Section 1: Implementing GPT-2 (124M parameter version) from scratch

The original code in OpenAI's GitHub is made using TensorFlow. However, a PyTorch implementation of GPT-2 from Hugging Face as it is the current industry standard for deep learning. More specifically the GPT2LMHeadModel model from transformers (Hugging Face) will be used as reference.

Parameters of GPT-2

IMPORTANT: ALWAYS USE POWERS OF 2

Below is the number of parameters in each neuron layer in GPT-2. Only the first few lines are shown.

```
transformer.wte.weight torch.Size([50257, 768])
transformer.wpe.weight torch.Size([1024, 768])
transformer.h.0.ln_1.weight torch.Size([768])
transformer.h.0.ln_1.bias torch.Size([768])
transformer.h.0.attn.c_attn.weight torch.Size([768, 2304])
transformer.h.0.attn.c_attn.bias torch.Size([2304])
transformer.h.0.attn.c_proj.weight torch.Size([768, 768])
transformer.h.0.attn.c_proj.bias torch.Size([768])
transformer.h.0.ln_2.weight torch.Size([768])
transformer.h.0.ln_2.bias torch.Size([768])
transformer.h.0.mlp.c_fc.weight torch.Size([768, 3072])
```

Line 1: Token embedding layer, where num_tokens=50257 and emb_dim=768.

Line 2: Positional embedding layer, where context_length=1024 and emb_dim=768.

Line 3 onward: Every neural network layer within the transformer block.

Positional Embeddings

- Size: 1024*768.
- The '1024' is the context length, which is the number of tokens each token can 'attend' to. Tokens can only attend tokens before it.
- The '768' is the number of embedding dimensions in the positional embedding space.
- In the original "Attention is all you need" paper, positional embeddings were represented by predetermined sinusoids. However, in GPT-2 the shape of these sinusoids are learned through training.

Difference between GPT2 and the original transformer model architecture

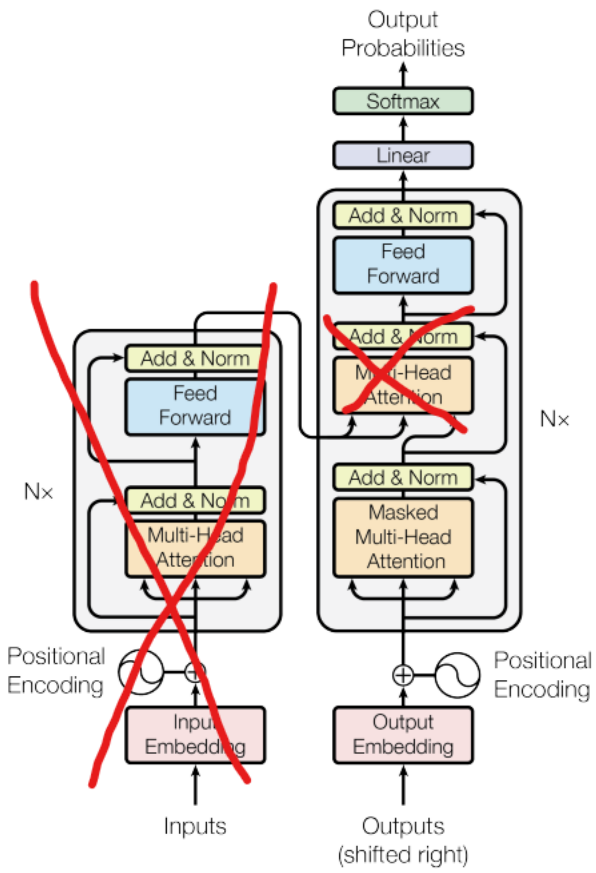


Figure 1: The Transformer - model architecture.

1. Decoder only model: GPT2 is a decoder only model. So, it doesn't have an encoder. This means that the entire left side of the transformer is removed. The attention block connecting to the encoder is also removed.
2. Revision of layer norms: Layer norms are now used before each attention block. A new layer norm is added before the final feedforward layer.

The Implementation

Libraries used

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

Implementation reference

The below image contains information about each layer, which will be used as reference for our model. This image was extracted from Hugging Face's "gpt2" model.

```
transformer.wte.weight torch.Size([50257, 768])
transformer.wpe.weight torch.Size([1024, 768])
transformer.h.0.ln_1.weight torch.Size([768])
transformer.h.0.ln_1.bias torch.Size([768])
transformer.h.0.attn.c_attn.weight torch.Size([768, 2304])
transformer.h.0.attn.c_attn.bias torch.Size([2304])
transformer.h.0.attn.c_proj.weight torch.Size([768, 768])
transformer.h.0.attn.c_proj.bias torch.Size([768])
transformer.h.0.ln_2.weight torch.Size([768])
transformer.h.0.ln_2.bias torch.Size([768])
transformer.h.0.mlp.c_fc.weight torch.Size([768, 3072])
transformer.h.0.mlp.c_fc.bias torch.Size([3072])
transformer.h.0.mlp.c_proj.weight torch.Size([3072, 768])
transformer.h.0.mlp.c_proj.bias torch.Size([768])
transformer.h.1.ln_1.weight torch.Size([768])
transformer.h.1.ln_1.bias torch.Size([768])
transformer.h.1.attn.c_attn.weight torch.Size([768, 2304])
transformer.h.1.attn.c_attn.bias torch.Size([2304])
transformer.h.1.attn.c_proj.weight torch.Size([768, 768])
transformer.h.1.attn.c_proj.bias torch.Size([768])
transformer.h.1.ln_2.weight torch.Size([768])
transformer.h.1.ln_2.bias torch.Size([768])
transformer.h.1.mlp.c_fc.weight torch.Size([768, 3072])
transformer.h.1.mlp.c_fc.bias torch.Size([3072])
transformer.h.1.mlp.c_proj.weight torch.Size([3072, 768])
...
transformer.h.11.mlp.c_proj.bias torch.Size([768])
transformer.ln_f.weight torch.Size([768])
transformer.ln_f.bias torch.Size([768])
lm_head.weight torch.Size([50257, 768])
```

The GPT block

1. GPTConfig contains all the dimensions of the model.
2. The transformer block is implemented using ModuleDict, which stores nns in dictionary format.
3. Self.lm_head contains the final nn layer.

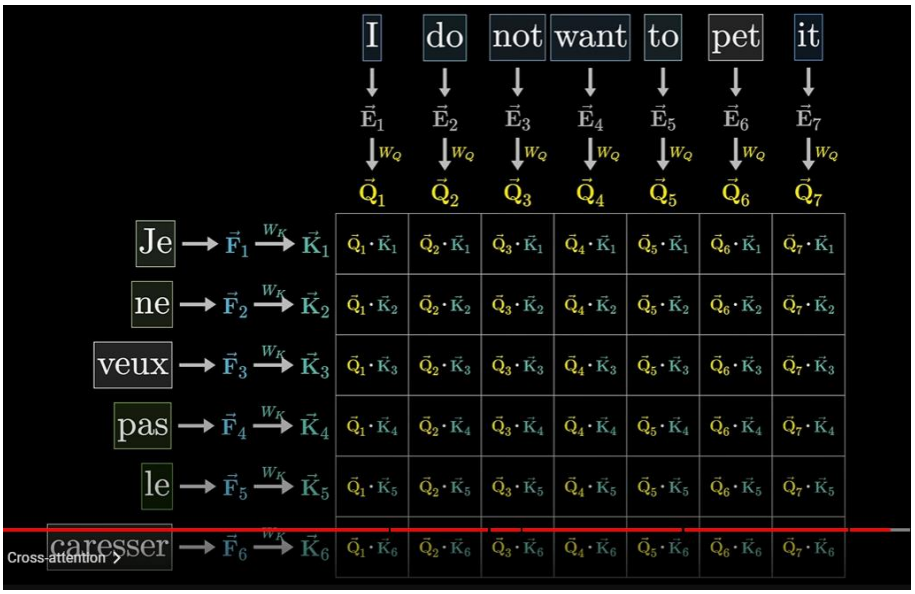
4. IMPORTANT: The final linear layer (`lm_head`) and the token emedding layer (`wte`) share the same weights. This is implemented by adding the following line at the very end:
    - `self.transformer.wte.weight = self.lm_head.weight`
    - This weight time scheme not only decreases the training time by reducing the number of parameters by around 30%, but also increase the model performance.

```python
@dataclass
class GPTConfig:
    block_size: int = 256
    vocab_size: int = 65
    n_layer: int = 6
    n_head: int = 6
    n_embd: int = 384

class GPT(nn.Module):

    def __init__(self, config):
        super().__init__()
        self.config = config

        self.transformer = nn.ModuleDict(dict(
            wte = nn.Embedding(config.vocab_size, config.n_embd),
            wpe = nn.Embedding(config.block_size, config.n_embd),
            h = nn.ModuleList([Block(config) for _ in range(config.n_layer)]),
            ln_f = nn.LayerNorm(config.n_embd),
        ))
        self.lm_head = nn.Linear(config.n_embd, config.vocab_size, bias=False)
```

Forward function in page 8.

The block (within self.transformer.h)

1. Attention is a communication mechanism that allows tokens to exchange information. In the below image, keys are the tokens within context and queries are the tokens each key can pay attention to. For each key, a dot product is used to find the relationship between itself and all the queries. The dot product highlights the strength and nature of the relationship between tokens and are used as weights. The weights are used to calculate a weighted sum for query with regards to all keys within context. Therefore, attention can be thought of as a weighted sum operation or a reduce operation.



2. The linear layer is map operation that maps the embedding space.
3. There the transformer block can be thought of as a series of repeated reduce and map operations.

```python
class Block(nn.Module):

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

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

The MLP

1. A GELU activation function is sandwiched between two linear layers.

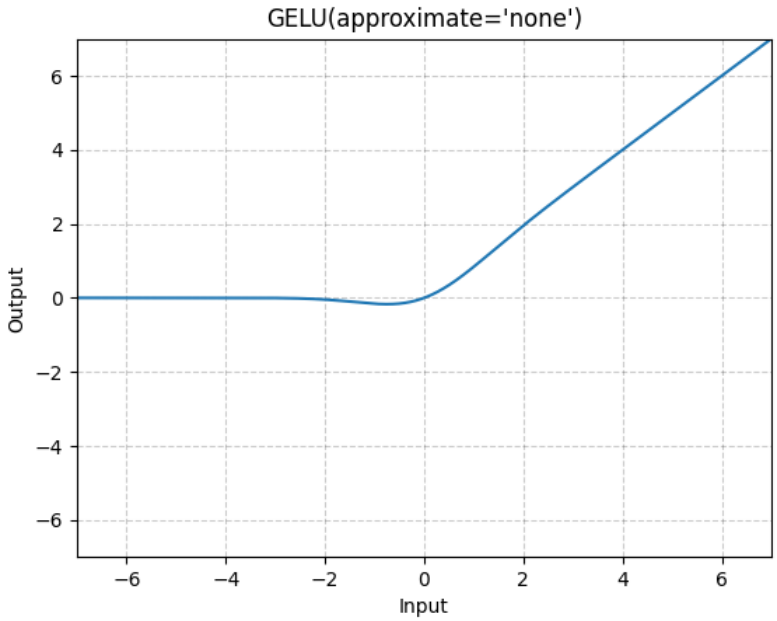
Applies the Gaussian Error Linear Units function.

$$\text{GELU}(x) = x * \Phi(x)$$

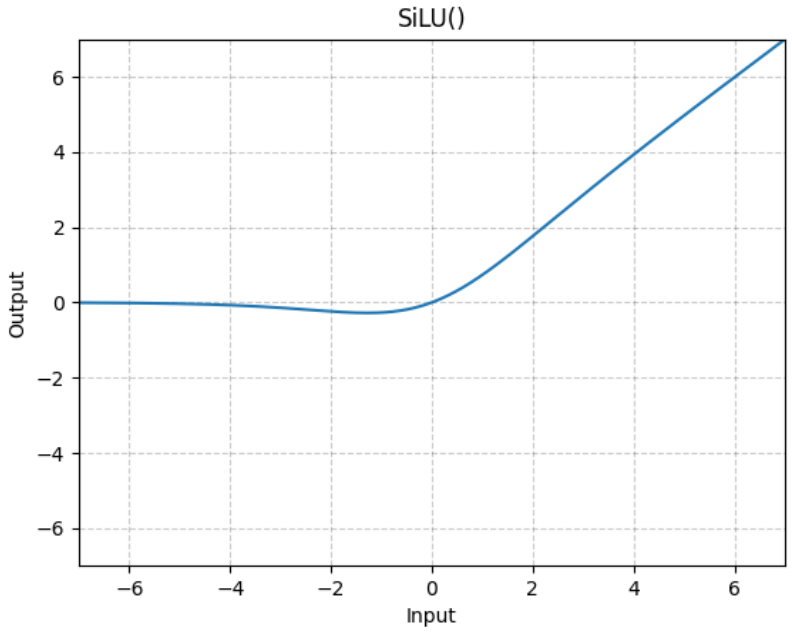
where $\Phi(x)$ is the Cumulative Distribution Function for Gaussian Distribution.

When the approximate argument is 'tanh', Gelu is estimated with:

$$\text{GELU}(x) = 0.5 * x * (1 + \text{Tanh}(\sqrt{2/\pi} * (x + 0.044715 * x^3)))$$



2. In GPT2 and Bert an approximation of GELU was used as at that time it was inefficient to computer GELU.
3. Why not use RELU?: It's because of the dead RELU neuron problem. In RELU if the input is less than 0, the output will always be 0. This causes the model to not learn anything in a lot of situations. GELU solves this by adding a slight curve for when the input is less than 0 (The bump slowly approaches 0). This allows the model to learn something instead of nothing. The research papers of GPT2 and Bert highlight the effectiveness in swapping to GELU.
4. Additional: In newer models such as LLaMA 3.1, this activation function further changes into SiLU. SiLU approaches 0 much slower and hence has a smoother bump.




