学习心得

（因主要使用 C 语言，对 python 编程语言仅有涉猎了解，并不熟练，所以在学习中遇到的 python 函数与库也会进行注明，pytorch 学自 B 站 up 主刘二大人）

学习内容：1，全连接神经网络 2，卷积神经网络 3，循环神经网络

贯穿始终的思路图：

贯穿始终的思路：

引入 torch，numpy，torchvision 等库——>定义数据内容（自定义数据列表，或引入 datasets 与 dataloader ——>定义模型类（全连接或卷积或循环）——>引用模型，设置损失函数（BCELoss 或 NULLLoss 或 CrossEntoupyLoss）与优化器（optimizer）——>epoch 运行进行训练（引入数据-引入损失-清零-前馈-反馈-优化）——>test 测试集进行测试——>准确率，损失值，返回率，f1 评估模型

全连接神经网络

线性模型（Linear modle）：

线性模型：y = w * x + b

线性模型作为全连接神经网络最基础的单元，在后期作为线性单元而被引入和重复使用，通过线性模型建立输入数据与输出数据之间的线性关系，并通过建立的线性关系对测试集给入的数据进行预测，其最主要的地方在于最佳权重 w

（weight）的寻找，而偏置量 b（bias）的寻找思路与其大致相同。

```python
"""线性模型"""
import numpy as np
import matplotlib.pyplot as plt

x_data = [1.0,2.0,3.0]
y_data = [2.0,4.0,6.0]

def forward(x):
    return x * w

def loss(x,y):
    y_pred = forward(x)
    return (y_pred - y)*(y_pred - y)

w_list = []
mse_list = []

for w in np.arange(0.0,4.1,0.1):
    print("w = ",w)
    l_sum = 0
    for x_val,y_val in zip(x_data,y_data):
        y_pred_val = forward(x_val)
        l_sum += loss(x_val,y_val)
        print("\t",x_val,y_val,y_pred_val)
    w_list.append(w)
    mse_list.append(l_sum/3)
plt.plot(w_list, mse_list)
plt.show()
```

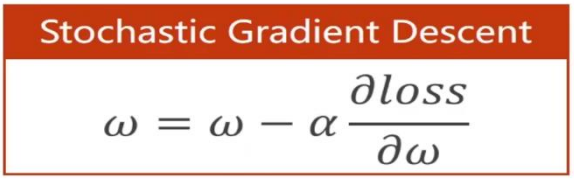
此处代码，在起始处引入 numpy 库与 matplotlib 库（图形绘画库），并建立 x_data 与 y_data 数据列表，此处 x_data 与 y_data 作用相当于后期训练集，定义的 forward 作用是对输入数据乘以相应权重，来作为输出端口，即前馈模型，之后定义损失函数，此处采用平均平方差（mse）作为损失函数，后续循环模块中在 w 在 0.0-4.1 之间以步幅 0.1 进行遍历，通过前馈，计算损失函数，寻找损失最小 w，即最佳权重点。

全局最优点：即使损失值最小的权重值在权重曲线上对应的点。

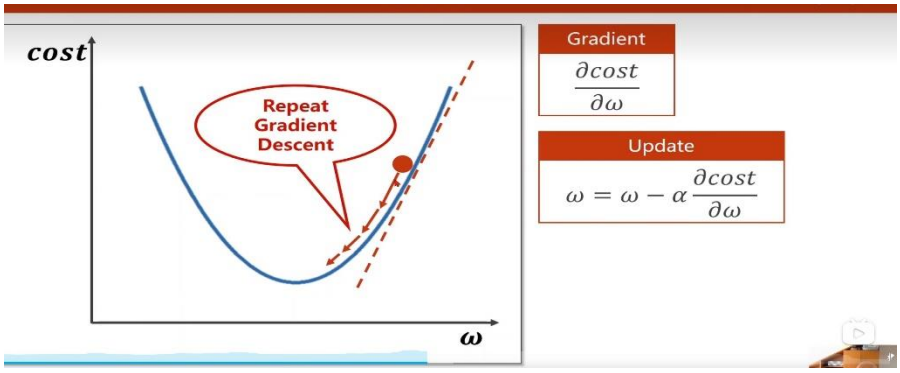
局部最优点：采取梯度下降的方法寻找全局最优点时，梯度设置不合适等原因造成 w 停留在局部最低点，而无法继续向全局最优点移动，在后期可采用动量率（momentum）的方法来缓解这种现象。

梯度下降：

即 w 在遍历的过程中，让 w 沿着 w-cost 权重曲线以曲线斜率 grad 为步幅使得 w 沿着曲线向下滚动，从而找到全局最优点的方法。即 w = w – a*grad（w 每滚动一次，更新一次 grad 的值，a 是学习率，控制梯度下降步幅的重要参数）

