一、模型搭建及部分原理

0. 导入库:

```
import copy
import math
import numpy as np
import torch
from torch import nn
import torch.nn.functional as F
```

1 Encoder

首先搭建出整个encoder的具体框架

```
class Encoder(nn.Module):
def __init__(self, layer, N):
    super(Encoder, self).__init__()
    self.layers = clones(layer, N)
    self.norm = LayerNorm(layer.size)

def forward(self, x, mask):
    for layer in self.layers:
        x = layer(x, mask)
    return self.norm(x)
```

1.1 EncoderLayer

每个EncoderLayer层的搭建，由一个self-attention + sublayer + feed forward组成

```
class EncoderLayer(nn.Module):
"""
    Encoder is made up of self-attn and feed forward (defined below)
"""

def __init__(self, size, self_attention, feed_forward, dropout):
    super(EncoderLayer, self).__init__()
    self.self_attention = self_attention
    self.feed_forward = feed_forward
    self.sublayer = clones(SublayerConnection(size, dropout), 2)
    self.size = size

def forward(self, x, mask):
    x = self.sublayer[0](x, lambda x_: self.self_attention(x_, x_, x_, mask))
    return self.sublayer[1](x, self.feed_forward)
```

1.2 SublayerConnection and LayerNorm

使用残差网络连接attention和forward层

```
class SublayerConnection(nn.Module):
"""
A residual connection followed by a layer norm.
Note for code simplicity the norm is first as opposed to last.
"""

def __init__(self, size, dropout):
    super(SublayerConnection, self).__init__()
    self.norm = LayerNorm(size)
    self.dropout = nn.Dropout(dropout)

def forward(self, x, sublayer):
    """
        Apply residual connection to any sublayer with the same size.
    """
    return x + self.dropout(sublayer(self.norm(x)))
```

批归一化的方法对数据进行归一化，便于计算以及降低算力要求

```
class LayerNorm(nn.Module):
"""
    搭建了一个批归一化模型，降低算力要求
"""

def __init__(self, features, eps=1e-6):
    super(LayerNorm, self).__init__()
    self.a_2 = nn.Parameter(torch.ones(features))
    self.b_2 = nn.Parameter(torch.zeros(features))
    self.eps = eps

def forward(self, x):
    mean = x.mean(-1, keepdim=True)
    std = x.std(-1, keepdim=True)
    """
        对输入x进行去中心化，同时a2，b2两个参数可随模型训练调整
    """
    return self.a_2 * (x - mean) / (std + self.eps) + self.b_2
```

2 Decoder

和Encoder的区别在于，还有一个multi-head attention层，连接方式和encoder一样

```
class Decoder(nn.Module):
"""
    Generic N layer decoder with masking.
"""

def __init__(self, layer, N):
    super(Decoder, self).__init__()
    self.layers = clones(layer, N)
    self.norm = LayerNorm(layer.size)
```

```
def forward(self, x, memory, src_mask, tgt_mask):
    """
        这个mask就像是transformer小时候的默写，你得先默写出来再去看答案，如果默写错了，就要吃一竹竿长记性（loss反向传递，系数更新）
    """
    for layer in self.layers:
        x = layer(x, memory, src_mask, tgt_mask)
    return self.norm(x)
```