```python
class MLP(nn.Module):

    def __init__(self, config):
        super().__init__()
        self.c_fc    = nn.Linear(config.n_embd, 4 * config.n_embd)
        self.gelu    = nn.GELU(approximate='tanh')
        self.c_proj  = nn.Linear(4 * config.n_embd, config.n_embd)

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

CasualSelfAttention

1. **IMPORTANT:** All variable names follow the hugging face GPT2 naming convention.

```python
class CausalSelfAttention(nn.Module):

    def __init__(self, config):
        super().__init__()
        assert config.n_embd % config.n_head == 0
        # key, query, value projections for all heads, but in a batch
        self.c_attn = nn.Linear(config.n_embd, 3 * config.n_embd)
        # output projection
        self.c_proj = nn.Linear(config.n_embd, config.n_embd)
        # regularization
        self.n_head = config.n_head
        self.n_embd = config.n_embd
        # not really a 'bias', more of a mask, but following the OpenAI/HF naming though
        self.register_buffer("bias", torch.tril(torch.ones(config.block_size, config.block_size))
                                     .view(1, 1, config.block_size, config.block_size))

    def forward(self, x):
        B, T, C = x.size() # batch size, sequence length, embedding dimensionality (n_embd)
        # calculate query, key, values for all heads in batch and move head forward to be the batch
        # nh is "number of heads", hs is "head size", and C (number of channels) = nh * hs
        # e.g. in GPT-2 (124M), n_head=12, hs=64, so nh*hs=C=768 channels in the Transformer
        qkv = self.c_attn(x)
        q, k, v = qkv.split(self.n_embd, dim=2)
        k = k.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
        q = q.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
        v = v.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
        # attention (materializes the large (T,T) matrix for all the queries and keys)
        att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))
        att = att.masked_fill(self.bias[:,:,:T,:T] == 0, float('-inf'))
        att = F.softmax(att, dim=-1)
        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 si
        # output projection
        y = self.c_proj(y)
        return y
```

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

What to do next?:

1. Fully understand the above code.
2. Implement it with minimal references to code. Try to only refer to the notes and hugging face variable names.
3. Continue video (Should be the part where he downloads and uploads the weights from hugging face.)

GPT block: Forward function

```python
def forward(self, idx):
    # idx is of shape (B, T)
    B, T = idx.size()
    assert T <= self.config.block_size, f"Cannot forward sequence of length {T}, block size i
    # forward the token and posisition embeddings
    pos = torch.arange(0, T, dtype=torch.long, device=idx.device) # shape (T)
    pos_emb = self.transformer.wpe(pos) # position embeddings of shape (T, n_embd)
    tok_emb = self.transformer.wte(idx) # token embeddings of shape (B, T, n_embd)
    x = tok_emb + pos_emb
    # forward the blocks of the transformer
    for block in self.transformer.h:
        x = block(x)
    # forward the final layernorm and the classifier
    x = self.transformer.ln_f(x)
    logits = self.lm_head(x) # (B, T, vocab_size)
    return logits
```

Document was

Important notes:

1. In the line where `pos` is defined, device=idx.device ensures that the tensor is stored in the GPU.
2. In the line where `x` is defined, tok_emb is broadcasted so that's its dimensions are compatible with pos_emb. The pos_embs are identical for each batch.
3. The forward function returns logits (not probabilities). These logits are later converted into probabilities during text generation.

Generating Text

```python
num_return_sequences = 5
max_length = 30

model = GPT.from_pretrained('gpt2')
model.eval()
model.to('cuda')

# prefix tokens
import tiktoken
enc = tiktoken.get_encoding('gpt2')
tokens = enc.encode("Hello, I'm a language model,")
tokens = torch.tensor(tokens, dtype=torch.long) # (8,)
tokens = tokens.unsqueeze(0).repeat(num_return_sequences, 1) # (5, 8)
x = tokens.to('cuda')
```

1. Note: To generate a new model, initialize it as GPT(GPTconfig()).
2. tiktoken is the tokenizer used in GPT-2.
3. In this code, "Hello, I'm a language model" is the prompt for all 5 batches.
4. Set model to `eval()` mode.

```
# generate! right now x is (B, T) where B = 5, T = 8
# set the seed to 42
torch.manual_seed(42)
torch.cuda.manual_seed(42)
while x.size(1) < max_length:
    # forward the model to get the logits
    with torch.no_grad():
        logits = model(x) # (B, T, vocab_size)
        # take the logits at the last position
        logits = logits[:, -1, :] # (B, vocab_size)
        # get the probabilities
        probs = F.softmax(logits, dim=-1)
        # do top-k sampling of 50 (huggingface pipeline default)
        # topk_probs here becomes (5, 50), topk_indices is (5, 50)
        topk_probs, topk_indices = torch.topk(probs, 50, dim=-1)
        # select a token from the top-k probabilities
        ix = torch.multinomial(topk_probs, 1) # (B, 1)
        # gather the corresponding indices
        xcol = torch.gather(topk_indices, -1, ix) # (B, 1)
        # append to the sequence
        x = torch.cat((x, xcol), dim=1)
```

5. max_length is total length of the `prompt + generated_text`.
6. `torch.no_grad()` indicates to PyTorch that no backpropagation takes place. This reduces the amount of space used by PyTorch in caching the intermediate tensors needed for backpropagation.
7. `torch.topk`, where k=50, is used to filter the 50 tokens with the highest probability and renormalize the probability distribution.
8. The next token is sampled using `torch.multinomial()`.

```
# print the generated text
for i in range(num_return_sequences):
    tokens = x[i, :max_length].tolist()
    decoded = enc.decode(tokens)
    print(">", decoded)
```

9. Decode and print the generated text of length max_length tokens.

Section 2: Debugging and setup

```
# get a data batch
import tiktoken
enc = tiktoken.get_encoding('gpt2')
with open('input.txt', 'r') as f:
    text = f.read()
text = text[:1000]
tokens = enc.encode(text)
B, T = 4, 32
buf = torch.tensor(tokens[:B*T + 1])
x = buf[:-1].view(B, T)
y = buf[1:].view(B, T)
```

1. The goal here is to overfit a single batch and ensure that the training loss decreases.
2. The `buf` tensor contains an additional entry because it also contains the target.
3. `y` is the target and `x` is used to predict `y`.

```
# get logits
model = GPT(GPTConfig())
model.to(device)
logits, loss = model(x, y)
```

4. At random initialization, each token in the vocabulary should be roughly given the same probability. Therefore, the initial loss approximately -ln(1/vocab_size).

```
# optimize!
optimizer = torch.optim.AdamW(model.parameters(), lr=3e-4)
for i in range(50):
    optimizer.zero_grad()
    logits, loss = model(x, y)
    loss.backward()
    optimizer.step()
    print(f"step {i}, loss: {loss.item()}")
```

5. `AdamW` fixes a bug in `Adam`
6. Adam is faster than SGD for LLMs
7. `lr=3e-4` is generally considered a good learning rate for debugging.
8. REMEMBER: Initialize gradients to 0 at every iteration as `optimizer.backward()` accumulates the gradients.
9. `optimizer.backward()` calculates the gradients via backpropagation.
10. `optimizer.step()` updates the parameters using the calculated gradients.
11. `loss.item()` converts the single valued tensor stored in the GPU to a float stored in the CPU.

DataLoader

```python
class DataLoaderLite:
    def __init__(self, B, T):
        self.B = B
        self.T = T

        # at init load tokens from disk and store them in memory
        with open('input.txt', 'r') as f:
            text = f.read()
        enc = tiktoken.get_encoding('gpt2')
        tokens = enc.encode(text)
        self.tokens = torch.tensor(tokens)
        print(f"loaded {len(self.tokens)} tokens")
        print(f"1 epoch = {len(self.tokens) // (B * T)} batches")

        # state
        self.current_position = 0

    def next_batch(self):
        B, T = self.B, self.T
        buf = self.tokens[self.current_position : self.current_position+B*T+1]
        x = (buf[:-1]).view(B, T) # inputs
        y = (buf[1:]).view(B, T) # targets
        # advance the position in the tensor
        self.current_position += B * T
        # if loading the next batch would be out of bounds, reset
        if self.current_position + (B * T + 1) > len(self.tokens):
            self.current_position = 0
        return x, y
```

Model Initialization

```python
# initialize weights
def _init_weights(self, module):
        if isinstance(module, nn.Linear):
                nn.init.normal_(module.weight, mean=0.0, std=0.02)
                if module.bias is not None:
                        nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
                nn.init.normal_(module.weight, mean=0.0, std=0.02)
        # no need to initialize layernorm as PyTorch default is what we want
```

1. Initializes all weight using a normal distribution with mean 0 and std 0.02.
2. The layernorm isn't initialized as PyTorch default is needed.

Section 3: Optimizing Training

Introduction

In order to optimize training, you must always start by asking (refer to the official GPU documentation);

1. What hardware do you have?
2. What does it offer?
3. Are you fully utilizing it?

My GPU: NVIDIA GeForce RTX 4080 laptop, 12GB VRAM, 80W

Theoretical Performance	
Pixel Rate:	280.6 GPixel/s
Texture Rate:	761.5 GTexel/s
FP16 (half):	48.74 TFLOPS (1:1)
FP32 (float):	48.74 TFLOPS
FP64 (double):	761.5 GFLOPS (1:64)

GPU Specifications

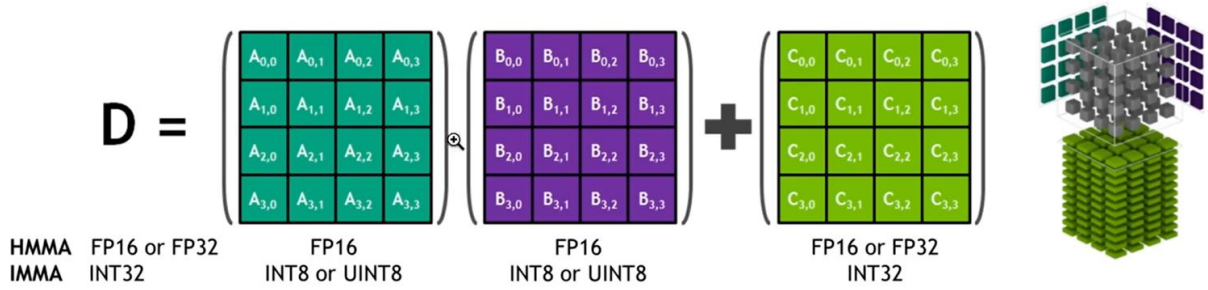
NVIDIA A100 TENSOR CORE GPU SPECIFICATIONS (SXM4 AND PCIE FORM FACTORS)