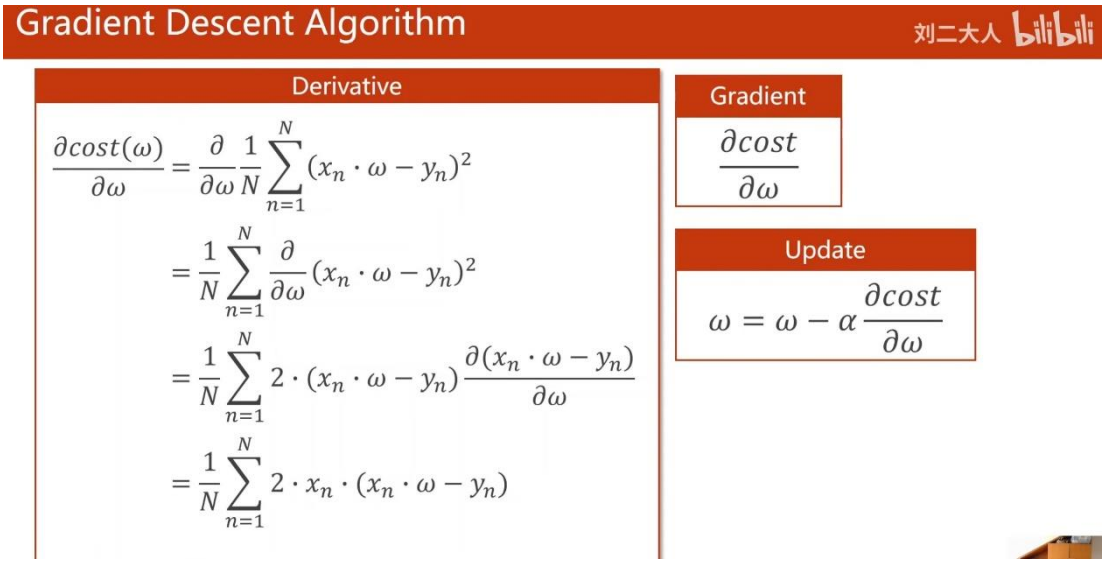


即 SGD 模型在后续全连接神经网络与卷积神经网络优化器的重要公式



此处即为梯度下降的核心滚动或说下降代码。a 为学习率。

在梯度下降中我们不再采用线性模型中的 mse 损失而是采用更为适合的损失韩式



损失表达式如图所示：

```
"""
import numpy as np
import matplotlib.pyplot as plt

x_data = [1.0,2.0,3.0]
y_data = [2.0,4.0,6.0]

w = 1.0

def forward(x):
    return x * w

def cost(xs,ys):
    cost = 0
    for x_val,y_val in zip(xs,ys):
        y_pred = forward(x_val)
        cost += (y_pred - y_val)**2
    return cost / len(xs)

def gra(xs,ys):
    gra = 0
    for x_val,y_val in zip(xs,ys):
        gra += 2*x_val*(x_val * w - y_val)
    return gra / len(xs)

print('Predict (before learning)',4,forward(4))
for epoch in range(100):
    cost_val = cost(x_data,y_data)
    gra_val = gra(x_data,y_data)
    w -= 0.01*gra_val
    print('Epoch', epoch,'w = ',w,'loss = ',cost_val)
print('Predict (after learning)',4,forward(4))
"""
```

此处代码主要改动在于损失函数的编写，换成了新的 cost 损失，梯度下降作为贯穿始终的线元寻找全局最佳权重的主要方法，在运行中会出现各种各样的问题比如局部最优点，震荡等等，在后续的学习中会继续优化代码，比如更改学习率（lr），引入冲量率（momentum）等方法来缓解震荡与局部最优点的情况，使得训练的效率更加高效，结果更加优化

随机梯度下降：（与梯度下降的主要区别）

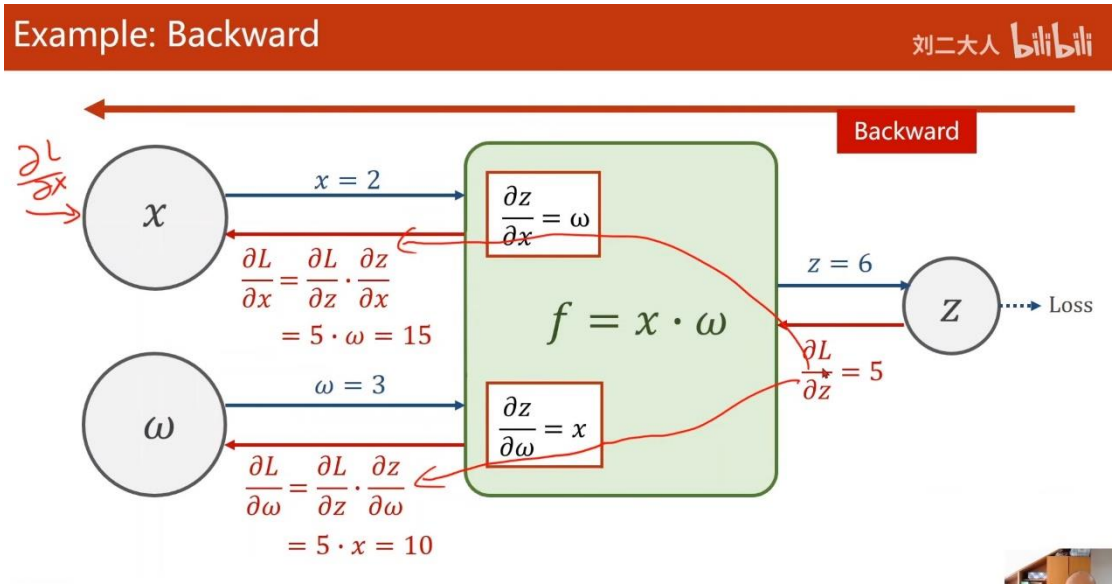
随机梯度下降是梯度下降的一种，其与梯度下降之间的关系可以理解为，梯度下降是设置一个 w 的起始点后，梯度应采用全局所有 w 损失的平均值，对 w 进行遍历，而随机梯度下降则是将一个 w 的损失值计算下降的梯度，拿出进行操作的方法

```
"""随机梯度递减"""
"""
import numpy as np
import matplotlib.pyplot as plt

x_data = [1.0,2.0,3.0]
y_data = [2.0,4.0,6.0]

w = 1.0

def forward(x):
    return x * w

def loss(x,y):
    y_pred = forward(x)
    return (y_pred - y)**2

def grad(x,y):
"梯度斜率计算函数"
    grad = 2 * x * (x * w - y)
    return grad

print('Predict (before learning)',4,forward(4))
for epoch in range(100):
    for x,y in zip(x_data,y_data):
        grad_val = grad(x,y)
        w -= 0.01 * grad_val
        print("\tgrad: ",x,y,grad)
        loss_val = loss(x, y)
    print('\t',"Epoch = ",epoch,"w = ",w,"loss_val = ",loss_val)
print('Predict (after learning)',4,forward(4))
"""
```

