# Encoder-Decoder架构

最初版本transformer由6组完全相同的encoder和6组完全相同decoder组成。

## 位置编码



这种方法属于相对位置编码，优点是能够扩展至任意序列长度

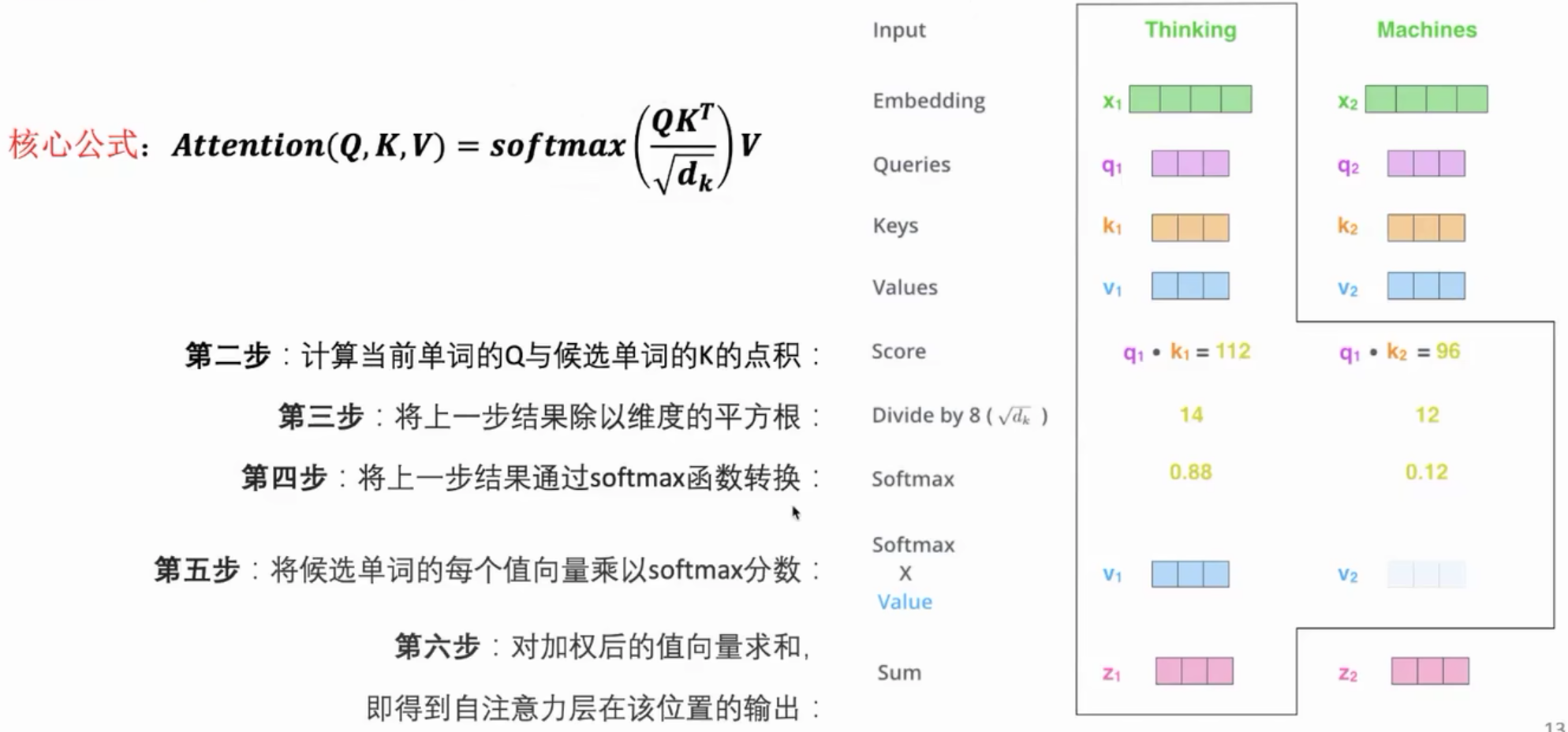


## Self-Attention

1）计算每个单词的Q、K、V



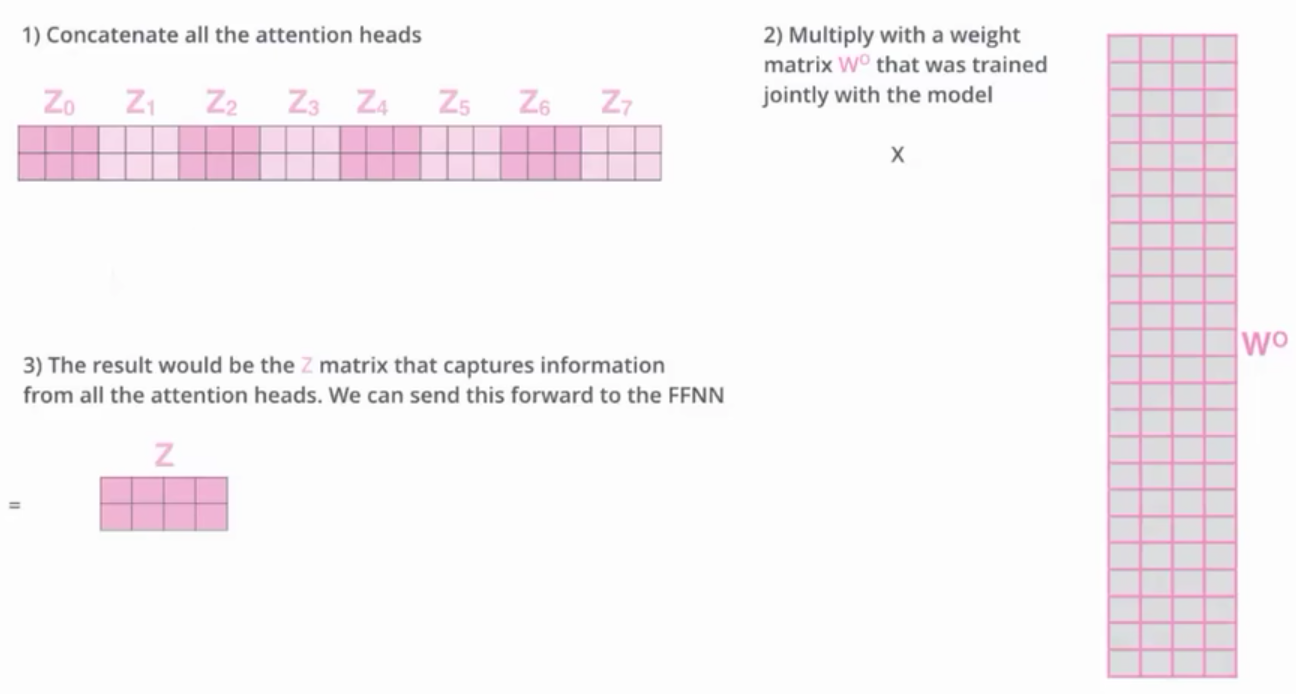
2）按公式计算剩余步骤，这里除以维度（64）的平方根，为了使梯度更稳定



## Muti-Head Attention

为了获取不同层次的信息。

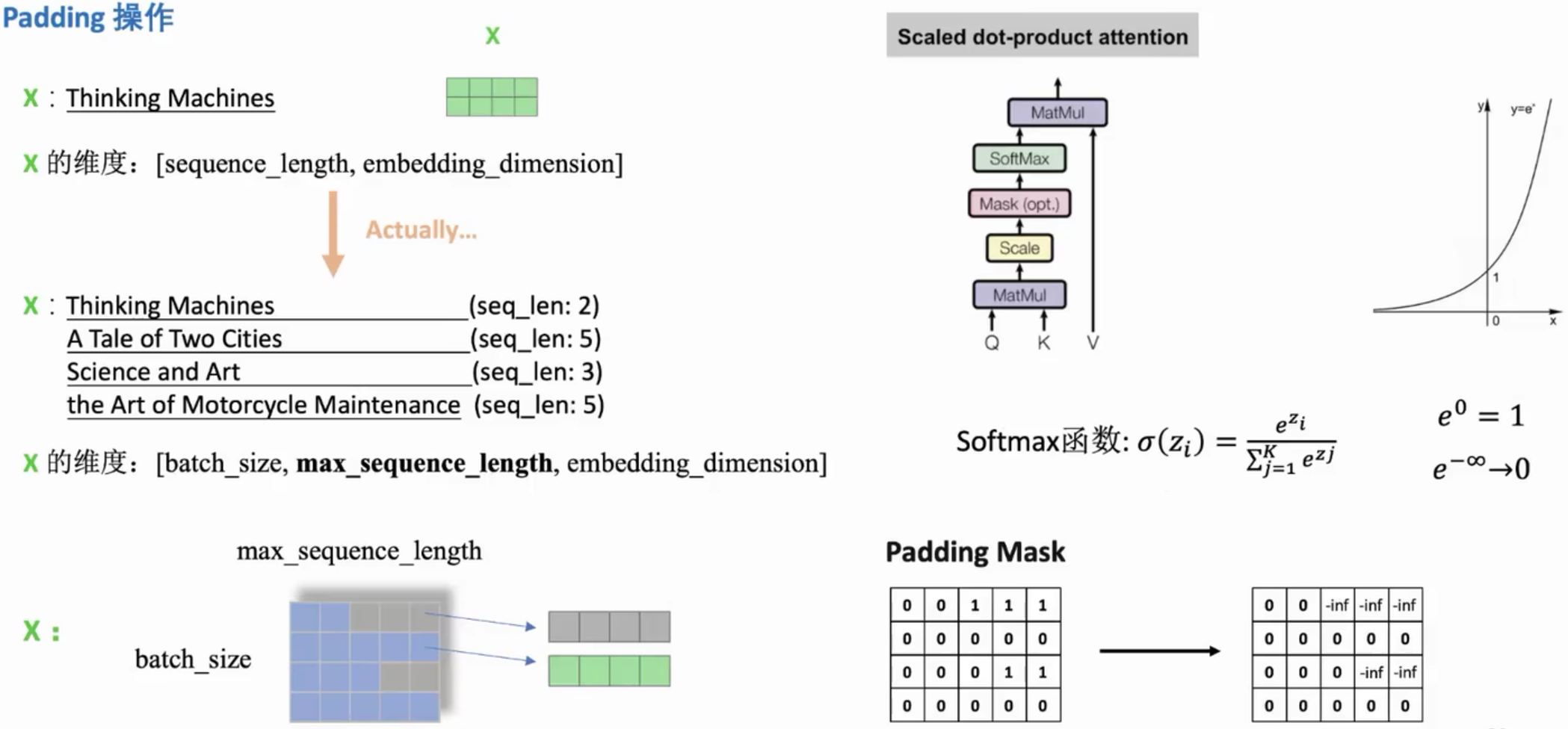
使用8个头，本质上使用了8组不同的转化矩阵WQ、WK、WV，然后合并。



## Padding

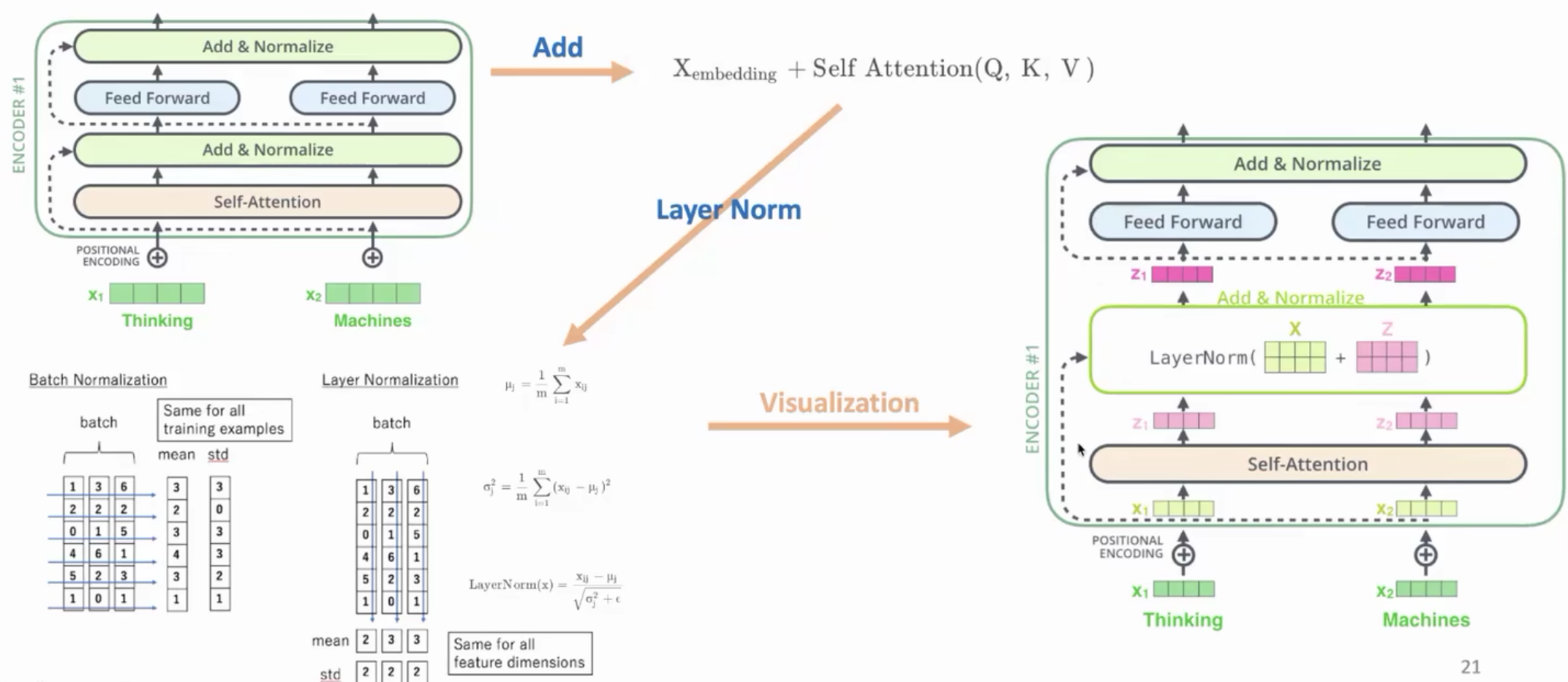
为了让batch中序列长度统一。

将短句用-inf补齐，通过数学方式让该位置输出为0，对全局不产生影响

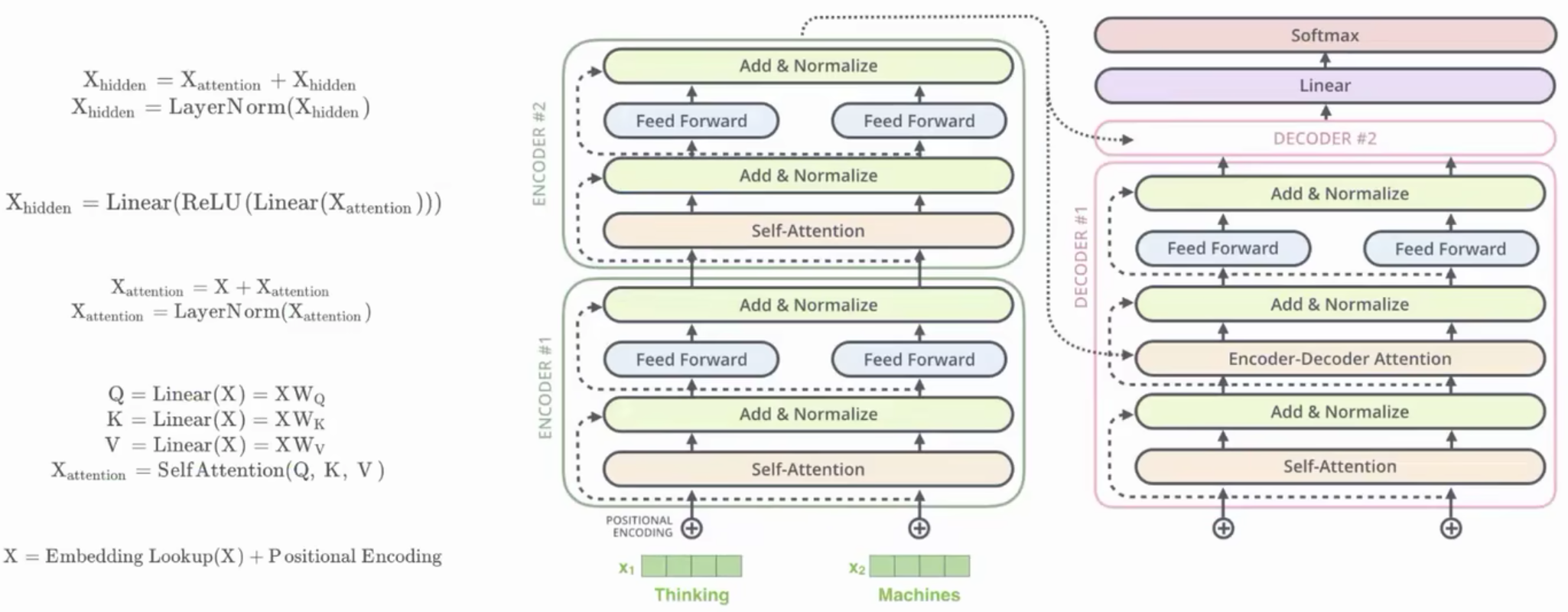


## 残差结构&层归一化

encoder和decoder都用该结构。

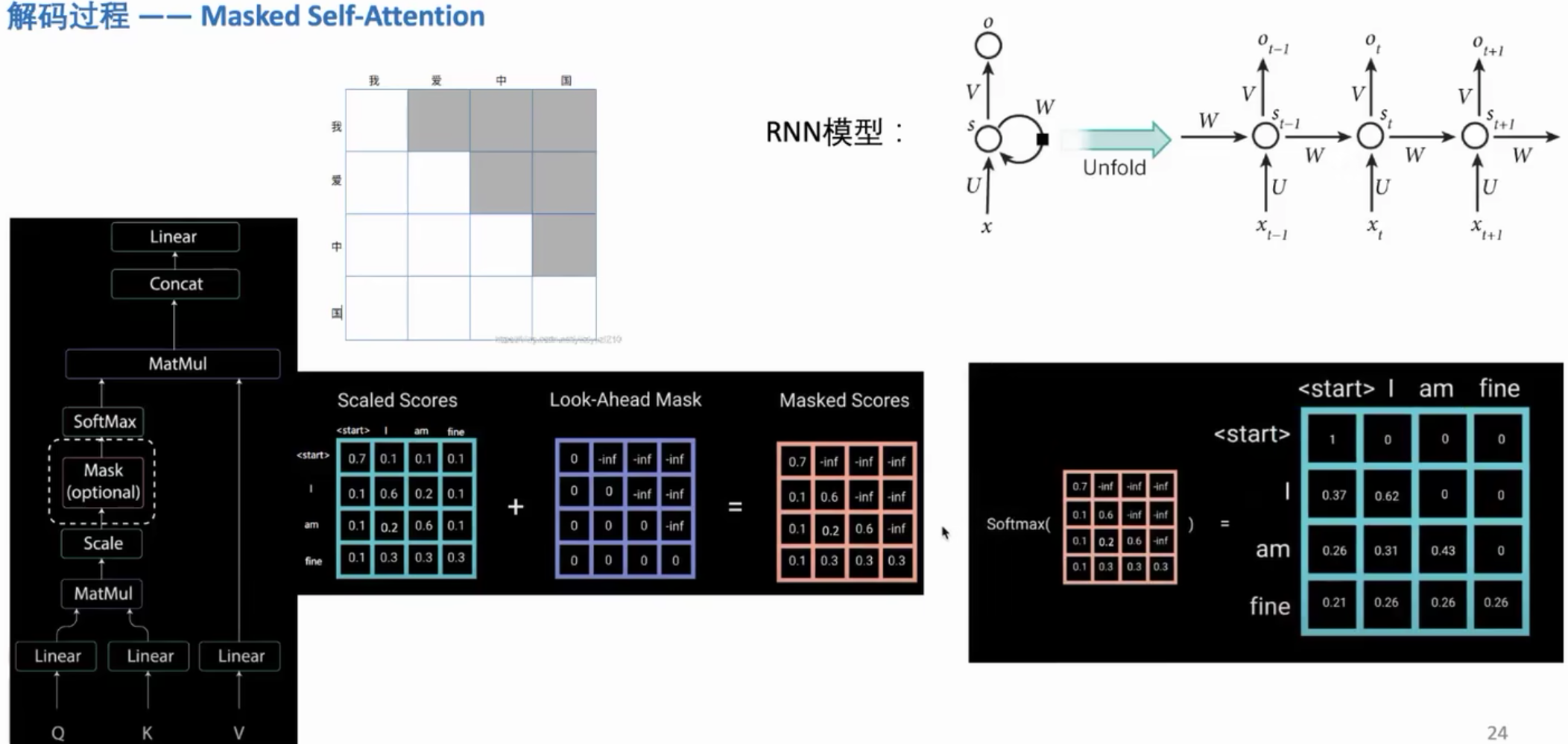


Add+Norm编码流程

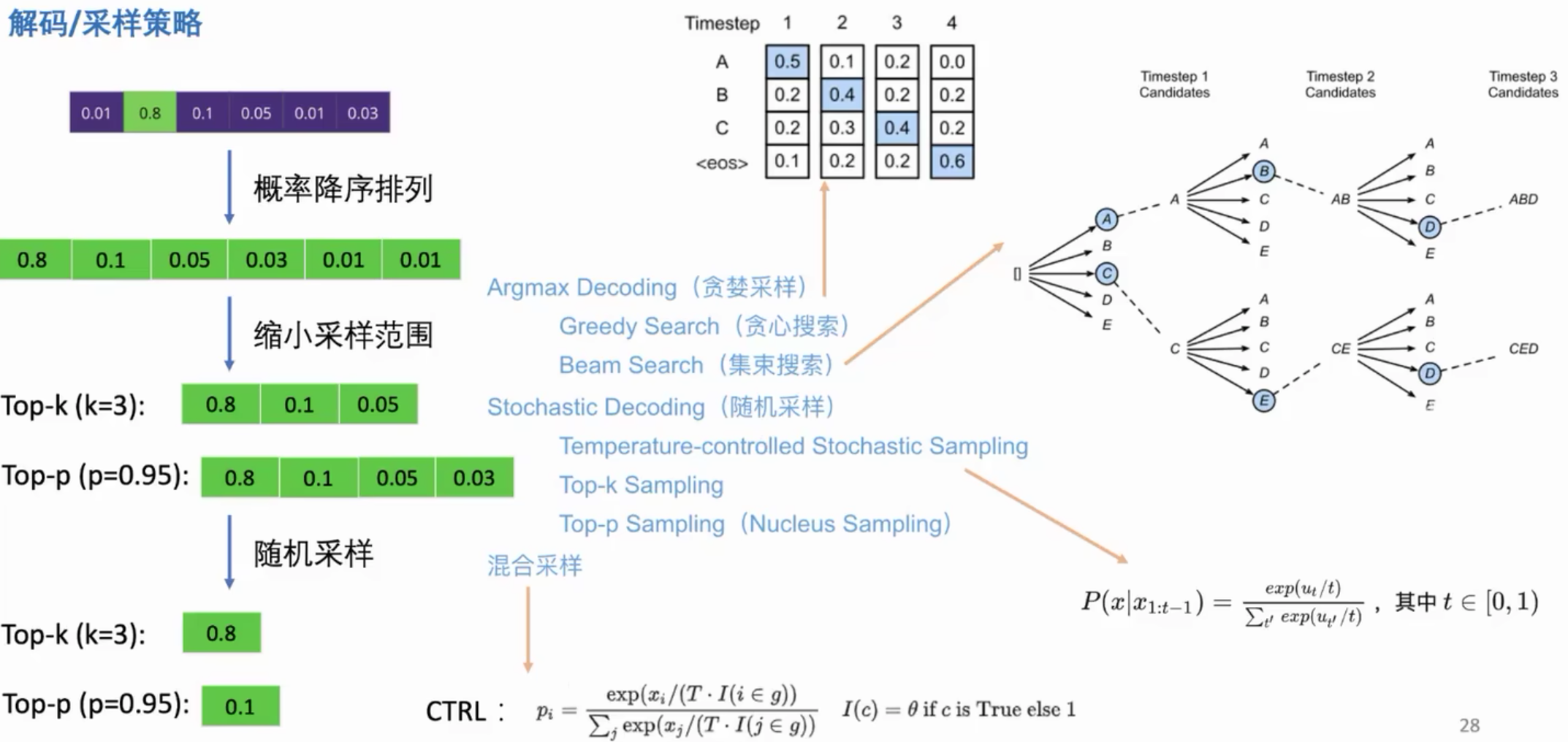


## Masked

Attention内部操作。屏蔽未来信息。