	A100 40GB PCIe	A100 80GB PCIe	A100 40GB SXM	A100 80GB SXM
FP64	9.7 TFLOPS			
FP64 Tensor Core	19.5 TFLOPS			
FP32	19.5 TFLOPS			
Tensor Float 32 (TF32)	156 TFLOPS | 312 TFLOPS*			
BFLOAT16 Tensor Core	312 TFLOPS | 624 TFLOPS*			
FP16 Tensor Core	312 TFLOPS | 624 TFLOPS*			
INT8 Tensor Core	624 TOPS | 1248 TOPS*			
GPU Memory	40GB HBM2	80GB HBM2e	40GB HBM2	80GB HBM2e
GPU Memory Bandwidth	1,555GB/s	1,935GB/s	1,555GB/s	2,039GB/s
Max Thermal Design Power (TDP)	250W	300W	400W	400W
Multi-Instance GPU	Up to 7 MIGs @ 5GB	Up to 7 MIGs @ 10GB	Up to 7 MIGs @ 5GB	Up to 7 MIGs @ 10GB
Form Factor	PCIe		SXM	
Interconnect	NVIDIA® NVLink® Bridge for 2 GPUs: 600GB/s ** PCIe Gen4: 64GB/s		NVLink: 600GB/s PCIe Gen4: 64GB/s	
Server Options	Partner and NVIDIA-Certified Systems™ with 1-8 GPUs		NVIDIA HGX™ A100-Partner and NVIDIA-Certified Systems with 4,8, or 16 GPUs NVIDIA DGX™ A100 with 8 GPUs	

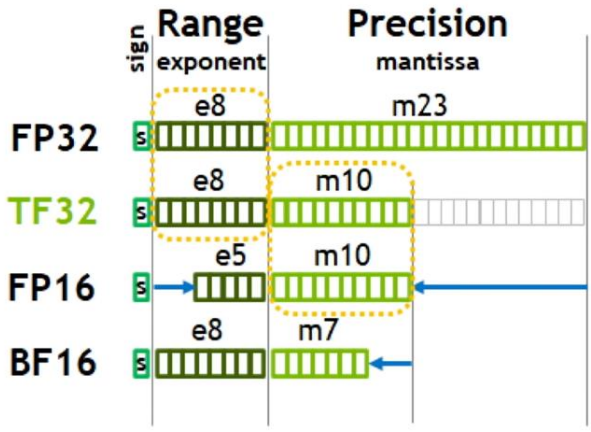
* With sparsity

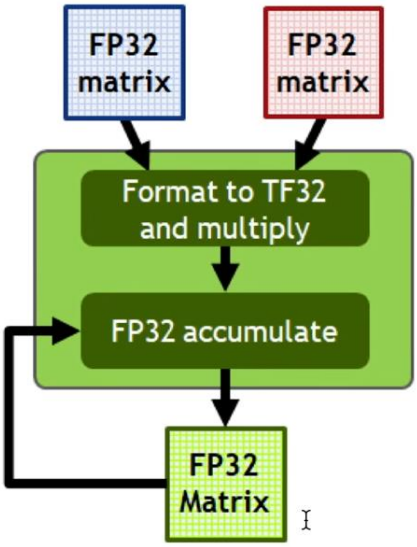
** SXM4 GPUs via HGX A100 server boards; PCIe GPUs via NVLink Bridge for up to two GPUs

1. Tensor core precision: By default, PyTorch creates float32 tensors. However, this level of accuracy isn't needed for deep learning tasks. Therefore, the precision is reduced. BF16 is used for training and Int8 is used for inference.
   In memory, floats are stored as two components; a) range – exponent b) precision – mantissa. Range indicates the range of possible values and precision highlights the number of possibilities the float can take within that particular range.
   Using lower precisions allow the model to be trained much faster. This allows the model to be trained on much more data, which can more than compensate the loss in model performance due to lower precision.
2. GPU memory: There's a limit to the number of bits a GPU could store.
3. GPU memory bandwidth: There's a limit to the speed that the GPU can access its memory. When well optimized the memory bandwidth would usually be the bottleneck. This is because it cannot load the data fast enough to the tensor cores. This results in many tensor cores to be idle about half the time.
4. Tensor cores: It is a structure in the GPU architecture. Each tensor core performs accelerated 4*4 matrix multiplications and accumulation by breaking down the input matrices into smaller 4*4 matrices. In this 124M parameter version of GPT-2, the very first matrix multiplication of transforming the 50257-dimensional space into a 768-dimensional space dominates the number of tensor core operations.



5. TF32 vs FP32: The exponent bits are the same, but for TF32 the last 13 mantissa bits get truncated. Thus, saving memory. Despite the loss in precision, empirically, it does not affect the training of the model.
   Code (TF32): `torch.set_float32_matmul_precision('high')`





6. BF16: Compared to TF32, the number of bits for the range is reduced by 3. Is used as compromise for FP16 given the implementation complexity of FP16 (implementation of gradient scalers, etc... is required).
   Implementation: Andrej Karpathy recommends using the 'automatic mixed precision' package in PyTorch. The statement `torch.autocast()` is used alongside the `with` statement. It must be ensured that on the model forward pass and the loss calculation are within the `with` statement. The optimizer and backpropagation must be not be inside.

```
# Enables autocasting for the forward pass (model + loss)
with torch.autocast(device_type="cuda"):
    output = model(input)
    loss = loss_fn(output, target)
```

   Mostly, matrices involved in matrix multiplication are converted to float16.

CUDA Ops that can autocast to `float16`

`__matmul__`, `addbmm`, `addmm`, `addmv`, `addr`, `baddbmm`, `bmm`, `chain_matmul`, `multi_dot`, `conv1d`, `conv2d`, `conv3d`, `conv_transpose1d`, `conv_transpose2d`, `conv_transpose3d`, `GRUCell`, `linear`, `LSTMCell`, `matmul`, `mm`, `mv`, `prelu`, `RNNCell`

CUDA Ops that can autocast to `float32`

`__pow__`, `__rdiv__`, `__rpow__`, `__rtruediv__`, `acos`, `asin`, `binary_cross_entropy_with_logits`, `cosh`, `cosine_embedding_loss`, `cdist`, `cosine_similarity`, `cross_entropy`, `cumprod`, `cumsum`, `dist`, `erfinv`, `exp`, `expm1`, `group_norm`, `hinge_embedding_loss`, `kl_div`, `l1_loss`, `layer_norm`, `log`, `log_softmax`, `log10`, `log1p`, `log2`, `margin_ranking_loss`, `mse_loss`, `multilabel_margin_loss`, `multi_margin_loss`, `nll_loss`, `norm`, `normalize`, `pdist`, `poisson_nll_loss`, `pow`, `prod`, `reciprocal`, `rsqrt`, `sinh`, `smooth_l1_loss`, `soft_margin_loss`, `softmax`, `softmin`, `softplus`, `sum`, `renorm`, `tan`, `triplet_margin_loss`

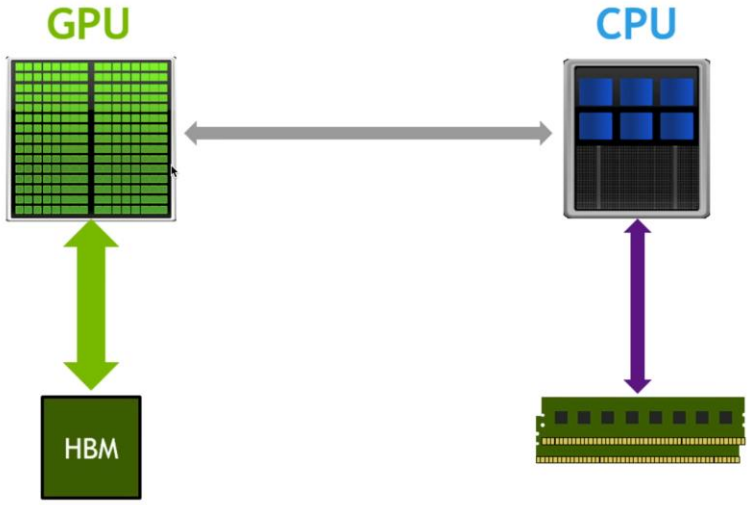
```
(env-gpt2) (base) lakindu_linux@lakindu-legion-Pro-7-16IRX9H:~/Desktop/GPT2-124M/Code$ python train_gpt2.py
loaded 338025 tokens
1 epoch = 82 batches
step 0, loss: 10.9281005859375, dt: 490.74ms, tok/sec: 8346.610042729333
step 1, loss: 9.55609130859375, dt: 301.59ms, tok/sec: 13581.160450221269
step 2, loss: 9.059257507324219, dt: 284.73ms, tok/sec: 14385.331525802943
step 3, loss: 8.874286651611328, dt: 287.41ms, tok/sec: 14251.215828758475
step 4, loss: 8.495040893554688, dt: 292.83ms, tok/sec: 13987.854723868038
step 5, loss: 8.065093994140625, dt: 287.14ms, tok/sec: 14265.04893912232
step 6, loss: 7.984556198120117, dt: 285.72ms, tok/sec: 14335.600108811204
step 7, loss: 7.792964935302734, dt: 290.57ms, tok/sec: 14096.245089456042
step 8, loss: 7.684320449829102, dt: 290.07ms, tok/sec: 14120.831166013088
step 9, loss: 7.3902587890625, dt: 285.32ms, tok/sec: 14355.820823285017
step 10, loss: 7.410587310791016, dt: 288.25ms, tok/sec: 14209.94752204916
step 11, loss: 7.434528350830078, dt: 291.53ms, tok/sec: 14049.91370768433
step 12, loss: 7.5016326904296875, dt: 287.68ms, tok/sec: 14237.94049191713
step 13, loss: 7.400749206542969, dt: 286.52ms, tok/sec: 14295.733534817615
step 14, loss: 7.04498291015625, dt: 290.12ms, tok/sec: 14118.487045490967
step 15, loss: 7.057903289794922, dt: 289.56ms, tok/sec: 14145.677864727426
step 16, loss: 6.8657989501953125, dt: 286.43ms, tok/sec: 14300.064911801723
step 17, loss: 6.71055793762207, dt: 288.99ms, tok/sec: 14173.511488634293
step 18, loss: 6.840118408203125, dt: 290.67ms, tok/sec: 14091.562374093639
step 19, loss: 6.850734710693359, dt: 288.21ms, tok/sec: 14211.78129573056
step 20, loss: 7.045553207397461, dt: 288.45ms, tok/sec: 14200.022799558292
step 21, loss: 6.860483169555664, dt: 290.64ms, tok/sec: 14093.007324646134
step 22, loss: 16.1326961517334, dt: 289.60ms, tok/sec: 14143.663157617426
step 23, loss: 6.9165496826171875, dt: 288.15ms, tok/sec: 14214.95625351136
step 24, loss: 6.962497711181641, dt: 289.01ms, tok/sec: 14172.307186684859
step 25, loss: 6.965065002441406, dt: 288.73ms, tok/sec: 14186.292026797268
step 26, loss: 6.784961700439453, dt: 289.05ms, tok/sec: 14170.588780074582
step 27, loss: 6.818553924560547, dt: 288.13ms, tok/sec: 14215.626703736802
step 28, loss: 6.829769134521484, dt: 289.56ms, tok/sec: 14145.82928208958
step 29, loss: 6.65318489074707, dt: 289.28ms, tok/sec: 14159.411781577459
step 30, loss: 6.63051700592041, dt: 289.14ms, tok/sec: 14166.136889791893
step 31, loss: 6.549281120300293, dt: 288.86ms, tok/sec: 14179.676509428979
step 32, loss: 6.567153453826904, dt: 288.89ms, tok/sec: 14178.307341497613
step 33, loss: 6.642053604125977, dt: 289.47ms, tok/sec: 14149.755617528699
```

`torch.compile(model)`

A compiler for neural networks that makes the code run much faster by reducing the python overhead and GPU read/write. Always use it unless you are debugging.

Normally, the python interpreter runs line by line. It would not know what code comes next. But with `torch.compile()` PyTorch “looks at” the entire code and performs the necessary optimizations as follows;

1. Compiles the entire model as a single object with no Python interpreter involved. The compiled code is much more efficient.
2. Kernel fusion: When performing a series of operations, `torch.compile()` minimizes the number of times the data travels between the HBM/VRAM and GPU.

