self-attention, numpy method:

```
import numpy as np

def compute_qkv(X, W_q, W_k, W_v):
     Q = X @ W_q
     K = X @ W_k
     V = X @ W_v
     return Q, K, V

def self_attention(Q, K, V):
     dim_K = K.shape[-1]
     scores = (Q @ K.T)/np.sqrt(dim_K)
     max_scores = np.max(scores, axis = -1, keepdims = True)
     new_scores = scores - max_scores
     attention_weights = np.exp(new_scores) /
np.sum(np.exp(new_scores),axis = -1, keepdims = True)
     attention_output = attention_weights @ V
     return attention_output
```

self-attention, pytorch method:

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

class SelfAttention(nn.Module):
  def __init__(self, embed_dim):
    super().__init__()
    self.embedding_dim = embed_dim
    self.get_q = nn.Linear(embed_dim, embed_dim, bias= False)
    self.get_k = nn.Linear(embed_dim, embed_dim, bias= False)
    self.get_v = nn.Linear(embed_dim, embed_dim, bias= False)

  def forward(self, x, mask= False):  # x.shape is [B,s,e] B: batch
size, s: sequence length, e: embedding size
    Q = self.get_q(x)
    K = self.get_k(x) # dim K is in each batch the embedding size =
self.embedding_dim
    V = self.get_v(x) # has shape [B,s,e]
    attention_scores = torch.matmul(Q,K.transpose(-2,-
1))/(self.embedding_dim**(-0.5))
    max_scores = torch.max(attention_scores, dim =-1, keepdim= True)
[0] # get max value of each row, not indices
    new_scores = attention_scores - max_scores # torch will auto
expand dim so every entry in a row - max value, and in softmax
exp(negative value) won't be huge
    attention_weights = torch.softmax(new_scores, dim =-1) # has shape
[B,s,s]
```

```
    attention_output = torch.matmul(attention_weights, V) # out put
has same dim as x
    return attention_output



# --- 2) Create a dummy batch of embeddings ---
B, T, C = 2, 5, 16           # batch size 2, sequence length 5, embed
dim 16
x = torch.randn(B, T, C)     # random input

# --- 3) Instantiate and run ---
attn_layer = SelfAttention(embed_dim=C)
output = attn_layer(x)

print("output shape:", output.shape)   # → torch.Size([2, 5, 16])

# --- 4) (Optional) Visualize one attention matrix ---
print("attention weights for sample 0:\n", output[0])
```

```
output shape: torch.Size([2, 5, 16])
attention weights for sample 0:
 tensor([[ 0.4882, -0.1974,  0.4953,  0.0751,  0.3127, -0.0228, -
0.0331,  0.3205,
         -0.1828,  0.3713, -0.3189, -0.0096, -0.2000,  0.1807,
0.4232,  0.1530],
        [ 0.5012,  0.0043, -0.5890, -0.1467, -0.4039, -0.6152, -
0.8488,  0.2181,
         -1.1083, -0.2581,  1.3502,  0.3465, -0.9484, -0.3295,
0.2146, -0.1683],
        [ 0.6680, -0.3255,  0.2919,  0.0046,  0.2490, -0.2500, -
0.0585,  0.1657,
          0.0210,  0.3828, -0.3388,  0.0551, -0.1739, -0.0188,
0.2955,  0.2897],
        [-0.0268,  0.1069,  1.0452,  0.2255,  0.4878,  0.6085, -
0.0106,  0.7298,
         -0.7010,  0.3331, -0.2881, -0.1790, -0.3185,  0.7287,
0.7232, -0.2406],
        [-0.2968, -0.5228,  0.5184, -0.4663,  0.2590,  0.4869, -
0.9638,  0.4480,
         -0.3621,  0.0240, -0.0476,  0.0554, -1.2374,  0.4955, -
0.1072, -0.5704]],
       grad_fn=<SelectBackward0>)
```

Multi-head attention

```
# --- 2) Create a dummy batch of embeddings ---
B, T, C = 2, 5, 16           # batch size 2, sequence length 5, embed
dim 16
```

```
x = torch.randn(B, T, C)     # random input

# --- 3) Instantiate and run ---
attn_layer = SelfAttention(embed_dim=C)
output = attn_layer(x)

print("output shape:", output.shape)   # → torch.Size([2, 5, 16])

