Transformer

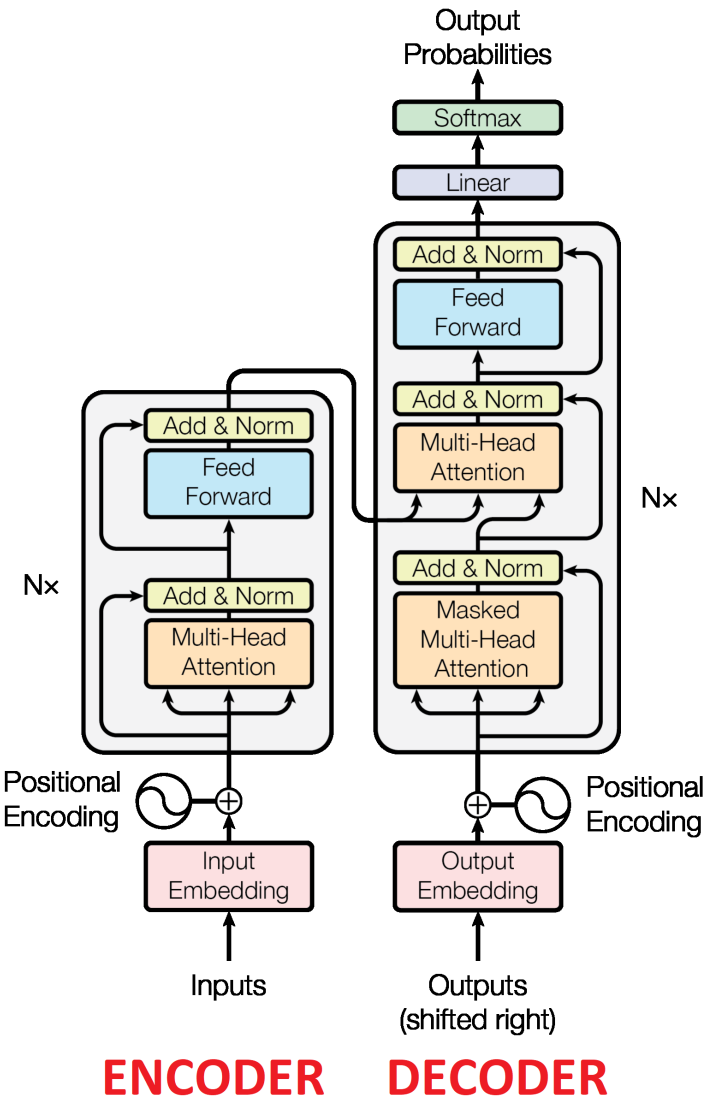
[ver. 25 March 2023] open in colab: Transformer.ipynb

this notebook is based on minGPT project of Andrej Karpathy (tiny Shakespeare char-level GPT example)

1. INTRO

Attention Is All You Need (Vaswani et al., 2017) introduced the Transformer, as -

> *a model architecture eschewing recurrence and instead **relying entirely on an attention mechanism** to draw global dependencies between input and output. The Transformer allows for significantly more parallelization ... the Transformer is the first transduction model relying entirely on self-attention to compute representations of its input and output without using sequence-aligned RNNs or convolution*



The Transformer was originally designed as a sequence-to-sequence translation model, where the **encoder** processes the input sequence in one language, and the **decoder** generates the output sequence in other language. It was designed to address the limitations of recurrent neural networks (RNNs) and convolutional neural networks (CNNs) in sequence-to-sequence NLP tasks. The Transformer eliminates the need for recurrent layers, enabling better parallelization, which significantly speeds up the training process

2. ***GPT - a Generative Pre-Trained Transformer***

In 2018, OpenAI introduced GPT in *Improving Language Understanding by Generative Pre-Training* (Radford et al., 2018). The GPT model is based on the Transformer architecture. However, it modifies the original design by discarding the encoder and solely utilizing the decoder part. This adaptation is designed to make GPT **"Autoregressive"** in the Generative mode, i.e., predict new tokens conditioning on its own previous predictions

There is an important distinction between the working of GPT in *Training* mode vs. *Generative* mode:

> **in training mode**, the objective of the model is to learn the language. The model is fed large amounts of text data and its only task is to predict the next token in the sequence, given the context of the previous tokens. In training mode, GPT does not generate tokens but only predictions, which are compared to the actual tokens fed as input. it is only in this phase that the model learnable parameters are updated through backpropagation, which minimizes the loss between the model predictions (aka ***logits***) and the actual input tokens

> **In generative mode**, the objective of the GPT model is to generate tokens using the pre-trained model in an **autoregressive** manner. This process begins by providing the model an initial input (prompt) as a starting point. The model predicts the next tokens given the initial prompt and continues to predict subsequent tokens **conditioning on its own previous predictions** in an autoregressive process. The prediction is made by converting the logits into a probability distribution using the **Softmax** operation, and sampling the next token from the distribution based on the probability scores. In this mode, the model's *weights are frozen* and not updated. (note: Generative mode is also known as **"Inference"** mode)

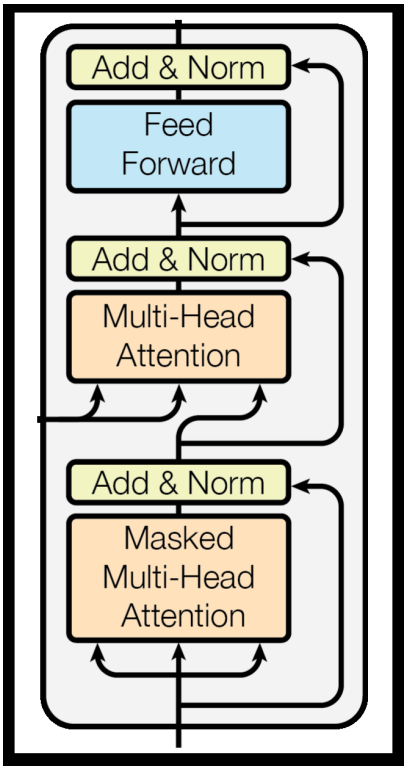
This is how it works:

> **1. Tokenization**: first, the input text data is tokenized into a sequence of tokens. In the original Transformer paper, the authors used a byte-pair encoding (BPE) tokenizer, which is similar to the WordPiece tokenizer used in BERT. In the GPT-3 paper, the authors used a byte-level BPE tokenizer, which is similar to the Byte-Pair Encoding (BPE) tokenizer used in GPT-2. In both cases, the tokenizer splits the input text into a sequence of tokens, where each token is a subword unit (e.g., a word, a character, or a subword). The tokenizer may also add special tokens to the beginning and end of the sequence

> **2. Token Embeddings**: after tokenization, each token is converted into an **embedding vector**. The concept of Word Embedding was introduced by Mikolov et al., 2013 in the **Word2Vec** paper *Distributed Representations of Words and Phrases and their Compositionality*. In the original GPT paper, the size of the embedding vector is 768; in GPT-3, the size is 1,248. in general, the larger the embedding vector, the more information the model can capture about the language. The representation of each token by a long embedding vector aims to capture the semantic information of the token in the context of the language as a whole. The embeddings represent the inherent properties and meaning of the token based on its co-occurrence patterns and relationships with other tokens in the training data. The token embeddings are part of the model's learnable parameters, and during the training process, the embeddings are updated through backpropagation in each batch iteration.

3. Positional Encoding: the embedding vectors are passed through a positional encoding layer, which adds positional information to the embedding vectors. This positional information is important for the model to understand the order of the tokens in the input sequence. The resulting matrix, containing both the embedding vectors and positional encodings, is fed as input **(X)** into the Transformer's first Transformer block.

4. Transformer Blocks - GPT applies **multiple** Transformer Blocks over the embeddings of input sequences. Each Transformer Block applies the following layers in sequence:



4.1 **Masked Multi-Head Attention layer**: Computes self-attention weights and generates a new representation of the input sequence (more on that below)

4.2 **Add & Norm** - Adds Residual Connection to the input to the Self-Attention Layer to its output and then apply Layer Normalization to the result

4.3 **Feed-Forward layer** Applies a *pointwise* feed-forward layer independently to each vector in the sequence. (the term "pointwise" refers to the fact that the FFN operates on each token in the input sequence independently, without considering the other tokens in the sequence; this is in contrast to the convolutional layers in CNNs, which operate on a local neighborhood of the input sequence, or the recurrent layers in RNNs, which operate on the entire input sequence at once)

4.4 **Add & Norm** - same as in step 4.2

5. The output of the Transformer blocks - the self-attention block's output represent each of the input tokens, like the input token embedding, but they capture different aspects of the token's context. The input token embedding is responsible for representing the token's general meaning in the language, while the self-attention block's output is a more refined representation of the token, which takes into account its specific contextual relationships with all other tokens in the input sequence. This output vector reflects the model's understanding of the token's role and significance within the given sequence, considering the broader context, dependencies, and interactions with other tokens

6. The output of the GPT Model - the output of the last Transformer block (which contains contextual vector representations for each token in the input sequence), is passed through a Linear layer, which generates a ***logits vector***. Each element in the logits vector is a scalar value that represents the model's unnormalized confidence for the corresponding token in the vocabulary being the next token. The size of the logits vector is equal to the size of the vocabulary. From here, the working of the model depends on its mode of operation:

6.1 in Training mode, the logits are fed into a Cross-Entropy Loss function, which computes the loss by comparing the logits to the actual tokens in the input sequence. This loss is then used to update the model learnable parameters through backpropagation (after each batch iteration).

6.2 in generative / inference mode, the logits are passed through a Softmax layer, which creates a probability distribution over the vocabulary for each token in the input sequence. The selection of the next token is made by sampling from this probability distribution, using methods such as top-k or top-p sampling, or other sampling techniques like beam search.

Some notes on the above:

the Feed-Forward layer (FFN) - the FFN consists of two linear (dense) layers with a non-linear activation function, such as GeLU (Gaussian Error Linear Units) in between. The purpose of this FFN is to introduce non-linearity into the model and combine features learned by the self-attention mechanism within the Transformer. The first linear layer of the FFN increases the dimensionality of the input (commonly X4), while the second linear layer reduces it back to the original dimension. The non-linear activation function helps the model capture complex relationships in the data

Residual Connections - In the GPT model, residual connections are found in two places within each Transformer block: the Self-Attention layer and the Feed-Forward layer. The purpose of these residual connections is to help the model learn more efficiently, by allowing gradients to flow more easily through the network during backpropagation, and mitigating the vanishing gradient problem that can occur in deep architectures.

Dropout - dropout regularization technique works by randomly "dropping out" or setting a fraction of the neurons to zero during training, forcing the network to learn more robust features. In the Transformer architecture and GPT, dropout is typically applied at several points: (i) after the Self-Attention layer - after computing the self-attention scores and generating a new representation of the input sequence, dropout is applied to the output before the first Add & Norm; (ii) after the Feed-Forward layer - after applying the FFN to each vector in the sequence, dropout is applied to the output before the second Add & Norm; and (iii) in the Multi-Head Attention - dropout can also be applied to the attention scores before they are used to compute the weighted sum of the value vectors

Layer Normalization - this layer apply normalization to the output values (activations) of the self-attention layer and the output values of the FFN by the mean and variance of the these activatations. This normalization technique helps the model learn more efficiently, by allowing gradients to flow more easily through the network during backpropagation, and mitigating the vanishing gradient problem that can occur in deep architectures. In PyTorch, layer normalization can be implemented using the torch.nn.LayerNorm module, which takes the number of features (neurons) as input and applies normalization across these features. *(note: "activations" typically refers to the output values of the current layer, before the activation function is applied)*