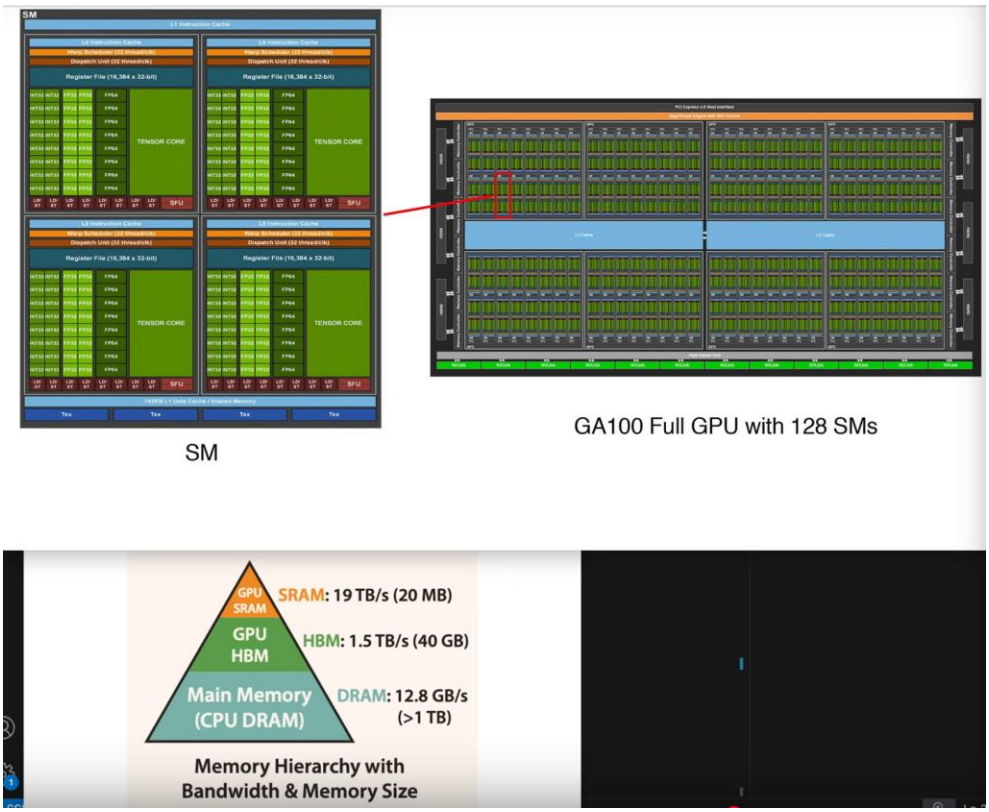


IMPORTANT: Requires Linux

```
(env-gpt2) (base) lakindu_linux@lakindu-legion-Pro-7-16IRX9H:~/Desktop/GPT2-124M/Code$ python train_gpt2.py
loaded 338025 tokens
1 epoch = 47 batches
step 0, loss: 10.936777114868164, dt: 10264.93ms, tok/sec: 698.3000687806998
step 1, loss: 9.533014297485352, dt: 257.03ms, tok/sec: 27888.001038909
step 2, loss: 8.871259689331055, dt: 265.60ms, tok/sec: 26987.56764626942
step 3, loss: 8.607799530029297, dt: 264.79ms, tok/sec: 27070.57588507906
step 4, loss: 8.420676231384277, dt: 261.83ms, tok/sec: 27376.357966931402
step 5, loss: 8.224714279174805, dt: 258.24ms, tok/sec: 27756.768051730553
step 6, loss: 8.203825950622559, dt: 265.47ms, tok/sec: 27001.09215107794
step 7, loss: 8.121841430664062, dt: 264.30ms, tok/sec: 27120.367060929053
step 8, loss: 7.821747303009033, dt: 260.96ms, tok/sec: 27468.0011767522
step 9, loss: 7.580530643463135, dt: 260.50ms, tok/sec: 27516.621824558577
step 10, loss: 7.4632954597473145, dt: 264.85ms, tok/sec: 27063.923260838255
step 11, loss: 7.515355587005615, dt: 263.84ms, tok/sec: 27167.76375206256
step 12, loss: 7.319307804107666, dt: 260.50ms, tok/sec: 27516.3951664496
step 13, loss: 7.250756740570068, dt: 263.21ms, tok/sec: 27232.977173582225
step 14, loss: 7.192774295806885, dt: 265.62ms, tok/sec: 26985.944647200853
step 15, loss: 9.692870140075684, dt: 264.57ms, tok/sec: 27092.555210118717
step 16, loss: 6.999518394470215, dt: 260.45ms, tok/sec: 27521.508473924627
step 17, loss: 6.888975620269775, dt: 264.01ms, tok/sec: 27150.025756824583
step 18, loss: 6.826714992523193, dt: 265.79ms, tok/sec: 26969.047991187563
step 19, loss: 6.8369035720825195, dt: 261.61ms, tok/sec: 27399.710801209552
step 20, loss: 6.741504192352295, dt: 261.29ms, tok/sec: 27433.287805803873
step 21, loss: 6.851746082305908, dt: 264.91ms, tok/sec: 27058.46713898969
step 22, loss: 6.658259868621826, dt: 264.30ms, tok/sec: 27120.88082147171
step 23, loss: 6.683382987976074, dt: 261.21ms, tok/sec: 27441.651086310652
step 24, loss: 6.582030773162842, dt: 262.62ms, tok/sec: 27294.093093856445
step 25, loss: 6.709610462188721, dt: 265.71ms, tok/sec: 26976.815887774155
step 26, loss: 6.55488920211792, dt: 263.32ms, tok/sec: 27221.61001407947
step 27, loss: 6.6046552658081055, dt: 260.95ms, tok/sec: 27468.80425689763
step 28, loss: 6.524278163909912, dt: 262.75ms, tok/sec: 27280.743878459347
step 29, loss: 6.589548587799072, dt: 264.58ms, tok/sec: 27091.700740983706
step 30, loss: 6.6378865242004395, dt: 263.96ms, tok/sec: 27155.347175605613
step 31, loss: 6.570703506469727, dt: 262.02ms, tok/sec: 27357.101910416524
step 32, loss: 6.713608264923096, dt: 263.71ms, tok/sec: 27181.494086748186
step 33, loss: 6.633688926696777, dt: 264.35ms, tok/sec: 27115.352801700617
```

Note: Increased batch_size to 8 and decreased context length to 896 (so that batch_size can be increased)

GPU memory hierarchy



The data stored in the SRAM is accessed fastest by the GPU because it is inside the GPU chip but is also the smallest. HBM on the other hand is outside the chip and is accessed much slower (dependent on memory bandwidth). The main memory is slowest.

The part boxed in red is an SM. The picture on the left shows the structure of an SM.

There's L2 cache on the GPU chip and L1 cache/registers on each SM. Despite all this, there's very little memory in the chip.

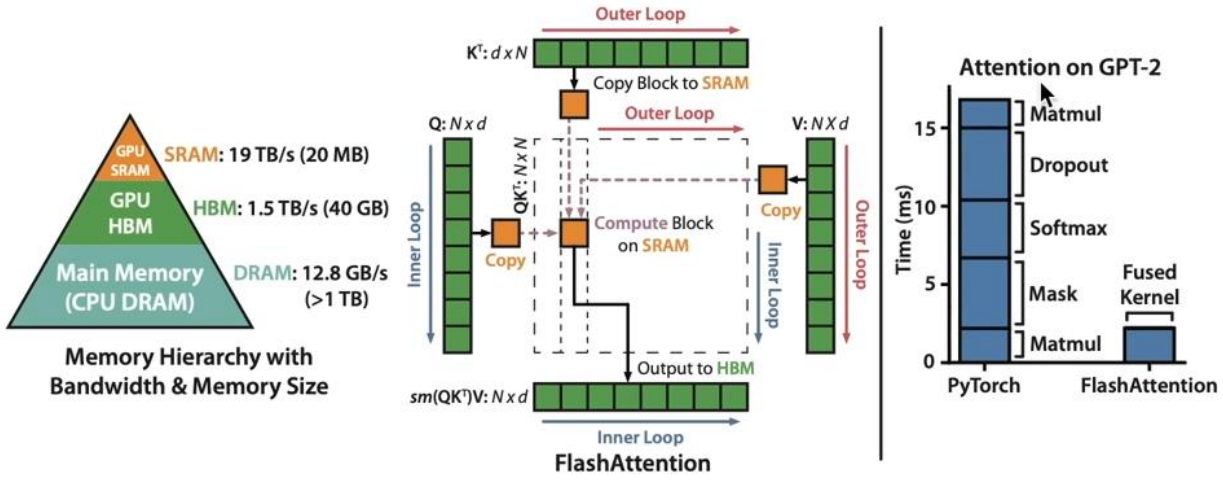
Flash-attention

Original attention:

```python
class CausalSelfAttention(nn.Module):

    def __init__(self, config):
        super().__init__()
        assert config.n_embd % config.n_head == 0
        # key, query, value projections for all heads, but in a batch
        self.c_attn = nn.Linear(config.n_embd, 3 * config.n_embd)
        # output projection
        self.c_proj = nn.Linear(config.n_embd, config.n_embd)
        # regularization
        self.n_head = config.n_head
        self.n_embd = config.n_embd
        # not really a 'bias', more of a mask, but following the OpenAI/HF naming though
        self.register_buffer("bias", torch.tril(torch.ones(config.block_size, config.block_size))
                                     .view(1, 1, config.block_size, config.block_size))

    def forward(self, x):
        B, T, C = x.size() # batch size, sequence length, embedding dimensionality (n_embd)
        # calculate query, key, values for all heads in batch and move head forward to be the batch
        # nh is "number of heads", hs is "head size", and C (number of channels) = nh * hs
        # e.g. in GPT-2 (124M), n_head=12, hs=64, so nh*hs=C=768 channels in the Transformer
        qkv = self.c_attn(x)
        q, k, v = qkv.split(self.n_embd, dim=2)
        k = k.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
        q = q.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
        v = v.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
        # attention (materializes the large (T,T) matrix for all the queries and keys)
        att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))
        att = att.masked_fill(self.bias[:,:,:T,:T] == 0, float('-inf'))
        att = F.softmax(att, dim=-1)
        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 si
        # output projection
        y = self.c_proj(y)
        return y
```

Flash-attention:

```python
y = F.scaled_dot_product_attention(q, k, v, is_causal=True) # flash-attention (7.6x faster than regul
```




```python
#---------------- Flash attention code removal -------------------------
        # att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1))) #standard attention formula
        # att = att.masked_fill(self.bias[:,:,:T,:T] == 0, float('-inf')) #masking
        # att = F.softmax(att, dim=-1) #standard attention formula
        # y = att @ v #standard attention formula
```

1. `torch.compile()` does not detect the kernel fusion for attention as it requires as an algorithmic rewrite. Therefore, flash-attention must be implemented.
2. Flash-attention is a kernel fusion operation that implements the above 4 lines very efficiently (7.6x faster)
3. Flash-attention achieves high efficiency by being very mindful of the memory hierarchy and thereby minimizing the number of read/writes between the HBM and SRAM.
4. Majority of the efficiency is contributed by not materializing the very big n*n `att` matrix. This achieved by the 'online SoftMax trick', which allows for the incremental calculation of the SoftMax without having all the inputs simultaneously.

```
(env-gpt2) (base) lakindu_linux@lakindu-legion-Pro-7-16IRX9H:~/Desktop/GPT2-124M/Code$ python train_gpt2.py
loaded 338025 tokens
1 epoch = 41 batches
step 0, loss: 10.934800148010254, dt: 12047.53ms, tok/sec: 679.9735031543808
step 1, loss: 9.547595977783203, dt: 220.81ms, tok/sec: 37099.57940676932
step 2, loss: 8.786746978759766, dt: 231.96ms, tok/sec: 35316.09831024134
step 3, loss: 8.56620979309082, dt: 237.68ms, tok/sec: 34465.99703686465
step 4, loss: 8.40669059753418, dt: 237.90ms, tok/sec: 34434.35762096463
step 5, loss: 8.342044830322266, dt: 227.45ms, tok/sec: 36016.27072366421
step 6, loss: 8.276555061340332, dt: 228.34ms, tok/sec: 35877.08805302666
step 7, loss: 7.9643096923828125, dt: 233.71ms, tok/sec: 35052.69509738011
step 8, loss: 7.672549724578857, dt: 236.25ms, tok/sec: 34675.14347303069
step 9, loss: 7.6112141609191895, dt: 229.82ms, tok/sec: 35645.87726562087
step 10, loss: 7.578176498413086, dt: 229.92ms, tok/sec: 35629.68731289346
step 11, loss: 11.070937156677246, dt: 232.28ms, tok/sec: 35267.70544467311
step 12, loss: 7.543802261352539, dt: 235.94ms, tok/sec: 34720.34385932293
step 13, loss: 7.367273330688477, dt: 231.67ms, tok/sec: 35360.7299103008
step 14, loss: 7.253439903259277, dt: 228.48ms, tok/sec: 35854.51301562125
step 15, loss: 7.112689018249512, dt: 231.44ms, tok/sec: 35395.079848652946
step 16, loss: 7.035983085632324, dt: 234.02ms, tok/sec: 35005.59153176099
step 17, loss: 7.051296234130859, dt: 234.66ms, tok/sec: 34909.95415544395
step 18, loss: 6.91270637512207, dt: 228.80ms, tok/sec: 35804.261289734335
step 19, loss: 6.764350891113281, dt: 232.55ms, tok/sec: 35227.063908490956
step 20, loss: 6.803074359893799, dt: 233.10ms, tok/sec: 35143.580794030044
step 21, loss: 6.707216262817383, dt: 234.84ms, tok/sec: 34883.65599742533
step 22, loss: 6.782576560974121, dt: 231.70ms, tok/sec: 35355.70868155953
step 23, loss: 6.641103744506836, dt: 230.27ms, tok/sec: 35575.90052753374
step 24, loss: 6.586719512939453, dt: 231.84ms, tok/sec: 35334.65688614565
step 25, loss: 6.630091190338135, dt: 234.03ms, tok/sec: 35004.165046506176
step 26, loss: 6.6863908767700195, dt: 231.68ms, tok/sec: 35359.05600892421
step 27, loss: 6.578578948974609, dt: 230.92ms, tok/sec: 35474.74508062926
step 28, loss: 6.768855094909668, dt: 230.92ms, tok/sec: 35475.07471710581
step 29, loss: 6.605204105377197, dt: 234.28ms, tok/sec: 34966.86809879945
step 30, loss: 6.59507942199707, dt: 233.28ms, tok/sec: 35116.85782265647
step 31, loss: 6.6183695793151855, dt: 230.07ms, tok/sec: 35605.835803619266
step 32, loss: 6.490643501281738, dt: 232.93ms, tok/sec: 35168.724538560506
```

Note: batch_size = 8 and increased context length back to 1024 (4GB+ more VRAM available)

Nice numbers

IMPORTANT: Make sure that the change of hyperparameters doesn't break any port of the model or the code.