# --- 4) (Optional) Visualize one attention matrix ---
print("attention weights for sample 0:\n", output[0])
```

```
output shape: torch.Size([2, 5, 16])
attention weights for sample 0:
 tensor([[ 0.4882, -0.1974,  0.4953,  0.0751,  0.3127, -0.0228, -
0.0331,  0.3205,
         -0.1828,  0.3713, -0.3189, -0.0096, -0.2000,  0.1807,
0.4232,  0.1530],
        [ 0.5012,  0.0043, -0.5890, -0.1467, -0.4039, -0.6152, -
0.8488,  0.2181,
         -1.1083, -0.2581,  1.3502,  0.3465, -0.9484, -0.3295,
0.2146, -0.1683],
        [ 0.6680, -0.3255,  0.2919,  0.0046,  0.2490, -0.2500, -
0.0585,  0.1657,
          0.0210,  0.3828, -0.3388,  0.0551, -0.1739, -0.0188,
0.2955,  0.2897],
        [-0.0268,  0.1069,  1.0452,  0.2255,  0.4878,  0.6085, -
0.0106,  0.7298,
         -0.7010,  0.3331, -0.2881, -0.1790, -0.3185,  0.7287,
0.7232, -0.2406],
        [-0.2968, -0.5228,  0.5184, -0.4663,  0.2590,  0.4869, -
0.9638,  0.4480,
         -0.3621,  0.0240, -0.0476,  0.0554, -1.2374,  0.4955, -
0.1072, -0.5704]],
       grad_fn=<SelectBackward0>)
```

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

class MultiHeadAttention(nn.Module):
  def __init__(self, d_model, num_heads):
    super().__init__()
    assert d_model % num_heads == 0
    self.d_model = d_model
    self.num_heads = num_heads
    self.d_k = d_model // num_heads

    self.W_Q = nn.Linear(d_model, d_model) # in theory, Q is
multiplied by num_heads small head W^{i}_Q. Here W_Q is concatenation
```

```
of W^{i}_Q of all heads.
    self.W_K = nn.Linear(d_model, d_model)
    self.W_V = nn.Linear(d_model, d_model)
    self.W_O = nn.Linear(d_model, d_model)

  def forward(self, Q,K,V, mask = None): #each of Q,K,V has shape
[B,s,d_model]:batch size, sequence length, embedding feature dimension
size
    batch_size = Q.size(0)
    seq_len = Q.size(1)

    Q = self.W_Q(Q).reshape(batch_size, seq_len, self.num_heads,
self.d_k) #project Q onto all heads, and reshape to sepreate them
    Q = Q.transpose(-2,-3) #now last two dims are (seq_len,d_k)
    K = self.W_K(K).reshape(batchsize, seq_len, self.num_heads,
self.d_k)
    K = K.transpose(-2,-3)
    V = self.W_V(V).reshape(batchsize, seq_len, self.num_heads,
self.d_k)
    V = V.transpose(-2,-3)

    scores = torch.matmul(Q,K.transpose(-2,-1))/ math.sqrt(self.d_k)

    # decide if use mask, if use, mask position that mask entry == 0,
by adding -inf.
    if mask is not None:
      scores = scores.masked_fill(mask == 0, float('-inf'))

    attention_weights = torch.softmax(scores, dim = -1) #softmax
applied on each token's scores vector
    attention_output = torch.matmul(attention_weights, V) # it's shape
(batch_size, num_heads, seq_len, d_k)
    attention_output.transpose(-2,-3)
    attention_output.reshape(batch_size,seq_len,self.d_model) # it
actually does concatenation of heads
    output = self.W_O(attention_output) # 用来融合不同head 的信息（这是论文
中的“output projection”）。

    return output




class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super().__init__()
        assert d_model % num_heads == 0
        self.num_heads = num_heads
        self.d_k = d_model // num_heads
```

```
        self.W_Q = nn.Linear(d_model, d_model)
        self.W_K = nn.Linear(d_model, d_model)
        self.W_V = nn.Linear(d_model, d_model)
        self.W_O = nn.Linear(d_model, d_model)

    def forward(self, Q, K, V, mask=None):
        batch_size = Q.size(0)

        Q = self.W_Q(Q).view(batch_size, -1, self.num_heads,
self.d_k).transpose(1, 2)
        K = self.W_K(K).view(batch_size, -1, self.num_heads,
self.d_k).transpose(1, 2)
        V = self.W_V(V).view(batch_size, -1, self.num_heads,
self.d_k).transpose(1, 2)

        scores = torch.matmul(Q, K.transpose(-2, -1)) /
math.sqrt(self.d_k)

        if mask is not None:
            scores = scores.masked_fill(mask == 0, float('-inf'))

        attn = torch.softmax(scores, dim=-1)
        out = torch.matmul(attn, V)  # [batch, heads, seq_len, d_k]
        out = out.transpose(1, 2).contiguous().view(batch_size, -1,
self.num_heads * self.d_k)
        return self.W_O(out)
```