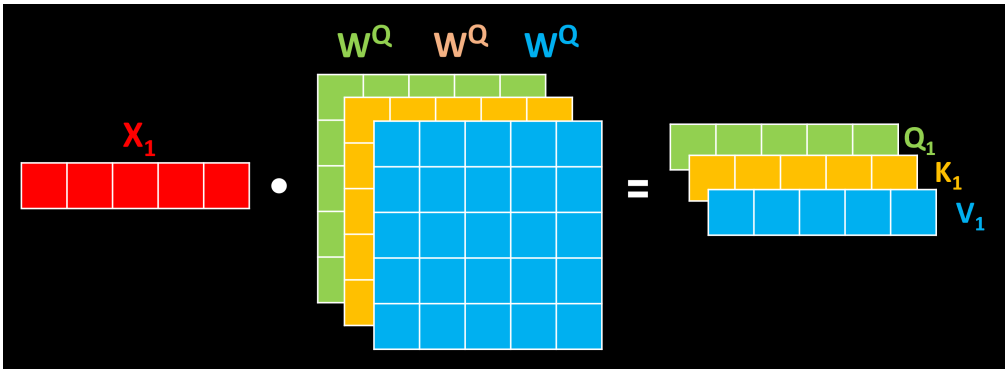
3. ***How the Masked Multi-Head Attention Work***

> Self-Attention is the fundamental operation of the Transformer. It is designed to weigh and relate the tokens of the input sequence to better capture the relationships and dependencies between them, by computing scores for each pair of elements in the input sequence. These scores determine how much "attention" each element should pay to other elements in the sequence. Higher scores indicate stronger relationships between elements, while lower scores suggest weaker relationships.

> The Self-Attention mechanism transforms each token, initially represented by its embedding and positional encoding vector, into three vectors: a query vector (Q), a key vector (K), and a value vector (V). This transformation is achieved by applying linear transformations, specifically by multiplying the input sequence of the embedding vectors (X) with their corresponding weight matrices (W_Q, W_K, and W_V). In PyTorch, linear transformations can be implemented using the torch.nn.Linear module, which takes the number of input features and the number of output features as input. more on this here. (note: the authors of *Attention Is All You Need* introduced the Query/Key/Value concept, drawing an analogy to a retrieval system, which might be somewhat confusing.)

Linear Transformation of the input sequence X to Q, K, V vectors:

```
Q = X @ W_Q
K = X @ W_K
V = X @ W_V`
```



> **Attention Scores** are computed by multiplying each Q vector with each K vector. This is done by taking the dot product between the Q matrix and the transpose of K matrix (`Q @ K.transpose()`). The result is a score matrix that represents the relationships or affinities between each token in the input sequence. These scores quantify the degree to which each token should "pay attention" to the others.

> **Attention Weights** Attention weights are normalized attention scores, after applying Softmax so that score values sum up to 1. The Softmax function can be sensitive to very large input values that kill the gradient and slow down learning. Since the average value of the dot product grows with the embedding dimension, it helps to scale down the dot product to stop the inputs to the Softmax function from growing too large, so we divide the dot product by the square root of the embedding dimension. So the attention weights are computed as follows:

```
A = softmax(Q @ K.transpose() / sqrt(dim_K))
```

> The Attention Output (O) is then computed by multiplying the attention scores (A) by the value vectors (V)

```
O (Output) = A @ V
```

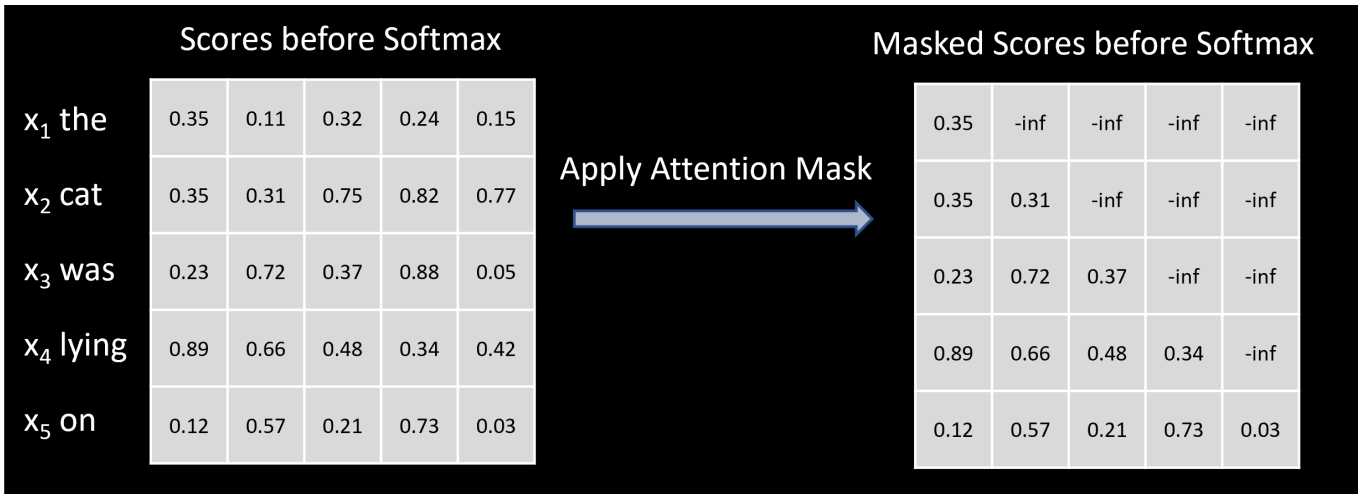
> The output O is a matrix of the same shape as the input X - each output vector represents a single input token and has the same size as the embedding and positional encoding vectors

> **In summary, the attention mechanism in GPT uses learned weight matrices (W_Q, W_K, W_V) to transform the input sequence into query, key, and value vectors, then uses the dot product of the query and key tensors to compute attention weights, which are used to compute a weighted sum of the value tensor to produce the final output.**

Some notes on the above:

> **Multi-Head Attention** - In each Transformer block, the Self-Attention is conducted multiple times using Multi-Head Attention, by dividing the Q, K, and V vectors into multiple subspaces (or "heads"), and applying the attention mechanism independently to each subspace, and then concatenates the results back into a single vector. This approach allows the model to focus on various relationships within the data simultaneously, leading to more expressive and powerful contextual representations.

