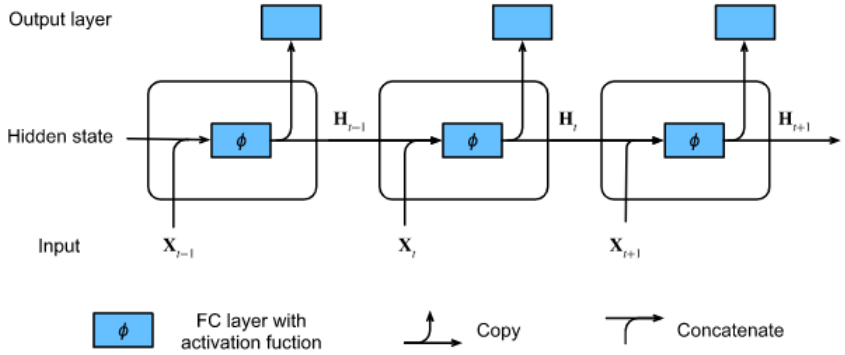
CS231n Assignment #3

- RNN:

RNN 多用于对时序模型建模，其通过对上一个时间步的隐藏状态，当前时间步的输入进行计算学习，从而实现对当前时间步输出的预测。其模型如下图所示：



其某一时间步计算公式如下：

$$h_t = f_W(h_{t-1}, x_t)$$

$$\downarrow$$

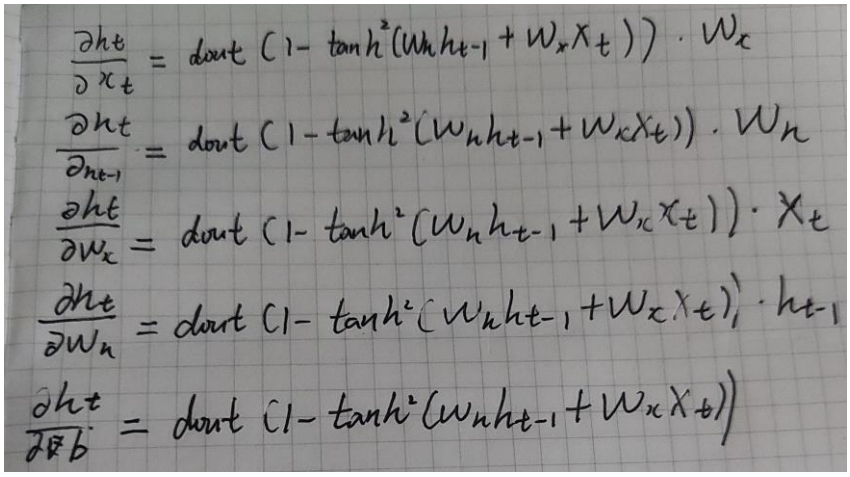
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

由此可得单时间步正向传播代码如下：

```
next_h = np.tanh(x.dot(Wx) + prev_h.dot(Wh) + b)
cache = (x, prev_h, Wx, Wh, b, next_h)
```

反向传播推导过程如下：



相应代码如下：

```
x, prev_h, Wx, Wh, b, tanh = cache
dtanh = 1 - tanh ** 2    # [NxH]
dnext_tanh = dnext_h * dtanh    # [NxH]
dx = dnext_tanh.dot(Wx.T)    # [NxD]
dprev_h = dnext_tanh.dot(Wh.T)    # [NxH]
dWx = (x.T).dot(dnext_tanh)    # [DxH]
dWh = (prev_h.T).dot(dnext_tanh)    # [DxH]
db = dnext_tanh.sum(axis=0)    # [1xH]
```

整个 rnn 层可视为按时间步先后顺序排列的 MLP，因此完整的 rnn 的正向传播和反向传播可类比 MLP 的思路实现（稍后在 LSTM 中展示）。

总结：

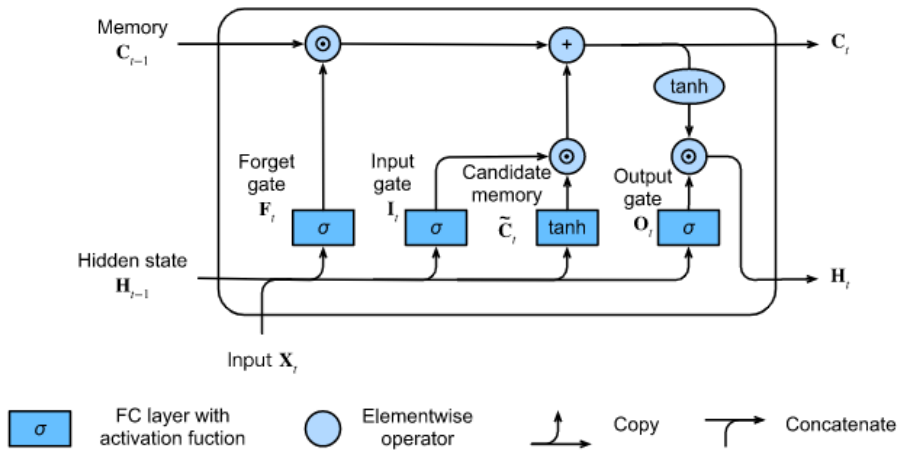
1、 对隐藏状态使用循环计算的神经网络称为循环神经网络（RNN）。
2、 循环神经网络的隐藏状态可以捕获直到当前时间步的序列的历史信息。
3、 循环神经网络模型的参数数量不会随着时间步的增加而增加。
4、 我们可以使用循环神经网络创建字符级、单词级语言模型。

● LSTM:

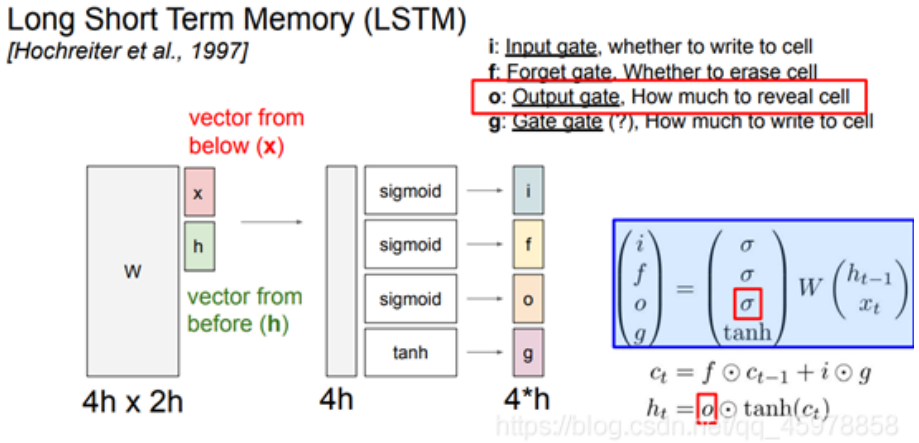
长短期记忆（Long short-term memory，LSTM）是一种特殊的 RNN，主要是为了解决长序列训练过程中的长期信息保存和短期输入跳跃、梯度消失和梯度爆炸问题。简单来说，就是相比普通的 RNN，LSTM 能够在更长的序列中有更好的表现。

可以说，长短期记忆网络的设计灵感来自于计算机的逻辑门。长短期记忆网络引入了存储单元，或简称为**单元**（cell）。有些文献认为存储单元是隐藏状态的一种特殊类型，它们与隐藏状态具有相同的形状，其设计目的是用于记录附加的信息。为了控制存储单元，我们需要许多门。其中一个门用来从单元中读出条目。我们将其称为**输出门**（output gate）。 另外一个门用来决定何时将数据读入单元。我们将其称为**输入门**（input gate）。最后，我们需要一种机制来重置单元的内容，由**遗忘门**（forget gate）来管理。这种设计的动机与门控循环单元相同，即能够通过专用机制决定什么时候记忆或忽略隐藏状态中的输入。

其模型如下图所示：



在本实验中，我们将各门的权重矩阵集中到大矩阵 W 中，用该矩阵进行运算，其中从左到右依次为输入门，遗忘门，输出门，候选记忆单元的权重，模型公式如下：

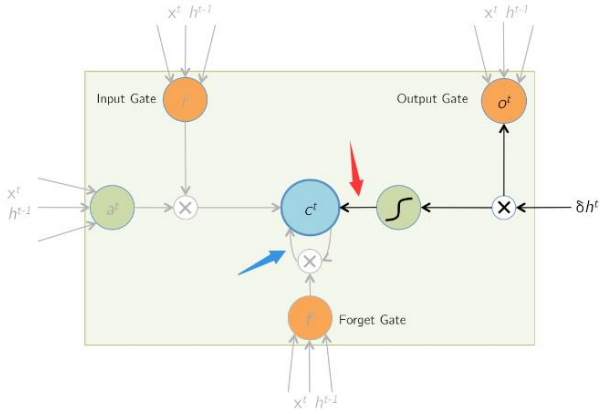


故其单时间步正向传播代码实现如下：

```
N, H = prev_h.shape
h = x.dot(Wx) + prev_h.dot(Wh) + b    # [Nx4H]
h[:, 0:3 * H] = sigmoid(h[:, 0:3 * H])    # [Nx4H]
h[:, 3 * H:4 * H] = np.tanh(h[:, 3 * H:4 * H])    # [Nx4H]
i, f, o, g = h[:, :H], h[:, H:2 * H], h[:, 2 * H:3 * H], h[:, 3 * H:4 * H]    # [NxH]
next_c = f * prev_c + i * g    # [NxH]
next_c_tanh = np.tanh(next_c)    # [NxH]
next_h = o * next_c_tanh    # [NxH]
cache = (x, prev_h, prev_c, Wx, Wh, b, h, next_c_tanh)
```

相较于 RNN，LSTM 单时间步的反向传播过程较为复杂，难以通过直接求导实现，故需分阶段进行推导：

A. Output:

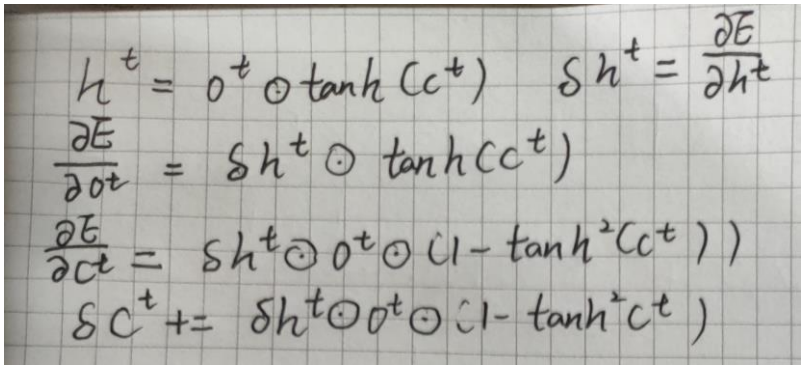


该阶段反向传播部分如图中黑色实线所示，已知条件为：

$$\text{Forward Pass: } h^t = o^t \odot \tanh(c^t)$$

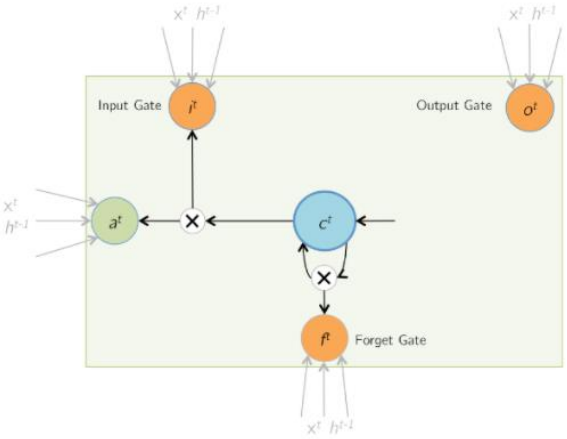
$$\text{Given } \delta h^t = \frac{\partial E}{\partial h^t}, \text{find } \delta o^t, \delta c^t$$