## 解码



# 训练

Decoder本质Autoregressive，自己做单字接龙。开始时需要给一个begin token，输出结束需要给出end token。输出字典包含{中文字符+end}，使用softmax分类。使用masked屏蔽未来信息，符合输出时“单字接龙”情形。

Cross Attention的k, v来自encoder，而q来自decoder的输入。

扔给Decoder input那部分是正确答案GT。此种训练方式叫Teacher Forcing。

## 数据示例

sentences = [  
 *# enc\_input dec\_input dec\_output* [**'ich mochte ein bier P'**, **'S i want a beer .'**, **'i want a beer . E'**],  
 [**'ich mochte ein cola P'**, **'S i want a coke .'**, **'i want a coke . E'**]  
]  
*# Padding Should be Zero*src\_vocab = {**'P'**: 0, **'ich'**: 1, **'mochte'**: 2, **'ein'**: 3, **'bier'**: 4, **'cola'**: 5}  
src\_vocab\_size = len(src\_vocab)  
  
tgt\_vocab = {**'P'**: 0, **'i'**: 1, **'want'**: 2, **'a'**: 3, **'beer'**: 4, **'coke'**: 5, **'S'**: 6, **'E'**: 7, **'.'**: 8}  
idx2word = {i: w for i, w in enumerate(tgt\_vocab)}  