1. Nice numbers: Numbers divisible by 2 many times.
2. Ugly numbers: Prime numbers. Odd numbers.

When dealing with Cuda or neural networks, nice numbers must always be used. This is because many kernels in Cuda use blocks tiles. Each block tile performs a small chunk of the calculation and is usually of size 2^n.

When a calculation is done, the data is broken down into smaller chunks. If these chunks aren't divisible by 2, there will be some leftover data. Another phase of calculations will be performed on this data. However, the kernels written for this phase are usually very inefficient. Therefore, by using nice numbers these kernels wouldn't have to be used. Thus. increasing the computational efficiency.

Changes

1. Increase vocab_size from 50257 to 50304: Adds dummy tokens that are never used. Despite slightly increasing the memory used, the time taken decreases by about 4%.

```
(env-gpt2) (base) lakindu_linux@lakindu-legion-Pro-7-16IRX9H:~/Desktop/GPT2-124M/Code$ python train_gpt2.py
loaded 338025 tokens
1 epoch = 41 batches
step 0, loss: 10.937861442565918, dt: 8705.23ms, tok/sec: 941.043797185551
step 1, loss: 9.576279640197754, dt: 219.54ms, tok/sec: 37314.85065040844
step 2, loss: 8.814200401306152, dt: 222.07ms, tok/sec: 36890.064802895395
step 3, loss: 8.655116081237793, dt: 225.00ms, tok/sec: 36409.63736106534
step 4, loss: 8.454036712646484, dt: 228.01ms, tok/sec: 35927.65725841447
step 5, loss: 8.386557579040527, dt: 219.36ms, tok/sec: 37345.633829576094
step 6, loss: 8.343727111816406, dt: 220.94ms, tok/sec: 37077.800704224595
step 7, loss: 8.009283065795898, dt: 223.34ms, tok/sec: 36679.14404146184
step 8, loss: 7.728996276855469, dt: 227.18ms, tok/sec: 36059.587313981065
step 9, loss: 7.6877593994140625, dt: 222.11ms, tok/sec: 36882.85778630322
step 10, loss: 7.656818866729736, dt: 219.82ms, tok/sec: 37266.85087072339
step 11, loss: 7.452986717224121, dt: 221.41ms, tok/sec: 37000.06285320468
step 12, loss: 14.519731521606445, dt: 227.55ms, tok/sec: 36000.76105936923
step 13, loss: 7.255451679229736, dt: 223.86ms, tok/sec: 36593.82817171114
step 14, loss: 7.196025848388672, dt: 221.79ms, tok/sec: 36935.470326702954
step 15, loss: 7.067119121551514, dt: 221.54ms, tok/sec: 36978.201816853194
step 16, loss: 6.927388668060303, dt: 226.15ms, tok/sec: 36224.541542042825
step 17, loss: 6.935396194458008, dt: 224.18ms, tok/sec: 36541.71739722298
step 18, loss: 6.79019832611084, dt: 221.74ms, tok/sec: 36943.49233491961
step 19, loss: 6.6626877784729, dt: 220.76ms, tok/sec: 37108.67477822106
step 20, loss: 6.669270038604736, dt: 224.17ms, tok/sec: 36543.310819391074
step 21, loss: 6.519961833953857, dt: 225.77ms, tok/sec: 36284.71507199973
step 22, loss: 6.613542079925537, dt: 223.50ms, tok/sec: 36653.945468840175
step 23, loss: 6.452114105224609, dt: 221.70ms, tok/sec: 36951.6370452661
step 24, loss: 6.416601657867432, dt: 222.63ms, tok/sec: 36796.67153365968
step 25, loss: 6.482755184173584, dt: 225.45ms, tok/sec: 36335.48009834765
step 26, loss: 6.643374443054199, dt: 225.04ms, tok/sec: 36402.96392731218
step 27, loss: 6.52678108215332, dt: 222.44ms, tok/sec: 36827.98687644631
step 28, loss: 6.725359916687012, dt: 222.50ms, tok/sec: 36817.48990672331
step 29, loss: 6.511846542358398, dt: 223.56ms, tok/sec: 36643.58654287097
step 30, loss: 6.5172319412231445, dt: 224.28ms, tok/sec: 36526.45090950067
step 31, loss: 6.510656356811523, dt: 223.24ms, tok/sec: 36696.26281388573
step 32, loss: 6.384012699127197, dt: 223.05ms, tok/sec: 36727.60370656602
```

Note: Runs about 10ms faster.

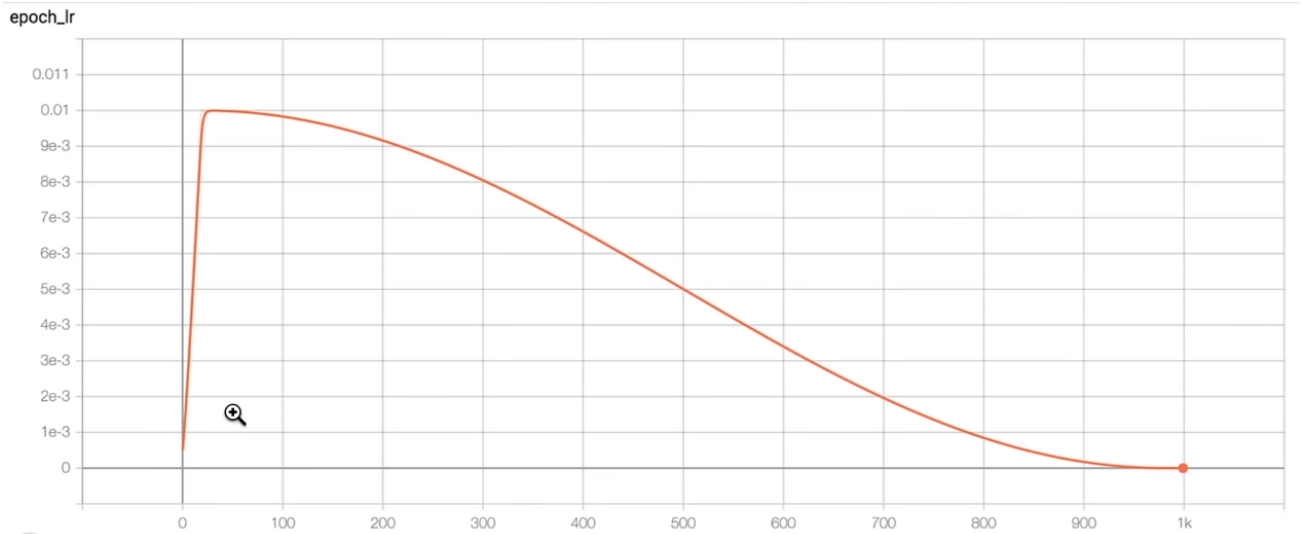
Section 4: Training optimisation II (Algorithmic)

Since the training details of GPT 2 weren't properly mentioned, the training details of GPT 3 would be used.

1. Adam: betas = (0.9, 0.95), eps=1e-8
2. Normalize the sum of backpropagated gradients using L2 norm such that they don't exceed 1.0. This greatly reduces the effect of cases with unusually high loss on the model weights. This improves the training of the model by reducing the effect of unexpected "shocks" during training.

   ```
   norm = torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
   ```

   The norm of the gradients is also printed. The norm allows us to detect anything unusual that happens during training. For example, a continuous increase in the norm highlights a problem with the model and frequent random surges of the norm highlights model instability.
3. Learning_rate: Cosine decay is used. The learning first ramps over the first few epochs, where afterwards it decays down to 10% of its initial value, where it stays constant.



Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
175B or “GPT-3”	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

4. Increasing batch_size with time: Intially, the gradients are highly correlated. Therefore, a batch_size of 32k would yield a similar result to a batch_size of 300M.
5. Sampling batches without replacement: Avoids overfitting.
6. Weight decay: All weights that aren’t one-dimensional, biases or layer norms have a decay of 0.1 for regularization.
7. adamw fused kernel: Usually, adamw uses a for loop to iterate over and update all the weights one by one. This calls a large number of kernels which has a huge overhead. This is made more efficient by calling a fused kernel that loads all the weights into the SRAM.
   NOTE: 7 and 8 are combined

```
optimizer = model.configure_optimizers(weight_decay=0.1, learning_rate=6e-4, device=device)
```

```
def configure_optimizers(self, weight_decay, learning_rate, device):
    # start with all of the candidate parameters (that require grad)
    param_dict = {pn: p for pn, p in self.named_parameters()}
    param_dict = {pn: p for pn, p in param_dict.items() if p.requires_grad}
    # create optim groups. Any parameters that is 2D will be weight decayed, otherwise no.
    # i.e. all weight tensors in matmuls + embeddings decay, all biases and layernorms don't.
    decay_params = [p for n, p in param_dict.items() if p.dim() >= 2]
    nodecay_params = [p for n, p in param_dict.items() if p.dim() < 2]
    optim_groups = [
        {'params': decay_params, 'weight_decay': weight_decay},
        {'params': nodecay_params, 'weight_decay': 0.0}
    ]
    num_decay_params = sum(p.numel() for p in decay_params)
    num_nodecay_params = sum(p.numel() for p in nodecay_params)
    print(f"num decayed parameter tensors: {len(decay_params)}, with {num_decay_params:,} param
    print(f"num non-decayed parameter tensors: {len(nodecay_params)}, with {num_nodecay_params:
    # Create AdamW optimizer and use the fused version if it is available
    fused_available = 'fused' in inspect.signature(torch.optim.AdamW).parameters
    use_fused = fused_available and 'cuda' in device
    print(f"using fused AdamW: {use_fused}")
    optimizer = torch.optim.AdamW(optim_groups, lr=learning_rate, betas=(0.9, 0.95), eps=1e-8,
    return optimizer
```

```
(env-gpt2) (base) lakindu_linux@lakindu-legion-Pro-7-16IRX9H:~/Desktop/GPT2-124M/Code$ /home/lakindu_linux/miniconda3/envs/
env-gpt2/bin/python /home/lakindu_linux/Desktop/GPT2-124M/Code/train_gpt2.py
loaded 338025 tokens
1 epoch = 41 batches
num decayed parameter tensors: 50, with 124,354,560 parameters
num non-decayed parameter tensors: 98, with 121,344 parameters
step    0 | loss: 10.937861 | lr: 6.0000e-05 | norm: 29.6022 | dt: 3288.28ms | tok/sec: 2491.27
step    1 | loss: 9.653714 | lr: 1.2000e-04 | norm: 10.4723 | dt: 213.26ms | tok/sec: 38412.53
step    2 | loss: 8.981644 | lr: 1.8000e-04 | norm: 5.3606 | dt: 214.88ms | tok/sec: 38124.45
step    3 | loss: 9.651280 | lr: 2.4000e-04 | norm: 9.5574 | dt: 212.42ms | tok/sec: 38564.50
step    4 | loss: 8.993284 | lr: 3.0000e-04 | norm: 4.2455 | dt: 216.83ms | tok/sec: 37780.54
step    5 | loss: 8.693342 | lr: 3.6000e-04 | norm: 3.0998 | dt: 220.68ms | tok/sec: 37121.95
step    6 | loss: 8.616785 | lr: 4.2000e-04 | norm: 3.5135 | dt: 214.61ms | tok/sec: 38170.87
step    7 | loss: 8.190947 | lr: 4.8000e-04 | norm: 2.5762 | dt: 213.35ms | tok/sec: 38396.69
step    8 | loss: 7.789444 | lr: 5.4000e-04 | norm: 2.8062 | dt: 213.58ms | tok/sec: 38355.29
step    9 | loss: 7.580828 | lr: 6.0000e-04 | norm: 2.3724 | dt: 218.39ms | tok/sec: 37511.45
step   10 | loss: 7.366563 | lr: 6.0000e-04 | norm: 1.9812 | dt: 218.54ms | tok/sec: 37484.56
step   11 | loss: 7.295155 | lr: 5.9917e-04 | norm: 24.1104 | dt: 215.67ms | tok/sec: 37984.11
step   12 | loss: 7.035192 | lr: 5.9668e-04 | norm: 2.2137 | dt: 213.25ms | tok/sec: 38414.46
step   13 | loss: 6.784040 | lr: 5.9254e-04 | norm: 0.9953 | dt: 216.68ms | tok/sec: 37806.81
step   14 | loss: 6.688363 | lr: 5.8679e-04 | norm: 1.2391 | dt: 220.65ms | tok/sec: 37125.92
step   15 | loss: 6.480438 | lr: 5.7945e-04 | norm: 1.1190 | dt: 215.96ms | tok/sec: 37933.57
step   16 | loss: 6.573632 | lr: 5.7057e-04 | norm: 1.1212 | dt: 214.79ms | tok/sec: 38140.03
step   17 | loss: 6.627208 | lr: 5.6021e-04 | norm: 1.1040 | dt: 214.97ms | tok/sec: 38108.43
step   18 | loss: 6.556674 | lr: 5.4843e-04 | norm: 1.5719 | dt: 219.25ms | tok/sec: 37363.38
step   19 | loss: 6.333993 | lr: 5.3531e-04 | norm: 0.9338 | dt: 218.62ms | tok/sec: 37471.69
step   20 | loss: 6.450834 | lr: 5.2092e-04 | norm: 1.3142 | dt: 215.62ms | tok/sec: 37993.22
step   21 | loss: 6.304412 | lr: 5.0535e-04 | norm: 1.6042 | dt: 214.57ms | tok/sec: 38178.80
step   22 | loss: 6.412543 | lr: 4.8870e-04 | norm: 1.1368 | dt: 216.57ms | tok/sec: 37826.79
step   23 | loss: 6.245270 | lr: 4.7107e-04 | norm: 1.1667 | dt: 220.90ms | tok/sec: 37084.28
step   24 | loss: 6.236846 | lr: 4.5258e-04 | norm: 1.0298 | dt: 216.27ms | tok/sec: 37878.42
step   25 | loss: 6.266149 | lr: 4.3332e-04 | norm: 0.7124 | dt: 214.70ms | tok/sec: 38155.02
step   26 | loss: 6.584507 | lr: 4.1343e-04 | norm: 1.3417 | dt: 215.12ms | tok/sec: 38081.52
```