推导过程如下：



需要注意的是，本阶段求出的 δc^t 并不是完整的 δc^t，仅为图中的虹色箭头所示部分，δc^t 的另一部分由蓝色箭头标出，将在下个阶段进行求解。

B. LSTM Memory Cell Update

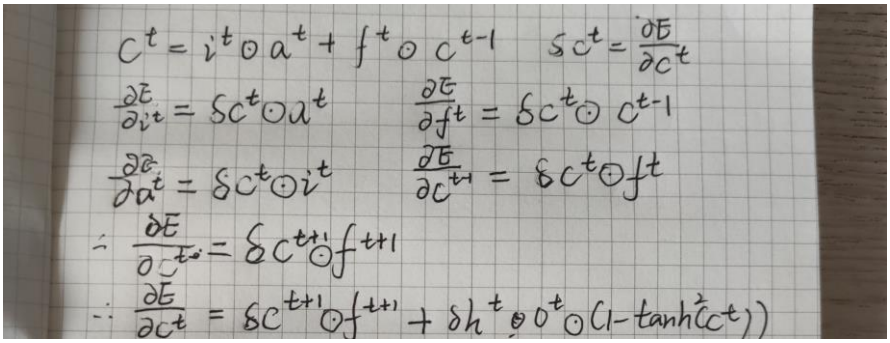


该阶段所求反向传播部分如上图实线所示，已知条件如下：

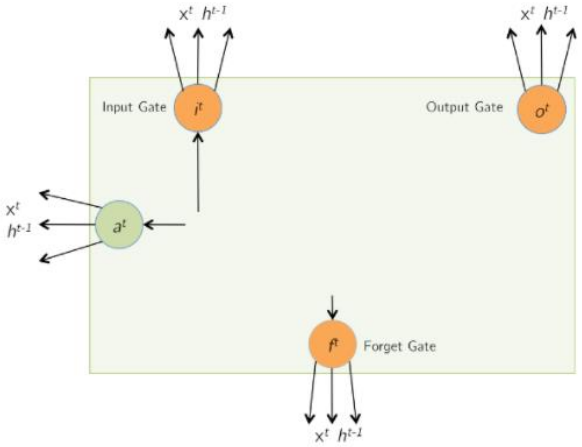
Forward Pass: $c^t = i^t \odot a^t + f^t \odot c^{t-1}$

Given $\delta c^t = \frac{\partial E}{\partial c^t}$, find $\delta i^t, \delta a^t, \delta f^t, \delta c^{t-1}$

推导过程如下：



C. Input and Gate Computation



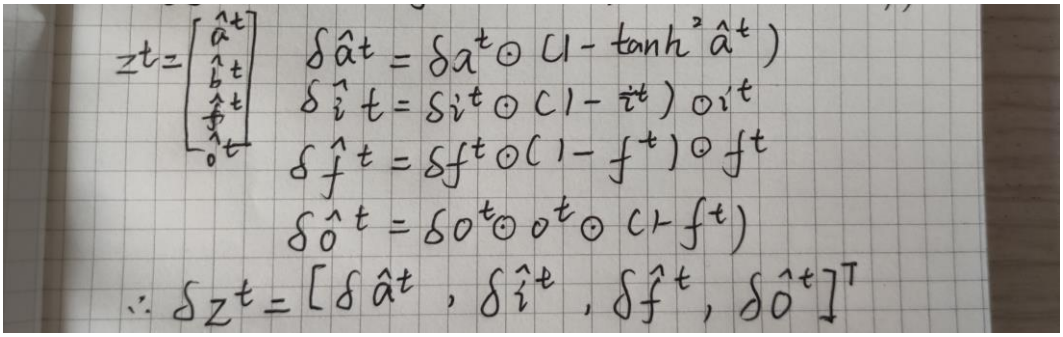
该阶段所求反向传播部分如上图实线所示，参考正向传播公式本阶段方向传播分两步推导。

第一步已知条件如下：

$$\text{Forward Pass: } z^t = \begin{bmatrix} \hat{a}^t \\ \hat{i}^t \\ \hat{f}^t \\ \hat{o}^t \end{bmatrix} = W \times I^t$$

$$\text{Given } \delta a^t, \delta i^t, \delta f^t, \delta o^t, \text{ find } \delta z^t$$

推导过程如下：

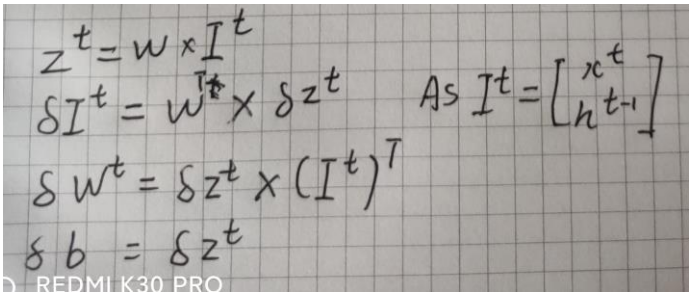


第二步已知条件如下：

$$\text{Forward Pass: } z^t = W \times I^t$$

$$\text{Given } \delta z^t, \text{ find } \delta W^t, \delta h^{t-1}$$

推导过程如下：



相应代码实现如下：

```
dgate = h.copy()

# sigmoid gradient
dgate[:, :3 * H] = dgate[:, :3 * H] * (1 - dgate[:, :3 * H])

# tanh gradient
dgate[:, 3 * H:4 * H] = 1 - dgate[:, 3 * H:4 * H] ** 2
dnc_tanh = 1 - next_c_tanh ** 2

# calculate gradients in common
dnc_prod = dnext_h * o * dnc_tanh + dnext_c
dgate[:, :H] *= dnc_prod * g
dgate[:, H:2 * H] *= dnc_prod * prev_c
dgate[:, 2 * H:3 * H] *= dnext_h * next_c_tanh
dgate[:, 3 * H:4 * H] *= dnc_prod * i

# calculate final gradients
dx = dgate.dot(Wx.T)
dprev_h = dgate.dot(Wh.T)
dprev_c = dnext_c * f + dnext_h * o * dnc_tanh
dWx = x.T.dot(dgate)
dWh = prev_h.T.dot(dgate)
db = dgate.sum(axis=0)
```