tgt\_vocab\_size = len(tgt\_vocab)  
src\_len = 5 *# enc\_input max sequence length*tgt\_len = 6 *# dec\_input(=dec\_output) max sequence length*

def make\_data(sentences):  
 enc\_inputs, dec\_inputs, dec\_outputs = [], [], []  
 for i in range(len(sentences)):  
 enc\_input = [[src\_vocab[n] for n in sentences[i][0].split()]] *# [[1, 2, 3, 4, 0], [1, 2, 3, 5, 0]]* dec\_input = [[tgt\_vocab[n] for n in sentences[i][1].split()]] *# [[6, 1, 2, 3, 4, 8], [6, 1, 2, 3, 5, 8]]* dec\_output = [[tgt\_vocab[n] for n in sentences[i][2].split()]] *# [[1, 2, 3, 4, 8, 7], [1, 2, 3, 5, 8, 7]]* enc\_inputs.extend(enc\_input)  
 dec\_inputs.extend(dec\_input)  
 dec\_outputs.extend(dec\_output)  
 return torch.LongTensor(enc\_inputs), torch.LongTensor(dec\_inputs), torch.LongTensor(dec\_outputs)  
enc\_inputs, dec\_inputs, dec\_outputs = make\_data(sentences)  
  
class MyDataSet(Data.Dataset):  
 def \_\_init\_\_(self, enc\_inputs, dec\_inputs, dec\_outputs):  
 super(MyDataSet, self).\_\_init\_\_()  
 self.enc\_inputs = enc\_inputs  
 self.dec\_inputs = dec\_inputs  
 self.dec\_outputs = dec\_outputs  
 def \_\_len\_\_(self):  
 return self.enc\_inputs.shape[0]  
 def \_\_getitem\_\_(self, idx):  
 return self.enc\_inputs[idx], self.dec\_inputs[idx], self.dec\_outputs[idx]  
  
loader = Data.DataLoader(MyDataSet(enc\_inputs, dec\_inputs, dec\_outputs), 2, True)

## 训练示例

# 模型超参数  
model = Transformer().to(device)  
criterion = nn.CrossEntropyLoss(ignore\_index=0)  
optimizer = optim.SGD(model.parameters(), lr=1e-3, momentum=0.99)

for epoch in range(1000):  
 for enc\_inputs, dec\_inputs, dec\_outputs in loader:  
 **'''  
 enc\_inputs: [batch\_size, src\_len]  
 dec\_inputs: [batch\_size, tgt\_len]  
 dec\_outputs: [batch\_size, tgt\_len]  
 '''** enc\_inputs, dec\_inputs, dec\_outputs = enc\_inputs.to(device), dec\_inputs.to(device), dec\_outputs.to(device)  
 *# outputs: [batch\_size \* tgt\_len, tgt\_vocab\_size]* outputs, enc\_self\_attns, dec\_self\_attns, dec\_enc\_attns = model(enc\_inputs, dec\_inputs)  
 loss = criterion(outputs, **dec\_outputs.view(-1)**)  
 print(**'Epoch:'**, **'%04d'** % (epoch + 1), **'loss ='**, **'{:.6f}'**.format(loss))  
 optimizer.zero\_grad()  
 loss.backward()# 每个batch反向计算前先zero\_grad将梯度清零  
 optimizer.step()

