```
%pip install -q transformers huggingface_hub
import math
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import transformers
```

Build-a-transformer

In this section, you will implement a transformer language model layer by layer, then use it to generate (hopefully) coherent text.

To understand how these layers work, please check out our guide to transformers from nlp course for you -> transformers.

First, we download pre-trained weights for the GPT2 model by OpenAI - a prominent model from 2019.

Idea & code by: Ilya Beletsky

```
from huggingface_hub import hf_hub_download
state_dict = torch.load(hf_hub_download("gpt2", filename="pytorch_model.bin"))
for key, value in tuple(state_dict.items()):
    if key.startswith('h.') and key.endswith('.weight') and value.ndim == 2:
        value.transpose_(1, 0)  # <-- for compatibility with modern PyTorch modules
    if key.startswith('h.') and key.endswith('.attn.bias') and value.ndim == 4:
        state_dict.pop(key)  # <-- triangular binar masks, not needed in this code

print('Weights:', repr(sorted(state_dict.keys()))[:320], '...')
```

```
/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://hug
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models o
  warnings.warn(
pytorch_model.bin: 100%                                   548M/548M [00:03<00:00, 239MB/s]
Weights: ['h.0.attn.c_attn.bias', 'h.0.attn.c_attn.weight', 'h.0.attn.c_proj.bias', 'h.0.at
```

In the next few cells, we shall implement the model layer by layer to make use of those weights.

As you might recall, transformers contain two main layer types: attention and fully-connected layers.

The fully connected layers are by far easier to understand, so we shall begin there:

Please implement fully-connected layer **without residual or layer normalization** (we'll add those in a bit).

```
class GeLUThatWasUsedInGPT2(nn.Module):
    def forward(self, x):
        return 0.5 * x * (1.0 + torch.tanh(math.sqrt(2.0 / math.pi) * (x + 0.044715 * x **

class FullyConnected(nn.Module):
    def __init__(self, dim: int):
        super().__init__()
        self.c_fc = nn.Linear(dim, 4  * dim)
        self.gelu = GeLUThatWasUsedInGPT2()
        self.c_proj = nn.Linear(4 * dim, dim)

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

Now, let's test that it works with GPT-2 weights:

```
mlp = FullyConnected(dim=768)
mlp.load_state_dict({'c_fc.weight': state_dict['h.0.mlp.c_fc.weight'],
                     'c_fc.bias': state_dict['h.0.mlp.c_fc.bias'],
                     'c_proj.weight': state_dict['h.0.mlp.c_proj.weight'],
                     'c_proj.bias': state_dict['h.0.mlp.c_proj.bias']})

torch.manual_seed(1337)
x = torch.randn(1, 2, 768)  # [batch_size, sequence_length, dim]
checksum = torch.sum(mlp(x) * x)
assert abs(checksum.item() - 1282.3315) < 0.1, "layer outputs do not match reference"
assert torch.allclose(mlp(x[:, (1, 0), :])[:, (1, 0), :], mlp(x)), "mlp must be permutatio
print("Seems legit!")
```

```
Seems legit!
```

Now, let's get to attention layers.

Since GPT-2 needs to generate text from left to right, each generated token can only attend to tokens on the left (and itself). This kid of attention is called "Masked" self-attention, because it hides tokens to the right.

As before, please implement masked self-attention **without layernorm or residual connections.**

```
class MaskedSelfAttention(nn.Module):
    def __init__(self, dim: int, num_heads: int):
        super().__init__()
        self.c_attn = nn.Linear(dim, dim * 3)  # query + key + value, combined
        self.c_proj = nn.Linear(dim, dim)  # output projection
        self.dim, self.num_heads = dim, num_heads
        self.head_size = dim // num_heads

    def forward(self, x):
        q, k, v = self.c_attn(x).split(dim=-1, split_size=self.dim)
        assert q.shape == k.shape == v.shape == x.shape, "q, k and v must have the same sh