在获得单时间步的反向传播公式后，可将单层 LSTM 网络类比沿时间步方向的 MLP，从而获得其整体的反向传播函数，需要注意的是，在一次完整的 LSTM 网络反向传播过程中，在任意时间步阶段 W、b 是通用的，而 h、c 会随着时间步的不同发生改变，相关代码如下：

```
# use of lstm step backward per timestep
for i in range(T - 1, -1, -1):
    dxi, dprev_h, dprev_c, dWxi, dWhi, dbi = lstm_step_backward(dnext_h + dh[:, i, :], dnext_c, cache[i])
    dx[:, i, :] = dxi
    dWx += dWxi
    dWh += dWhi
    db += dbi
    dnext_h = dprev_h
    dnext_c = dprev_c
dh0 = dnext_h
```

总结：

1、 长短期记忆网络是典型的具有重要状态控制的隐变量自回归模型。多年来已经提出了其许多变体，例如，多层、残差连接、不同类型的正则化。然而，由于序列的长距离依赖性，训练长短期记忆网络和其他序列模型（例如门控循环单元）的成本是相当高的。

2、 长短期记忆网络的隐藏层输出包括“隐藏状态”和“记忆单元”。只有隐藏状态会传递到输出层，而记忆单元完全属于内部信息。

3、 长短期记忆网络可以缓解梯度消失和梯度爆炸。

● Network_Visualization：

Q1: Saliency Maps

A1: Saliency Maps 可以理解为是用做模型的解释，可以用来知道哪些变量对于模型来说是重要的。我们也可以将其理解为特征图，它告诉我们图像中的像素点对图像分类结果的影响。

即：给定一张图片 I_0, 其对应的分类是 c, 有一个模型给出图片 I_0 的概率

值是 *S_c(I)*,我们想要衡量 *I_0* 的某个像素点对分类器 *S_c(I)*的影响.

对于复杂的网络来说，模型 *S_c(I)*是一个复杂的非线性模型。但是对于给的图像 *I_0*， 我们可以在 ***I_0* 的周围对模型 *S_c(I)*进行一阶泰勒展开**，如下所示：

$$S_c(I) \approx w^T I + b,$$

其中的 W 就是模型 *S_c(I)*对 *I* 的导数：

$$w = \left.\frac{\partial S_c}{\partial I}\right|_{I_0}.$$

最终我们要做的就是计算 *W* 的值。

其实现步骤如下：

1、 计算与图像像素对应的正确分类中的标准化分数的梯度。如果图像的形状是(3, H, W)，这个梯度的形状也是(3, H, W)；
2、 对于图像中的每个像素点，这个梯度告诉我们当像素点发生轻微改变时，正确分类分数变化的幅度。
3、 计算出梯度的绝对值，然后再取三个颜色通道的最大值；因此最后的 saliency map 的形状是(H, W)为一个通道的灰度图。

相应代码实现如下：

```
scores=model(X)
scores=scores.gather(1,y.view(-1,1)).squeeze()
scores.backward(torch.FloatTensor([1.0,1.0,1.0,1.0,1.0]))
saliency=X.grad.data
saliency=saliency.abs()
saliency,i=torch.max(saliency,dim=1)
saliency=saliency.squeeze()
```

Q2: Fool Image

A2：在深层的神经网络中，真正影响特征信息的，不是个体单元，而是空间信息。神经网络对输入图像的分析是不连续的，所以通过这个特点可以在原始图像上加上一种特定噪声来实现误导。

给定了一张图片和一个目标的类，我们可以在图片上做梯度上升来最大化目标类的分数，直到神经网络把这个图片预测为目标类。

相应代码实现如下：

```
scores = model(X_fooling)
_, index = scores.max(dim=1)
if index == target_y:
    break
target_score = scores[0, target_y]
target_score.backward()
im_grad = X_fooling.grad
X_fooling.data += learning_rate * (im_grad / im_grad.norm())
X_fooling.grad.zero_()
```

Q3：Class Visualization

A3：旨在通过产生一个随机噪声的图片，然后在目标类上做梯度上升，从而生成一张模型会认为是目标类的图片。具体来说，假设 I 是一张图片，y 是目标类. 假设 sy(I)是卷积网络在图片 I 是目标类 y 上面的打分（注意这些都是未归一化的打分，并不是类的概率。）我们希望通过下面这个公式来生成一张图片 I*能够

在目标类 y 上得分高。

$$I^* = \arg\max_I s_y(I) - R(I)$$

可以用梯度上升来解决这个问题,即计算对于生成图片上的梯度，并最大化与目标类相关的部分。其中 R 是一个正则项，本实验使用显式的 L2 正则项，相应代码如下：

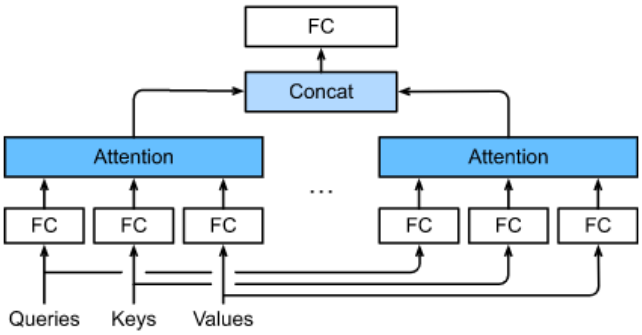
```
score = model(img)
score[0, target_y].backward()
im_grad = img.grad
im_grad -= 2 * l2_reg * img
img.data += learning_rate * im_grad / im_grad.norm()
img.grad.zero_()
```