> **Causal Attention Mask** - the attention mechanism in GPT uses an attention mask to prevent the model from attending to tokens that come after the current token in the input sequence. Masking is done by creating a triangular matrix where the lower triangular part is preserved, and the upper triangular part is masked. As a result, when the model processes a given token, it can only attend to the tokens that came before it or the current token itself, but not the future tokens.This is crucial for maintaining a **causal structure** to preventing information leakage from future tokens during training and inference, and enforcing the autoregressive property and causal structure of the model. The attention mask triangle is applied on the Attention Scores, before the Softmax operation, by setting the values of the cells we want to mask to -infinity.



4. ***Code***

Following is a PyTorch implementation of a basic GPT model, based on Andrej Karpathy's minGPT project. Thanks Andrej!

- The model is trained on 1 MB txt file of Shakespeare's writings, and after training it is able to generate new sonnets that resemble Shakespeare's

- The model is using a simple character-level tokenizer, and the vocabulary size is relatively small - 65 unique characters: `!$&',-.3:;?ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz`
- The model is configured by default to run with 6 Transformer blocks (`C.n_layer`), each with 6 masked self-attention heads (`C.n_head`), and and tokens (characters) embedding size is 192 (`C.n_embed`)
- The model has ~1.1 million learnable parameters, and on a single NVIDIA GeForce RTX 3090 it takes about 5 min to train
- The model is using the Adam optimizer with a learning rate of 0.0001 (`C.lr`), and the loss function is Cross Entropy Loss
- The training loop is configured to run 5000 batch iterations (`C.max_iters`); prints out info on the loss and perplexity every 200 iterations (`trainer.iter_num % 200 == 0`); and generates a new sonnet every 1000 iterations (`trainer.iter_num % 1000 == 0`).
- in the Generative mode, the input sequence is initialized with a given prompt (`context = "In void of faith, "`).

