The Transformer

this notebook is based on the tiny shakespeare char-level GPT example in the minGPT project of Andrej Karpathy.

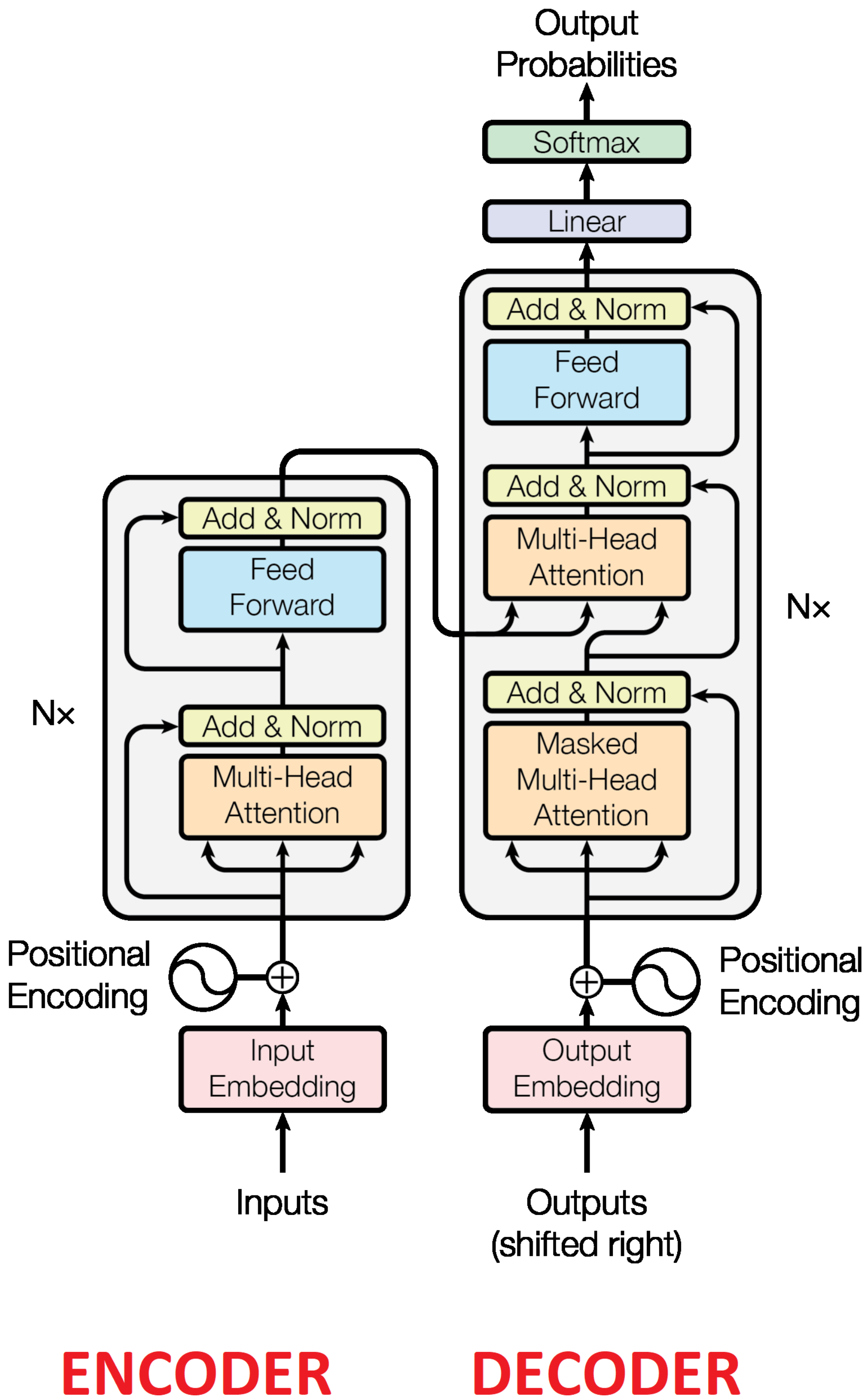
open in colab: Transformer.ipynb

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 model was designed to address the limitations of recurrent neural networks (RNNs) and convolutional neural networks (CNNs) in sequence-to-sequence tasks. The Transformer architecture eliminates the need for recurrent layers, enabling better parallelization, which significantly speeds up the training process.