● Transformer:

Q1:多头注意力(multihead_attention)

A1: 在实践中，当给定相同的查询、键和值的集合时，我们希望模型可以基于相同的注意力机制学习到不同的行为，然后将不同的行为作为知识组合起来，例如捕获序列内各种范围的依赖关系（短距离依赖和长距离依赖）。因此，允许注意力机制组合使用查询、键和值的不同子空间表示可能是有益的。

为此，与使用单独一个注意力汇聚不同，我们可以用独立学习得到的h组不同的线性投影来变换查询、键和值。将这h组变换后的查询、键和值将并行地送到注意力汇聚中。最后将这h个注意力汇聚的输出拼接在一起，并且通过另一个可以学习的线性投影进行变换，以产生最终输出。这种设计被称为**多头注意力**，其中h个注意力汇聚输出中的每一个输出都被称作一个头。模型如下图所示;



在 multi-headed 注意的情况下，我们学习每个 head 的参数矩阵，使模型更能表达注意输入的不同部分。设h为 head 数，Yi为 head_i的注意力输出。因此，我们学习单个矩阵Qi，Ki和Vi。保持我们的整体计算的 head 的情况下,我们选择$Qi \in \mathbb{R}^{d \times d/h}$ $Ki \in \mathbb{R}^{d \times d/h}$和$Vi \in \mathbb{R}^{d \times d/h}$。在上面简单的点积上加上一个比例项 1 /sqrt(d/h)，从而得到:

$$Y_i = \text{softmax}\left(\frac{(XQ_i)(XK_i)^\top}{\sqrt{d/h}}\right)(XV_i)$$

其中$Yi \in \mathbb{R}^{\ell \times d/h}$，$\ell$ 为序列长度。
最后，注意力的输出是 heads 级联的线性变换:

$$Y=[Y1;\ldots;Yh]A \ (A \in \mathbb{R}^{d \times d} 和 [Y1;\ldots;Yh] \in \mathbb{R}^{\ell \times d})$$

相应代码实现如下：

```
H = self.num_heads
Q = self.query(query).view(N, S, H, D // H).transpose(1, 2)    # N, H, S, K
K = self.key(key).view(N, T, H, D // H).transpose(1, 2)    # N, H, T, K
V = self.value(value).view(N, T, H, D // H).transpose(1, 2)    # N, H, T, K

# N, H, S, T <- N, H, S, K * N, H, K, T
E = Q.matmul(K.transpose(2, 3)) / torch.sqrt(torch.Tensor([D / H]))
# mask before softmax
if attn_mask is not None:    # attn_mast: T, S, so maybe here is a mistake
      E = E.masked_fill(attn_mask == False, -float('inf'))

A = torch.softmax(E, dim=3)    # N, H, S, ->T<-
A = self.dropout(A)
# N, H, S, K <- N, H, S, T * N, H, T, K
Y = A.matmul(V)
output = self.proj(Y.transpose(1, 2).reshape(N, S, D))
```

Q2：位置编码（positional-encoding）

A2：在处理词元序列时，循环神经网络是逐个的重复地处理词元的，而自注意力则因为并行计算而放弃了顺序操作。为了使用序列的顺序信息，我们通过在输入表示中添加位置编码来注入绝对的或相对的位置信息。位置编码可以通过学习得到也可以直接固定得到。本实验中使用的是基于正弦函数和余弦函数的固定位置编码.

假设输入表示$\mathbf{X} \in \mathbb{R}^{n \times d}$包含一个序列中$n$个词元的$d$维嵌入表示。位置编码使用相同形状的位置嵌入矩阵$\mathbf{P} \in \mathbb{R}^{n \times d}$并输出$\mathbf{X}+\mathbf{P}$，编码公式如下：

$$p_{i,2j} = \sin\left(\frac{i}{10000^{2j/d}}\right),$$
$$p_{i,2j+1} = \cos\left(\frac{i}{10000^{2j/d}}\right).$$

相应代码如下：

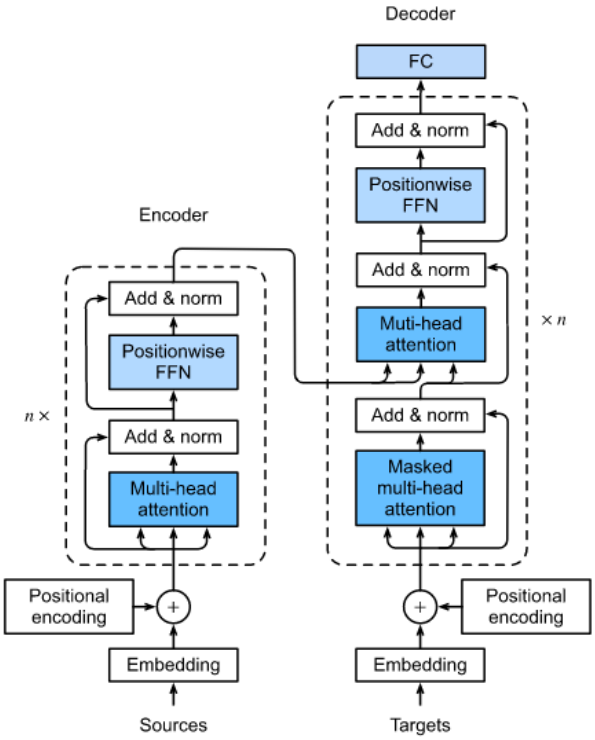
```
col = torch.arange(0, max_len).unsqueeze(1)
row = 10000**(-torch.arange(0, embed_dim, 2) / embed_dim)
m = col * row
pe[:, :, 0::2] = torch.sin(m)
pe[:, :, 1::2] = torch.cos(m)
output = x + self.pe[:, :S, :]
output = self.dropout(output)
```