2.1 DecoderLayer

由self—attention multi-head attention 以及feed forward 三个层组成

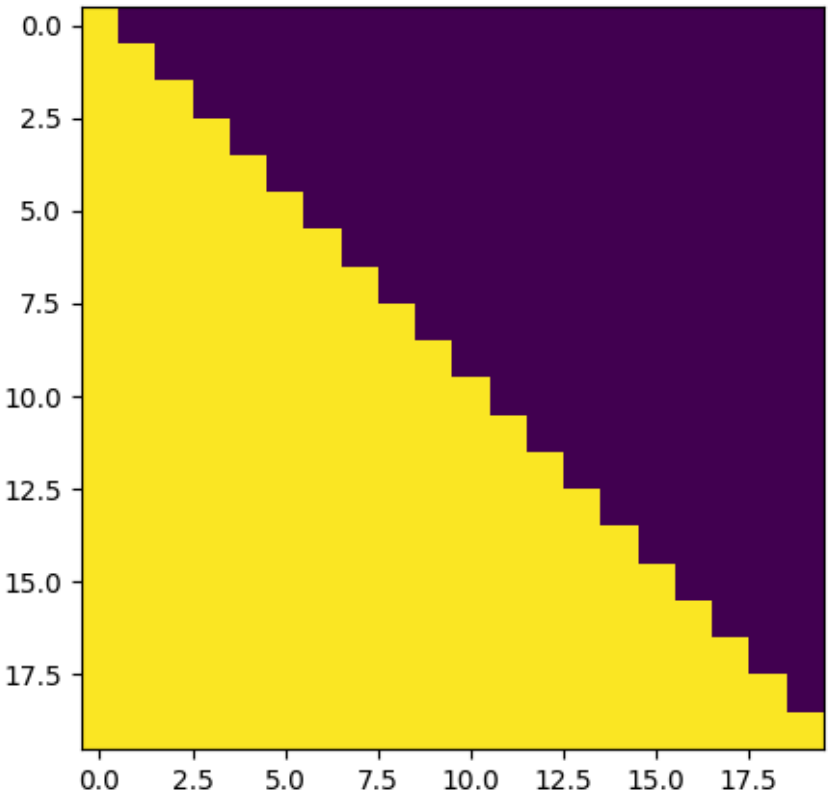
```
class DecoderLayer(nn.Module):
"""
    Decoder is made of self-attn, src-attn, and feed forward
"""

def __init__(self, size, self_attention, src_attention, feed_forward, dropout):
    super(DecoderLayer, self).__init__()
    self.size = size
    self.self_attention = self_attention
    self.src_attention = src_attention
    self.feed_forward = feed_forward
    self.sublayer = clones(SublayerConnection(size, dropout), 3)

def forward(self, x, memory, src_mask, tgt_mask):
    x = self.sublayer[0](x, lambda x_: self.self_attention(x_, x_, x_, tgt_mask))
    x = self.sublayer[1](x, lambda x_: self.src_attention(x_, memory, memory,
src_mask))
    return self.sublayer[2](x, self.feed_forward)
```

2.2 mask

注意掩码下面显示了每个tgt单词（行）被允许查看的位置（列）。在训练过程中，单词会被屏蔽，以便关注将来的单词。

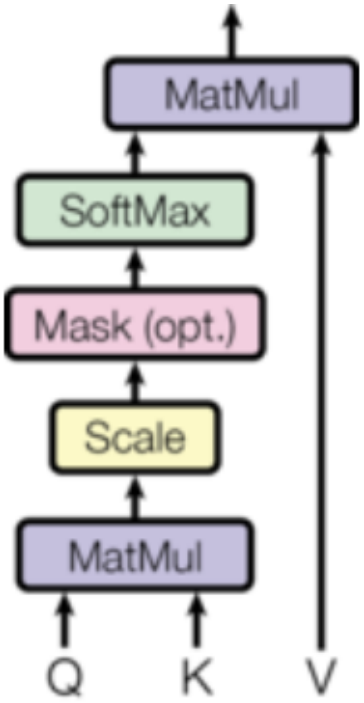


transformer修改了解码器堆栈中的自注意子层，以防止位置关注后续位置。这种掩蔽，再加上输出嵌入偏移一个位置的事实，确保了位置i的预测只依赖于小于i位置的已知输出。

```
def subsequent_mask(size):
"""
    Mask out subsequent positions.
"""
attention_shape = (1, size, size)
sub_mask = np.triu(np.ones(attention_shape), k=1).astype('int8')
return torch.from_numpy(sub_mask) == 0
```

3 attention

attention函数可以描述为将query和一组key-value对映射到输出，其中query、key、value和输出都是向量。输出是作为值的加权和计算的。



计算公式如下：

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

两个最常用的注意力函数是additive attention和dot-product (multiplicative) attention。除了乘上一个比例因子$\sqrt{d_k}$，dot-product attention与transformer算法相同。additive attention使用一个具有单个隐藏层的前馈网络来计算兼容性函数。虽然两者在理论复杂性上相似，但在实践中，dot-product attention更快，更节省空间，因为它可以使用高度优化的矩阵乘法代码来实现。

虽然对于较小的d_k值，这两种机制的表现相似；但是对于d_k值较大且dot-product attention不缩放d_k的情况下，additive attention优于dot-product attention