Note: About 7ms faster

8. Gradient accumulation:

```
total_batch_size = 524288 # 2**19, ~0.5M, in number of tokens
B = 16 # micro batch size
T = 1024 # sequence length
assert total_batch_size % (B * T) == 0, "make sure total_batch_size is divisible by B * T"
grad_accum_steps = total_batch_size // (B * T)
print(f"total desired batch size: {total_batch_size}")
print(f"=> calculated gradient accumulation steps: {grad_accum_steps}")
```

```
for step in range(max_steps):
    t0 = time.time()
    optimizer.zero_grad()
    loss_accum = 0.0
    for micro_step in range(grad_accum_steps):
        x, y = train_loader.next_batch()
        x, y = x.to(device), y.to(device)
        with torch.autocast(device_type=device, dtype=torch.bfloat16):
            logits, loss = model(x, y)
        # we have to scale the loss to account for gradient accumulation,
        # because the gradients just add on each successive backward().
        # addition of gradients corresponds to a SUM in the objective, but
        # instead of a SUM we want MEAN. Scale the loss here so it comes out ri
        loss = loss / grad_accum_steps
        loss_accum += loss.detach()
        loss.backward()
    norm = torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
    # determine and set the learning rate for this iteration
    lr = get_lr(step)
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr
    optimizer.step()
    torch.cuda.synchronize() # wait for the GPU to finish work
```

Section 5: Parallelizing over many GPUs (`nn.parallel.DistributedDataParallel)

This part was not implemented by me as I only have access to 1 GPU.

1. For n GPUs, n processes are launched.
2. Each process is assigned to 1 GPU.
3. Each processes involves calculating loss and backpropagating the gradients for 1/n-th of the data.
4. After completion, the backpropagated gradients are accumulated and normalised (averaged).

NOTE: The program is now run using the `torchrun` command. `torchrun` creates environmental variables such as `LOCAL_RANK`, `RANK` and `WORLD_RANK` that can be accessed by each process.

```
# run the training loop
from torch.distributed import init_process_group, destroy_process_group
from torch.nn.parallel import DistributedDataParallel as DDP
import torch.distributed as dist
```

```
# set up DDP (distributed data parallel).
# torchrun command sets the env variables RANK, LOCAL_RANK, and WORLD_SIZE
ddp = int(os.environ.get('RANK', -1)) != -1 # is this a ddp run?
if ddp:
    # use of DDP atm demands CUDA, we set the device appropriately according to rank
    assert torch.cuda.is_available(), "for now i think we need CUDA for DDP"
    init_process_group(backend='nccl')
    ddp_rank = int(os.environ['RANK'])
    ddp_local_rank = int(os.environ['LOCAL_RANK'])
    ddp_world_size = int(os.environ['WORLD_SIZE'])
    device = f'cuda:{ddp_local_rank}'
    torch.cuda.set_device(device)
    master_process = ddp_rank == 0 # this process will do logging, checkpointing etc.
else:
    # vanilla, non-DDP run
    ddp_rank = 0
    ddp_local_rank = 0
    ddp_world_size = 1
    master_process = True
    # attempt to autodetect device
    device = "cpu"
    if torch.cuda.is_available():
        device = "cuda"
    elif hasattr(torch.backends, "mps") and torch.backends.mps.is_available():
        device = "mps"
    print(f"using device: {device}")
```

- Each process runs on the same script and has a different `. GPU 0 has a ddp rank of 1. The ddp ranks is used to ensure that no more than 1 runs on the same part of the data.
- `ddp_local_rank`: Used in a multi-node setting. Each node has multiple GPUs and `local_rank` corresponds to the rank of the GPU within that particular node.
- `ddp_world_size`: Total number of processes running.
- `master_process`: Does additional work such as logging, checkpointing, etc...

```
total_batch_size = 524288 # 2**19, ~0.5M, in number of tokens
B = 16 # micro batch size
T = 1024 # sequence length
assert total_batch_size % (B * T * ddp_world_size) == 0, "make sure total_batch_size is divisible by B * T * ddp_world_size"
grad_accum_steps = total_batch_size // (B * T * ddp_world_size)
if master_process:
    print(f"total desired batch size: {total_batch_size}")
    print(f"=> calculated gradient accumulation steps: {grad_accum_steps}")
```

```
class DataLoaderLite:
    def __init__(self, B, T, process_rank, num_processes):
        enc = tiktoken.get_encoding('gpt2')
        tokens = enc.encode(text)
        self.tokens = torch.tensor(tokens)
        print(f"loaded {len(self.tokens)} tokens")

        # state
        self.current_position = self.B * self.T * self.process_rank

    def next_batch(self):
        B, T = self.B, self.T
        buf = self.tokens[self.current_position : self.current_position+B*T+1]
        x = (buf[:-1]).view(B, T) # inputs
        y = (buf[1:]).view(B, T) # targets
        # advance the position in the tensor
        self.current_position += B * T * self.num_processes
        # if loading the next batch would be out of bounds, reset
        if self.current_position + (B * T + 1) > len(self.tokens):
            self.current_position = 0
        return x, y
```

```
# create model
model = GPT(GPTConfig(vocab_size=50304))
model.to(device)
model = torch.compile(model)
if ddp:
    model = DDP(model, device_ids=[ddp_local_rank])
```

- Each process creates a different GPT model. By, initializing the same seed we can ensure that all the models are identical.
- Each model is wrapped inside the `DDP` container.
- For each model the forward pass is identical. However, the backward pass is different. `DDP ` averages the backpropagated gradients and deposits them to every single rank (using `dist.all_reduce()`).
- `DDP` has more inbuilt optimizations which won't be discussed here.

```
for step in range(max_steps):
    t0 = time.time()
    optimizer.zero_grad()
    loss_accum = 0.0
    for micro_step in range(grad_accum_steps):
        x, y = train_loader.next_batch()
        x, y = x.to(device), y.to(device)
        with torch.autocast(device_type=device, dtype=torch.bfloat16):
            logits, loss = model(x, y)
        # we have to scale the loss to account for gradient accumulation,
        # because the gradients just add on each successive backward().
        # addition of gradients corresponds to a SUM in the objective, but
        # instead of a SUM we want MEAN. Scale the loss here so it comes out right
        loss = loss / grad_accum_steps
        loss_accum += loss.detach()
        if ddp:
            model.require_backward_grad_sync = (micro_step == grad_accum_steps - 1)
        loss.backward()
    if ddp:
        dist.all_reduce(loss_accum, op=dist.ReduceOp.AVG)
    norm = torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
    # determine and set the learning rate for this iteration
    lr = get_lr(step)
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr
    optimizer.step()
    torch.cuda.synchronize() # wait for the GPU to finish work
    t1 = time.time()
    dt = t1 - t0 # time difference in seconds
```

```
    dt = t1 - t0 # time difference in seconds
    tokens_processed = train_loader.B * train_loader.T * grad_accum_steps * ddp_world_size
    tokens_per_sec = tokens_processed / dt
    if master_process:
        print(f"step {step:4d} | loss: {loss_accum.item():.6f} | lr {lr:.4e} | norm: {norm:.4f} | dt: {dt*

if ddp:
    destroy_process_group()
```

```
# create model
model = GPT(GPTConfig(vocab_size=50304))
model.to(device)
model = torch.compile(model)
if ddp:
    model = DDP(model, device_ids=[ddp_local_rank])
raw_model = model.module if ddp else model # always contains the "raw" unwrapped model
```

Section 6: Datasets used to train GPT 3

Figure 2.2: Total compute used during training. Based on the analysis in Scaling Laws For Neural Language Models [KMH+20] we train much larger models on many fewer tokens than is typical. As a consequence, although GPT-3 3B is almost 10x larger than RoBERTa-Large (355M params), both models took roughly 50 petaflop/s-days of compute during pre-training. Methodology for these calculations can be found in Appendix D.

Dataset	Quantity (tokens)	Weight in training mix	Epochs elapsed when training for 300B tokens
Common Crawl (filtered)	410 billion	60%	0.44
WebText2	19 billion	22%	2.9
Books1	12 billion	8%	1.9
Books2	55 billion	8%	0.43
Wikipedia	3 billion	3%	3.4

Table 2.2: Datasets used to train GPT-3. "Weight in training mix" refers to the fraction of examples during training that are drawn from a given dataset, which we intentionally do not make proportional to the size of the dataset. As a result, when we train for 300 billion tokens, some datasets are seen up to 3.4 times during training while other datasets are seen less than once.

These datasets however are very outdated. Therefore, we use more refined datasets such as the:

1. RedPajama
2. SlimPajama (Cleaned and deduplicated subset of RedPajama)
3. FineWeb
4. FineWeb-edu (Subset of FineWeb with educational content)

For this project we will be using a 10BT subset of 4 as it approximates to very good GPT2 performance.

Section 7: Model evaluation, validation loss logging and sampling

Every 100th step is used to evaluate the model. The validation loss is calculated, printed and stored. For this project, given that there's only 1 epoch and the small size of the model, it is unlikely for it to overfit. However, the validation curve can be used to check for overfitting when this is not the case.