Q3：Transformer

A3：由于自注意力同时具有并行计算和最短的最大路径长度这两个优势。因此，使用自注意力来设计深度结构是很有吸引力的。对比之前仍然依赖循环神经网络实现输入表示的自注意力模型,Transformer 模型完全基于注意力机制，没有任何卷积层或循环神经网络层。尽管 Transformer 最初是应用于序列到序列的学习文本数据，但现在已经推广到各种现代的深度学习中，例如语言、视觉、语音

和强化学习领域。

Transformer 作为编码器－解码器结构的一个实例，其整体结构图在下图中展示。Transformer 是由编码器和解码器组成的。其编码器和解码器是基于自注意力的模块叠加而成的，源(输入)序列和目标(输出)序列的 嵌入(embedding)表示将加上位置编码(positional encoding)，再分别输入到编码器和解码器中。



从宏观角度来看，Transformer 的编码器是由多个相同的层叠加而成的，每个层都有两个子层（sublayer）。第一个子层是**多头自注意力**；第二个子层是**基于位置的前馈网络**。具体来说，在计算编码器的自注意力时，查询、键和值都来自前一个编码器层的输出。受 ResNet 的启发，每个子层都采用了**残差连接**。在 Transformer 中，对于序列中任何位置的任何输入$\mathbf{x} \in \mathbb{R}d$都要求满足 sublayer$(\mathbf{x}) \in \mathbb{R}d$，以便残差连接满足 $\mathbf{x}+$sublayer$(\mathbf{x}) \in \mathbb{R}d$。在残差连接的加法计算后，紧接着应用**层归一化**。因此，输入序列对应的每个位置，Transformer 编码器都将输出一个d维表示向量。

Transformer 解码器也是由多个相同的层叠加而成的，并且层中使用了残差连接和层归一化。除了编码器中描述的两个子层之外，解码器还在这两个子层之间插入了第三个子层，称为**编码器－解码器注意力**层。在编码器－解码器注意力中，查询来自前一个解码器层的输出，而键和值来自整个编码器的输出。在解码器自注意力中，查询、键和值都来自上一个解码器层的输出。但是，解码器中的每个位置只能考虑该位置之前的所有位置。

这种遮蔽注意力保留了自回归属性，确保预测仅依赖于已生成的输出词元。

总结：

1、 多头注意力融合了来自于相同的注意力汇聚产生的不同的知识，这些知识的不同来源于相同的查询、键和值的不同的子空间表示。通过适当的张量操作，可以实现多头注意力的并行计算。
2、 为了使用序列的顺序信息，我们可以通过在输入表示中添加位置编码来注入绝对的或相对的位置信息。
3、 Transformer 是编码器－解码器结构的一个实践，在该模型中，多头自注意力用于表示输入序列和输出序列，不过解码器还必须通过遮蔽机制来保留自回归属性。模型中基于位置的前馈网络使用同一个多层感知机，作用是对所有的序列位置的表示进行转换。

● GAN：

2014 年，Goodfellow 等人提出了一种生成模型训练方法，简称生成对抗网络(generative Adversarial Networks，简称 GANs)。在 GAN 中，我们构建两种不同的神经网络。我们的第一个网络是传统的分类网络，称为判别器。我们将训练判别器来识别图像，并将其分类为真实(属于训练集)或虚假(不存在于训练集)。我们的另一个网络称为生成器，它将随机噪声作为输入，并使用神经网络对其进行变换以生成图像。生成器的目的是让判别器误以为它产生的图像是真实的。

我们可以将生成器(G)试图愚弄判别器(D)和判别器试图正确区分真实与虚假的这种来回过程视为 minimax 游戏:

$$\underset{G}{\text{minimize}}\ \underset{D}{\text{maximize}}\ \mathbb{E}_{x\sim p_{\text{data}}}\left[\log D(x)\right] + \mathbb{E}_{z\sim p(z)}\left[\log\left(1 - D(G(z))\right)\right]$$

其中$z\tilde{}p(z)$为随机噪声样本，$G(z)$为利用神经网络生成器G生成的图像，D为判别器的输出，表示输入为真实的概率。

为了优化这个极大极小博弈，我们将交替使用G目标上的梯度下降步骤和D目标上的梯度上升步骤:

1. 更新生成器(G)以最小化判别器做出正确选择的概率。

2. 更新判别器(D)以使判别器做出正确选择的概率最大化。

虽然这些分析对更新很有用，但它们在实践中表现不佳。相反，当更新生成器时，通常使用一个不同的目标:最大化判别器做出错误选择的概率。**这个小变化有助于缓解当判别器确定时生成器梯度消失的问题。**

a) Vanilla GAN:

判别器：

```
model = nn.Sequential(
      Flatten(),
      nn.Linear(784, 256),
      nn.LeakyReLU(),
      nn.Linear(256, 256),
      nn.LeakyReLU(),
      nn.Linear(256, 1)
)
```

生成器：

```
model = nn.Sequential(
      nn.Linear(noise_dim, 1024)
      nn.ReLU(),
      nn.Linear(1024, 1024),
      nn.ReLU(),
      nn.Linear(1024, 784),
      nn.Tanh()
)
```

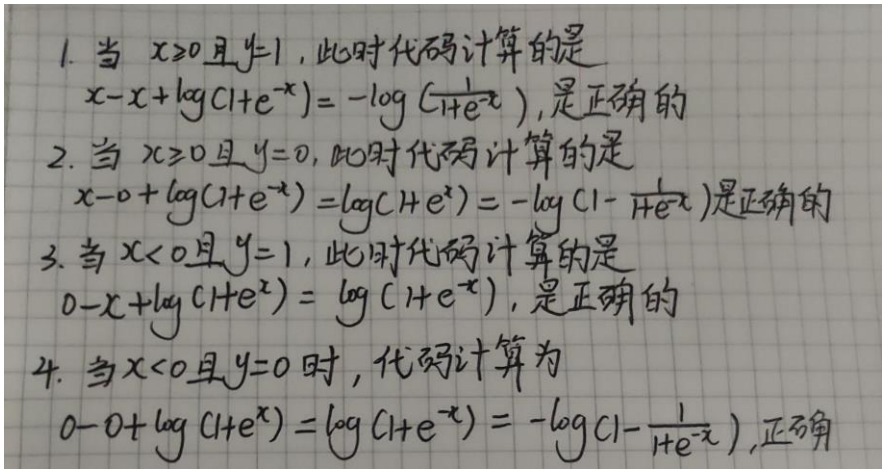
考虑判别器仅需判别生成图片是否有效，可看做二分类问题，故可使用 bceloss:

$$bce(s, y) = -y * \log(s) - (1 - y) * \log(1 - s)$$

需要注意的是原e^x在x是较大正数的时候会出现溢出，也就是超出计算机能表达的范围，从而造成计算误差产生数值稳定性性问题。故需采用以下代码实现：

```
neg_abs = - input.abs()
loss = input.clamp(min=0) - input * target + (1 + neg_abs.exp()).log()
```

