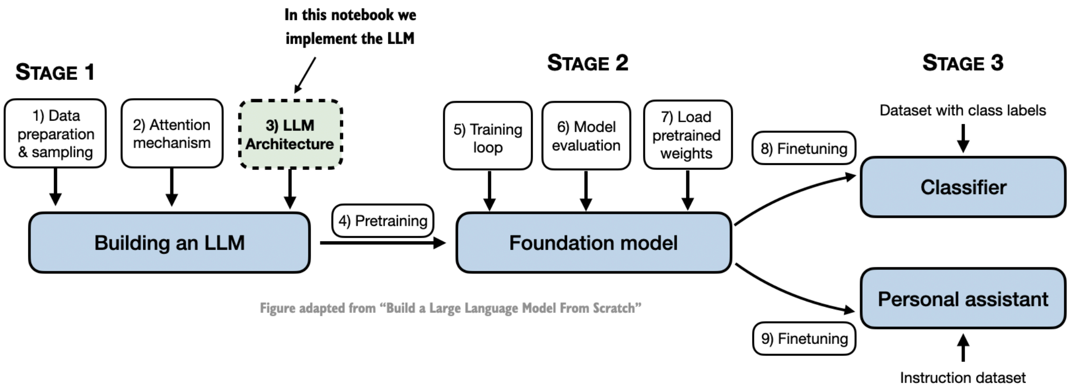
3. Coding an LLM architecture

In [1]:
```
from importlib.metadata import version


print("torch version:", version("torch"))
print("tiktoken version:", version("tiktoken"))
```

```
torch version: 2.8.0
tiktoken version: 0.11.0
```

- In this notebook, we implement a GPT-like LLM architecture; the next notebook will focus on training this LLM



3.1 Coding an LLM architecture

- Models like GPT, Gemma, Phi, Mistral, Llama etc. generate words sequentially and are based on the decoder part of the original transformer architecture
- Therefore, these LLMs are often referred to as "decoder-like" LLMs
- Compared to conventional deep learning models, LLMs are larger, mainly due to their vast number of parameters, not the amount of code
- We'll see that many elements are repeated in an LLM's architecture