如果不是每一个batch就清除掉原有的梯度，而是比如说两个batch再清除掉梯度，这是一种变相提高batch\_size的方法，对于计算机硬件不行，但是batch\_size可能需要设高的领域比较适合，比如目标检测模型的训练。

# 编码实现

*# Transformer Parameters*d\_model = 512 *# Embedding Size*d\_ff = 2048 *# FeedForward dimension*d\_k = d\_v = 64 *# dimension of K(=Q), V*n\_layers = 6 *# number of Encoder of Decoder Layer*n\_heads = 8 *# number of heads in Multi-Head Attention*

## 位置编码

class PositionalEncoding(nn.Module):  
 def \_\_init\_\_(self, d\_model, dropout=0.1, max\_len=5000):  
 super(PositionalEncoding, self).\_\_init\_\_()  
 self.dropout = nn.Dropout(p=dropout)  
 pe = torch.zeros(max\_len, d\_model)  
 position = torch.arange(0, max\_len, dtype=torch.float).unsqueeze(1)  
 div\_term = torch.exp(torch.arange(0, d\_model, 2).float() \* (-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).transpose(0, 1)  
 self.register\_buffer(**'pe'**, pe)  
 def forward(self, x):  
 *'''  
 x: [seq\_len, batch\_size, d\_model]  
 '''* x = x + self.pe[:x.size(0), :]  
 return self.dropout(x)

## Encoder

class Encoder(nn.Module):  
 def \_\_init\_\_(self):  
 super(Encoder, self).\_\_init\_\_()  
 self.src\_emb = nn.Embedding(src\_vocab\_size, d\_model)  
 self.pos\_emb = PositionalEncoding(d\_model)  
 self.layers = nn.ModuleList([EncoderLayer() for \_ in range(n\_layers)])  
 def forward(self, enc\_inputs):  
 *'''  
 enc\_inputs: [batch\_size, src\_len]  
 '''* enc\_outputs = self.src\_emb(enc\_inputs) *# [batch\_size, src\_len, d\_model]* enc\_outputs = self.pos\_emb(enc\_outputs.transpose(0, 1)).transpose(0, 1) *# [batch\_size, src\_len, d\_model]* enc\_self\_attn\_mask = get\_attn\_pad\_mask(enc\_inputs, enc\_inputs) *# [batch\_size, src\_len, src\_len]* enc\_self\_attns = []  
 for layer in self.layers:  
 *# enc\_outputs: [batch\_size, src\_len, d\_model], enc\_self\_attn: [batch\_size, n\_heads, src\_len, src\_len]* enc\_outputs, enc\_self\_attn = layer(enc\_outputs, enc\_self\_attn\_mask)  
 enc\_self\_attns.append(enc\_self\_attn)  
 return enc\_outputs, enc\_self\_attns

class EncoderLayer(nn.Module):  
 def \_\_init\_\_(self):  
 super(EncoderLayer, self).\_\_init\_\_()  
 self.enc\_self\_attn = MultiHeadAttention()  
 self.pos\_ffn = PoswiseFeedForwardNet()  
 def forward(self, enc\_inputs, enc\_self\_attn\_mask):  
 *'''  
 enc\_inputs: [batch\_size, src\_len, d\_model]  
 enc\_self\_attn\_mask: [batch\_size, src\_len, src\_len]  
 '''  
 # enc\_outputs: [batch\_size, src\_len, d\_model], attn: [batch\_size, n\_heads, src\_len, src\_len]* enc\_outputs, attn = self.enc\_self\_attn(enc\_inputs, enc\_inputs, enc\_inputs,  
 enc\_self\_attn\_mask) *# enc\_inputs to same Q,K,V* enc\_outputs = self.pos\_ffn(enc\_outputs) *# enc\_outputs: [batch\_size, src\_len, d\_model]* return enc\_outputs, attn

### MultiHeadAttention

class MultiHeadAttention(nn.Module):  
 def \_\_init\_\_(self):  
 super(MultiHeadAttention, self).\_\_init\_\_()  
 self.W\_Q = nn.Linear(d\_model, d\_k \* n\_heads, bias=False)  
 self.W\_K = nn.Linear(d\_model, d\_k \* n\_heads, bias=False)  
 self.W\_V = nn.Linear(d\_model, d\_v \* n\_heads, bias=False)  
 self.fc = nn.Linear(n\_heads \* d\_v, d\_model, bias=False)  
  