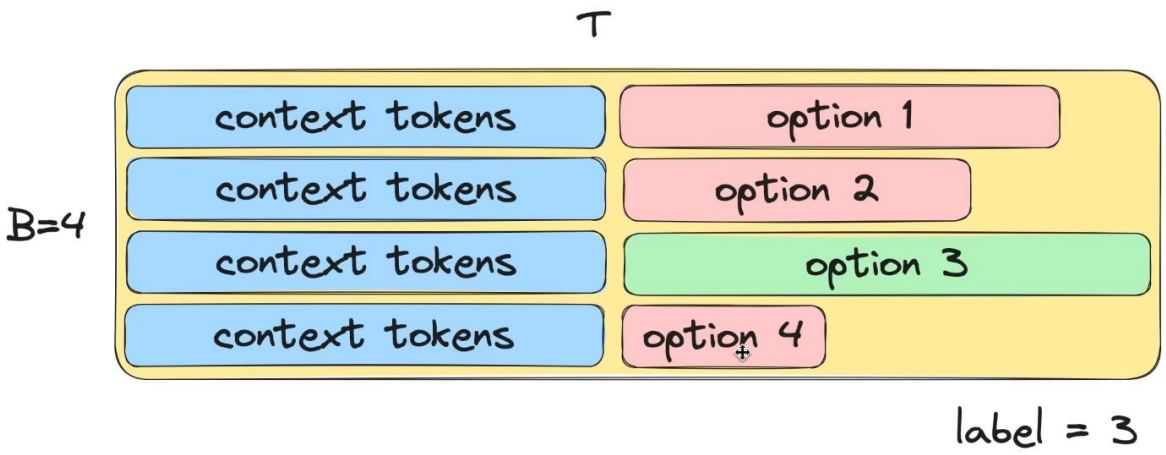
```
for step in range(max_steps):
        t0 = time.time()

        # validation loop (once every 100 steps) -------------------------------------------------------
        if step % 100 == 0:
                model.eval()
                val_loader.reset()
                with torch.no_grad():
                    val_loss_accum = 0.0
                    val_loss_steps = 20
                    for _ in range(val_loss_steps):
                        x, y = val_loader.next_batch()
                        x, y = x.to(device), y.to(device)
                        with torch.autocast(device_type=device, dtype=torch.bfloat16):
                            logits, loss = model(x, y)
                        loss = loss / val_loss_steps
                        val_loss_accum += loss.detach()

                print(f"validation loss: {val_loss_accum.item():.4f}")


        # sampling loop (once every 100 steps) ---------------------------------------------------------
```

```python
# sampling loop (once every 100 steps) ------------------------------------------------------
if step > 0 and step % 100 == 0:
        model.eval()
        max_length = T
        num_return_sequences = B
        tokens = enc.encode("Hello, I'm a language model,")
        tokens = torch.tensor(tokens, dtype=torch.long)
        tokens = tokens.unsqueeze(0).repeat(num_return_sequences, 1)
        xgen = tokens.to(device)
        sample_rng = torch.Generator(device=device)
        sample_rng.manual_seed(42)
        while xgen.size(1) < max_length:

            # forward the model to get the logits
            # torch.no_grad() indicates to PyTorch that no backpropagation takes place.
            # This reduces the amount of space used by PyTorch in caching the intermediate tensors needed for
            with torch.no_grad():
                logits, loss = model(xgen) # (B, T, vocab_size)

                # take the logits corresponding to the last position (poistion closest to token being predicte
                logits = logits[:, -1, :] # (B, vocab_size)

                # convert the logits to probabilities
                probs = F.softmax(logits, dim=-1)

                # `torch.topk`, where k=50, is used to filter the 50 tokens with the highest probability
                #and renormalize the probability distribution.
                # Use of topk ensures that tokens with really small probabilities aren't chosen.
                # probs here have shape (B, 24), tok_indices is (B, 24)
                topk_probs, topk_indices = torch.topk(probs, 50, dim=-1)

                # model the distribution as multinomial and sample the next token.
                # Sampling ensures that the same token isn't predicted all the time.
                ix = torch.multinomial(topk_probs, 1, generator=sample_rng)

                # gather the corresponding tokens
                xcol = torch.gather(topk_indices, -1, ix) # (B, 1)

                # concatenate the predicted tokens into the tensor, so the model can predict another token.
                x = torch.cat((x, xcol), dim=1)

        # print the generated text
        generated_texts = []
        for i in range(num_return_sequences):
            tokens = x[i, :max_length].tolist()
            decoded = enc.decode(tokens)
            generated_texts.append({'Sequence_Number': i, 'Generated_Text': decoded})
            print(f"sample {i}: {decoded}")
        df = pd.DataFrame(generated_texts)
        df.to_csv('outputs.csv', mode='a', index=False, header=False)
```

HellaSwag: A multiple-choice sentence completion evaluation dataset. Each choice is adversarially generated using GANs such that AI models find it difficult to tell part. What makes HellaSwag good is the fact that the improvements happen gradually, that is, the accuracy slowly increases from 25% (random guessing). Therefore, it gives early signal on whether or not there are issues with the model.

For larger model, we can the prompt the entire question into the mdoel with all the choices. However, smaller models are unable to learn the concept of “associating a label with a question”. Therefore, for GPT2 (124M) we compromise as follows:

1. The question (context tokens) will be prompted before each choice.
2. For each option, the probabilities for each of its tokens are found and averaged.
3. The option with the highest average probability is chosen. Alternatively, the option with the lowest cross-entropy loss can be selected.
4. Keeping in mind that the length T is equal to the length of the longest sequence and that all the additional tokens at the end are padded.



Section 8: Analysing how the outputs of the model changes across pretraining

Prompt: “Hello, I’m a language model, ”

1. At step 20, it seems that the model has learnt that certain words such as “the”, “that”, “is”, etc appear much more frequently compared to other words in the English words. So, the model therefore, predicts the tokens associated with these words.

```
validation loss: 7.6565
HellaSwag accuracy: 2510/10042=0.2500
sample 0: Hello, I'm a language model, the the for the. is to
 he your a to and was it to the of the in. of the to
sample 1: Hello, I'm a language model, and was A-.
 of by the. his) and toThe and., to a:.
.
sample 2: Hello, I'm a language model, and the it of the:The in the that the the is and, the is a a the the the and it
sample 3: Hello, I'm a language model,. that the and the in a of a but the in the that to.S in the and as the, it

step   20 | loss: 7.679338 | lr: 3.7800e-04 | norm: 1.1156 | dt: 66682.49ms | tok/sec: 7862.45
```

2. At step 100, the model has learnt to predict tokens in a similar pattern to which appears in the training data. This can be observed by the intelligible text generated. However, due to training just beginning, it is unable to understand what the prompt actually means.

```
validation loss: 6.4594
HellaSwag accuracy: 2436/10042=0.2426
sample 0: Hello, I'm a language model, for a man. The National Center's Information for the course. And there are a own al). If the United
sample 1: Hello, I'm a language model, so what may be the mind is he was not say and how that also you have you could not in your report of
sample 2: Hello, I'm a language model, but a part of. A century (The country, and in an use of the state of the following the first).
sample 3: Hello, I'm a language model, or in the example, or to give to encourage a variety of its bodies of some or an attention at the last times

step  100 | loss: 6.527895 | lr: 1.8000e-03 | norm: 0.5770 | dt: 74836.36ms | tok/sec: 7005.79
```

3. At 200-300 steps, the model learns to associate the prompt to itself and uses words such as “I” and “my”.

```
200,0,"Hello, I'm a language model, I'm talking about my mind I could go down a. I've the word, the writing that we
need to see"
200,1,"Hello, I'm a language model, so if I'm a bit a free, you with what you don’t want it. There are always don"
200,2,"Hello, I'm a language model, and the words and if he wants it is why some of their work is a new. It is a
good and an"
200,3,"Hello, I'm a language model, it becomes a new, that I'd have had a little true with my students know I will
have this is the topic"
300,0,"Hello, I'm a language model, though I was a good tool and you'd do with ""I love as the ""a number,"" ""What
is I"
300,1,"Hello, I'm a language model, one group of the same language and with a computer on other languages (some two
language). A child does not consider,"
300,2,"Hello, I'm a language model, and I never tried any real word to be of the same word for some of the word
""the word ""in or"
300,3,"Hello, I'm a language model, a whole language for a language, that can even be a more easy. It may like
other language than a great world"
```

4. At steps 400-500, the model has learnt a simiplied semantic understanding of the word “language” and other similar words. It uses words such as “word”, “argument”, “text file” that are related to “language”.

```
400,0,"Hello, I'm a language model, but I think that I had some way was an argument to say “the world of an opinion. I’"
400,1,"Hello, I'm a language model, and if you are looking at the following. The world it’s what the two countries have written by the main"
400,2,"Hello, I'm a language model, and I like it you should need a language as a language in language and language. I’ll be able to"
400,3,"Hello, I'm a language model, a word which is a language. It can communicate with a language where I have started the different languages is a type code"
500,0,"Hello, I'm a language model, and I will be a student’s life-giving learning of kids about the school age group and a child who"
500,1,"Hello, I'm a language model, since i'm a bit more,” said Aussi’s teacher who can use his students to become"
500,2,"Hello, I'm a language model, I'm trying to solve these problems by using some arguments. For me, I'm going to be a great example."
500,3,"Hello, I'm a language model, or as a text file, you should have written, or from this language language-level programming languages-based programming;"
```

5. Step 2800. The generated text doesn’t make complete sense to the prompt but is starting to make some sense. The model is definitely learning.

```
2800,0,"Hello, I'm a language model, and I don't want to write you and share my knowledge, however, and I can have fun and be a very"
2800,1,"Hello, I'm a language model, I need to create an R package, I'm starting for a programming background with the C compiler, a programming background with"
2800,2,"Hello, I'm a language model, so I’m interested in a language to play with?<|endoftext|>- The National Science Foundation (NSF) is"
2800,3,"Hello, I'm a language model, I got a very nice, I have some sample code, I am using a software model in the real world and if"
```

6. At step 6000, the generated text is much more coherent and makes some sense despite not being that related to the prompt.

```
6000,0,"Hello, I'm a language model, and I need to learn my new language(s) just to know and remember what is inside and what does not have"
6000,1,"Hello, I'm a language model, I use GPT to describe things in a language."" My name is Nicki, a mother tongue English professor at the"
6000,2,"Hello, I'm a language model, I'm here with you today.<|endoftext|>The American Petroleum Association’s (API) Annual Report on the State of"
6000,3,"Hello, I'm a language model, I feel like I'm going to share a joke with you. Thanks for the explanation. Bye, thanks for sharing too"
6100,0,"Hello, I'm a language model, and I was a student of CCDOS last year. I don't like to talk about languages and don't think"
6100,1,"Hello, I'm a language model, I like to make my learning easy for you to read the text on my page and understand the meaning of my sentences."
6100,2,"Hello, I'm a language model, I've already worked on various languages and I still use it today in my classes.
```

7. At 8000 steps, the text generated is more related to “language”, despite not being related to “language model”. The model has also learnt that there “programming languages". This is shown by words such as “python” and “syntax”.

```
7900,0,"Hello, I'm a language model, and I know you guys do understand your topic more than I did last year and I
can't think you are a model"
7900,1,"Hello, I'm a language model, I'm not a linguist, and language programming is really similar to the other
types. As an example,"
7900,2,"Hello, I'm a language model, I'm actually talking about building a good model...so I will create an object
with a model, and then we'll"
7900,3,"Hello, I'm a language model, I use the syntax of the syntax in my textbook. I get stuck and need an extra
guide to do the right language"
8000,0,"Hello, I'm a language model, and I need to learn Python in order to help me learn. Please be as concise and
short as I can. Thanks"
8000,1,"Hello, I'm a language model, so, I'm going to use a language to introduce all of you to your new world.
My new language is"
8000,2,"Hello, I'm a language model, so I started out playing. I did a ton of research about languages and
languages, but I didn't know how people"
8000,3,"Hello, I'm a language model, I use it. I'm interested in how. I'm getting good at writing
The last time I wrote about that"
8100,0,"Hello, I'm a language model, and I think it is amazing, right? Can you prove it here? And if not, then, you
can just"
8100,1,"Hello, I'm a language model, I understand people, but I have a problem that was going on for me to
understand and then it will go away."
8100,2,"Hello, I'm a language model, I'm talking a great thing about my language is as a person I speak a language
and I can understand it, but"
8100,3,"Hello, I'm a language model, and what I've learned is how to make sound. I've already created the three
sounds I can control, so they"
```

8. Over the next steps, the text becomes more and more coherant and related to the prompt. Here's some screenshots of the text generated at different training steps.