作者怀疑，对于较大的d_k值，点积的幅度会变得很大，将softmax函数推向其梯度极小的区域（为了说明为什么点积会变得很大，假设 q 和k的分量是均值为0、方差为1的独立随机变量（即二者满足正态分布）。那么它们的点积$q \cdot k = \sum_{i=1}^{d_k} q_i k_i$的均值为0，方差为$d_k$）。为了抵消这种效应，通过$d_k$对点积进行缩放。

代码如下：

```
def attention(query, key, value, mask=None, dropout=None):
    """
    :param query:
    :param key:
    :param value: vocab vector
    :return: value_(i.e z) and A
    """
    d_k = key.size(-1)  # d_k表示key矩阵的维度
    scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)  # key是一个二维矩阵
    if mask is not None:
        scores = scores.masked_fill(mask == 0, 1e-9)
    p_attn = F.softmax(scores, dim=-1)  # 按列进行softmax，即计算每个词在语句中出现的概率
    if dropout is not None:
```

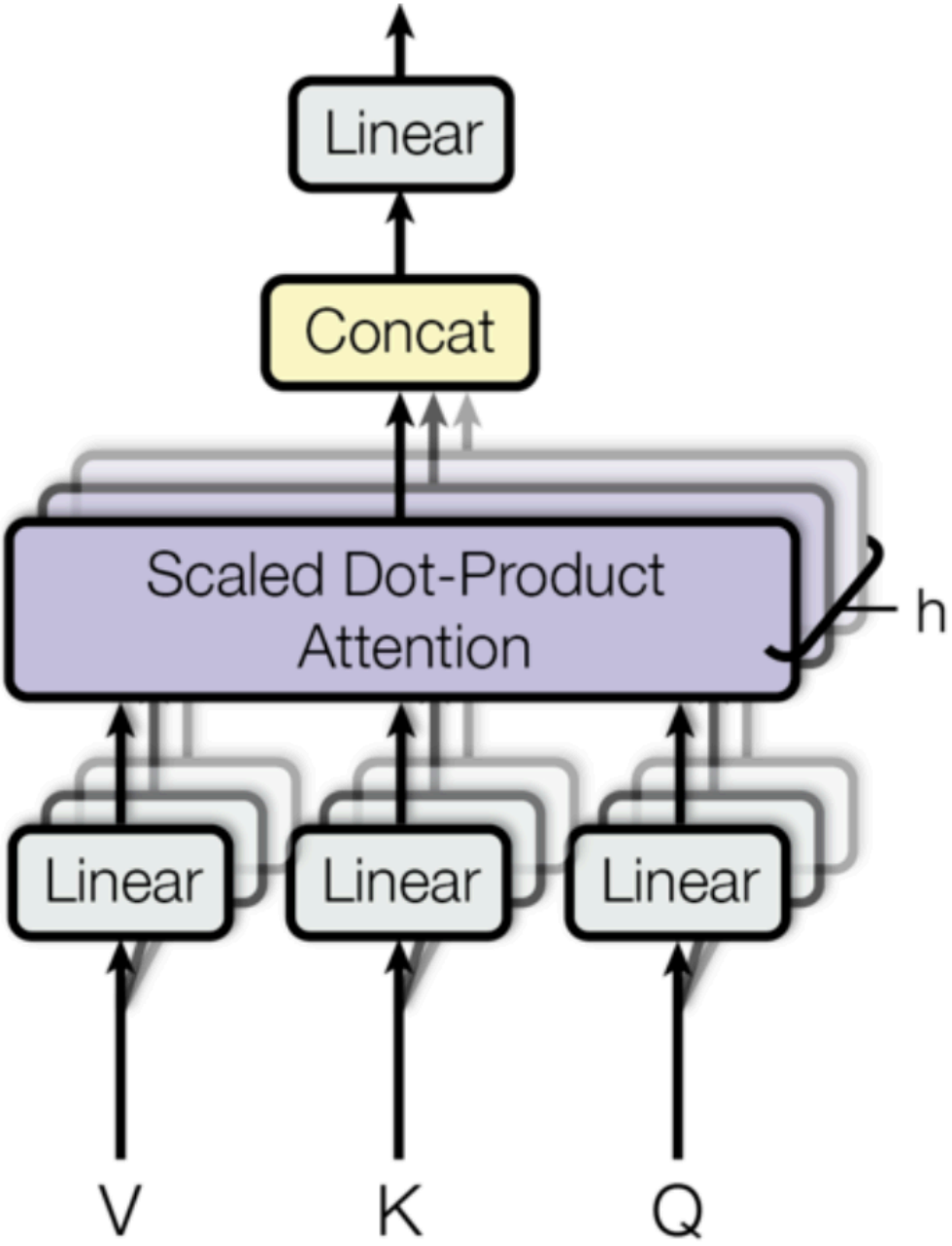
```
        p_attn = dropout(p_attn)
    return torch.matmul(p_attn, value), p_attn
```