 def forward(self, input\_Q, input\_K, input\_V, attn\_mask):  
 *'''  
 input\_Q: [batch\_size, len\_q, d\_model]  
 input\_K: [batch\_size, len\_k, d\_model]  
 input\_V: [batch\_size, len\_v(=len\_k), d\_model]  
 attn\_mask: [batch\_size, seq\_len, seq\_len]  
 '''* residual, batch\_size = input\_Q, input\_Q.size(0)  
 *# (B, S, D) -proj-> (B, S, D\_new) -split-> (B, S, H, W) -trans-> (B, H, S, W)* **Q = self.W\_Q(input\_Q).view(batch\_size, -1, n\_heads, d\_k).transpose(1, 2)** *# Q: [batch\_size, n\_heads, len\_q, d\_k]* K = self.W\_K(input\_K).view(batch\_size, -1, n\_heads, d\_k).transpose(1, 2) *# K: [batch\_size, n\_heads, len\_k, d\_k]* V = self.W\_V(input\_V).view(batch\_size, -1, n\_heads, d\_v).transpose(1, 2) *# V: [batch\_size, n\_heads, len\_v(=len\_k), d\_v]* attn\_mask = attn\_mask.unsqueeze(1).repeat(1, n\_heads, 1, 1) *# attn\_mask : [batch\_size, n\_heads, seq\_len, seq\_len]  
 # context: [batch\_size, n\_heads, len\_q, d\_v], attn: [batch\_size, n\_heads, len\_q, len\_k]* context, attn = ScaledDotProductAttention()(Q, K, V, attn\_mask)  
 **context = context.transpose(1, 2).reshape(batch\_size, -1, n\_heads \* d\_v)** *# context: [batch\_size, len\_q, n\_heads \* d\_v]* **output = self.fc(context)** *# [batch\_size, len\_q, d\_model]* return nn.LayerNorm(d\_model).to(device)(output + residual), attn

class **ScaledDotProductAttention**(nn.Module):  
 def \_\_init\_\_(self):  
 super(ScaledDotProductAttention, self).\_\_init\_\_()  
 def forward(self, Q, K, V, attn\_mask):  
 *'''  
 Q: [batch\_size, n\_heads, len\_q, d\_k]  
 K: [batch\_size, n\_heads, len\_k, d\_k]  
 V: [batch\_size, n\_heads, len\_v(=len\_k), d\_v]  
 attn\_mask: [batch\_size, n\_heads, seq\_len, seq\_len]  
 '''* **scores = torch.matmul(Q, K.transpose(-1, -2)) / np.sqrt(d\_k)** *# scores : [batch\_size, n\_heads, len\_q, len\_k]* **scores.masked\_fill\_(attn\_mask, -1e9)** *# Fills elements of self tensor with value where mask is True.* attn = nn.Softmax(dim=-1)(scores)  
 context = torch.matmul(attn, V) *# [batch\_size, n\_heads, len\_q, d\_v]* return context, attn

### PoswiseFeedForwardNet

class PoswiseFeedForwardNet(nn.Module):  
 def \_\_init\_\_(self):  
 super(PoswiseFeedForwardNet, self).\_\_init\_\_()  
 self.fc = nn.Sequential(  
 nn.Linear(d\_model, d\_ff, bias=False),  
 nn.ReLU(),  
 nn.Linear(d\_ff, d\_model, bias=False)  
 )  
 def forward(self, inputs):  
 *'''  
 inputs: [batch\_size, seq\_len, d\_model]  
 '''* residual = inputs  
 output = self.fc(inputs)  
 return **nn.LayerNorm(d\_model).to(device)(output + residual)** *# [batch\_size, seq\_len, d\_model]*

## Decoder

class Decoder(nn.Module):  
 def \_\_init\_\_(self):  
 super(Decoder, self).\_\_init\_\_()  
 self.tgt\_emb = nn.Embedding(tgt\_vocab\_size, d\_model)  
 self.pos\_emb = PositionalEncoding(d\_model)  
 self.layers = nn.ModuleList([DecoderLayer() for \_ in range(n\_layers)])  
 def forward(self, dec\_inputs, enc\_inputs, enc\_outputs):  
 *'''  
 dec\_inputs: [batch\_size, tgt\_len]  
 enc\_intpus: [batch\_size, src\_len]  
 enc\_outputs: [batsh\_size, src\_len, d\_model]  
 '''* dec\_outputs = self.tgt\_emb(dec\_inputs) *# [batch\_size, tgt\_len, d\_model]* dec\_outputs = self.pos\_emb(dec\_outputs.transpose(0, 1)).transpose(0, 1).to(device) *# [batch\_size, tgt\_len, d\_model]* dec\_self\_attn\_pad\_mask = get\_attn\_pad\_mask(dec\_inputs, dec\_inputs).to(device) *# [batch\_size, tgt\_len, tgt\_len]* dec\_self\_attn\_subsequence\_mask = get\_attn\_subsequence\_mask(dec\_inputs).to(device) *# [batch\_size, tgt\_len, tgt\_len]* dec\_self\_attn\_mask = torch.gt((dec\_self\_attn\_pad\_mask + dec\_self\_attn\_subsequence\_mask), 0).to(device) *# [batch\_size, tgt\_len, tgt\_len]* dec\_enc\_attn\_mask = get\_attn\_pad\_mask(dec\_inputs, enc\_inputs) *# [batc\_size, tgt\_len, src\_len]* dec\_self\_attns, dec\_enc\_attns = [], []  
 for layer in self.layers:  
 *# dec\_outputs: [batch\_size, tgt\_len, d\_model], dec\_self\_attn: [batch\_size, n\_heads, tgt\_len, tgt\_len], dec\_enc\_attn: [batch\_size, h\_heads, tgt\_len, src\_len]* dec\_outputs, dec\_self\_attn, dec\_enc\_attn = layer(dec\_outputs, enc\_outputs, dec\_self\_attn\_mask, dec\_enc\_attn\_mask)  
 dec\_self\_attns.append(dec\_self\_attn)  
 dec\_enc\_attns.append(dec\_enc\_attn)  
 return dec\_outputs, dec\_self\_attns, dec\_enc\_attns