相应分析如下：



其判别器 loss 为：

```
labels_real = torch.ones(logits_real.shape).type(dtype)
labels_fake = torch.zeros(logits_fake.shape).type(dtype)
loss = bce_loss(logits_real, labels_real) + bce_loss(logits_fake, labels_fake)
```

生成器 loss 为：

```
labels_fake = torch.ones(logits_fake.shape).type(dtype)
loss = bce_loss(logits_fake, labels_fake)
```

b) LSGAN

最小二乘 GAN 是一种更新、更稳定的 GAN 损失函数。对于这一部分，本实验中 LSGAN 仅需要改变 Vanilla GAN 损失函数。新的损失函数如下：

$$\ell_G = \frac{1}{2}\mathbb{E}_{z\sim p(z)}\left[(D(G(z))-1)^2\right]$$

$$\ell_D = \frac{1}{2}\mathbb{E}_{x\sim p_{\text{data}}}\left[(D(x)-1)^2\right] + \frac{1}{2}\mathbb{E}_{z\sim p(z)}\left[(D(G(z)))^2\right]$$

故判别器 loss 代码如下：

```
loss = 0.5 * torch.mean((scores_real - 1)**2) + 0.5 * torch.mean(scores_fake**2)
```

生成器 loss 代码如下：

```
loss = 0.5 * torch.mean((scores_fake - 1)**2)
```

c) DCGAN

DCGAN 旨在通过引入卷积层，增强模型的空间推理能力，使其能更好的推理出纹理，边缘等信息。

判别器：　　生成器：

```
model = nn.Sequential(
        Unflatten(batch_size, 1, 28, 28), #
        nn.Conv2d(1, 32, 5, 1), # out: 32 *
        nn.LeakyReLU(),
        nn.MaxPool2d(2, 2), # out: 32 * 12 *
        nn.Conv2d(32, 64, 5, 1), # out: 64 *
        nn.LeakyReLU(),
        nn.MaxPool2d(2, 2), # out: 64 * 4 *
        Flatten(),
        nn.Linear(64 * 4 * 4, 4 * 4 * 64),
        nn.LeakyReLU(),
        nn.Linear(4 * 4 * 64, 1)
)
```

```
model = nn.Sequential(
        nn.Linear(noise_dim, 1024),
        nn.ReLU(),
        nn.BatchNorm1d(1024),
        nn.Linear(1024, 7 * 7 * 128),
        nn.ReLU(),
        nn.BatchNorm1d(7 * 7 * 128),
        nn.Unflatten(1, (128, 7, 7)),
        nn.ConvTranspose2d(128, 64, 4, 2, 1),
        nn.ReLU(),
        nn.BatchNorm2d(64),
        nn.ConvTranspose2d(64, 1, 4, 2, 1),
        nn.Tanh(),
        nn.Flatten()
)
```

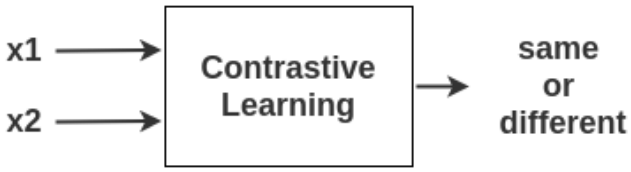
总结：

不同 GAN 模型可能在不同的数据集上能取得较好的结果，但没有实证证据能证明这些 GAN 变体在所有数据集上明显优于原版。（Are GANs Created Equal? A Large-Scale Study
Mario Lucic, Karol Kurach, Marcin Michalski, Sylvain Gelly, Olivier Bousquet https://arxiv.org/abs/1711.1033）

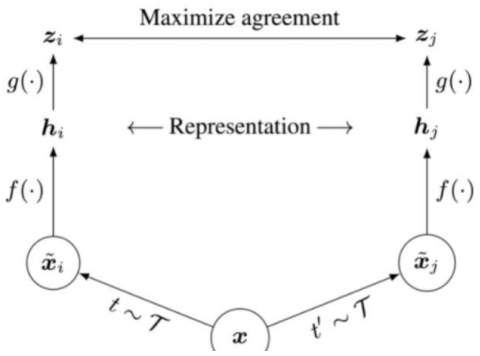
● SimCLR：

现代机器学习需要大量的标记数据。但通常情况下，获取大量人类标记数据是具有挑战性和昂贵的。自我监督学习(SSL)允许模型使用给定数据集中的数据自动学习“好的”表现空间，而不需要标签。具体来说，如果我们的数据集是一堆图像，那么自我监督学习允许模型学习并生成“好的”图像表现向量。SSL 方法之所以如此流行，是因为学习后的模型在其他数据集上仍然表现良好，一个“好的”表现向量需要捕获图像的重要特征，因为它与数据集的其余部分相关。这意味着表现语义相似实体的数据集中的图像应该有相似的表现向量，数据集中不同的图像应该有不同的表现向量。

SimCLR 是一种新兴的自我监督学习模型，意在通过对比学习，使机器区分相似和不相同的东西，对于数据集中的每个图像，SimCLR 生成该图像的两个不同的增强视图，称为正对。然后，鼓励该模型为这对图像生成相似的表现向量。