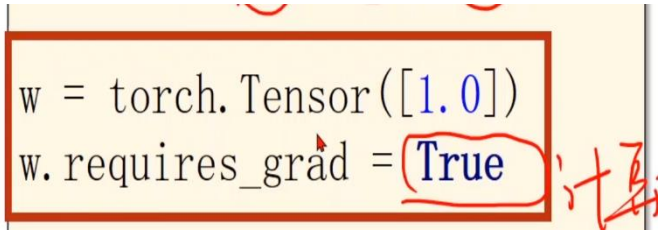
此处为随机梯度下降的代码，与梯度下降代码相比，主要区别在于提前更定一个 w 的权重值，剩余代码操作则大相径庭。

反向传播：（反馈的实际定义）

反向传播即为反馈（backward）是全连接神经网络，卷积神经网络与循环神经网络训练模型中极为重要的一环，其主要和前馈搭配使用



如图中展示，在 x 与 w 的线性运算得到 z（其实就是预测值）后，z 再通过损失函数得到最后的损失值，由 x 与 w 得到 y_pred 的过程即为前馈，而反馈则是通过积分（个人认为）与偏微分与多元积分的方法从最终的损失函数向前进行反馈回溯给出 w 与 x 的过程。



在反向传播当中也首次出现了 Tensor 数据类型的定义，Tensor 数据类型是一种张量数据类型，其主要用来定义张量，比如向量，矩阵，三维空间（包括后期图像中（channels，height 与 weight）组成的三维空间）此处虽然只定义了 1.0 一个数值，但是因为是张量的缘故仍然需要加上中括号。新建的 Tensor 变量默认不会进行梯度操作，而 w.requires_grad=True 则表示 w 需要并可以进行梯度操作

```
"""反向传播"""
"""
import numpy as np
import torch
import matplotlib.pyplot as plt

x_data = [1.0,2.0,3.0]
y_data = [2.0,4.0,6.0]
"tensor变量可以是点，线，面，三维变量"
w = torch.tensor([1.0])
""创建的tensor默认的不会计算梯度，所以要写一个需要梯度""
w.requires_grad = True

def forward(x):
    return x * w

def loss(x,y):
    y_pred = forward(x)
    return (y_pred - y)**2

print("Predict (before learning)",4,forward(4).item())
"item()用来将tensor转化为python的标量"
for epoch in range(100):
    for x,y in zip(x_data,y_data):
        l = loss(x,y)
        l.backward()
        print('\t\n',x,y,w.grad.item())
        w.data = w.data - 0.01*w.grad.data
        w.grad.data.zero_()
    print("process",epoch,l.item())
print("Predict (after learning)",4,forward(4).item())

plt.plot(l.item(),w.data.item())
plt.show()
```

循环进行一轮后，将 w 梯度清零。

此处代码与之前相比首次引入了 torch 库，其前馈函数（forward）与损失函数与之前保持一致，print 操作中，因为 w 是 tensor，其与 w 进行线性操作后，得到的前馈结果即有 y_pred 也自动进行类型转换变换成 tensor 变量，此时想要打印 tensor 变量中的数值则需要在 tensor 变量后赘述 item（），表示提取张量中数据，而 w.data 则表示返回与 w 相同的 tensor 并于原 w 共享，即一方数据改变，另一方数据也发生改变，所起的作用是将 tensor 从原有的计算图中分离出来，因其仍未 tensor 的缘故，故调用其中元素时仍需加入.item（）

此处也首次出现了清零函数，即 w.grad.data.zero_（）此处表示

用 Pytorch 实现线性回归：

```
import  torch

x_data = torch.Tensor([[1.0],[2.0],[3.0]])
y_data = torch.Tensor([[2.0],[4.0],[6.0]])

class LinearModle(torch.nn.Module):    #继承自Module

    def __init__(self):  #__表示构造
        super(LinearModle,self).__init__()  #调用父类的构造
        self.linear = torch.nn.Linear(1,1) # (1, 1) 分别对应y_pred与输入x的维度
        ""Linear(input,output,bias = True)
        其中，input与output分别对应输入与输出的维度，bias是一个bool型变量，表示是否加入偏置量
        ""

    def forward(self,x):
        y_pred = self. linear(x)
        return y_pred

modle = LinearModle()

criterion = torch.nn.MSELoss(size_average = False)
""MSELoss是Module自带的类，求损失值，size_average表示是否对Loss求均值""
optimizer = torch.optim.SGD(modle.parameters(),lr = 0.01)
""optim表示优化，torch.optim.SGD表示优化模块。
parameters用来搜索linear中表示权重的量，
lr是梯度学习率""

for epoch in range(10000):
    y_pred = modle(x_data)   #直接调用LinearModle中的forward函数
    loss = criterion(y_pred,y_data)
    print(epoch,loss)

    optimizer.zero_grad()   #清零
    loss.backward()         #反馈
    optimizer.step()        #更新

print('w = ',modle.linear.weight.item())  #调用modle中的linear中的weight
print('b = ',modle.linear.bias.item())

x_test = torch.Tensor([[4]])
y_test = modle(x_test)
print('y_pred = ',y_test.data)
```

