distribution
    sampled_token = np.random.choice(len(probs), p=probs)

    return sampled_token
```

```
def generate(inputs, params, n_head, n_tokens_to_generate):
    from tqdm import tqdm

    # print(inputs)
    # print(params)
    # print(n_head)
    # print(n_tokens_to_generate)

    for _ in tqdm(range(n_tokens_to_generate), "generating"):  # auto-regressive decode loop
        logits = gpt2(inputs, **params, n_head=n_head)  # model forward pass

        # print(logits)

        # Choose the current token from logits. Note what the shape of the logit
        # variable is (hint: you only want to use the logits for the token that
        # we are predicting at this step, and ignore the previous logits)
        # ALSO, MAKE SURE once you predict the current token, you append the id of
        # predicted token to inputs to update inputs.

        # There are also different stratgies of predicting tokens givne logits.
        # Here, just implement the greedy strategy, i.e. choose the index
        # that has the largest logit.

        last_input_token = logits[-1, :]
        predicted_token = np.argmax(last_input_token)
        # predicted_token = temporal_sampling(last_input_token, 100)

        inputs = np.append(inputs, predicted_token)

    return inputs[-n_tokens_to_generate:]  # only return generated ids

# "124M", "355M", "774M", "1558M"
def main(prompt, n_tokens_to_generate = 40, model_size = "1558M", models_dir = "gpt2_models"):

    # load encoder, hparams, and params from the released open-ai gpt-2 files
    encoder, hparams, params = load_encoder_hparams_and_params(model_size, models_dir)

    # encode the input string using the BPE tokenizer
    input_ids = encoder.encode(prompt)

    # make sure we are not surpassing the max sequence length of our model
    assert len(input_ids) + n_tokens_to_generate < hparams["n_ctx"]

    # generate output ids
    output_ids = generate(input_ids, params, hparams["n_head"], n_tokens_to_generate)

    # decode the ids back into a string
    output_text = encoder.decode(output_ids)

    return output_text



prompt = "Alan Turing theorized that computers would one day become"
output = main(prompt)
print(output)
```

```
Fetching checkpoint: 1.00kb [00:00, 3.66Mb/s]
Fetching encoder.json: 1.04Mb [00:00, 2.42Mb/s]
Fetching hparams.json: 1.00kb [00:00, 3.54Mb/s]
Fetching model.ckpt.data-00000-of-00001: 6.23Gb [10:19, 10.1Mb/s]
Fetching model.ckpt.index: 21.0kb [00:00, 434kb/s]
Fetching model.ckpt.meta: 1.84Mb [00:00, 3.26Mb/s]
Fetching vocab.bpe: 457kb [00:00, 1.28Mb/s]
generating: 100%|██████████| 40/40 [10:31<00:00, 15.80s/it]
'
:
. and " in, in- a.
 and-a Aalsam on,
,'-50,?2-ro.s (L for "are,
```

```
prompt = "One day a golden retriever sees another golden retriever, they"
output = main(prompt)
print(output)
```

```
generating: 100%|██████████| 40/40 [11:03<00:00, 16.58s/it]
 sw B its I ( should-.
, and home as a the up [ Ds in, and to the [
 on (be-
 and the [-and-t : "
```

Your Tasks:

1. Complete all the <### Your Code Here> blocks in the cells above. (1.5 pt)
2. Show the generated text given the provided prompt in the previous cell. (0.25 pt)
3. Currently, you implemented a greedy approach for decoding. Is there any randomness in the greedy approach? I.e. given a trained (fixed) model, if we run the model multiple times given the same prompt, will we get different generated text? (0.25 pt)

No, with the greedy approach we will never generate different text.

4. Using the greedy approach, generate a text of 40 tokens with the 124M model based on the following prompt:

```
One day a golden retriever sees another golden retriever, they
```

Do you see any problem with the generated text? What do you think causes the problem? (0.5 pt)

5. *Temporal sampling*: Next, we will implement a slightly more interesting sampling technique that allows us to control how creative the generated text will be. Temporal sampling is quite straightforward: it integrates a temperature parameter into the softmax operation. Given the logits $l = l_1, \cdots, l_{|V|}$, a temporal softmax (operating on each li) takes the form of

$$s_i = \frac{\exp(l_i/\tau)}{\sum_{k \in 1 \ldots |V|} exp(l_k/\tau)}$$

where τ is a temperature parameter. We can then sample the index from a multinomidal distribution with parameters $s = s_1, \cdots, s_{|V|}$.

- Show that when $\tau \to 0$, temporal sampling is equivalent to the greedy approach.
- Show that when $\tau \to \infty$, temporal sampling is equivalent to sampling an index from the uniform distribution. Therefore, when $\tau \to \infty$, what do you think the quality of the generated text will be? Therefore, think of τ as a knob on how creative the generated text will be.
- Implement the temporal sampling. For the prompt

```
Alan Turing theorized that computers would one day become
```

try three different τ: 0.01, 1, and 100 to generate three 100-token texts. Show your generations here. (1 pt)

6. You can see 124M is the smallest model. Experiment with some of the bigger models and whatever prompts you like. Comparing the generated text from the different sized models on the same prompt, what conclusion can you draw? (0.5 pt)

ANSWERS

1. See code blocks.
2. ◦ the the. when 's are (the or no.-) the the- theque,the,-'s-, the-, will are
3. No, with the greedy approach the model will never generate different text.
4. them,— M, Phoenix the" an (the and and to the W [. -' it the- ((and,(also was or the and and and- in-.
5. See responses for different temperatures.

0.01: is/-..: N

- the the. when 's are (the or no.-) the the- theque,the,-'s-, the-, will are

1: juutes- feud, and W so the for,.....,ai)- fault ' or things to outside minded is), a about to distress: you misplaced let change see often

100: archaeocity Such slippedoliberal Thai year CAMquist Shipsassadicals shop Blades Creed nudity Lamador Douglas policemenFocus shovelgreat forestryB Compan Eliane forgivenVersion田etheus312...? Boot OFFIC detract curb ver ...

6. Models with more parameters were not much better than the 124M parameter model. None of the models produced coherent outputs.