#Complete transformer model

##Transformer 模型（ 和论文一致 ）

###sublayer：multi-head attention + position encoding + feedforward + residual + layernorm ）。

Comments: laynorm 在代码中我们实际使用 pre-norm， 即对每一个 multi-head attention 和 position encoding + feedforward sublayer，先做 layernorm 再做 sublayer，然后做 add residual。 ：

特性 Post-Norm（论文原版）vs Pre-Norm（后续改进）

性能差异 小规模时差别不大 大规模模型普遍用 Pre-Norm

：

1. 归一化位置 残差加法后 vs 残差加法前
2. 梯度流动 梯度必须经过 LayerNorm，可能被削弱 vs 梯度可以直接通过残差支路传播
3. 训练稳定性 在深层 Transformer（> 6 层）中容易梯度爆炸或消失 vs 更稳定，支持更深层（甚至 100+ 层）

：Pre-Norm 改进了 梯度流动路径，从而显著提升了深层 Transformer 的可训练性。

```
'''Encoder-decoder architecture
│
├── Encoder (N 层)
│    ├── Self-Attention
│    ├── FeedForward
│    └── Add & Norm
│
└── Decoder (N 层)
     ├── Masked Self-Attention  ← causal
     ├── Cross-Attention (对Encoder 输出)
     ├── FeedForward
     └── Add & Norm'''
```