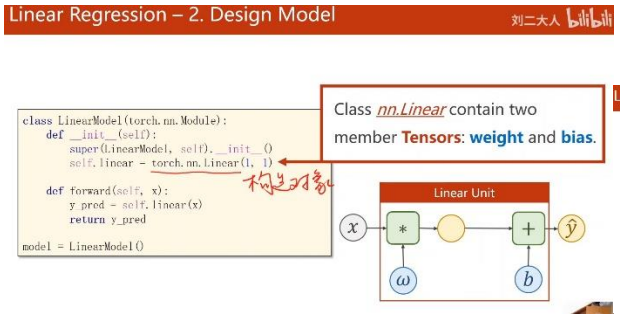
Pytorch 作为深度学习的重要工具，在本节正式开始进行训练模型的构建，训练模型是整个深度学习过程的核心部分，其不仅是算法层级集中定义体现的部分（全连接层，卷积层，池化层，残差块等等），也是定义训练过程的部分，后续的训练操作都是在此基础上展开的。此处代码中，定义了一个线性模型，linear（1，1），linear

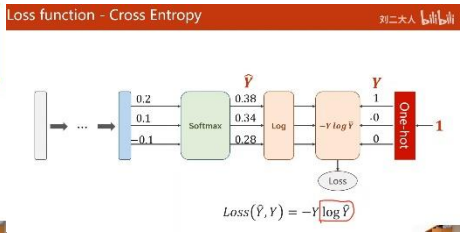
表示线性单元，前一个参数表示输入 x 的维度，后一个参数表示输出的维度。模型继承自 torch.nn.Module，super（类名称，self）.__init__()

是继承的代码，此处模型引入了一个全连接层。Model = LinearModel（）是把模型实列化，criterion 定义了损失函数的实例化，采用 MSELoss 损失（size_average=False）表示不对损失求均值，optimizer 是优化器的使用,后续即训练模型操作（前馈，清零，反馈，更新）。