其模型基本框架如下：



即取一幅图像，对其进行随机变换，得到一对增广图像 x_i 和 x_j。该对中的每个图像都通过编码器以获得图像的表示。然后用一个非线性全连接层来获得图像表示 z，其任务是最大化相同图像的 z_i 和 z_j 两种表征之间的相似性。

具体实现步骤如下：

A. **图像增广**

本实验应用以下图像增广方法：

1. 随机调整大小并裁剪到 32x32。
2. 以 0.5 的概率水平翻转图像。
3. 使用 0.8 的概率，应用颜色抖动。
4. 概率为 0.2，将图像转换为灰度

相应代码如下：

```
transforms.RandomResizedCrop(32),
transforms.RandomHorizontalFlip(p=0.5),
transforms.RandomApply(torch.nn.ModuleList([color_jitter]), p=0.8),
transforms.RandomGrayscale(p=0.2),
```

B. **使用编码器和投影头**

基础编码器f提取增强样本的表示向量。SimCLR 论文发现，使用更深更广的模型可以提高性能，因此选择 ResNet 作为基础编码器。基本编码器的输出是表示向量$hi=f(\hat{x}i)$和$hj=f(\hat{x}j)$。

```
for name, module in resnet50().named_children():
    if name == 'conv1':
        module = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False)
    if not isinstance(module, nn.Linear) and not isinstance(module, nn.MaxPool2d):
        self.f.append(module)
# encoder
self.f = nn.Sequential(*self.f)
```

投影头g是一个小的神经网络，它将表示向量hi和hj映射到应用对比损耗的空间。采用非线性投影头可以提高前一层的表示质量。具体来说，使用了一个带有一个隐藏层的 MLP 作为投影头g。然后根据输出$zi=g(hi)$和$zj=g(hj)$计算对比损失。

```
self.g = nn.Sequential(nn.Linear(2048, 512, bias=False), nn.BatchNorm1d(512),
                       nn.ReLU(inplace=True), nn.Linear(512, feature_dim, bias=True))
```

C. **模型调优**

1) **计算余弦相似度**

用余弦相似度计算图像的两个增强的图像之间的相似度。对于两个增强的图像 x_i 和 x_j，在其投影表示 z_i 和 z_j 上计算余弦相似度。

$$sim(z_i, z_j) = \frac{z_i \cdot z_j}{\|z_i\| \|z_j\|}$$

相应代码如下：

```
z_i_normalized = z_i / torch.linalg.vector_norm(z_i)
z_j_normalized = z_j / torch.linalg.vector_norm(z_j)
norm_dot_product = torch.dot(z_i_normalized, z_j_normalized)
```

2) NT-Xent 损失(归一化温度-尺度交叉熵损失)

即将 batch 的增强对逐个取出。使用 softmax 函数来得到这两个图像相似的概率。此时，batch 中所有剩余的图像都被采样为不相似的图像(负样本对)。

$$l\left(i,j\right)=-\log\frac{\exp\left(\text{ sim}\left(z_i,z_j\right)\ /\ \tau\right)}{\sum_{k=1}^{2N}1_{k\neq i}\exp\left(\text{ sim}\left(z_i,z_k\right)\ /\ \tau\right)}$$

然后通过取上述计算的对数的负数来计算这一对图像的损失。最后计算 batch 内所有正对图片的损失并计算平均值。

$$L=\frac{1}{2N}\sum_{k=1}^{N}\left[\ l(k,\ k+N)+l(k+N,\ k)\ \right]$$

相应代码实现如下：

```
# Step 1: Use sim_matrix to compute the denominator value for all augmented samples.
# Hint: Compute e^(sim / tau) and store into exponential, which should have shape 2N x 2N.
exponential = torch.exp(sim_matrix.to(device) / tau)    #分子

# This binary mask zeros out terms where k=i.
:orch.eye创建一个2维张量，对角线数字为1，  其他位置为0，也就是一个单位矩阵
mask = (torch.ones_like(exponential, device=device) - torch.eye(2 * N, device=device)).to(device).bool()

# We apply the binary mask.
exponential = exponential.masked_select(mask).view(2 * N, -1)    # [2*N, 2*N-1]

# Hint: Compute the denominator values for all augmented samples. This should be a 2N x 1 vector.
denom = torch.sum(exponential, dim=1, keepdims=True)

# Step 2: Compute similarity between positive pairs.
# You can do this in two ways:
# Option 1: Extract the corresponding indices from sim_matrix.
# Option 2: Use sim_positive_pairs().
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

pos_pairs = sim_positive_pairs(out_left, out_right).to(device)
pos_pairs = torch.cat([pos_pairs, pos_pairs], dim=0)

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# Step 3: Compute the numerator value for all augmented samples.
numerator = None
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

numerator = torch.exp(pos_pairs / tau)

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# Step 4: Now that you have the numerator and denominator for all augmented samples, compute the total loss.
loss = torch.mean(-torch.log(numerator / denom))
```

基于这种损失，编码器和投影头表示法会随着时间的推移而改进，所获得的表示法会将相似的图像放在空间中更相近的位置。之后，预训练过的 SimCLR 模型即可被用到迁移学习中。

总结：

1、 SimCLR 为自监督学习方向的进一步研究和改善计算机视觉的自监督学习状态提供了一个强有力的框架。

2、 该框架可以作为很多视觉相关的任务的预训练模型，可以在少量标注样本的情况下，拿到比较好的结果。

● 本章总结：

本章节实验大部分可以使用 pytorh，因此在实现方面感觉难度在三次作业中最低，仅 LSTM 的反向传播部分较难实现；但本章内容较多，有部分是《动手深度学习》，《深度学习与计算机视觉》中没有提到的，因此存在一定理解难度，需查阅其他资料。通过本章实验的学习，复习巩固了 RNN，LSTM，Transformer，attention 等模型，学习和掌握了 GAN，SimCLR 等模型的原理和基础实现，应用方法。并锻炼了应用 pytorch 的编码能力。