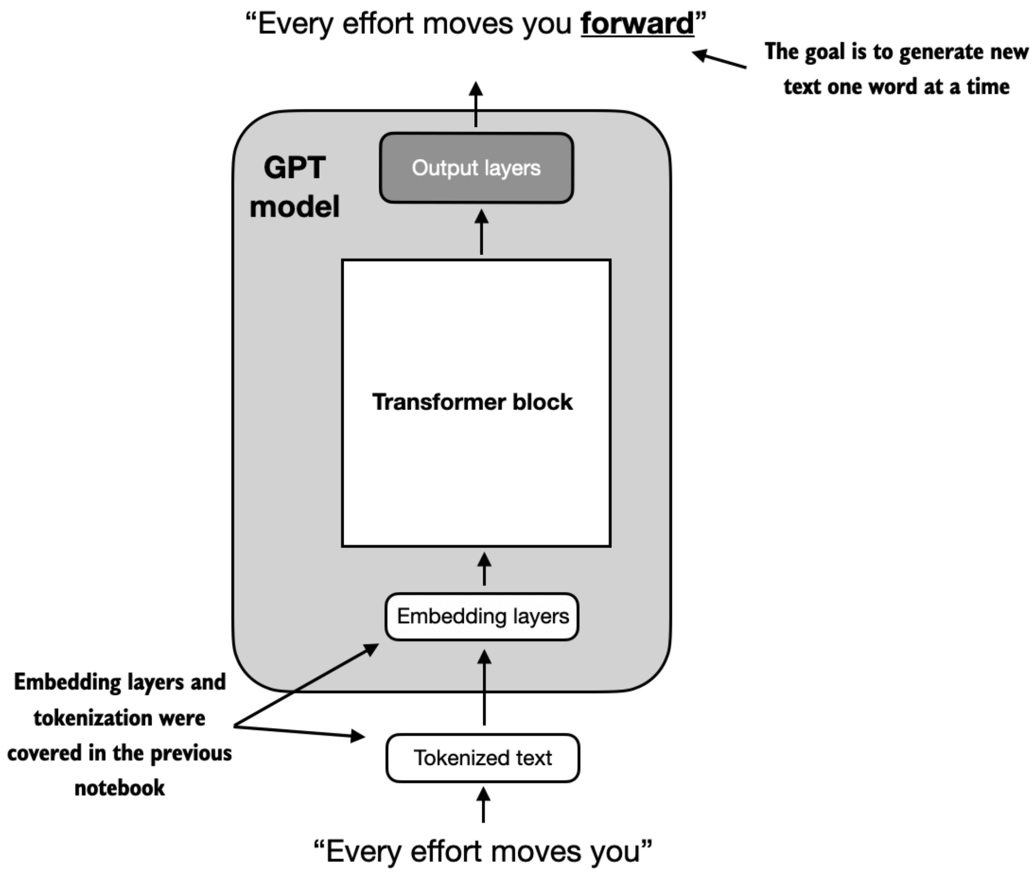
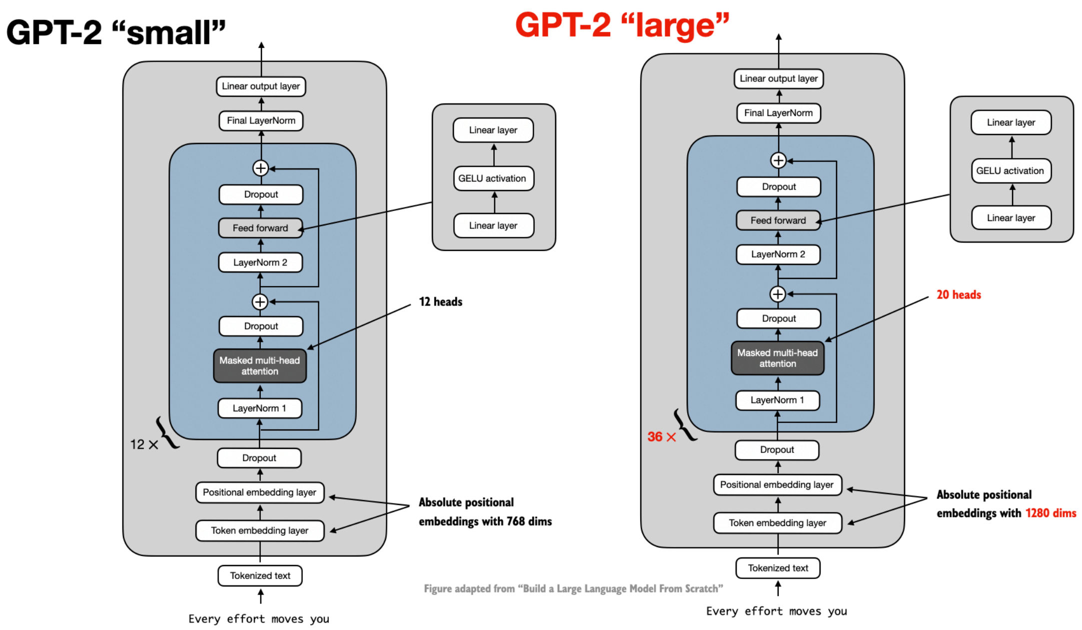
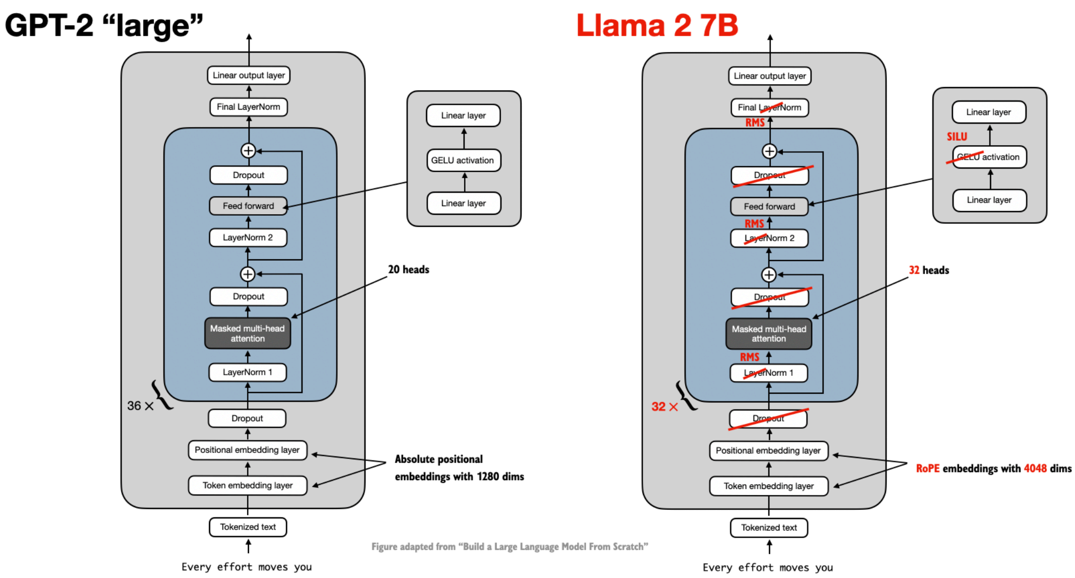