Here's a sample output:

```
In void of faith, by the tune of that thou hate'en;
An entreaty is not broad, but not ambitious
Fitted in this bastarding thoughts!

ARIEL:
Alas!
I think, I was confessed mind,--that spright me;
In her perfections by children him no farther.

ROMEO:
No.

JULIET:
But she must I wound the way is the mind.

RICHARD:
Ay, be so feast.

WARWICK:
Ah deadly I have not been sick my love,
Thou dost redress it for a little:
I have seen to thee a hand, a bloody gentleman
I thought it in my brother's shall be all,
By the bond
```

```python
# some imports..
import math
import torch
import numpy as np
import torch.nn as nn
```

```
from torch.nn import functional as F
from torch.utils.data import Dataset, DataLoader
import time
from collections import defaultdict
import random
import platform
from IPython.display import clear_output
```

```
# some utils..
class NewGELU(nn.Module):
    """  Implementation of the GELU activation function currently in Google BERT
repo (identical to OpenAI GPT). Reference: Gaussian Error Linear Units (GELU)
paper: https://arxiv.org/abs/1606.08415. GELU is a smooth approximation of the
ReLU function """
    def forward(self, x):
        return 0.5 * x * (1.0 + torch.tanh(math.sqrt(2.0 / math.pi) * (x +
0.044715 * torch.pow(x, 3.0))))

def set_seed(seed):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)

class CfgNode:
    """ a lightweight configuration class inspired by yacs """
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

    def __str__(self):
        return self._str_helper(0)

    def _str_helper(self, indent):
        parts = []
        for k, v in self.__dict__.items():
            if isinstance(v, CfgNode):
                parts.append("%s:\n" % k)
                parts.append(v._str_helper(indent + 1))
            else:
                parts.append("%s: %s\n" % (k, v))
        parts = [' ' * (indent * 4) + p for p in parts]
        return "".join(parts)

    def to_dict(self):
        """ return a dict representation of the config """
        return { k: v.to_dict() if isinstance(v, CfgNode) else v for k, v in
self.__dict__.items() }

    def merge_from_dict(self, d):
        self.__dict__.update(d)
```

```
class CausalSelfAttention(nn.Module):
    ''' a vanilla multi-head masked self-attention layer with a projection at the
end (instead of using torch.nn.MultiheadAttention) the causal mask is applied to
the attention weights to ensure that the model cannot "cheat" and look into the
future when making predictions. the mask is applied to the attention weights
before they are normalized with a softmax.  '''
    def __init__(self, config):
        super().__init__()
        assert config.n_embd % config.n_head == 0 # make sure that the number of
heads evenly divides the embedding size
        ''' key, query, value projections for all heads, but in a batch. The
self.c_attn is a nn.Linear module, where the input size is config.n_embd (the
token embedding size) and the output size is 3 * config.n_embd. In the forward
step, the output tensor will split into 3 tensors along the last dimension, each
with shape (batch_size, seq_len, config.n_embd). These 3 tensors will be used as
the query, key, and value vectors in the self-attention mechanism '''
        self.c_attn = nn.Linear(config.n_embd, 3 * config.n_embd)
        # projection of the concatenated multi-head attention output back to
embedding size (i.e. the output size of the self-attention layer is 3 *
config.n_embd, but we want to project it back to config.n_embd, which is the size
of the token embeddings)
        self.c_proj = nn.Linear(config.n_embd, config.n_embd)
        # dropout regularization
        self.attn_dropout = nn.Dropout(config.attn_pdrop)
        # next is a dropout regularization to the residual connection, which is a
bypass that allows the gradient to flow more easily through the network
        # the residual connection is the addition of the input to the output of
the layer
        self.resid_dropout = nn.Dropout(config.resid_pdrop)
        # causal mask to ensure that attention is only applied to the left in the
input sequence
        self.register_buffer("bias", torch.tril(torch.ones(config.block_size,
config.block_size))
                                     .view(1, 1, config.block_size,
config.block_size))
        self.n_head = config.n_head
        self.n_embd = config.n_embd

    ''' this is called in the forward method of the Block class. x is the input
tensor of shape (batch_size, seq_len, 3 * config.n_embd) i.e. ([64, 128, 576g])
in this tiny shakespear example '''
    def forward(self, x):
        B, T, C = x.size()  # B=Batch size, T=block size or sequence length, i.e.
the number of Tokens, C=num of Channels or embedding size
        # calculate query, key, values for all heads in batch and move head
forward to be the batch dim
        # we split the x input tensor along the last dimension into 3 tensors of
size (batch_size, seq_len, config.n_embd)
        q, k ,v  = self.c_attn(x).split(self.n_embd, dim=2)
        # now we split the tensors along the second dimension into n_head tensors
of size (batch_size, seq_len, config.n_embd // n_head)
        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)
        att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))
        # here we apply the causal mask to the attention weights, before the
softmax normalization
        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 by side

        # apply residual connection and projection (linear transoformation) to
the input tensor y to finalize the self-attention layer
        y = self.resid_dropout(self.c_proj(y))
        return y
```

```
class Block(nn.Module):
    ''' a single block of the transformer model '''
    def __init__(self, config):
        super().__init__()
        self.ln_1 = nn.LayerNorm(config.n_embd)
        self.attn = CausalSelfAttention(config) # init the self-attention layer
        self.ln_2 = nn.LayerNorm(config.n_embd)
        self.mlp = nn.ModuleDict(dict(
            # c_fc is the first linear layer in the MLP, which expands the input
to a larger vector size
            # Specifically, it maps from a vector of size `config.n_embd` to a
larger vector of size 4 * `config.n_embd`
            c_fc    = nn.Linear(config.n_embd, 4 * config.n_embd),
            # c_proj is the second linear layer in the MLP, which maps the output
of the first linear layer back to the original size of `config.n_embd`
            # Specifically, it maps from a larger vector of size 4 *
`config.n_embd` to a vector of size `config.n_embd`
            c_proj  = nn.Linear(4 * config.n_embd, config.n_embd),
            # act is the activation function used in the MLP. Here, act will
apply the GELU activation function to the output of the FIRST linear layer in
this MLP. The purpose of this is to introduce non-linearity to the self-attention
computation. The self-attention mechanism is essentially a weighted sum of the
input tokens, and without non-linearity, it may not be able to capture complex
relationships between the tokens.
            act = NewGELU(),
            # dropout layer, which in this case will apply to the output of the
second linear layer (c_proj).
            dropout = nn.Dropout(config.resid_pdrop),
        ))
        # assign self.mlp to 'm' so that we can use it in the forward function.
```

```
        m = self.mlp
        # mlpf is the forward function of the MLP, which is a composition of the
three layers in the MLP: c_fc, c_proj, and act
        # the lambda function is a shorthand for defining a function in Python.
in this case, it is equivalent to the following:
        # def mlpf(x):
        #     return m.dropout(m.c_proj(m.act(m.c_fc(x))))
        self.mlpf = lambda x: m.dropout(m.c_proj(m.act(m.c_fc(x)))) # MLP forward
- it is called in the forward function of the Block class

    # this is called in the GPT class forward function
    # it first applies the self-attention mechanism to the input,
    # and then applies the MLP to the output of the self-attention mechanism
    def forward(self, x):
        x = x + self.attn(self.ln_1(x)) # this is calling the forward method of
the CausalSelfAttention class. the addition of x is the residual connection
        x = x + self.mlpf(self.ln_2(x)) # this is calling the mlpf function
defined above. the addition of x is the residual connection
        return x # the output of the block is the input to the next block
```

```
class GPT(nn.Module):
    @staticmethod
    def get_default_config():
        C = CfgNode()
        C.model_type = 'gpt'
        C.n_layer = 6
        C.n_head = 6
        C.n_embd =  192
        C.vocab_size = None
        C.block_size = None
        # dropout hyperparameters
        C.embd_pdrop = 0.1
        C.resid_pdrop = 0.1
        C.attn_pdrop = 0.1
        return C

    def __init__(self, config):
        super().__init__()
        assert config.vocab_size is not None
        assert config.block_size is not None
        self.block_size = config.block_size
        type_given = config.model_type is not None
        params_given = all([config.n_layer is not None, config.n_head is not
None, config.n_embd is not None])

        # transformer is an nn.ModuleList container, and h is a contrainer used
to create a list of Block modules. wte is the token embedding layer, wpe is the
position embedding layer the position embedding layer is used to encode the
position of a token in the sequence.
        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.embd_pdrop),
            h = nn.ModuleList([Block(config) for _ in range(config.n_layer)]),
            ln_f = nn.LayerNorm(config.n_embd),
        ))
        #  the output of the transformer is the input to the linear layer
lm_head.
        self.lm_head = nn.Linear(config.n_embd, config.vocab_size, bias=False)

        ''' init all weights, and apply a special scaled init to the residual
projections, per GPT-2 paper. torch.nn.apply() applies a given function to all
the modules within a module container. The method takes two arguments: module -
the module container on which the function will be applied; and func - the
function to apply to each module. The function func is applied recursively to
each submodule within the module container. The function should take a single
module as input and should return the modified module. The output of
torch.nn.apply() is the modified module container.         '''
        self.apply(self._init_weights)

        '''next code is initializing the weight parameters for a specific layer
in the model, specifically, it is initializing the weight matrix c_proj.weight.
The named_parameters() method returns an iterator over the module's named
parameters along with their corresponding values. The initialization is performed
using the torch.nn.init.normal_() method, which sets the initial values of the
tensor with random samples from a normal distribution. This scheme is known as
the Xavier initialization, which aims to keep the scale of the gradients
approximately the same throughout the training process.'''
        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 (note we don't count the decoder parameters
in lm_head)
        n_params = sum(p.numel() for p in self.transformer.parameters())
        print("number of parameters: %.2fM" % (n_params/1e6,))

    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)
        elif isinstance(module, nn.LayerNorm):
            torch.nn.init.zeros_(module.bias)
            torch.nn.init.ones_(module.weight)

    def configure_optimizers(self, train_config):
        """  This long function is unfortunately doing something very simple and
is being very defensive: We are separating out all parameters of the model into
two buckets: those that will experience weight decay for regularization and those
that won't (biases, and layernorm/embedding weights). We are then returning the
PyTorch optimizer object. """
        # separate out all parameters to those that will and won't experience
regularizing weight decay
```

```
        decay = set()
        no_decay = set()
        whitelist_weight_modules = (torch.nn.Linear, )
        blacklist_weight_modules = (torch.nn.LayerNorm, torch.nn.Embedding)
        for mn, m in self.named_modules():
            for pn, p in m.named_parameters():
                fpn = '%s.%s' % (mn, pn) if mn else pn # full param name random
note: because named_modules and named_parameters are recursive we will see the
same tensors p many many times. but doing it this way allows us to know which
parent module any tensor p belongs to...
                if pn.endswith('bias'):
                    # all biases will not be decayed
                    no_decay.add(fpn)
                elif pn.endswith('weight') and isinstance(m,
whitelist_weight_modules):
                    # weights of whitelist modules will be weight decayed
                    decay.add(fpn)
                elif pn.endswith('weight') and isinstance(m,
blacklist_weight_modules):
                    # weights of blacklist modules will NOT be weight decayed
                    no_decay.add(fpn)

        # validate that we considered every parameter
        param_dict = {pn: p for pn, p in self.named_parameters()}
        inter_params = decay & no_decay
        union_params = decay | no_decay
        assert len(inter_params) == 0, "parameters %s made it into both
decay/no_decay sets!" % (str(inter_params), )
        assert len(param_dict.keys() - union_params) == 0, "parameters %s were
not separated into either decay/no_decay set!" \
                                                    % (str(param_dict.keys() -
union_params), )

        # create the pytorch optimizer object
        optim_groups = [