3.1 multi-head attention

multi-head attention允许模型在不同位置共同注意来自不同表示子空间的信息。对于单一注意力头，平均会抑制这一点。

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$
$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

其中，映射矩阵$W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$，，$W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$，$W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$，and $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$（不同的head之间映射矩阵并不相同）。在这项工作中，transformer使用了$h = 8$个并行注意力层（即8个head）。对于每个注意力层，使用。由于每层$d_k = d_v = \frac{d_{\text{model}}}{h} = 64$.的维度降低，总的计算成本与具有完整维度的单头注意力相似。



代码实现如下：

```
class MultiHeadAttention(nn.Module):
```

```
def __init__(self, h, d_model, dropout=0.1):
    """
    :param h: the number of head
    :param d_model: the dimension of the model
    """
    super(MultiHeadAttention, self).__init__()
    assert d_model % h == 0
    self.d_k = d_model // h  # We assume d_v always equals d_k
    self.h = h
    self.linears = clones(nn.Linear(d_model, d_model), 4)
    self.attention = None
    self.dropout = nn.Dropout(dropout)

def forward(self, query, key, value, mask):
    if mask is not None:
        mask = mask.squeeze(1)  # Same mask applied to all h heads.
    nbatches = query.size(0)

    # 1) Do all the linear projections in batch from d_model => h x d_k
    query, key, value = [linears(x).view(nbatches, -1, self.h, self.d_k).transpose(1,
2) for linears, x in
                         zip(self.linears, (query, key, value))]
    # the shape of query is [nbatches,the num of head,1,dimension of max]

    # 2) Apply attention on all the projected vectors in batch.
    x, self.attention = attention(query, key, value, mask=mask, dropout=self.dropout)

    # 3) "Concat" using a view and apply a final linear.
    x = x.transpose(1, 2).contiguous().view(nbatches, -1, self.d_k * self.h)

    # Linear change on the final result
    return self.linears[-1](x)
```

Transformer以三种不同的方式使用多头注意：

1)在“编码器-解码器注意”层中，查询来自前一个解码器层，而记忆键和值来自编码器的输出。这允许解码器中的每个位置都参与输入序列中的所有位置。这模仿了序列到序列模型中典型的编码器-解码器注意机制。

2)编码器包含self-attention层。在self-attention层中，所有的key、value和query都来自同一个地方，在这种情况下，是编码器中前一层的输出。编码器中的每个位置都可以处理编码器前一层中的所有位置。

3)类似地，解码器中的self-attention层允许解码器中的每个位置注意到解码器中的所有位置直至并包括该位置。需要防止解码器中的向左信息流以保持自回归特性。通过mask掩码来设置对应值为$-\infty$实现softmax输入中对应非法连接的所有值。

4.Position-wise Feed-Forward Networks

除了注意力子层之外，transformer的编码器和解码器的每一层都包含一个全连接的前馈网络，这个网络分别且相同地应用于每个位置。这包括两个线性变换，中间有一个ReLU激活函数。

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

虽然线性变换在不同位置是相同的，但它们在不同的层之间使用不同的参数（类似卷积核大小为1的卷积）。其中，输入和输出的维度都是$d_{model} = 512$，中间层维度设置为$d_{ff} = 2048$。

```
class PositionwiseFeedForward(nn.Module):
    def __init__(self, d_model, d_ff, dropout=0.1):
        super(PositionwiseFeedForward, self).__init__()
        self.w_1 = nn.Linear(d_model, d_ff)
        self.w_2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        return self.w_2(self.dropout(F.relu(self.w_1(x))))
```