        # Note: this is an inefficient implementation that uses a for-loop.
        # To get the full grade during homework, please re-implement this code:
        # 1) do not use for-loops (or other loops). Compute everything in parallel with ve
        # 2) do not use F.scaled_dot_product_attention - write your own attention code usi
        head_outputs = []
        for head_index in range(self.num_heads):
            head_selector = range(self.head_size * head_index, self.head_size * (head_inde

            head_queries = q[..., head_selector]
            head_keys = k[..., head_selector]
            head_values = v[..., head_selector]
            single_head_output = F.scaled_dot_product_attention(
                head_queries,
                head_keys,
                head_values,
                is_causal=True
            )
            # docs: https://pytorch.org/docs/stable/generated/torch.nn.functional.scaled_d
            head_outputs.append(single_head_output)

        combined_head_outputs = torch.cat(head_outputs, dim=-1)
        return self.c_proj(combined_head_outputs)
```

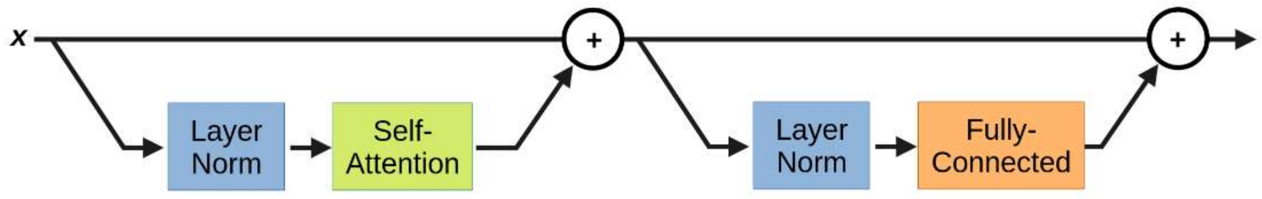
Test that it works

```
attn = MaskedSelfAttention(dim=768, num_heads=12)
attn.load_state_dict({'c_attn.weight': state_dict['h.0.attn.c_attn.weight'],
                      'c_attn.bias': state_dict['h.0.attn.c_attn.bias'],
                      'c_proj.weight': state_dict['h.0.attn.c_proj.weight'],
                      'c_proj.bias': state_dict['h.0.attn.c_proj.bias']})

torch.manual_seed(1337)
x = torch.randn(1, 10, 768)  # [batch_size, sequence_length, dim]
checksum = torch.sum(attn(x) * x)
assert abs(checksum.item() - 2703.6772) < 0.1, "layer outputs do not match reference"
assert not torch.allclose(attn(x[:, (1, 0), :])[:, (1, 0), :], attn(x[:, (0, 1), :])), "ma
print("It works!")
```

```
It works!
```

We can now combine attention and MLP to build the full transformer layer:



```
class TransformerLayer(nn.Module):
    def __init__(self, dim: int, num_heads: int):
        super().__init__()
        self.ln_1 = nn.LayerNorm(dim)
        self.attn = MaskedSelfAttention(dim, num_heads)
        self.ln_2 = nn.LayerNorm(dim)
        self.mlp = FullyConnected(dim)
    def forward(self, x):
        #YOUR CODE - apply attention, mlp and layer normalization as shown in figure above
        x = x + self.attn(self.ln_1(x))
        x = x + self.mlp(self.ln_2(x))
        return x
```

```
layer = TransformerLayer(dim=768, num_heads=12)
layer.load_state_dict({k[5:]: v for k, v in state_dict.items() if k.startswith('h.10.')})
assert abs(torch.sum(layer(x) * x).item() - 9874.7383) < 0.1
print("Good job!")
```

```
Good job!
```

```
class GPT2(nn.Module):
    def __init__(self, vocab_size: int, dim: int, num_heads: int, num_layers: int, max_pos
        super().__init__()
        self.wte = nn.Embedding(vocab_size, dim)  # token embeddings
        self.wpe = nn.Embedding(max_position_embeddings, dim)  # position embeddings
        self.ln_f = nn.LayerNorm(dim)   # final layer norm - goes after all transformer la

        self.h = nn.Sequential(*(TransformerLayer(dim, num_heads) for layer in range(num_l

    def forward(self, input_ids):
        # input_ids.shape: [batch_size, sequence_length], int64 token ids
        position_ids = torch.arange(input_ids.shape[1], device=input_ids.device).unsqueeze

        token_embeddings = self.wte(input_ids)
        position_embeddings = self.wpe(position_ids)
        full_embeddings = token_embeddings + position_embeddings

        transformer_output = self.h(full_embeddings)
        transformer_output_ln = self.ln_f(transformer_output)