class DecoderLayer(nn.Module):  
 def \_\_init\_\_(self):  
 super(DecoderLayer, self).\_\_init\_\_()  
 self.dec\_self\_attn = MultiHeadAttention()  
 self.dec\_enc\_attn = MultiHeadAttention()  
 self.pos\_ffn = PoswiseFeedForwardNet()  
 def forward(self, dec\_inputs, enc\_outputs, dec\_self\_attn\_mask, dec\_enc\_attn\_mask):  
 *'''  
 dec\_inputs: [batch\_size, tgt\_len, d\_model]  
 enc\_outputs: [batch\_size, src\_len, d\_model]  
 dec\_self\_attn\_mask: [batch\_size, tgt\_len, tgt\_len]  
 dec\_enc\_attn\_mask: [batch\_size, tgt\_len, src\_len]  
 '''  
 # dec\_outputs: [batch\_size, tgt\_len, d\_model], dec\_self\_attn: [batch\_size, n\_heads, tgt\_len, tgt\_len]* dec\_outputs, dec\_self\_attn = self.dec\_self\_attn(dec\_inputs, dec\_inputs, dec\_inputs, dec\_self\_attn\_mask)  
 *# dec\_outputs: [batch\_size, tgt\_len, d\_model], dec\_enc\_attn: [batch\_size, h\_heads, tgt\_len, src\_len]* dec\_outputs, dec\_enc\_attn = self.dec\_enc\_attn(dec\_outputs, enc\_outputs, enc\_outputs, dec\_enc\_attn\_mask)  
 dec\_outputs = self.pos\_ffn(dec\_outputs) *# [batch\_size, tgt\_len, d\_model]* return dec\_outputs, dec\_self\_attn, dec\_enc\_attn

## Padding

def get\_attn\_pad\_mask(seq\_q, seq\_k):  
 *'''  
 seq\_q: [batch\_size, seq\_len]  
 seq\_k: [batch\_size, seq\_len]  
 seq\_len could be src\_len or it could be tgt\_len  
 seq\_len in seq\_q and seq\_len in seq\_k maybe not equal  
 '''* batch\_size, len\_q = seq\_q.size()  
 batch\_size, len\_k = seq\_k.size()  
 *# eq(zero) is PAD token* pad\_attn\_mask = seq\_k.data.eq(0).unsqueeze(1) *# [batch\_size, 1, len\_k], False is masked* return pad\_attn\_mask.expand(batch\_size, len\_q, len\_k) *# [batch\_size, len\_q, len\_k]*

def get\_attn\_subsequence\_mask(seq):  
 *'''  
 seq: [batch\_size, tgt\_len]  
 '''* attn\_shape = [seq.size(0), seq.size(1), seq.size(1)]  
 subsequence\_mask = np.triu(np.ones(attn\_shape), k=1) *# Upper triangular matrix* subsequence\_mask = torch.from\_numpy(subsequence\_mask).byte()  
 return subsequence\_mask *# [batch\_size, tgt\_len, tgt\_len]*

## Transfomer

class Transformer(nn.Module):  
 def \_\_init\_\_(self):  
 super(Transformer, self).\_\_init\_\_()  
 self.encoder = Encoder().to(device)  
 self.decoder = Decoder().to(device)  
 self.projection = nn.Linear(d\_model, tgt\_vocab\_size, bias=False).to(device)  
 def forward(self, enc\_inputs, dec\_inputs):  
 *'''  
 enc\_inputs: [batch\_size, src\_len]  
 dec\_inputs: [batch\_size, tgt\_len]  
 '''  
 # tensor to store decoder outputs  
 # outputs = torch.zeros(batch\_size, tgt\_len, tgt\_vocab\_size).to(self.device)  
  
 # enc\_outputs: [batch\_size, src\_len, d\_model], enc\_self\_attns: [n\_layers, batch\_size, n\_heads, src\_len, src\_len]* enc\_outputs, enc\_self\_attns = self.encoder(enc\_inputs)  
 *# dec\_outpus: [batch\_size, tgt\_len, d\_model], dec\_self\_attns: [n\_layers, batch\_size, n\_heads, tgt\_len, tgt\_len], dec\_enc\_attn: [n\_layers, batch\_size, tgt\_len, src\_len]* dec\_outputs, dec\_self\_attns, dec\_enc\_attns = self.decoder(dec\_inputs, enc\_inputs, enc\_outputs)  
 dec\_logits = self.projection(dec\_outputs) *# dec\_logits: [batch\_size, tgt\_len, tgt\_vocab\_size]* return dec\_logits.view(-1, dec\_logits.size(-1)), enc\_self\_attns, dec\_self\_attns, dec\_enc\_attns

## 解码

def greedy\_decoder(model, enc\_input, start\_symbol):  
 *"""  
 For simplicity, a Greedy Decoder is Beam search when K=1. This is necessary for inference as we don't know the target sequence input. Therefore we try to generate the target input word by word, then feed it into the transformer.  
 """* enc\_outputs, enc\_self\_attns = model.encoder(enc\_input)  
 dec\_input = torch.zeros(1, 0).type\_as(enc\_input.data)  
 terminal = False  
 next\_symbol = start\_symbol  
 while not terminal:  
 dec\_input = torch.cat([dec\_input.detach(), torch.tensor([[next\_symbol]], dtype=enc\_input.dtype).to(device)], -1)# detach不参与梯度计算，cat拼接上一轮得到的字符  
 dec\_outputs, \_, \_ = model.decoder(dec\_input, enc\_input, enc\_outputs)  
 projected = model.projection(dec\_outputs)  
 prob = projected.squeeze(0).max(dim=-1, keepdim=False)[1]# squeeze压缩维度，[1]指ind  
 next\_word = prob.data[-1]  
 next\_symbol = next\_word  
 if next\_symbol == tgt\_vocab[**"."**]:  
 terminal = True  
 print(next\_word)  
 return dec\_input

enc\_inputs, \_, \_ = next(iter(loader))  
enc\_inputs = enc\_inputs.to(device)  
for i in range(len(enc\_inputs)):  
 greedy\_dec\_input = greedy\_decoder(model, enc\_inputs[i].view(1, -1), start\_symbol=tgt\_vocab[**"S"**])  
 predict, \_, \_, \_ = model(enc\_inputs[i].view(1, -1), greedy\_dec\_input)  
 predict = predict.data.max(1, keepdim=True)[1]  
 print(enc\_inputs[i], **'->'**, [idx2word[n.item()] for n in predict.squeeze()])