Architecture

模块	功能
`MultiHeadAttention`	实现 Q-K-V 注意力机制
`FeedForward`	两层前馈网络
`LayerNorm`	层归一化（可训练 γ, β）
`EncoderLayer`	Self-Attn + FFN
`DecoderLayer`	Masked Self-Attn + Cross-Attn + FFN
`Encoder/Decoder`	堆叠多层
`Transformer`	Encoder + Decoder + 输出层

```
import torch
import torch.nn as nn
import math

# ========= 1 位置编码Positional Encoding =========
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super().__init__()
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-
math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)  # [1, max_len, d_model]
        self.register_buffer('pe', pe)

    def forward(self, x):
        # x: [batch, seq_len, d_model]
        seq_len = x.size(1)
        return x + self.pe[:, :seq_len]

# ========= 2 多头注意力Multi-Head Attention =========
class MultiHeadAttention(nn.Module):
```

```
    def __init__(self, d_model, num_heads):
        super().__init__()
        assert d_model % num_heads == 0
        self.num_heads = num_heads
        self.d_k = d_model // num_heads

        self.W_Q = nn.Linear(d_model, d_model)
        self.W_K = nn.Linear(d_model, d_model)
        self.W_V = nn.Linear(d_model, d_model)
        self.W_O = nn.Linear(d_model, d_model)

    def forward(self, Q, K, V, mask=None):
        batch_size = Q.size(0)

        Q = self.W_Q(Q).view(batch_size, -1, self.num_heads,
self.d_k).transpose(1, 2)
        K = self.W_K(K).view(batch_size, -1, self.num_heads,
self.d_k).transpose(1, 2)
        V = self.W_V(V).view(batch_size, -1, self.num_heads,
self.d_k).transpose(1, 2)

        scores = torch.matmul(Q, K.transpose(-2, -1)) /
math.sqrt(self.d_k)

        if mask is not None:
            scores = scores.masked_fill(mask == 0, float('-inf'))

        attn = torch.softmax(scores, dim=-1)
        out = torch.matmul(attn, V)  # [batch, heads, seq_len, d_k]
        out = out.transpose(1, 2).contiguous().view(batch_size, -1,
self.num_heads * self.d_k)
        return self.W_O(out)

# ========= 3 前馈网络FeedForward =========
class FeedForward(nn.Module):
    def __init__(self, d_model, d_ff, dropout=0.1):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.ReLU(),
            nn.Linear(d_ff, d_model),
            nn.Dropout(dropout)
        )

    def forward(self, x):
        return self.net(x)

# ========= 4 LayerNorm =========
class LayerNorm(nn.Module):
    def __init__(self, d_model, eps=1e-6):
```

```
        super().__init__()
        self.gamma = nn.Parameter(torch.ones(d_model))
        self.beta = nn.Parameter(torch.zeros(d_model))
        self.eps = eps

    def forward(self, x):
        mean = x.mean(-1, keepdim=True)
        std = x.std(-1, keepdim=True)
        return self.gamma * (x - mean) / (std + self.eps) + self.beta

# ========= 5 Encoder Layer =========
class EncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout=0.1):
        super().__init__()
        self.norm1 = LayerNorm(d_model)
        self.norm2 = LayerNorm(d_model)
        self.attn = MultiHeadAttention(d_model, num_heads)
        self.ffn = FeedForward(d_model, d_ff, dropout)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        # Pre-Norm + 残差连接
        x2 = self.attn(self.norm1(x), self.norm1(x), self.norm1(x),
mask)
        x = x + self.dropout(x2)
        x2 = self.ffn(self.norm2(x))
        x = x + self.dropout(x2)
        return x

# ========= 6 Decoder Layer =========
class DecoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout=0.1):
        super().__init__()
        self.norm1 = LayerNorm(d_model)
        self.norm2 = LayerNorm(d_model)
        self.norm3 = LayerNorm(d_model)

        self.self_attn = MultiHeadAttention(d_model, num_heads)
        self.cross_attn = MultiHeadAttention(d_model, num_heads)
        self.ffn = FeedForward(d_model, d_ff, dropout)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, enc_out, tgt_mask=None, memory_mask=None):
        # Masked Self-Attention
        x2 = self.self_attn(self.norm1(x), self.norm1(x),
self.norm1(x), tgt_mask)
        x = x + self.dropout(x2)
        # Cross Attention
        x2 = self.cross_attn(self.norm2(x), enc_out, enc_out,
memory_mask)
```

```
        x = x + self.dropout(x2)
        # Feed Forward
        x2 = self.ffn(self.norm3(x))
        x = x + self.dropout(x2)
        return x

# ========= 7 Encoder Stack =========
class Encoder(nn.Module):
    def __init__(self, vocab_size, d_model, num_layers, num_heads,
d_ff, dropout=0.1):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, d_model)
        self.pos = PositionalEncoding(d_model)
        self.layers = nn.ModuleList([
            EncoderLayer(d_model, num_heads, d_ff, dropout)
        for _ in range(num_layers)])
        self.norm = LayerNorm(d_model)

    def forward(self, src, mask=None):
        x = self.embed(src)
        x = self.pos(x)
        for layer in self.layers:
            x = layer(x, mask)
        return self.norm(x)

# ========= 8 Decoder Stack =========
class Decoder(nn.Module):
    def __init__(self, vocab_size, d_model, num_layers, num_heads,
d_ff, dropout=0.1):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, d_model)
        self.pos = PositionalEncoding(d_model)
        self.layers = nn.ModuleList([
            DecoderLayer(d_model, num_heads, d_ff, dropout)
        for _ in range(num_layers)])
        self.norm = LayerNorm(d_model)

    def forward(self, tgt, enc_out, tgt_mask=None, memory_mask=None):
        x = self.embed(tgt)
        x = self.pos(x)
        for layer in self.layers:
            x = layer(x, enc_out, tgt_mask, memory_mask)
        return self.norm(x)

# ========= 9 整体Transformer =========
class Transformer(nn.Module):
    def __init__(self, src_vocab, tgt_vocab, d_model=512,
num_layers=6,
                 num_heads=8, d_ff=2048, dropout=0.1):
        super().__init__()
```

```
        self.encoder = Encoder(src_vocab, d_model, num_layers,
num_heads, d_ff, dropout)
        self.decoder = Decoder(tgt_vocab, d_model, num_layers,
num_heads, d_ff, dropout)
        self.output = nn.Linear(d_model, tgt_vocab)

    def forward(self, src, tgt, src_mask=None, tgt_mask=None,
memory_mask=None):
        enc_out = self.encoder(src, src_mask)
        dec_out = self.decoder(tgt, enc_out, tgt_mask, memory_mask)
        return self.output(dec_out)

# simple test
src = torch.randint(0, 100, (2, 10))  # batch=2, seq_len=10
tgt = torch.randint(0, 100, (2, 9))

model = Transformer(src_vocab=100, tgt_vocab=100)
out = model(src, tgt)
print(out.shape)
```