            {"params": [param_dict[pn] for pn in sorted(list(decay))],
"weight_decay": train_config.weight_decay},
            {"params": [param_dict[pn] for pn in sorted(list(no_decay))],
"weight_decay": 0.0},
        ]
        ''' note: we are using AdamW optimizer here, which is Adam with weight
decay fix. The optimizer takes in a list of parameter groups optim_groups, which
are the different subsets of model parameters that have different optimization
settings such as learning rate. The optimizer will update the parameters in each
group separately. '''
        optimizer = torch.optim.AdamW(optim_groups,
lr=train_config.learning_rate, betas=train_config.betas)
        return optimizer

    # in this forward method of the GPT module class, idx is the input tensor
(indices of tokens), and targets is the output tensor (also indices of tokens).
For our tiny shakespeares, both are of shape (b=64, t=128) where b is the batch
size and t is the sequence length (num of chars).
    # this forward is called by the trainer in logits, self.loss = model(x, y)
    def forward(self, idx, targets=None):
```

```
        device = idx.device
        b, t = idx.size()
        assert t <= self.block_size, f"Cannot forward sequence of length {t},
block size is only {self.block_size}"
        # This creates a tensor containing a sequence of integers starting from 0
to t with a step size of 1. The dtype argument specifies the data type of the
tensor to be long, which means the values will be 64-bit integers. The
unsqueeze(0) method call adds a new dimension of size 1 at the 0-th dimension of
the tensor, to make the tensor compatible with other tensors in the model that
have an additional batch dimension.
        pos = torch.arange(0, t, dtype=torch.long, device=device).unsqueeze(0) #
shape (1, 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 (1, t,
n_embd)
        # tok_emb and pos_emb are the token and positional embeddings,
respectively, that are summed together to produce the final input embeddings to
the transformer blocks. The drop method applies a dropout regularization - it
works by randomly setting a fraction of the input tensor's values to zero during
each training iteration. The drop method takes a single argument x and returns a
tensor with the same shape as x, but with some of its elements randomly set to
zero. tok_emb and pos_emb have the same shape, and when you add them together,
you are performing an element-wise addition between the corresponding elements of
the two tensors
        x = self.transformer.drop(tok_emb + pos_emb)
        # now we feed the embeddings into the transformer blocks contained in
self.transformer.h. block(x) is a call to the forward method of the Block class
        for block in self.transformer.h:
            x = block(x)

        # apply the final layer norm and the linear layer to get the logits. the
logits are the output of the linear layer of shape (batch_size, n_embed,
vocab_size), where each element of the tensor represents the predicted scores for
each word in the vocabulary at each position in the input sequence.
        x = self.transformer.ln_f(x)
        logits = self.lm_head(x)

        # if we are given some desired targets also calculate the loss.
        # The cross_entropy function takes two arguments: the logits and the
targets. The logits are the output of the linear layer, and the targets are the
desired output values. In this case, logits is a tensor of shape (batch_size,
seq_length, vocab_size) containing the predicted logits for each possible output
token at each position in the sequence, and targets is a tensor of shape
(batch_size, seq_length) containing the true (index of) target tokens for each
position in the sequence. The method view(-1, logits.size(-1)) is used to reshape
the logits tensor to have shape (batch_size * seq_length, vocab_size). This is
necessary because the F. cross_entropy() function expects the logits to have
shape (N, C) where N is the total number of predictions (i.e., batch_size *
seq_length) and C is the number of classes (i.e., vocab_size). The ignore_index
parameter is used to specify a token index that should be ignored in the loss
calculation. In this case, -1 is specified to ignore any padding tokens in the
targets tensor. The resulting loss is a scalar tensor that represents the average
cross-entropy loss over all tokens in the batch.
```

```
        loss = None
        if targets is not None:
            loss = F.cross_entropy(logits.view(-1, logits.size(-1)),
targets.view(-1), ignore_index=-1)
        return logits, loss

    @torch.no_grad() # don't track gradients for this method
    def generate(self, idx, max_new_tokens, temperature=1.0, do_sample=False,
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 time. """
        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.block_size else idx[:, -
self.block_size:]
            # 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, top_k)
                logits[logits < v[:, [-1]]] = -float('Inf')
            # apply softmax to convert logits to (normalized) probabilities
            probs = F.softmax(logits, dim=-1)
            # either sample from the distribution or take the most likely element
            if do_sample:
                idx_next = torch.multinomial(probs, num_samples=1)
            else:
                _, idx_next = torch.topk(probs, k=1, dim=-1)
            # append sampled index to the running sequence and continue
            idx = torch.cat((idx, idx_next), dim=1)

        return idx
```

```
class Trainer:

    @staticmethod
    def get_default_config():
        CN = CfgNode
        C = CN()
        # device to train on
        C.device = 'auto'
        # dataloder parameters
        C.num_workers = 4
        # optimizer parameters
        C.max_iters = 5000
        C.batch_size = 64
        C.learning_rate = 3e-4
```

```
        C.betas = (0.9, 0.95)
        C.weight_decay = 0.1 # only applied on matmul weights
        C.grad_norm_clip = 1.0
        return C

    def __init__(self, config, model, train_dataset):
        print(">>> init trainer")
        self.config = config
        self.model = model
        self.optimizer = None
        self.train_dataset = train_dataset
        self.callbacks = defaultdict(list)

        # determine the device we'll train on
        if config.device == 'auto':
            self.device = 'cuda' if torch.cuda.is_available() else 'cpu'
        else:
            self.device = config.device
        self.model = self.model.to(self.device)
        print("running on device", self.device)

        # variables that will be assigned to trainer class later for logging and
etc
        self.iter_num = 0
        self.iter_time = 0.0
        self.iter_dt = 0.0

    def add_callback(self, onevent: str, callback): # onevent is a string that
specifies when the callback should be called. callback is a function that takes
the trainer object as an argument. the callback function is added to the list of
callbacks for the specified event. the callback is defined
        self.callbacks[onevent].append(callback)

    def set_callback(self, onevent: str, callback):
        self.callbacks[onevent] = [callback]

    def trigger_callbacks(self, onevent: str):
        for callback in self.callbacks.get(onevent, []):
            callback(self)

    def run(self):
        model, config = self.model, self.config

        # setup the optimizer
        self.optimizer = model.configure_optimizers(config)

        # setup the dataloader, using the train_dataset that was passed in to the
Trainer class. the train_dataset is a CharDataset object that is created in
chargpt.py, in which the input tokens (x) and labels (y) are stored.
        train_loader = DataLoader(
            self.train_dataset,
            sampler=torch.utils.data.RandomSampler(self.train_dataset,
replacement=True, num_samples=int(1e10)), # When replacement is set to True, the
sampler can sample the same sample multiple times
            shuffle=False,
```

```
            pin_memory=True,
            batch_size=config.batch_size,
            num_workers= 0 if platform.system() == 'Windows' else
config.num_workers
        )
        ''' module.train() is a method in PyTorch's nn.Module class that sets the
module to training mode. This has any effect only on certain modules. See
documentations of particular modules for details of their behaviors in
training/evaluation modes, if they are affected, e.g. Dropout, BatchNorm, etc.
'''
        model.train()
        # iteration counter and timer. each iteration is one batch of data
        self.iter_num = 0
        self.iter_time = time.time()
        ''' The iter() function is used to create an iterator object from a
DataLoader object that can be used to get the next batch of data in the dataset.
The next() function is then used on this iterator object to retrieve the next
batch of data.'''
        data_iter = iter(train_loader)

        while True:
            ''' fetch the next batch (x, y) and re-init iterator if needed. The
except StopIteration is used to catch the StopIteration exception that is raised
by the next function when there is no more data to be returned from data_iter. If
there is no more data to be returned, data_iter is reset to the beginning of the
data by creating a new iterator from train_loader and the next batch of data is
retrieved using next(data_iter). Finally, batch is created as a list of tensors
that are transferred to the device (specified by self.device) using the to
method. '''
            try:
                batch = next(data_iter)
            except StopIteration:
                data_iter = iter(train_loader)
                batch = next(data_iter)
            batch = [t.to(self.device) for t in batch]

            ''' each batch contains two tensors, x and y. x is the input token
and y is the label. x and y are of shape (batch_size, seq_len). '''
            x, y = batch
            ''' forward the model - this will call the GPT class (model)
forward() method. the loss returned is a scalar tensor that represents the
average cross-entropy loss over all tokens in the batch. logits is a tensor of
shape (batch_size, n_embed, vocab_size) that contains the predicted scores for
each class in the output vocabulary. '''
            logits, self.loss = model(x, y)

            '''backprop and update the parameters.
model.zero_grad(set_to_none=True) is used to set the gradients of all the
parameters of the model to zero before computing the gradients for the current
batch.      '''
            model.zero_grad(set_to_none=True)
            '''In PyTorch, calling backward() on a tensor computes the gradients
of that tensor with respect to a scalar value.       '''
            self.loss.backward()
```

```
            # clip the gradients and update the parameters to avoid the issue of
exploding gradients during training.
            torch.nn.utils.clip_grad_norm_(model.parameters(),
config.grad_norm_clip)

            ''' the step() method updates the model parameters using the
gradients that were computed during the backward pass, scaled by the learning
rate and any other optimizer-specific scaling factors.'''
            self.optimizer.step()

            self.trigger_callbacks('on_batch_end')
            self.iter_num += 1
            tnow = time.time()
            self.iter_dt = tnow - self.iter_time
            self.iter_time = tnow

            # termination conditions
            if config.max_iters is not None and self.iter_num >=
config.max_iters:
                self.trigger_callbacks('on_finished_training')
                break
```

```
def get_config():

    CN = CfgNode
    C = CN()

    # system
    C.system = CN()
    C.system.seed = 3407

    # data
    C.data = CharDataset.get_default_config()

    # model
    C.model = GPT.get_default_config()
    C.model.model_type = 'gpt-mini'

    # trainer
    C.trainer = Trainer.get_default_config()
    C.trainer.learning_rate = 5e-4 # the model we're using is so small that we
can go a bit faster

    return C

# -----------------------------------------------------------------------------
# the Dataset class is responsible for loading the data and returning batches
class CharDataset(Dataset):
    """
    Emits batches of characters
    """
```

```
    @staticmethod
    def get_default_config():
        CN = CfgNode
        C = CN()
        C.block_size = 128
        return C

    def __init__(self, config, data):
        self.config = config
        # set() is a python built-in that returns a unique list of characters
        chars = sorted(list(set(data)))
        data_size, vocab_size = len(data), len(chars)
        print('data has %d characters, %d unique.' % (data_size, vocab_size))
        # create dictionaries to convert between characters and integers (stoi =
string to integer) and between integers and characters (itos = integer to string)
        self.stoi = { ch:i for i,ch in enumerate(chars) }
        self.itos = { i:ch for i,ch in enumerate(chars) }
        self.vocab_size = vocab_size
        self.data = data

    def get_vocab_size(self):
        return self.vocab_size

    def get_block_size(self):
        return self.config.block_size

    def __len__(self):
        return len(self.data) - self.config.block_size

    def __getitem__(self, idx):
        # grab a chunk of (block_size + 1) characters from the data
        chunk = self.data[idx:idx + self.config.block_size + 1]
        # encode every character to an integer
        dix = [self.stoi[s] for s in chunk]
        # return as tensors
        x = torch.tensor(dix[:-1], dtype=torch.long)
        y = torch.tensor(dix[1:], dtype=torch.long)
        return x, y

# -----------------------------------------------------------------------------
```