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:



GPT

In 2018, OpenAI introduced GPT (Generative Pre-Trained Transformer) 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 phase, GPT predicts the next token in a sequence, conditioning on its own previous predictions

The Distinction Between Training Phase and Generative Phase Is Important:

> **in training mode**, the objective of the model is to learn the language - in this phase the model is fed large amounts of text data and trained to predict the next token in the sequence, given the context of the previous tokens. It is only in this mode that the model weights are updated through backpropagation, which minimizes the loss between the output vectors (called **logits**) and the actual tokens

> **in generative mode**, the objective of the model is "only to predict the next token" - it starts by predicting next tokens based on an initial user prompt, but then it continues to generate tokens **based on its own previous predictions**, in an **autoregressive** process. The prediction is made by generating a probability distribution from the logits using **Softmax** operation, and selecting the token from the distribution based on the probability scores. In this mode, the model weights are frowzen and not updated

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 also adds special tokens to the beginning and end of the sequence, such as the [CLS] token for classification tasks and the [SEP] token for sentence pair classification tasks. In the original Transformer paper, the authors used the [PAD] token to pad the input sequences to a fixed length, while in the GPT-3 paper, the authors used the [EOS] token to mark the end of the sequence.

2. **Token Embeddings**: after tokenization, each token is converted into an **embedding vector**. 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 concept of Word Embedding was introduced in Mikolov et al., 2013 **Word2Vec** paper *Distributed Representations of Words and Phrases and their Compositionality*, where the authors used a neural network to learn the embedding vectors for words. The representation of each token by a long embedding vecturs 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 (i.e. the language). The token embeddings are part of the

model's learnable parameters, and during the training process, the embeddings are updated through backpropagation in each iteration (processing one batch of data).

3. **Positional Encoding**: the embedding vectors are then 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. In the original Transformer paper, the authors used a sinusoidal function to compute the positional encoding. In the GPT-3 paper, the authors used a learned embedding matrix to compute the positional encoding. The resulting matrix, containing both the embedding vectors and positional encodings, is fed as input **(X)** into the Transformer's first self-attention block.

4. **Multiple Transformer Blocks** - GPT applies multiple Transformer Blocks over the embeddings of input sequences. Each block applies the following in sequence (see Decoder part of the Transformer architecture above):

   __4.1 Masked Multi-Head Attention layer__: Computes self-attention weights and generates a new representation of the input sequence (see more 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 block (Y)** - the output of the Transformer layers is a representation of the input tokens (X) after going through the Transformer layers. A token output vector captures the contextual information of the token, as it takes into account the relationships between the token and other tokens in the input sequence. The output vector and the embedding vector are of the same size and both represent the same token, but in different contexts. The output representation is more refined and context-aware compared to the initial token embedding vector.

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 model can either be used in training mode or inference mode:

- in Training phase, 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 weights through backpropagation (in each batch iteration).
- in 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.

NOTES on the above:

> **Pointwise 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, 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.

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

> **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** - regularization technique that 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 residual connection and normalization layer; (ii) after the Feed-Forward layer: After applying the pointwise feed-forward layer to each vector in the sequence, dropout is applied to the output before the residual connection and normalization layer; 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)*

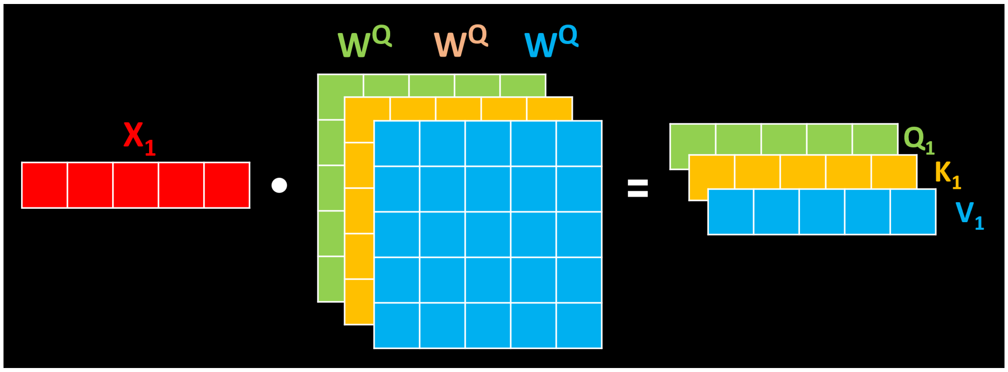
Masked Multi-Head Attention layer

- 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 (each represented by an embedding vector) into three vectors: a query vector Q, a key vector K, and a value vector V, by applying linear transformation, specifically by multiplying the input sequence of the embedding vectors (**X**) with corresponding weight matrices (W_Q, W_K, and W_V). *(the authors of Attention Is All You Need introduced the concept of Query/Key/Value, drawing an analogy to a retrieval system, which is arguably a bit confiusing)* [on Linear Transformation see here]