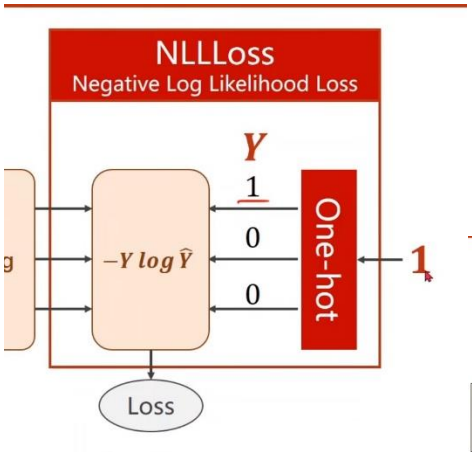
（时间缘故，心得编写时间不够，后面心得主要以贴图为主）

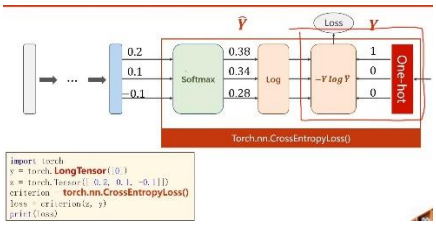
基础线性层





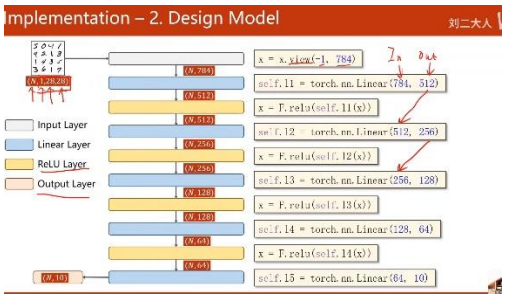
NLLLoss 损失与 CrossEntropyLoss 损失

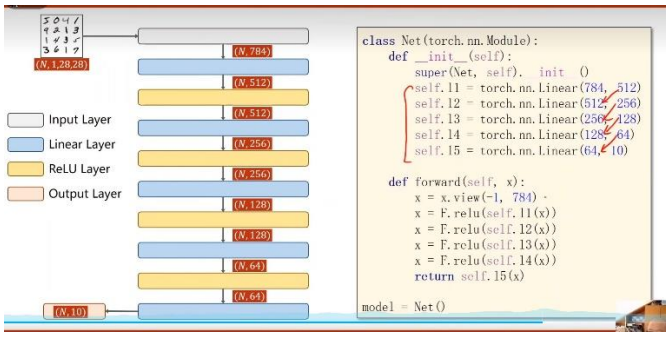




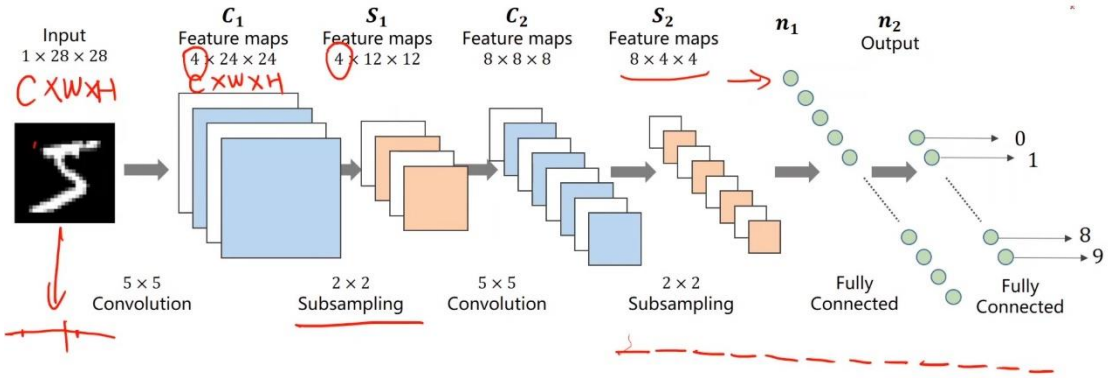
全连接层的使用与激活，引入非线性映射使得模型可以处理更为复杂的数据

$$Pixel_{norm} = \frac{Pixel_{origin} - mean}{std}$$

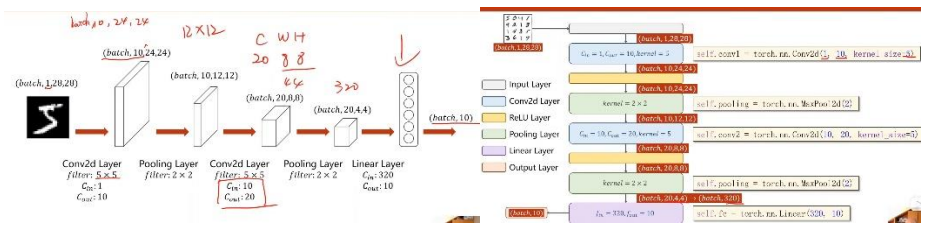


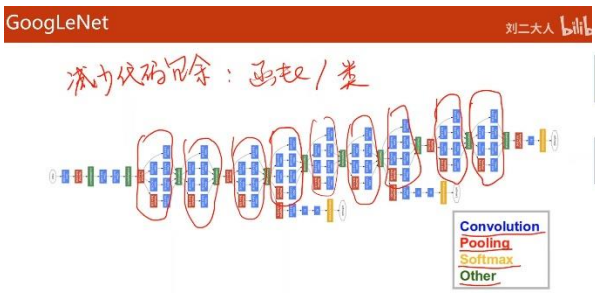


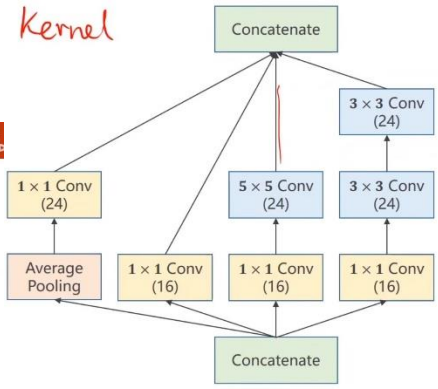
卷积

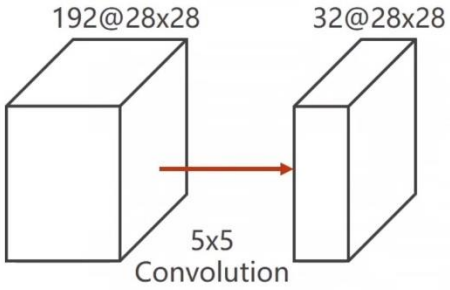


卷积核和池化层





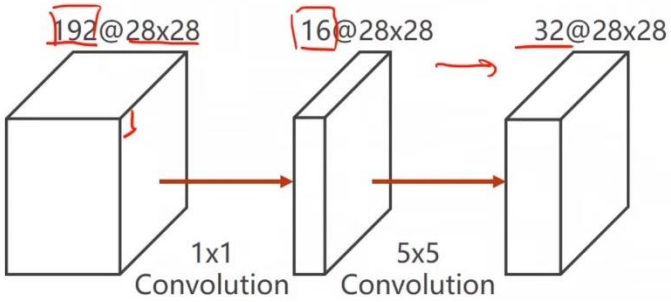




Operations:

$5^2 \times 28^2 \times 192 \times 32 =$

$120{,}422{,}400$

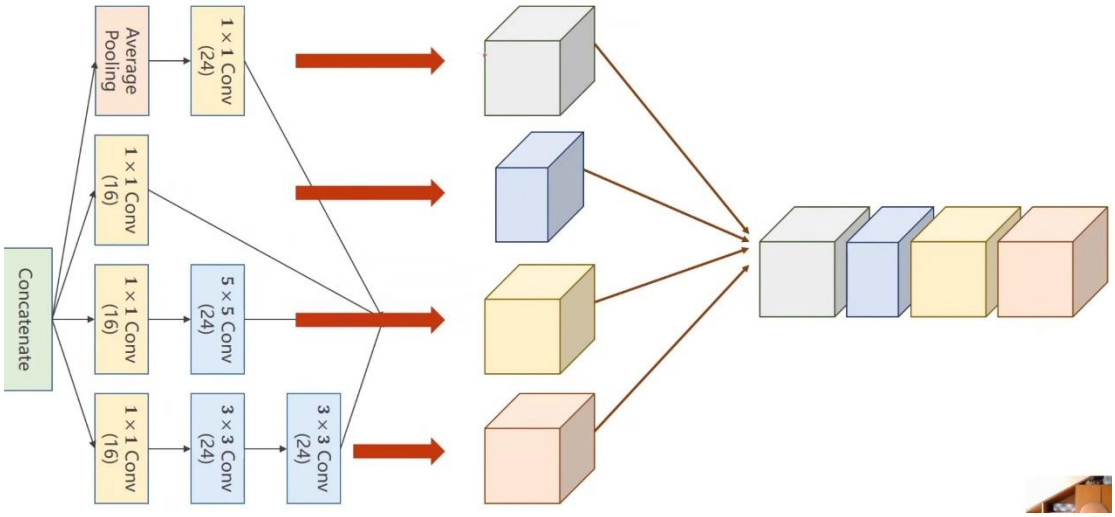


Operations:

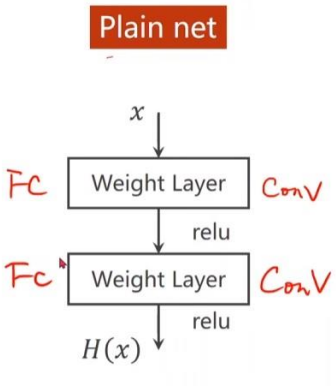
$1^2 \times 28^2 \times 192 \times 16 +$