prepare the data

```
import os
import pickle
import requests
import numpy as np

# download the tiny shakespeare dataset
input_file = 'input.txt'
```

```
if not os.path.exists(input_file):
    data_url = 'https://raw.githubusercontent.com/karpathy/char-
rnn/master/data/tinyshakespeare/input.txt'
    with open(input_file, 'w') as f:
        f.write(requests.get(data_url).text)
with open(input_file, 'r') as f:
    data = f.read()
print(f"length of dataset in characters: {len(data):,}")
```

```
# get all the unique characters that occur in this text
chars = sorted(list(set(data)))
vocab_size = len(chars)
print("all the unique characters:", ''.join(chars))
print(f"vocab size: {vocab_size:,}")
```

```
# create a mapping from characters to integers
stoi = { ch:i for i,ch in enumerate(chars) }
itos = { i:ch for i,ch in enumerate(chars) }
def encode(s):
    return [stoi[c] for c in s] # encoder: take a string, output a list of
integers
def decode(l):
    ''.join([itos[i] for i in l]) # decoder: take a list of integers, output a
string

# create the train and test splits
n = len(data)
train_data = data[:int(n*0.9)]
val_data = data[int(n*0.9):]

# encode both to integers
train_ids = encode(train_data)
val_ids = encode(val_data)
print(f"train has {len(train_ids):,} tokens")
print(f"val has {len(val_ids):,} tokens")
```

```
# 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('train.bin')
val_ids.tofile('val.bin')

# save the meta information as well, to help us encode/decode later
meta = {
    'vocab_size': vocab_size,
    'itos': itos,
    'stoi': stoi,
}
```

```
with open('meta.pkl', 'wb') as f:
    pickle.dump(meta, f)
```