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

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

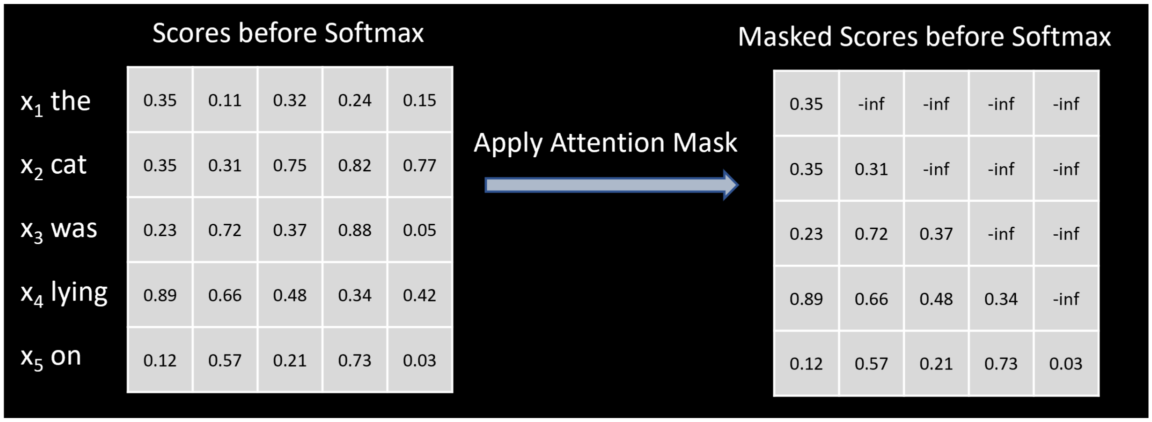


- **Attention Scores** are computed by multiplying each Q vector with each K vector. This is done by taking the dot product between the Q matrix and the transpose of K matrix (`Q @ K.transpose()`). The result is a score matrix that represents the relationships or affinities between each token in the input sequence. These scores quantify the degree to which each token should "pay attention" to the others.
- **Attention Weights** Attention weights are notmalized attention scores, after applying Softmax so that score values sum up to 1. The Softmax function can be sensitive to very large input values. These 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. Hence, the Attention Weights are computed as follows: `A (Attention Weights) = softmax(Q @ K.transpose(-1, -2) / 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 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.**

<u>**NOTES on the above:**</u>

- **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.



- **Multi-Head Attention** - In each Transformer block, the Self-Attention is conducted multiple times using Multi-Head Attention, which helps capture different aspects of the input data's context. Multi-Head Attention divides the Q, K, and V vectors into multiple subspaces (or "heads"), applies 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.

Following is the implementation of the Multi-Head Attention layer in PyTorch, based on Karpathy's minGPT project

In [ ]:
```
# 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
```

In [ ]:
```
# some utils..
class NewGELU(nn.Module):
    """  Implementation of the GELU activation function currently in Google BERT re
    def forward(self, x):
        return 0.5 * x * (1.0 + torch.tanh(math.sqrt(2.0 / math.pi) * (x + 0.044715

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)
```

```python
    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._

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

In [ ]:

```python
class CausalSelfAttention(nn.Module):
    ''' a vanilla multi-head masked self-attention layer with a projection at the e
    def __init__(self, config):
        super().__init__()
        assert config.n_embd % config.n_head == 0 # make sure that the number of he
        ''' key, query, value projections for all heads, but in a batch. The self.c
        self.c_attn = nn.Linear(config.n_embd, 3 * config.n_embd)
        # projection of the concatenated multi-head attention output back to embedd
        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 b
        # the residual connection is the addition of the input to the output of the
        self.resid_dropout = nn.Dropout(config.resid_pdrop)
        # causal mask to ensure that attention is only applied to the left in the i
        self.register_buffer("bias", torch.tril(torch.ones(config.block_size, confi
                                     .view(1, 1, config.block_size, config.block_si
        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 ten
    def forward(self, x):
        B, T, C = x.size()  # B=Batch size, T=block size or sequence length, i.e. t
        # calculate query, key, values for all heads in batch and move head forward
        # we split the x input tensor along the last dimension into 3 tensors of si
        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 o
        k = k.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T
        q = q.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T
        v = v.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T

        # causal self-attention; Self-attend: (B, nh, T, hs) x (B, nh, hs, T) -> (B
        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 softma
```

```
        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 out

        # apply residual connection and projection (linear transoformation) to the
        y = self.resid_dropout(self.c_proj(y))
        return y
```

In [ ]:
```
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
            # Specifically, it maps from a vector of size `config.n_embd` to a larg
            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 o
            # Specifically, it maps from a larger vector of size 4 * `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
            act = NewGELU(),
            # dropout layer, which in this case will apply to the output of the sec
            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 th
        # the lambda function is a shorthand for defining a function in Python. in
        # 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 -

    # 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
        x = x + self.mlpf(self.ln_2(x)) # this is calling the mlpf function defined
        return x # the output of the block is the input to the next block
```

In [ ]:
```
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,