Figure adapted from "Build a Large Language Model From Scratch"

- In the previous notebook, we used small embedding dimensions for token inputs and outputs for ease of illustration, ensuring they neatly fit on the screen
- In this notebook, we consider embedding and model sizes akin to a small GPT-2 model
- We'll specifically code the architecture of the smallest GPT-2 model (124 million parameters), as outlined in Radford et al.'s Language Models are Unsupervised Multitask Learners (note that the initial report lists it as 117M parameters, but this was later corrected in the model weight repository)



- The next notebook will show how to load pretrained weights into our implementation, which will be compatible with model sizes of 345, 762, and 1542 million parameters
- Models like Llama and others are very similar to this model, since they are all based on the same core concepts



- Configuration details for the 124 million parameter GPT-2 model (GPT-2 "small") include:

In [2]:
```
GPT_CONFIG_124M = {
    "vocab_size": 50257,    # Vocabulary size
    "context_length": 1024, # Context length
    "emb_dim": 768,         # Embedding dimension
    "n_heads": 12,          # Number of attention heads
    "n_layers": 12,         # Number of layers
    "drop_rate": 0.0,       # Dropout rate
    "qkv_bias": False       # Query-Key-Value bias
}
```

3.2 Coding the GPT model

- We are almost there: now let's plug in the transformer block into the architecture we coded at the very beginning of this notebook so that we obtain a useable GPT architecture
- Note that the transformer block is repeated multiple times; in the case of the smallest 124M GPT-2 model, we repeat it 12 times:

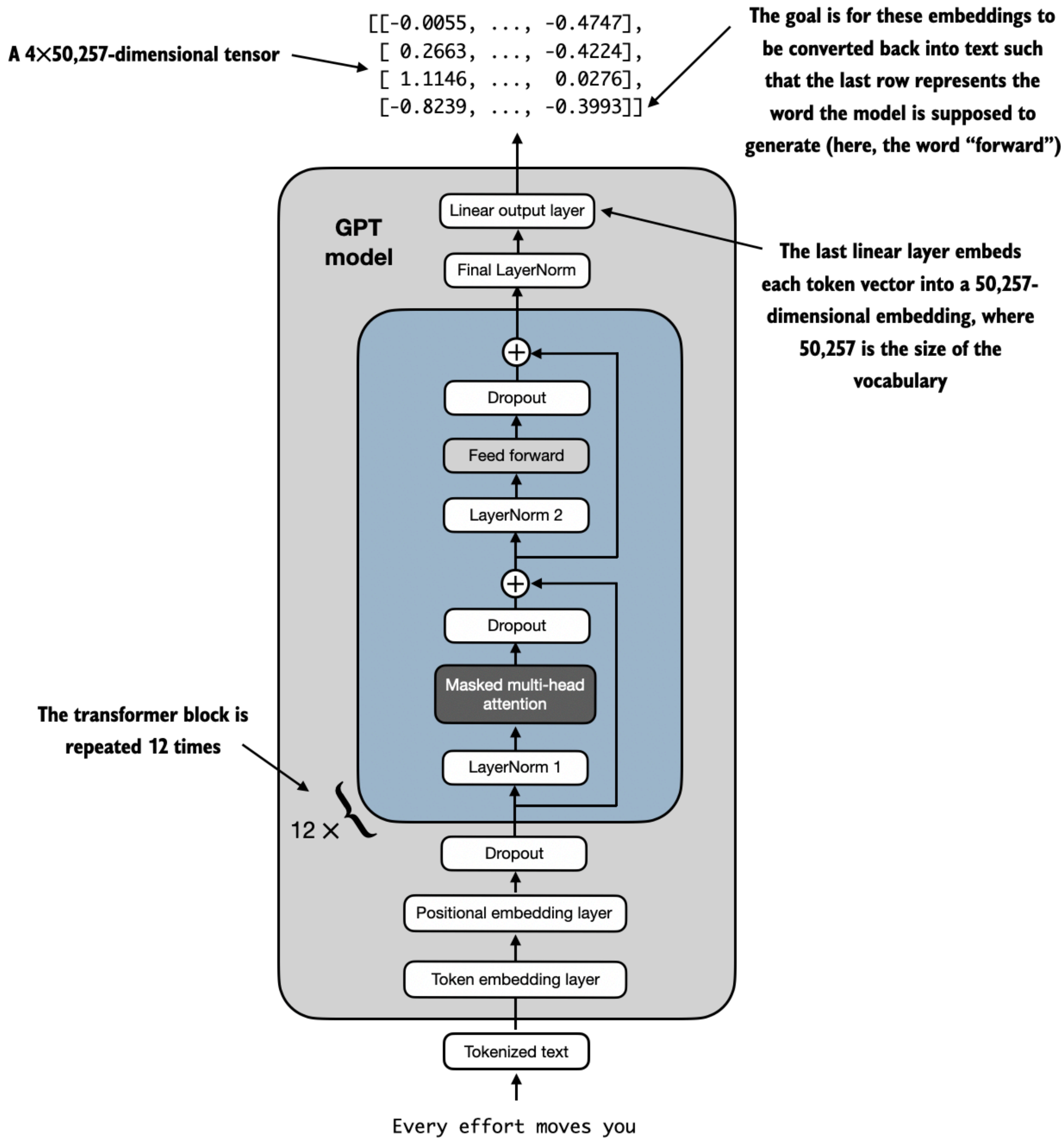


Figure adapted from "Build a Large Language Model From Scratch"

- The corresponding code implementation, where `cfg["n_layers"] = 12`:

In [3]:
```
import torch.nn as nn
from supplementary import TransformerBlock, LayerNorm


class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"]
        self.drop_emb = nn.Dropout(cfg["drop_rate"])

        self.trf_blocks = nn.Sequential(
            *[TransformerBlock(cfg) for _ in range(cfg["n_layers"])])

        self.final_norm = LayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"], bias=False
        )

    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        tok_embeds = self.tok_emb(in_idx)
        pos_embeds = self.pos_emb(torch.arange(seq_len, device=in_idx.dev
        x = tok_embeds + pos_embeds  # Shape [batch_size, num_tokens, emb
        x = self.drop_emb(x)
        x = self.trf_blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logits
```

- Using the configuration of the 124M parameter model, we can now instantiate this GPT model with random initial weights as follows:

In [4]:
```
import torch
import tiktoken

tokenizer = tiktoken.get_encoding("gpt2")

batch = []

txt1 = "Every effort moves you"
txt2 = "Every day holds a"

batch.append(torch.tensor(tokenizer.encode(txt1)))
batch.append(torch.tensor(tokenizer.encode(txt2)))
batch = torch.stack(batch, dim=0)
print(batch)
```

```
tensor([[6109, 3626, 6100,  345],
        [6109, 1110, 6622,  257]])
```