$5^2 \times 28^2 \times 16 \times 32 =$

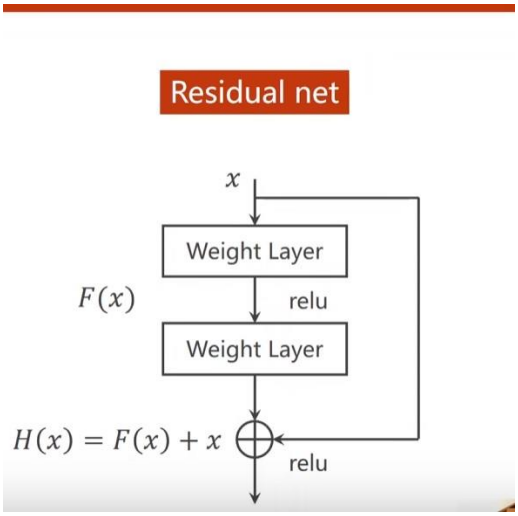
$12{,}433{,}648$

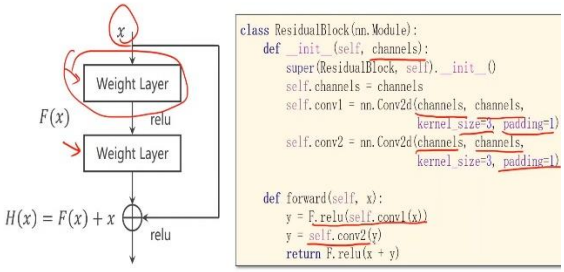


普通网络随着线性层的增多，权重相乘变多，梯度趋近于 0，即梯度消失

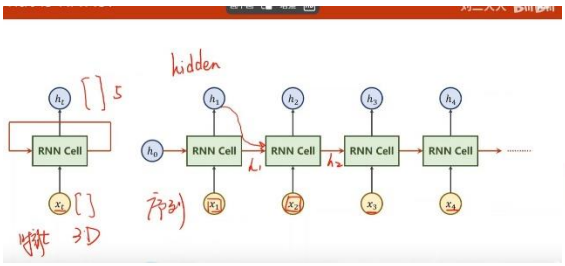


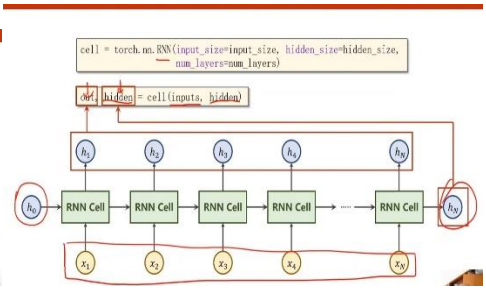
引入残差块，消除梯度消失

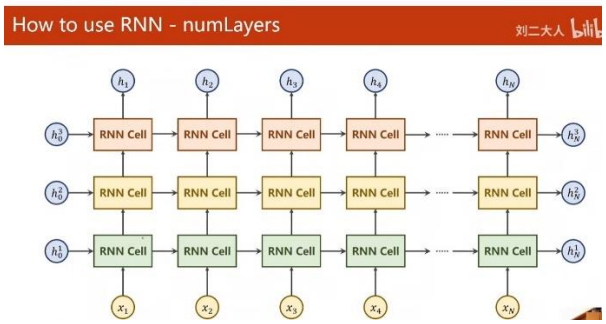




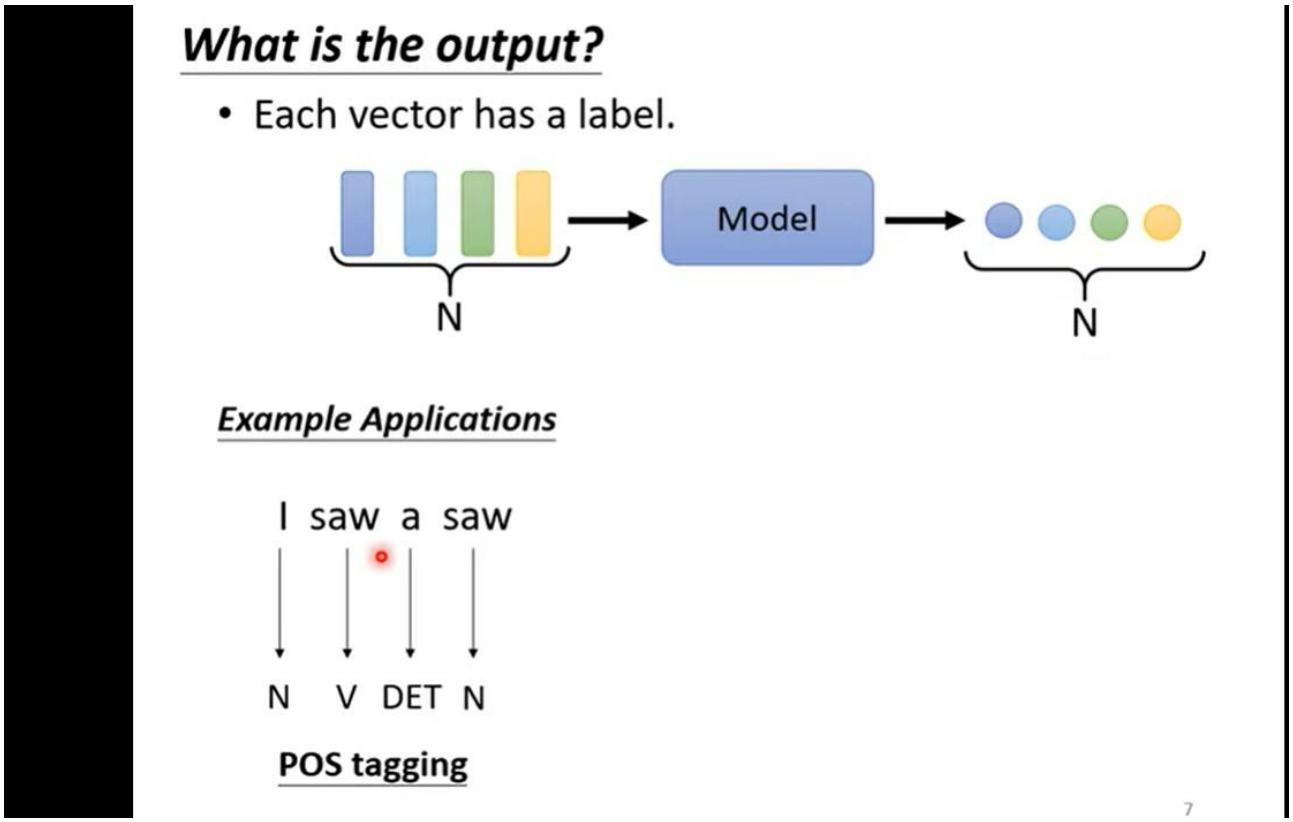
循环神经网络





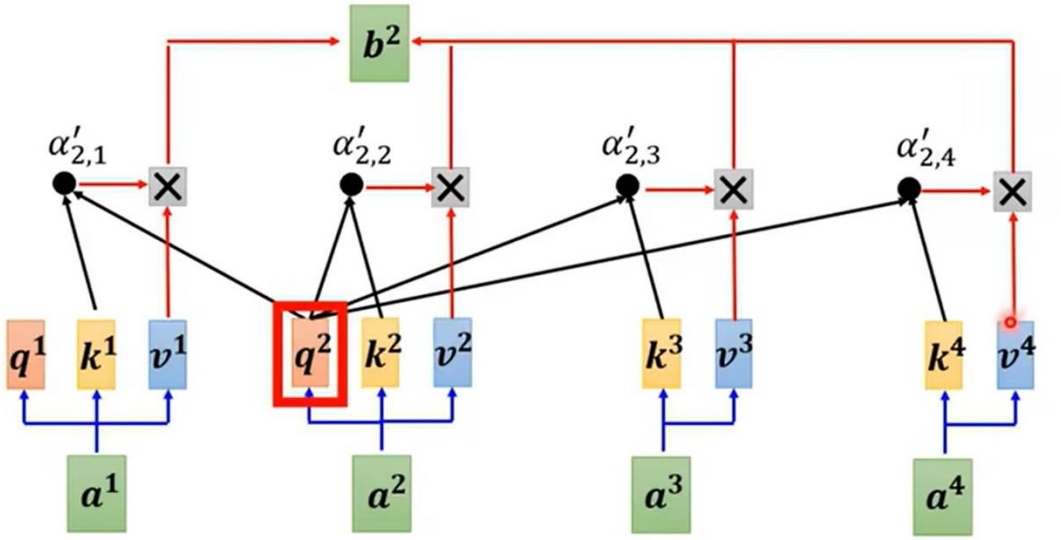