5.Embedding and softmax

与其他序列转换模型类似，transformer使用学习到的嵌入（embeddings）将输入的标记（tokens）和输出的标记转换为维度为d_{model}的向量。transformer还使用通常的学习到的线性变换和softmax函数将解码器的输出转换为预测下一个标记的概率。在transformer的模型中，transformer在两个嵌入层和预softmax线性变换之间共享相同的权重矩阵。在嵌入层中，transformer将这些权重乘以$\sqrt{d_{model}}$。

```
class Embeddings(nn.Module):
    def __init__(self, d_model, vocab):
        super(Embeddings, self).__init__()
        self.lut = nn.Embedding(vocab, d_model, padding_idx=1)
        self.d_model = d_model

    def forward(self, x):
        # Since dividing sqrt(d_model) in self-attention, transformer need to
multiply it back in the decoding phase
        return self.lut(x) * math.sqrt(self.d_model)
```

6. Positional Encoding

由于transformer的模型不包含递归（recurrence）和卷积（convolution），为了使模型能够利用序列的顺序，transformer必须注入一些关于序列中标记（tokens）的相对或绝对位置的信息。为此，transformer在编码器（encoder）和解码器（decoder）堆栈的底部的输入嵌入（input embeddings）中添加了“位置编码”。

transformer中采用相对位置编码，由于位置编码的维度与词向量维度一致，因此在transformer中直接采用相加的方法，原作者采用了以下公式计算位置编码：

$$PE_{(\mathrm{pos},2i)} = \sin\left(\frac{\mathrm{pos}}{10000^{2i/d_{\mathrm{model}}}}\right)$$

$$[PE_{(\mathrm{pos},2i+1)} = \cos\left(\frac{\mathrm{pos}}{10000^{2i/d_{\mathrm{model}}}}\right)$$

其中，pos 是位置，i 是维度。也就是说，位置编码的每个维度都对应一个正弦波。波长（wavelengths）从 2π 到10000·2π 形成几何级数。作者选择这个函数是因为作者假设它将允许模型轻松地通过相对位置学习注意力（attention），因为对于任何固定的偏移量k，PE_{pos+k}都可以表达为PE_{pos}的线性函数（采用正余弦函数的和角公式即可推导得到）。

$$(PE_{(pos+k,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\mathrm{model}}}}\right)\cos\left(\frac{k}{10000^{2i/d_{\mathrm{model}}}}\right) + \cos\left(\frac{pos}{10000^{2i/d_{\mathrm{model}}}}\right)\sin\left(\frac{k}{10000^{2i/d_{\mathrm{model}}}}\right)$$

$$(PE_{(pos+k,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)\cos\left(\frac{k}{10000^{2i/d_{\text{model}}}}\right) - \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)\sin\left(\frac{k}{10000^{2i/d_{\text{model}}}}\right)$$

同时，在Positional Encoding中，transformer采用dropout函数，$p_{dropout} = 1$。

```
class PositionalEncoding(nn.Module):
    """
        max_len decide the size of word matrix
    """

    def __init__(self, d_model, dropout, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(dropout)

        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1)  # 矩阵转置
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000) /
d_model))

        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe.unsqueeze(0)

        self.register_buffer('pe', pe)  # Set the location code unchanged

    def forward(self, x):
        pe = self.pe[:, :x.size(1)].detach
        x = x + pe
        return self.dropout(x)
```

7. Full model

7.1 EncoderDecoder

设置一个函数类，便于后续搭建模型

```
class EncoderDecoder(nn.Module):
"""
    A standard Encoder-Decoder architecture. (Base for this and many other models.)
"""
    def __init__(self, encoder, decoder, src_embed, tgt_embed, generator):
        super(EncoderDecoder, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed  # 源语言嵌入层
        self.tgt_embed = tgt_embed  # 目标语言嵌入层
        # 俩个语言嵌入层必定通过算法融合得到一个新的”潜空间“
        self.generator = generator

    def forward(self, src, src_mask, tgt, tgt_mask):
        """
            Take in and process masked src and target sequences.
            前向传播训练过程
        """
        return self.decode(self.encode(src, src_mask), src_mask, tgt, tgt_mask)
```

```
    def encode(self, src, src_mask):
        return self.encoder(self.src_embed(src), src_mask)

    def decode(self, memory, src_mask, tgt, tgt_mask):
        return self.decoder(self.tgt_embed(tgt), memory, src_mask, tgt_mask)
```