        # transformer is an nn.ModuleList container, and h is a contrainer used to
        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 proje
        self.apply(self._init_weights)

        '''next code is initializing the weight parameters for a specific layer in
        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_

        # report number of parameters (note we don't count the decoder parameters i
        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
        # separate out all parameters to those that will and won't experience regul
        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():
```

```python
            for pn, p in m.named_parameters():
                fpn = '%s.%s' % (mn, pn) if mn else pn # full param name random not
                if pn.endswith('bias'):
                    # all biases will not be decayed
                    no_decay.add(fpn)
                elif pn.endswith('weight') and isinstance(m, whitelist_weight_modul
                    # weights of whitelist modules will be weight decayed
                    decay.add(fpn)
                elif pn.endswith('weight') and isinstance(m, blacklist_weight_modul
                    # 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_de
        assert len(param_dict.keys() - union_params) == 0, "parameters %s were not
                                                    % (str(param_dict.keys() - unio

        # create the pytorch optimizer object
        optim_groups = [
            {"params": [param_dict[pn] for pn in sorted(list(decay))], "weight_deca
            {"params": [param_dict[pn] for pn in sorted(list(no_decay))], "weight_d
        ]
        ''' note: we are using AdamW optimizer here, which is Adam with weight deca
        optimizer = torch.optim.AdamW(optim_groups, lr=train_config.learning_rate,
        return optimizer

    # in this forward method of the GPT module class, idx is the input tensor (indi
    # 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
        # This creates a tensor containing a sequence of integers starting from 0 t
        pos = torch.arange(0, t, dtype=torch.long, device=device).unsqueeze(0) # sh
        # forward the GPT model itself
        tok_emb = self.transformer.wte(idx) # token embeddings of shape (b, t, n_em
        pos_emb = self.transformer.wpe(pos) # position embeddings of shape (1, t, n
        # tok_emb and pos_emb are the token and positional embeddings, respectively
        x = self.transformer.drop(tok_emb + pos_emb)
        # now we feed the embeddings into the transformer blocks contained in self.
        for block in self.transformer.h:
            x = block(x)

        # apply the final layer norm and the linear layer to get the logits. the lo
        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 target
        loss = None
        if targets is not None:
            loss = F.cross_entropy(logits.view(-1, logits.size(-1)), targets.view(-
        return logits, loss
```

```python
    @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
        """ take a conditioning sequence of indices idx (LongTensor of shape (b,t))
        for _ in range(max_new_tokens):
            # if the sequence context is growing too long we must crop it at block_
            idx_cond = idx if idx.size(1) <= self.block_size else idx[:, -self.bloc
            # 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
```

In [ ]:

```python
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 et
        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 spec
        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 T
        train_loader = DataLoader(
            self.train_dataset,
            sampler=torch.utils.data.RandomSampler(self.train_dataset, replacement=
            shuffle=False,
            pin_memory=True,
            batch_size=config.batch_size,
            num_workers= 0 if platform.system() == 'Windows' else config.num_worker
        )
        ''' module.train() is a method in PyTorch's nn.Module class that sets the m
        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 DataLoa
        data_iter = iter(train_loader)

        while True:
            ''' fetch the next batch (x, y) and re-init iterator if needed. The exc
            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
            x, y = batch
            ''' forward the model - this will call the GPT class (model) forward()
```

```
            logits, self.loss = model(x, y)

            '''backprop and update the parameters. model.zero_grad(set_to_none=True
            model.zero_grad(set_to_none=True)
            '''In PyTorch, calling backward() on a tensor computes the gradients of
            self.loss.backward()

            # clip the gradients and update the parameters to avoid the issue of ex
            torch.nn.utils.clip_grad_norm_(model.parameters(), config.grad_norm_cli

            ''' the step() method updates the model parameters using the gradients
            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
```

In [ ]:
```
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

    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 = st
        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

In [ ]:

```
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/tin
    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):,}")
```

In [ ]:

```
# 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:,}")
```

In [ ]:
```
# 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 stri

# 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")
```

In [ ]:
```
# 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

In [ ]:
```
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)
```

```python
# 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}

    if trainer.iter_num % 1000 == 0:
        # evaluate both the train and test score
        model.eval()
        with torch.no_grad():
            # sample from the model...
            context = "O God, O God!"
            x = torch.tensor([train_dataset.stoi[s] for s in context], dtype=to
            y = model.generate(x, 150, temperature=1.0, do_sample=True, top_k=1
            completion = ''.join([train_dataset.itos[int(i)] for i in y])
            print(completion)
        # 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 = "O God, O God!"
        x = torch.tensor([train_dataset.stoi[s] for s in context], dtype=torch.
        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 val
        print(completion)
        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()
```