Self attention 主要表示通过与前后上下文输入值一起分析预测的方法，其计算时为多分支并行运算



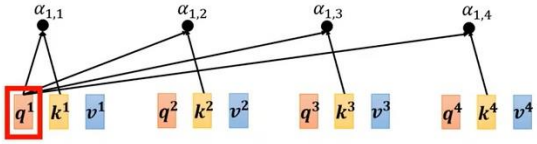
Q（查询），k（键），v（值）

每一个输出的 b 其运算方式如后图



Self-attention

$\alpha_{1,1} =$ k^1 q^1

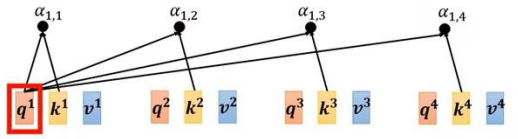


Self-attention

$\alpha_{1,1} =$ k^1 q^1 $\alpha_{1,2} =$ k^2 q^1

$\alpha_{1,3} =$ k^3 q^1 $\alpha_{1,4} =$ k^4 q^1

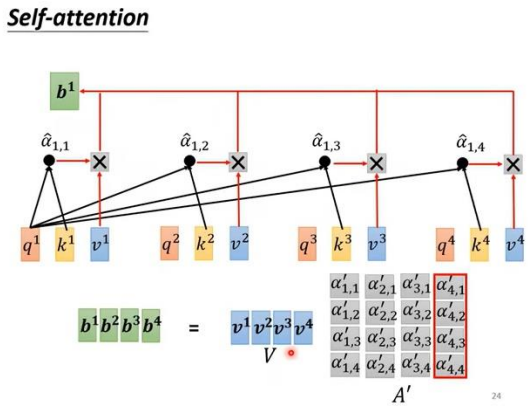
$\alpha_{1,1}$ $\alpha_{1,2}$ $\alpha_{1,3}$ $\alpha_{1,4}$ $= k^1$ k^2 k^3 k^4 q^1