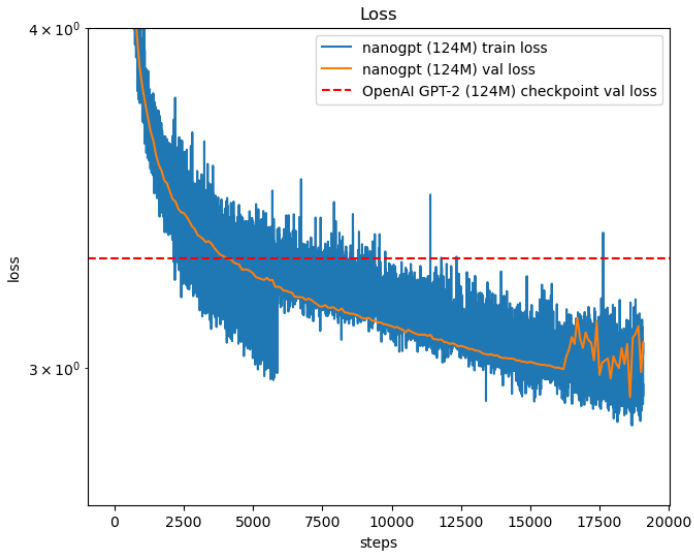
```
14100,1,"Hello, I'm a language model, I'll ask you, are you.
"" I guess I don't want to go through all the math, I"
14100,2,"Hello, I'm a language model, I'm building an image to represent this language; I'm creating this language
model with my image.
I can imagine"
14100,3,"Hello, I'm a language model, I work with computers, I use machine language at work, I don't have other
machines or people doing the job right"
14200,0,"Hello, I'm a language model, and I understand that there are different styles of expressions you can use;
my favourite is, um, you should use these"
```

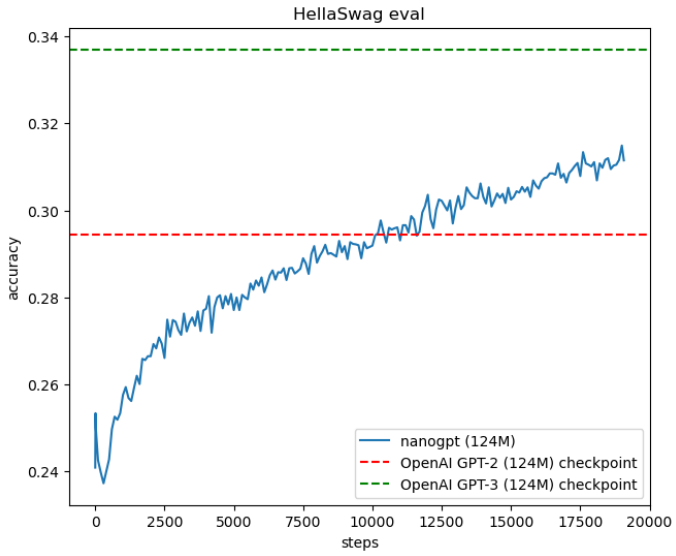
```
9900,2,"Hello, I'm a language model, I'm thinking. For many languages and languages like Dutch, French and English,
it is very important to know the most"
9900,3,"Hello, I'm a language model, so maybe I'll be able to speak another language.
Well, if we ever go online and take a language path"
10000,0,"Hello, I'm a language model, and I don't understand your idea of human characteristics. It's something
strange for me, but what makes you feel weird"
10000,1,"Hello, I'm a language model, I use objects and methods. I put objects into your head, you type and you do
this, what are you doing"
10000,2,"Hello, I'm a language model, so I got this as .NET class. If you want something different, you can do it
in the same class,"
```

```
16300,0,"Hello, I'm a language model, and I understand the syntax of Java. Because I can have Java statements when
I want them, my first thought is:"
16300,1,"Hello, I'm a language model, I do research, I do a grant, but, when I want to learn something I want to
ask someone. I"
16300,2,"Hello, I'm a language model, I'm part of the English language team. It's a lot more interesting than most
of the other languages.
When"
```

```
19000,0,"Hello, I'm a language model, and I understand the syntax of some text/concepts, and a basic concept of
what a model should look like for"
19000,1,"Hello, I'm a language model, I like to use the language for talking about language development... I can
always say, 'Okay. Language development is not"
19000,2,"Hello, I'm a language model, I'm curious what programming a language might be! I'm sure there are many,
many languages out there, and languages"
19000,3,"Hello, I'm a language model, and want to create a model of my model ...
I'm gonna show a general purpose of this assignment, but,"
19072,0,"Hello, I'm a language model, and I am a computer, because when I created the Python code code using Pygame
(C,C++, Python"
19072,1,"Hello, I'm a language model, I do type in words, and
I type math to write equations to math. And it is so hard to type"
19072,2,"Hello, I'm a language model, so I wrote an open(module) function call to the library that takes the object
and returns the object. I need"
19072,3,"Hello, I'm a language model, I see the word ""dictionary"" as representing a dictionary. Now, we know this
has been for a long,"
```

Section 9: The Pretrained model





Model Performance

The above graphs depict the training/validation curves and the HellaSwag evaluation curve throughout the 19071 epochs. The baselines for the 124M versions of GPT2 and GPT3 have been for comparison.

It can be observed that my model performed much better than GPT2 while falling short to GPT3. Additionally, this higher performance than GPT2 was achieved by training it across only 20GB (10B tokens) of data, which was half the 40GB used for GPT2. This higher performance achieved whilst training with half the amount of data can be attributed to the high quality of the fineweb-edu dataset. This also highlights the importants of using high quality training data in training LLMs.

GPU details

The pretraining was done on an RTX 4080 laptop GPU for approximately 71 hours distributed across 5 days. The GPU speed was quite stable during this process with a maximum temperature of 66 degrees Celsius without an external cooler. The GPU utilisation reached 100% many times consistently and the VRAM usage never exceeded 8.4/12 GB.

Text generation

Generated many examples and all of that had two things in common. The English is understandable and grammatically correct. Individually, some sentences generated make sense. However, the logic falls short halfway through the generation. Despite the context length being 1024 tokens, the model forgets what it says in the previous sentence making the next sentence make no sense in regard to context. Given the low scores in HellaSwag, my guess is that the model is lacks the complexity understand the complex patterns in human language. This could also be attributed to the fact that the model was trained with only 10B tokens.

However, for very simple prompts that are more open-ended like in the image below, the text generated makes more sense. The below image is infact the final text generated by the model right after ending pretraining.

```
19072,0,"Hello, I'm a language model, and I am a computer, because when I created
the Python code code using Pygame (C,C++, Python"
19072,1,"Hello, I'm a language model, I do type in words, and
I type math to write equations to math. And it is so hard to type"
19072,2,"Hello, I'm a language model, so I wrote an open(module) function call to
the library that takes the object and returns the object. I need"
19072,3,"Hello, I'm a language model, I see the word ""dictionary"" as
representing a dictionary. Now, we know this has been for a long,"
```

Text generation via Prompt Engineering

Using prompts seem to drastically increase model coherence and the text generated is more linked to the prompt. It seems that the tokens required to degenerate the context also increases. But as earlier not all text generated makes sense (sample 1) and as the model generates more text, it deviates from the original prompt. The factual information given also seems to be clearly wrong.

19072,0,"Question: What are cats? Answer: Let's think step by step.
Answer: Cats have many names depending on the cat breed and cat habitat where they are found. Some species are considered purebred and some are considered substandard. Most cats are classified as either mixed (wildcat, wild dog, domestic cat), or purebred (wildcat, panda/kungaroo).
1. Which species is considered purebred
Fluid and Blood: The cats that have blood type A and B are considered the purebred cats. In fact, purebred cats can be seen in many types of cats ranging from toy rabbits to wild dogs. They are an important part of the human pet supply, providing both the domestic cat and its companion those that are not wild.
Wildcat and domestic cat: The Wildcat and domestic cat have much in common. They share many of the characteristics that make them one of the many types of cats. They typically have a distinctive head shape and large claws. Their long, narrow bodies and dense manes typically give them an animal-like appearance to look at, often accompanied with a strong, sharp, rounded jaw.
Panda: Panda is a genus of giant pandas that were found by scientists in Africa"
19072,1,"Question: What are cats? Answer: Let's think step by step. cats is a cat that is not one of our favorites on this list. The cat that was the one to be named because of the color has the blue eye and the white eye is the big one. This is a very special cat and I really love it.
Question: How many cats are there? Answer: Let's think about this with you. There are eight different breeds of cats who are found in different parts of the country. Three of the breed are the Bengal, the Abyssinian and the Hottentot. Their colors can be easily separated.
Question: What is hair color and why do its so special? Answer: Well, this is where you'll find the mystery. Cats are colorless, so because they have such beautiful eyes, but they also have deep blue in the eye. This makes it very rare for the eyes of a cat to have such a blue in the eyes.
Question: Can you grow a cat like a pet? Answer: In most countries, there are not as many cat breeds as there are cats. However, you could try and create some unique cats like:<|endoftext|>A recent study by researchers conducted by the Harvard School of Public Health found"
19072,2,"Question: What are cats? Answer: Let's think step by step. Cats are members of the feline family. A cat is any living member of the family of mammals. Cats have whiskers, eyes, and ears that can be used for sensing, smelling, eating, and feeling. Let's get a closer look. Cows, for example, can sense if they are pregnant or they are not. They also have large, sensitive scents. So, when a cat goes for a walk, her nose is probably full. That cat's scent is different from that of an animal. So, her body heat and scent would be more sensitive and it can have a very powerful impact in her mood.
As you can see, cats can smell like an animal. And the way a cat moves her body is more sensitive and the amount of information she can absorb. The way a cat's body moves is related to how quickly she gets used to the smell.
Cats smell in the environment.
Olfactory receptors, for instance, on the nose and mouth are located on the sides of the tongue, the roof of the mouth, and on the roof of the mouth. These receptors are sensitive to the smell and the chemicals that a cat needs. So"
19072,3,"Question: What are cats? Answer: Let's think step by step.
There are more than 2 million different breeds of cats. Different breeds have a unique coat. The difference between them when it comes to coat colour alone is about the same as the colour of the dog.
This makes it very difficult for one person to identify the different colours of a dog. For instance, some breeds of dogs do not have a 'blue' coat and some are 'grey'.
Because the colour of the hair should always match the colour of the dog, these animals can all be considered 'wild' or 'wild'.<|endoftext|>As the sun sets on Thursday, the coldest winter of our lives has left snow on the ground: in a short period of time, we now know that the ground is frozen! After an active campaign for many years, the government has asked the federal government to reconsider its decision to remove greenhouse gas emissions from the emissions of some US cities, with an eye on Berlin, too.
The Government is also preparing for the implementation of the Montreal Protocol to combat pollution caused by diesel exhaust from the exhaust stacks in cities around the world. One option for dealing with these issues would be to find a “sustainable” solution that avoids the emission of"

The model seems to focus on predicting the next word instead of actually giving the answer. The text generated seems to what appears next in the question instead of what appears next in the answer.

```
19072,0,"Question: What is 1+1? Answer: Let's think step by step. _________________ = 1 + 1+1 is right.
_________________ = 1 + 1 + 1 is wrong. Answer: 1+1 +1 is correct. _________________ = 1 +1 + 1 is wrong.
Solution. 1 is correct. _________________ = 1 + 1, but we already know that 1+1 is incorrect.
3x-1 = 2n-1. (A-2)
3x-x = 3x1 - n^2 to get the answer . . . [M] 4x-12 = 1. If the answer is correct, then this solution is right. 1 +
2 + 2 + 2 + 2 + 2 + 4 = 1
3y+1 = 2y+2 to get the answer . . . [M] 2y +2 + 2 2y^11 + 5y^12 = 1. It is wrong.
5y +3y-1 =2y +3y-1 . . . [M] 5x2-5y2 + 5y = . . . . [M] 5x2^11 + 5y^12 = 7"
```

Section 10: Post-Pretraining ideas

Realized that GPT2 that it's not really worth investing time and effort to further optimise it. I will probably actualize these ideas in the near future with an already pretrained mode

1. Natural endings: Terminate after end of text token + hard limit
2. Supervised Finetuning for Chatting
3. Distillation: Fine-tune on DeepSeek v3 outputs (using api)

Finale: Future project

1. Distillation?: Fine-tune on DeepSeek r1 thought process.
2. Finetuning
3. RAG

References

1. https://arxiv.org/abs/2005.14165
2. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf
3. https://huggingface.co/datasets/HuggingFaceFW/fineweb-edu
4. https://github.com/karpathy/build-nanogpt/tree/master
5. https://www.youtube.com/watch?v=l8pRSuU81PU&t=6519s