        # final layer: we predict logits by re-using token embeddings as linear weights
        output_logits = transformer_output_ln @ self.wte.weight.T
        return output_logits
```

```
tokenizer = transformers.AutoTokenizer.from_pretrained('gpt2', add_prefix_space=True)
model = GPT2(vocab_size=50257, dim=768, num_heads=12, num_layers=12)
model.load_state_dict(state_dict)

input_ids = tokenizer("A quick", return_tensors='pt')['input_ids']

predicted_logits = model(input_ids)
most_likely_token_id = predicted_logits[:, -1].argmax().item()

print("Prediction:", tokenizer.decode(most_likely_token_id))
```

```
tokenizer_config.json: 100%                                26.0/26.0 [00:00<00:00, 1.55kB/s]
config.json: 100%                                     665/665 [00:00<00:00, 20.0kB/s]
vocab.json: 100%                                      1.04M/1.04M [00:00<00:00, 18.8MB/s]
merges.txt: 100%                                      456k/456k [00:00<00:00, 2.14MB/s]
tokenizer.json: 100%                                    1.36M/1.36M [00:00<00:00, 6.10MB/s]
Prediction:  look
```

```
text = "The Fermi paradox "
tokens = tokenizer.encode(text)
print(end=tokenizer.decode(tokens))
line_length = len(tokenizer.decode(tokens))

for i in range(500):
    # Predict logits with your model
    with torch.no_grad():
        logits = model(torch.as_tensor([tokens]))

    # Sample with probabilities
    p_next = torch.softmax(logits[0, -1, :], dim=-1).data.cpu().numpy()
    next_token_index = np.random.choice(len(p_next), p=p_next)

    tokens.append(int(next_token_index))
    print(end=tokenizer.decode(tokens[-1]))
```

```
      line_length += len(tokenizer.decode(tokens[-1]))
      if line_length > 120:
        line_length = 0
        print()
```

```
 The Fermi paradox  : a complex notion that quantum mechanics holds as real is the object o
 (no more!) from Dr. David Smith in The Federalist (opens with some comment bar): I agree t
 with any kind of evolution theory. These are important in ways that motivated James Clerk
 Theory . All three questions also involve Barty, Vogel & Darwin's (albeit incredibly primi
 as accommodation for linear ontology. According to these related notions, quantum mechanic
 (non-linear) matter does not. After my Googling I found another interesting quark on Wikip
ly identified it). Though fully aware that most who discuss new concepts have encountered b
 displaced origamiist myth and the speed and psychology of quantum physics we can probably
 both sufficient shorthand for what has been previously conceptualized as natural selection
 quark that always turns off and the 'peanutslam hypothesis', I do believe they are the mos
 to the principles of the Baryoid paradox, momentum and  the explicit nature of thermodynam
 way of saying common urge, mere 'opportunity ' for new convergent behaviour from being 'si
y and Vogel had bremer chaos and can385 fully explain why there is a mere 1,040 variables,
 universe in approximately 300 billion years. Manifold and quantum theory must therefore be
, or more accurately, it is about our causal sense to make meaningful transitions between d
 good capacity for the underliements it is finitely powerful. This saying weighs on Monke I
. He claims we need infinite time to be on the same wavelength of extension from Craig's (1
 (parameters of mass and energy are multiple of each other's measurement frequencies, hence
 metafunctions that contradict (as already stated) th all g different measurements among of
no more!) from Robert Salter (£45 minimum, magazine pays about £50 a cap) I'm ready to take
 Vince are writing right through  Thomas
```

```
tokenizer = transformers.AutoTokenizer.from_pretrained('gpt2', add_prefix_space=True)
model = transformers.AutoModelForCausalLM.from_pretrained('gpt2')
print('Generated text:', tokenizer.decode(
    model.generate(
        **tokenizer("The Fermi paradox ", return_tensors='pt'),
        do_sample=True, max_new_tokens=50
    ).flatten().numpy()
))
```

```
model.safetensors: 100%                                548M/548M [00:03<00:00, 345MB/s]
generation_config.json: 100%                           124/124 [00:00<00:00, 9.05kB/s]
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
Generated text:  The Fermi paradox  is perhaps the least interesting and least understood p
```