In [5]:
```
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)

out = model(batch)
print("Input batch:\n", batch)
print("\nOutput shape:", out.shape)
print(out)
```

```
Input batch:
 tensor([[6109, 3626, 6100,  345],
        [6109, 1110, 6622,  257]])

Output shape: torch.Size([2, 4, 50257])
tensor([[[ 6.4164e-02,  2.0443e-01, -1.6945e-01,  ...,  1.7887e-01,
           2.1921e-01, -5.8153e-01],
         [ 3.7736e-01, -4.2545e-01, -6.5874e-01,  ..., -2.5050e-01,
           4.6553e-01, -2.5760e-01],
         [ 8.8996e-01, -1.3770e-01,  1.4748e-01,  ...,  1.7770e-01,
          -1.2015e-01, -1.8902e-01],
         [-9.7276e-01,  9.7338e-02, -2.5419e-01,  ...,  1.1035e+00,
           3.7639e-01, -5.9006e-01]],

        [[ 6.4164e-02,  2.0443e-01, -1.6945e-01,  ...,  1.7887e-01,
           2.1921e-01, -5.8153e-01],
         [ 1.3433e-01, -2.1289e-01, -2.7021e-02,  ...,  8.1153e-01,
          -4.7410e-02,  3.1186e-01],
         [ 8.9996e-01,  9.5396e-01, -1.7896e-01,  ...,  8.3053e-01,
           2.7657e-01, -2.4577e-02],
         [-9.2814e-05,  1.9390e-01,  5.1217e-01,  ...,  1.1915e+00,
          -1.6431e-01,  3.7046e-02]]], grad_fn=<UnsafeViewBackward0>)
```

In [6]:
```
out[0][0].shape
```

Out[6]:
```
torch.Size([50257])
```

- We will train this model in the next notebook

3.4 Generating text

- LLMs like the GPT model we implemented above are used to generate one word at a time

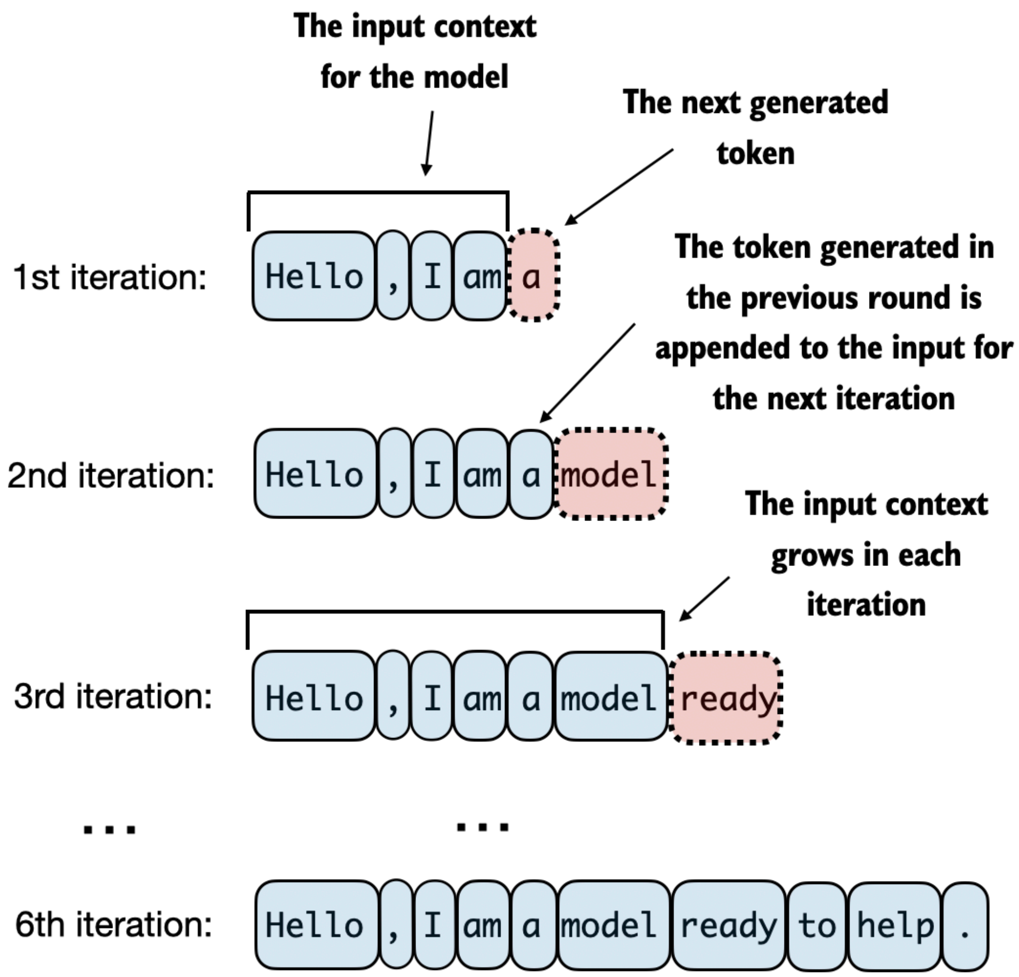


Figure adapted from "Build a Large Language Model From Scratch"