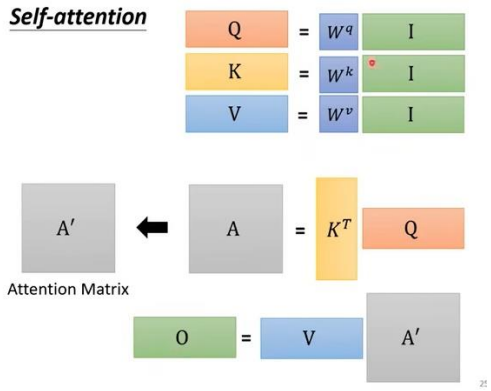
$$\begin{bmatrix} \alpha_{1,1} & \alpha_{2,1} & \alpha_{3,1} & \alpha_{4,1} \\ \alpha_{1,2} & \alpha_{2,2} & \alpha_{3,2} & \alpha_{4,2} \\ \alpha_{1,3} & \alpha_{2,3} & \alpha_{3,3} & \alpha_{4,3} \\ \alpha_{1,4} & \alpha_{2,4} & \alpha_{3,4} & \alpha_{4,4} \end{bmatrix} = \begin{matrix} k^1 \\ k^2 \\ k^3 \\ k^4 \end{matrix} \begin{matrix} q^1 & q^2 & q^3 & q^4 \end{matrix}$$

$$\begin{bmatrix} \alpha'_{1,1} & \alpha'_{2,1} & \alpha'_{3,1} & \alpha'_{4,1} \\ \alpha'_{1,2} & \alpha'_{2,2} & \alpha'_{3,2} & \alpha'_{4,2} \\ \alpha'_{1,3} & \alpha'_{2,3} & \alpha'_{3,3} & \alpha'_{4,3} \\ \alpha'_{1,4} & \alpha'_{2,4} & \alpha'_{3,4} & \alpha'_{4,4} \end{bmatrix} \leftarrow \text{softmax} \begin{bmatrix} \alpha_{1,1} & \alpha_{2,1} & \alpha_{3,1} & \alpha_{4,1} \\ \alpha_{1,2} & \alpha_{2,2} & \alpha_{3,2} & \alpha_{4,2} \\ \alpha_{1,3} & \alpha_{2,3} & \alpha_{3,3} & \alpha_{4,3} \\ \alpha_{1,4} & \alpha_{2,4} & \alpha_{3,4} & \alpha_{4,4} \end{bmatrix} =$$

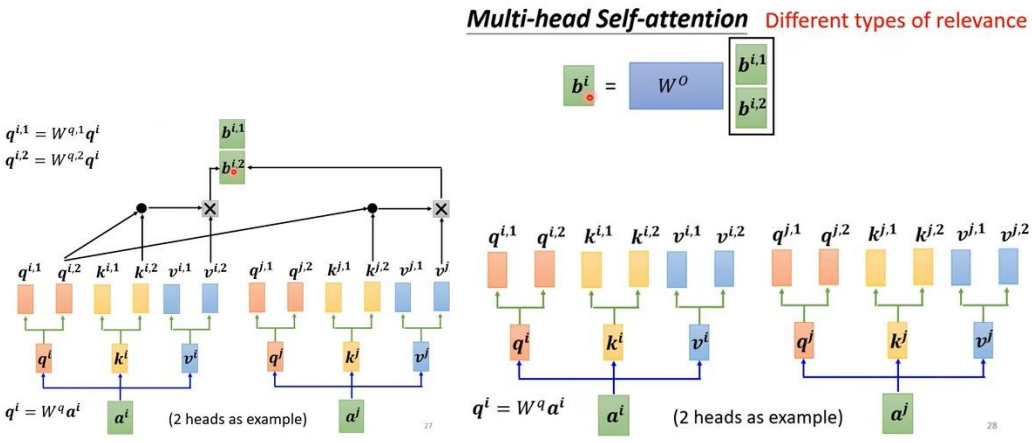
A' softmax A

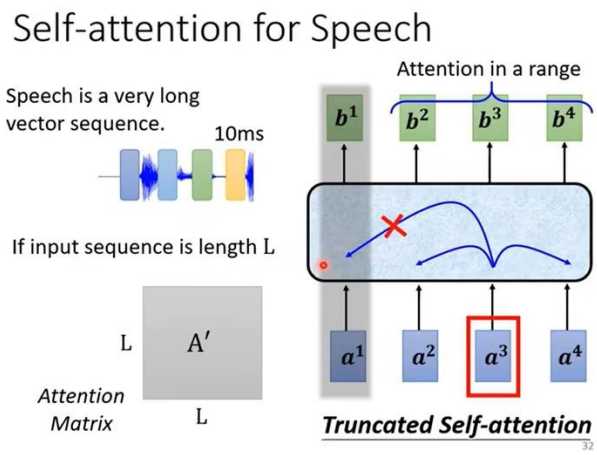


此图主要表示 Self attention 中主要的未知量为 wp，wk，wv 三个矩阵



此处为更为复杂的注意力机制





Self attention 与 CNN 的关系

Self-attention

1. **全局依赖性**：Self-attention机制能够捕捉序列中任意两个位置之间的依赖关系，这使得它在处理长序列数据时特别有效。相比之下，CNN通常关注局部邻域内的依赖关系。
2. **并行计算**：Self-attention允许模型在计算时并行处理整个序列，这与RNN的顺序处理形成对比，后者限制了计算的并行性。而CNN虽然也可以并行处理数据，但它的并行性受限于卷积核的大小和步长。
3. **灵活性**：Self-attention提供了更高的灵活性，因为它可以自适应地学习序列中不同部分的重要性。CNN的卷积核大小和步长是预先定义的，这限制了其对不同尺度特征的适应性。
4. **可扩展性**：在数据量较大的情况下，Self-attention能够更好地捕捉复杂的模式，而CNN可能受限于其固定的感受野大小。

CNN

1. **局部感受野**：CNN通过卷积层捕捉局部特征，这使得它在图像处理等任务中非常有效，因为它能够利用图像的局部性质。
2. **参数共享**：CNN中的卷积核通过参数共享机制减少了模型的复杂性，这有助于减少过拟合的风险并提高模型的泛化能力。
3. **计算效率**：在处理图像等具有明显局部结构的数据时，CNN通常比Self-attention更高效，因为它的卷积操作可以利用硬件加速。
4. **固定结构**：CNN的结构相对固定，卷积层、池化层和全连接层的组合形成了经典的CNN架构。而Self-attention机制则更加灵活，可以根据任务的需要进行调整。