7.2 Generator

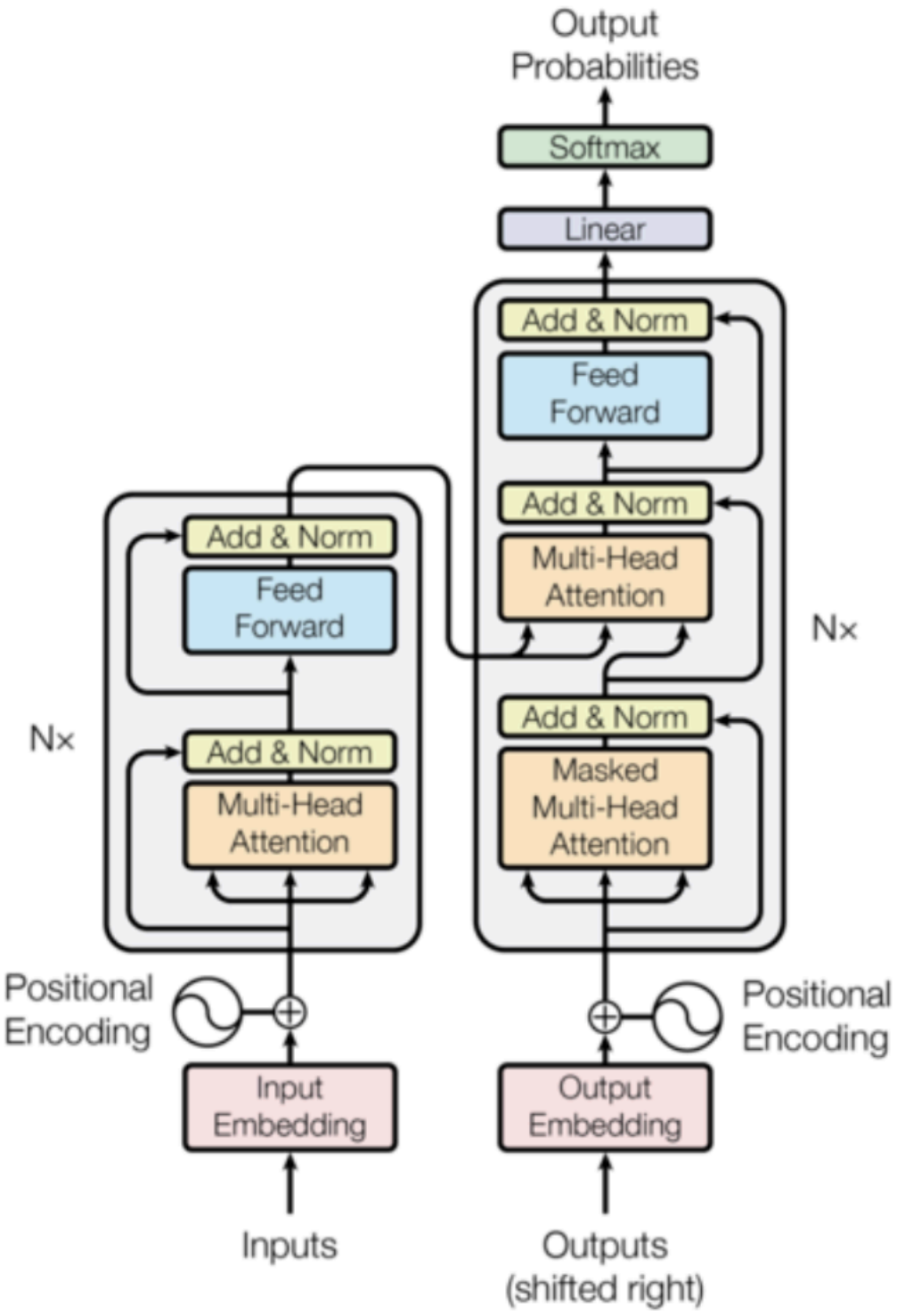
对于generator类设置了线性层和归一化层，保证最终输出结果与输入词向量的向量维度一致

```
class Generator(nn.Module):
    def __init__(self, d_model, vocab):
        super(Generator, self).__init__()
        self.linear = nn.Linear(d_model, vocab)

    def forward(self, x):
        return F.softmax(self.linear(x), dim=1)
```