训练循环（training loop）。

我们使用经典设置：

任务：序列到序列（seq2seq），比如机器翻译（n to m），文本分类（n to 1）；

：nn.CrossEntropyLoss；

：torch.optim.Adam；

mask：包含 padding mask 和 causal mask（上三角 mask）。

```
# 1. 准备工具函数

import torch
import torch.nn.functional as F

# ===== 生成mask =====

def generate_square_subsequent_mask(sz: int):
    """生成上三角causal mask（未来token 置-inf）"""
    mask = torch.triu(torch.ones(sz, sz), diagonal=1)
    mask = mask.masked_fill(mask == 1, float('-inf'))
    return mask  # [seq_len, seq_len]

def create_padding_mask(seq, pad_idx=0):
    """生成padding mask"""
    return (seq != pad_idx).unsqueeze(1).unsqueeze(2)  #
[batch,1,1,seq_len]
```

```python
# 2. 准备模型和优化器

from torch import nn, optim

SRC_VOCAB = 100
TGT_VOCAB = 100
PAD_IDX = 0

model = Transformer(src_vocab=SRC_VOCAB, tgt_vocab=TGT_VOCAB,
d_model=512)
optimizer = optim.Adam(model.parameters(), lr=1e-4)
criterion = nn.CrossEntropyLoss(ignore_index=PAD_IDX)

# 3. 构造一批数据(data example)

batch_size = 2
src_seq_len = 10
tgt_seq_len = 9

src = torch.randint(1, SRC_VOCAB, (batch_size, src_seq_len))
tgt = torch.randint(1, TGT_VOCAB, (batch_size, tgt_seq_len))
tgt_y = torch.randint(1, TGT_VOCAB, (batch_size, tgt_seq_len))  # 预测
目标

# 4. 生成mask

# 1 源序列padding mask
src_mask = create_padding_mask(src, pad_idx=PAD_IDX)

# 2 目标序列causal mask（防止看未来）
tgt_seq_len = tgt.size(1)
tgt_mask = generate_square_subsequent_mask(tgt_seq_len)

# 3 目标padding mask（防止计算无效token）
tgt_padding_mask = create_padding_mask(tgt, pad_idx=PAD_IDX)

# 5. 训练循环（一个epoch ）

model.train()
optimizer.zero_grad()

# 前向传播
out = model(src, tgt, src_mask=src_mask, tgt_mask=tgt_mask)

# 输出维度: [batch, tgt_len, vocab]
# 调整为CrossEntropyLoss 需要的[batch*tgt_len, vocab]
logits = out.view(-1, TGT_VOCAB)
targets = tgt_y.view(-1)

loss = criterion(logits, targets)
loss.backward()
```

```python
#  ()
torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

optimizer.step()

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

# 6. 完整最简训练循环模板

EPOCHS = 10
for epoch in range(EPOCHS):
    model.train()
    optimizer.zero_grad()

    src_mask = create_padding_mask(src)
    tgt_mask = generate_square_subsequent_mask(tgt.size(1))

    out = model(src, tgt, src_mask=src_mask, tgt_mask=tgt_mask)
    logits = out.view(-1, TGT_VOCAB)
    targets = tgt_y.view(-1)

    loss = criterion(logits, targets)
    loss.backward()
    torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
    optimizer.step()

    print(f"Epoch {epoch+1}/{EPOCHS}, Loss = {loss.item():.4f}")
```

##总结：训练流程

步骤	内容
1数据输入	`src`, `tgt`, `tgt_y`
2 Mask 构造	`src_mask` (padding)，`tgt_mask` (causal)
3 前向传播	`model(src, tgt, ...)`
4 计算损失	`CrossEntropyLoss(ignore_index=PAD_IDX)`
5 反向传播	`.backward()`
6 优化参数	`optimizer.step()`
7 重复训练	epoch 循环

Comments：最终输出： 每个目标 token 的预测概率分布 [batch, tgt_len, vocab_size]， 训练时用 CrossEntropyLoss 对比真实目标 tgt_y。

autoregressive inference (greedy and sampling) using the Transformer you wrote earlier.

Key ideas:

Encode the source once (enc_out).

Start tgt with a start token (BOS).

Repeatedly decode using current tgt to get logits for the next token, pick next token (greedy / sample), append, stop on EOS or max length.

Use causal mask for target self-attention so each step cannot attend to future tokens.

Provide memory_mask (padding mask) for encoder outputs so cross-attention ignores padded positions.

Use torch.no_grad() and model.eval().