- The following `generate_text_simple` function implements greedy decoding, which is a simple and fast method to generate text
- In greedy decoding, at each step, the model chooses the word (or token) with the highest probability as its next output (the highest logit corresponds to the highest probability, so we technically wouldn't even have to compute the softmax function explicitly)
- The figure below depicts how the GPT model, given an input context, generates the next word token

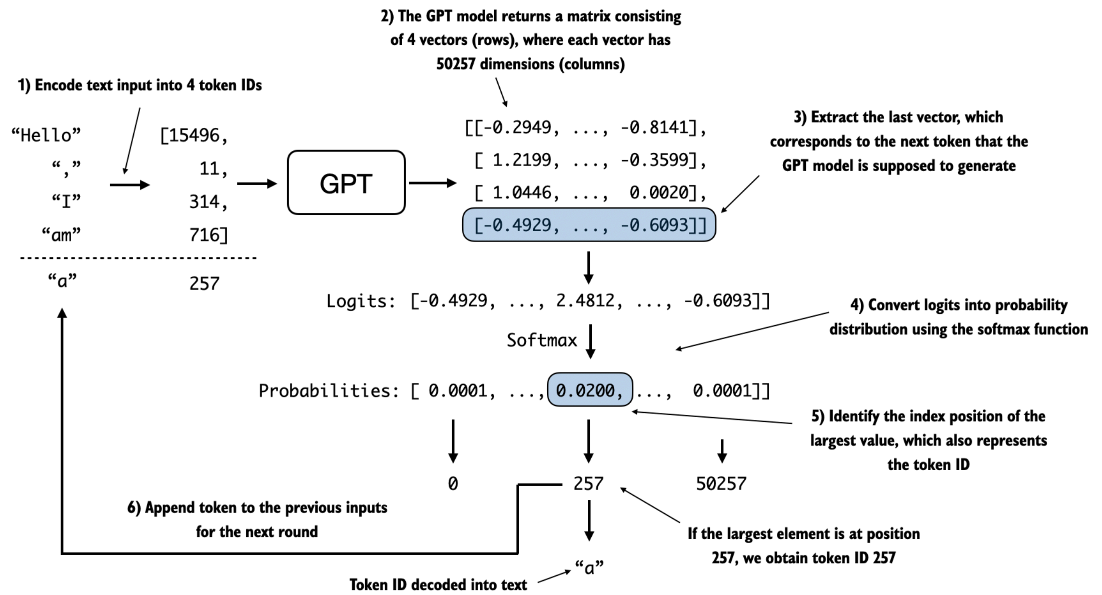


Figure adapted from "Build a Large Language Model From Scratch"

In [7]:
```
def generate_text_simple(model, idx, max_new_tokens, context_size):
    # idx is (batch, n_tokens) array of indices in the current context
    for _ in range(max_new_tokens):

        # Crop current context if it exceeds the supported context size
        # E.g., if LLM supports only 5 tokens, and the context size is 10
        # then only the last 5 tokens are used as context
        idx_cond = idx[:, -context_size:]

        # Get the predictions
        with torch.no_grad():
            logits = model(idx_cond)

        # Focus only on the last time step
        # (batch, n_tokens, vocab_size) becomes (batch, vocab_size)
        logits = logits[:, -1, :]

        # Apply softmax to get probabilities
        probas = torch.softmax(logits, dim=-1)  # (batch, vocab_size)

        # Get the idx of the vocab entry with the highest probability val
        idx_next = torch.argmax(probas, dim=-1, keepdim=True)  # (batch,

        # Append sampled index to the running sequence
        idx = torch.cat((idx, idx_next), dim=1)  # (batch, n_tokens+1)

    return idx
```

- The `generate_text_simple` above implements an iterative process, where it creates one token at a time