7.3 full_model

这里定义了一个函数，它接受超参数并产生一个完整的模型。



代码如下所示:

```
def transformer(src_vocab, tgt_vocab, N=6, d_model=512, d_ff=2048, h=8, dropout=0.1):
    c = copy.deepcopy
    multi_head_attention = MultiHeadAttention(h, d_model)
    FFD = PositionwiseFeedForward(d_model, d_ff)
    PE = PositionalEncoding(d_model, dropout)

    model = EncoderDecoder(
        Encoder(
            EncoderLayer(size=d_model, self_attention=c(multi_head_attention),
feed_forward=c(FFD), dropout=dropout),
            N),
        Decoder(
```

```
            DecoderLayer(size=d_model, self_attention=c(multi_head_attention),
src_attention=c(multi_head_attention),
                         feed_forward=FFD, dropout=dropout), N),
        nn.Sequential(Embeddings(d_model, src_vocab), c(PE)),
        nn.Sequential(Embeddings(d_model, tgt_vocab), c(PE)),
        Generator(d_model, tgt_vocab))

    # This was important from their code.
    # Initialize parameters with Glorot / fan_avg.
    for p in model.parameters():
        # 过滤偏差层
        if p.dim() > 1:
            nn.init.xavier_normal_(p)

    return model
```

二、 Training

1. Batches and Masking

```
class Batch:
    "Object for holding a batch of data with mask during training."
    def __init__(self, src, trg=None, pad=0):
        self.src = src
        self.src_mask = (src != pad).unsqueeze(-2)
        if trg is not None:
            self.trg = trg[:, :-1]
            self.trg_y = trg[:, 1:]
            self.trg_mask = \
                self.make_std_mask(self.trg, pad)
            self.ntokens = (self.trg_y != pad).data.sum()

    @staticmethod
    def make_std_mask(tgt, pad):
        "Create a mask to hide padding and future words."
        tgt_mask = (tgt != pad).unsqueeze(-2)
        tgt_mask = tgt_mask & Variable(
            subsequent_mask(tgt.size(-1)).type_as(tgt_mask.data))
        return tgt_mask
```

2. Training Loop

```
def run_epoch(data_iter, model, loss_compute):
    "Standard Training and Logging Function"
    start = time.time()
    total_tokens = 0
    total_loss = 0
    tokens = 0
    for i, batch in enumerate(data_iter):
        out = model.forward(batch.src, batch.trg,
                            batch.src_mask, batch.trg_mask)
        loss = loss_compute(out, batch.trg_y, batch.ntokens)
        total_loss += loss
```

```
        total_tokens += batch.ntokens
        tokens += batch.ntokens
        if i % 50 == 1:
            elapsed = time.time() - start
            print("Epoch Step: %d Loss: %f Tokens per Sec: %f" %
                    (i, loss / batch.ntokens, tokens / elapsed))
            start = time.time()
            tokens = 0
    return total_loss / total_tokens
```

3. Training Data and Batching

WMT 2014 English-German dataset

WMT 2014 English-French dataset

```
global max_src_in_batch, max_tgt_in_batch
def batch_size_fn(new, count, sofar):
    "Keep augmenting batch and calculate total number of tokens + padding."
    global max_src_in_batch, max_tgt_in_batch
    if count == 1:
        max_src_in_batch = 0
        max_tgt_in_batch = 0
    max_src_in_batch = max(max_src_in_batch,  len(new.src))
    max_tgt_in_batch = max(max_tgt_in_batch,  len(new.trg) + 2)
    src_elements = count * max_src_in_batch
    tgt_elements = count * max_tgt_in_batch
    return max(src_elements, tgt_elements)
```

4. Optimizer

使用了Adam优化器，其中 $\beta_1 = 0.9$，$\beta_2 = 0.98$，$\epsilon = 10^{-9}$。transformer在训练过程中根据以下公式调整学习率：$\text{lrate} = d_{\text{model}}^{-0.5} \cdot \min\left(\text{step\_num}^{-0.5}, \text{step\_num} \cdot \text{warmup\_steps}^{-1.5}\right)$。这相当于在前 $warmup_steps$训练步中线性增加学习率，之后则按照步数的倒数平方根比例递减。使用了 $warmup\_steps = 4000$。

```
class NoamOpt:
    "Optim wrapper that implements rate."
    def __init__(self, model_size, factor, warmup, optimizer):
        self.optimizer = optimizer
        self._step = 0
        self.warmup = warmup
        self.factor = factor
        self.model_size = model_size
        self._rate = 0

    def step(self):
        "Update parameters and rate"
        self._step += 1
        rate = self.rate()
        for p in self.optimizer.param_groups:
            p['lr'] = rate
        self._rate = rate
        self.optimizer.step()

    def rate(self, step = None):
```

```
        "Implement `lrate` above"
        if step is None:
            step = self._step
        return self.factor * \
            (self.model_size ** (-0.5) *
            min(step ** (-0.5), step * self.warmup ** (-1.5)))

def get_std_opt(model):
    return NoamOpt(model.src_embed[0].d_model, 2, 4000,
            torch.optim.Adam(model.parameters(), lr=0, betas=(0.9, 0.98), eps=1e-9))
```

