Here's a **minimal GPT-2-like Transformer model** implemented in **PyTorch** for **text generation**, with the following core components:

- Token embedding
- Positional encoding
- Multi-head self-attention
- Transformer blocks
- Language modeling head (causal LM)

🧠 GPT-2-like Transformer in PyTorch

✅ Prerequisites

```
pip install torch torchvision torchaudio
```

📦 Full Code Implementation

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

# Config
class GPT2Config:
    vocab_size = 50257       # same as GPT-2
    block_size = 128         # context window
    n_layer = 4
    n_head = 4
    n_embd = 256             # embedding size
    dropout = 0.1

# Causal Self-Attention
class CausalSelfAttention(nn.Module):
    def __init__(self, config):
        super().__init__()
        assert config.n_embd % config.n_head == 0
        self.n_head = config.n_head
        self.head_dim = config.n_embd // config.n_head

        self.qkv = nn.Linear(config.n_embd, 3 * config.n_embd)
        self.out_proj = nn.Linear(config.n_embd, config.n_embd)
        self.dropout = nn.Dropout(config.dropout)

        # causal mask
        self.register_buffer("mask",
torch.tril(torch.ones(config.block_size,
```

PROF

```
config.block_size)).unsqueeze(0).unsqueeze(0))

    def forward(self, x):
        B, T, C = x.size()
        qkv = self.qkv(x)
        q, k, v = qkv.chunk(3, dim=-1)

        q = q.view(B, T, self.n_head, self.head_dim).transpose(1, 2)
        k = k.view(B, T, self.n_head, self.head_dim).transpose(1, 2)
        v = v.view(B, T, self.n_head, self.head_dim).transpose(1, 2)

        att = (q @ k.transpose(-2, -1)) / math.sqrt(self.head_dim)
        att = att.masked_fill(self.mask[:, :, :T, :T] == 0, float('-
inf'))
        att = F.softmax(att, dim=-1)
        att = self.dropout(att)

        out = (att @ v).transpose(1, 2).contiguous().view(B, T, C)
        return self.out_proj(out)

# Transformer Block
class TransformerBlock(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.ln1 = nn.LayerNorm(config.n_embd)
        self.attn = CausalSelfAttention(config)
        self.ln2 = nn.LayerNorm(config.n_embd)
        self.mlp = nn.Sequential(
            nn.Linear(config.n_embd, 4 * config.n_embd),
            nn.GELU(),
            nn.Linear(4 * config.n_embd, config.n_embd),
            nn.Dropout(config.dropout),
        )

    def forward(self, x):
        x = x + self.attn(self.ln1(x))
        x = x + self.mlp(self.ln2(x))
        return x

# GPT-2 Model
class GPT2(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.token_embed = nn.Embedding(config.vocab_size,
config.n_embd)
        self.pos_embed = nn.Parameter(torch.zeros(1, config.block_size,
config.n_embd))
        self.blocks = nn.Sequential(*[TransformerBlock(config) for _ in
range(config.n_layer)])
        self.ln_f = nn.LayerNorm(config.n_embd)
        self.head = nn.Linear(config.n_embd, config.vocab_size,
bias=False)
```

```
        self.block_size = config.block_size
        self.apply(self._init_weights)

    def forward(self, idx, targets=None):
        B, T = idx.size()
        assert T <= self.block_size

        tok_emb = self.token_embed(idx)                # (B, T, C)
        pos_emb = self.pos_embed[:, :T, :]             # (1, T, C)
        x = tok_emb + pos_emb                          # (B, T, C)
        x = self.blocks(x)
        x = self.ln_f(x)
        logits = self.head(x)

        if targets is None:
            return logits, None
        else:
            loss = F.cross_entropy(logits.view(-1, logits.size(-1)),
targets.view(-1))
            return logits, loss

    def _init_weights(self, module):
        if isinstance(module, (nn.Linear, nn.Embedding)):
            nn.init.normal_(module.weight, mean=0.0, std=0.02)

    @torch.no_grad()
    def generate(self, idx, max_new_tokens):
        for _ in range(max_new_tokens):
            idx_cond = idx[:, -self.block_size:]
            logits, _ = self(idx_cond)
            next_token = torch.argmax(logits[:, -1, :], dim=-1,
keepdim=True)
            idx = torch.cat([idx, next_token], dim=1)
        return idx
```

PROF

🔧 Example Usage (with Dummy Data)

```
config = GPT2Config()
model = GPT2(config)

# Simulated dummy input (batch size 1, seq length 8)
x = torch.randint(0, config.vocab_size, (1, 8))
logits, loss = model(x, x)
print("Loss:", loss.item())

# Generate tokens from prompt
generated = model.generate(x, max_new_tokens=20)
print("Generated token IDs:", generated)
```

🧠 Notes

- This model is **causal**, meaning each token only attends to previous tokens — exactly like GPT-2.
- For production, replace token embedding with a tokenizer like Hugging Face's `GPT2Tokenizer`.
- You can scale layers or use `nn.TransformerDecoderLayer` if you want a shortcut.
- The full GPT-2 uses:
  - 12 layers
  - 768 hidden units
  - 12 heads
  - Positional embeddings, LayerNorm, GELU, etc.