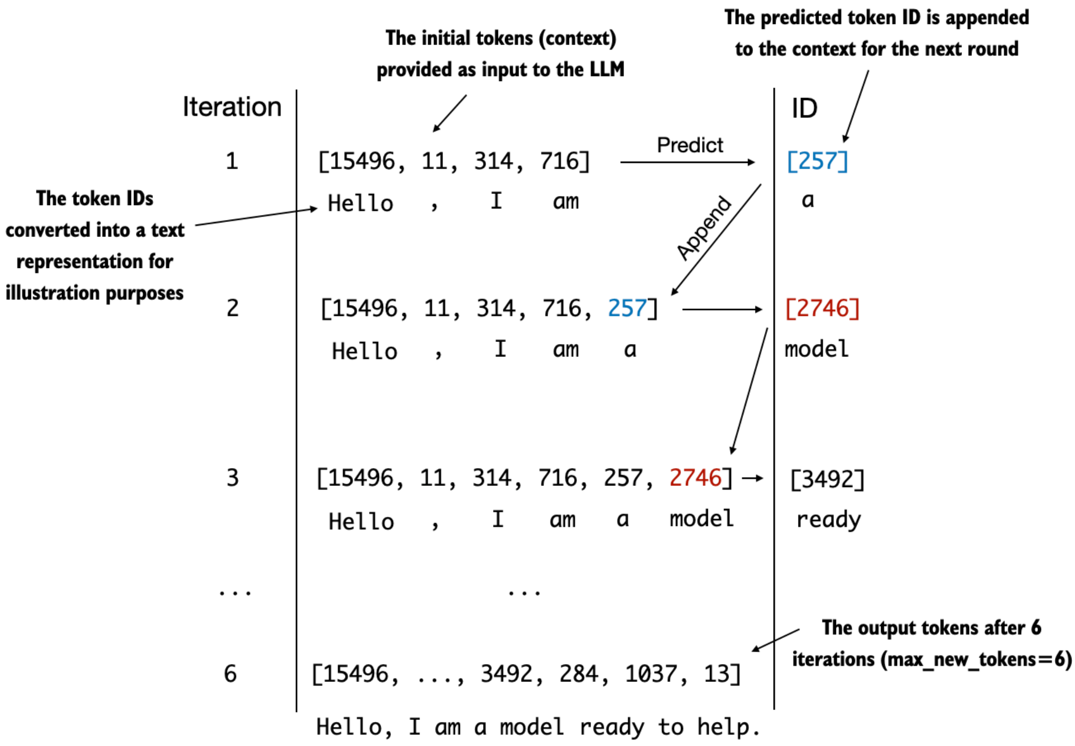


Figure adapted from "Build a Large Language Model From Scratch"

Exercise: Generate some text

1. Use the `tokenizer.encode` method to prepare some input text
2. Then, convert this text into a pytprch tensor via (`torch.tensor`)
3. Add a batch dimension via `.unsqueeze(0)`
4. Use the `generate_text_simple` function to have the GPT generate some text based on your prepared input text
5. The output from step 4 will be token IDs, convert them back into text via the `tokenizer.decode` method

In [8]:
```
model.eval();  # disable dropout
```

Solution

In [9]:
```
start_context = "Hello, I am"

encoded = tokenizer.encode(start_context)
print("encoded:", encoded)

encoded_tensor = torch.tensor(encoded).unsqueeze(0)
print("encoded_tensor.shape:", encoded_tensor.shape)
```

```
encoded: [15496, 11, 314, 716]
encoded_tensor.shape: torch.Size([1, 4])
```

In [10]:
```
out = generate_text_simple(
    model=model,
    idx=encoded_tensor,
    max_new_tokens=6,
    context_size=GPT_CONFIG_124M["context_length"]
)

print("Output:", out)
print("Output length:", len(out[0]))
```

```
Output: tensor([[15496,    11,   314,   716, 27018, 24086, 47843, 30961, 4
2348,  7267]])
Output length: 10
```

- Remove batch dimension and convert back into text:

In [11]:
```
decoded_text = tokenizer.decode(out.squeeze(0).tolist())
print(decoded_text)
```

```
Hello, I am Featureiman Byeswickattribute argue
```

- Note that the model is untrained; hence the random output texts above
- We will train the model in the next notebook

NEXT: Pretraining LLMs