5. Regularization

5.1Label Smoothing

在训练过程中，transformer采用了标签平滑技术，其值为 $\epsilon_{ls} = 0.1$。这会降低困惑度，因为模型学会了更加不确定，但提高了准确率和BLEU分数。

通过KL散度损失来实现标签平滑。transformer没有使用一个独热编码的目标分布，而是创建了一个分布，该分布对正确单词有信心，并将其余的平滑质量分布在整个词汇表中。

```
class LabelSmoothing(nn.Module):
    "Implement label smoothing."
    def __init__(self, size, padding_idx, smoothing=0.0):
        super(LabelSmoothing, self).__init__()
        self.criterion = nn.KLDivLoss(size_average=False)
        self.padding_idx = padding_idx
        self.confidence = 1.0 - smoothing
        self.smoothing = smoothing
        self.size = size
        self.true_dist = None

    def forward(self, x, target):
        assert x.size(1) == self.size
        true_dist = x.data.clone()
        true_dist.fill_(self.smoothing / (self.size - 2))
        true_dist.scatter_(1, target.data.unsqueeze(1), self.confidence)
        true_dist[:, self.padding_idx] = 0
        mask = torch.nonzero(target.data == self.padding_idx)
        if mask.dim() > 0:
            true_dist.index_fill_(0, mask.squeeze(), 0.0)
        self.true_dist = true_dist
        return self.criterion(x, Variable(true_dist, requires_grad=False))
```

三、Example

1. Synthetic Data

```
def data_gen(V, batch, nbatches):
    "Generate random data for a src-tgt copy task."
    for i in range(nbatches):
        data = torch.from_numpy(np.random.randint(1, V, size=(batch, 10)))
        data[:, 0] = 1
        src = Variable(data, requires_grad=False)
        tgt = Variable(data, requires_grad=False)
        yield Batch(src, tgt, 0)
```

2.SimpleLossCompute

```
class SimpleLossCompute:
    "A simple loss compute and train function."
    def __init__(self, generator, criterion, opt=None):
        self.generator = generator
        self.criterion = criterion
        self.opt = opt

    def __call__(self, x, y, norm):
        x = self.generator(x)
        loss = self.criterion(x.contiguous().view(-1, x.size(-1)),
                              y.contiguous().view(-1)) / norm
        loss.backward()
        if self.opt is not None:
            self.opt.step()
            self.opt.optimizer.zero_grad()
        return loss.data[0] * norm
```

3. Greedy Decoding

```
V = 11
criterion = LabelSmoothing(size=V, padding_idx=0, smoothing=0.0)
model = make_model(V, V, N=2)
model_opt = NoamOpt(model.src_embed[0].d_model, 1, 400,
        torch.optim.Adam(model.parameters(), lr=0, betas=(0.9, 0.98), eps=1e-9))

for epoch in range(10):
    model.train()
    run_epoch(data_gen(V, 30, 20), model,
              SimpleLossCompute(model.generator, criterion, model_opt))
    model.eval()
    print(run_epoch(data_gen(V, 30, 5), model,
                    SimpleLossCompute(model.generator, criterion, None)))
```

```
def greedy_decode(model, src, src_mask, max_len, start_symbol):
    memory = model.encode(src, src_mask)
    ys = torch.ones(1, 1).fill_(start_symbol).type_as(src.data)
    for i in range(max_len-1):
        out = model.decode(memory, src_mask,
                           Variable(ys),
```

```
                            Variable(subsequent_mask(ys.size(1))
                                     .type_as(src.data)))
        prob = model.generator(out[:, -1])
        _, next_word = torch.max(prob, dim = 1)
        next_word = next_word.data[0]
        ys = torch.cat([ys,
                        torch.ones(1, 1).type_as(src.data).fill_(next_word)], dim=1)
    return ys

model.eval()
src = Variable(torch.LongTensor([[1,2,3,4,5,6,7,8,9,10]]) )
src_mask = Variable(torch.ones(1, 1, 10) )
print(greedy_decode(model, src, src_mask, max_len=10, start_symbol=1))
```