start training

```
if __name__ == '__main__':

    # get default config and overrides from the command line, if any
    config = get_config()
    set_seed(config.system.seed)

    # construct the training dataset
    text = open(input_file, 'r').read()
    train_dataset = CharDataset(config.data, text)

    # construct the model
    config.model.vocab_size = train_dataset.get_vocab_size()
    config.model.block_size = train_dataset.get_block_size()
    model = GPT(config.model)

    # construct the trainer object
    trainer = Trainer(config.trainer, model, train_dataset)

    # iteration callback
    def batch_end_callback(trainer):

        if trainer.iter_num % 200 == 0:
            print(f"iter_dt {trainer.iter_dt * 1000:.2f}ms; iter
{trainer.iter_num}: train loss {trainer.loss.item():.5f}")

        if trainer.iter_num % 1000 == 0:
            # evaluate both the train and test score
            model.eval()
            with torch.no_grad():
                # sample from the model...
                context = "In void of faith, "
                x = torch.tensor([train_dataset.stoi[s] for s in context],
dtype=torch.long)[None,...].to(trainer.device)
                y = model.generate(x, 150, temperature=1.0, do_sample=True,
top_k=10)[0]
                completion = ''.join([train_dataset.itos[int(i)] for i in y])
                print('\n===>', completion, '<===\n ')
            # revert model to training mode
            model.train()

    # finished_training_callback
    def finished_training_callback(trainer):
        model.eval()
        with torch.no_grad():
            # sample from the model...
            context = "In void of faith, "
            x = torch.tensor([train_dataset.stoi[s] for s in context],
```

```
dtype=torch.long)[None,...].to(trainer.device)
            y = model.generate(x, 500, temperature=1.0, do_sample=True, top_k=10)
[0]
            completion = ''.join([train_dataset.itos[int(i)] for i in y])
            clear_output(wait=True)  # Clear the output before printing the new
value
            print('\n', completion, '\n ')
            time.sleep(1)  # Wait for 1 second

    # register the callbacks
    trainer.set_callback('on_batch_end', batch_end_callback)
    trainer.set_callback('on_finished_training', finished_training_callback)

    # run the optimization
    trainer.run()
```