学习记录

（本身有一些 c 语言基础和 python 的编程基础，对于大篇幅的 python 程序以及需要进行许多 import 的程序也完成过一个，但是编写过程耗时很长。对神经网络与机器学习除了耳闻就是一无所知了，这段时间在 b 站跟着 up“刘二大人”自学）

贯穿始终的流程图:

分别是 ①准备数据集

②设计模型，继承自 torch.nn.Module

③构造优化器和损失函数

④循环训练，三步走，前馈，反馈，更新

如下是一些课程外的知识点（网课中没说的），以及一些异常情况的处理：

一、反向传播：按照网课老师的代码，发现会报错无法运行，显示问题在函数（）

backward 一处，

理解：有部分非张量参与了张量的运算，换言之，没有像老师说的那样解释器会对一些数值自动进行转换。

```python
import torch
x_list = [1.0, 2.0, 3.0]
y_list = [2.0, 4.0, 6.0]
w = torch.Tensor([1.0])
w.requires_grad_ = True

def forward(x):
    return x*w

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

print("predict (before training):", 4,forward(4).item())
for epoch in range(100):
    for x, y in zip(x_list, y_list):
        los = loss(x, y)
        los.backward()
        print("\tgrad:", f"{x:.2f}", f"{y:.2f}", f"{w.grad.item():.2f}")
        w.data = w.data - 0.01*w.grad.data
        w.grad.data.zero_()
    print("progress:", epoch, los.item())
print("predict (after training):", 4,forward(4).item())
```

解决方法：张量进行运算前，将参与计算的变量强制转换为张量，同时，发生了“get float”的报错，将 tensor 后方的参数改为带有小数点的 float 类型即可。同时引入了一种更安全的函数 torch.no_grad，指明这里不需要进行梯度的计算，只是调用了 grad 的数值。

```python
import torch

x_list = [1.0, 2.0, 3.0]  # 输入数据
y_list = [2.0, 4.0, 6.0]  # 目标数据
w = torch.tensor([1.0], requires_grad=True)  # 初始化权重，且允许梯度累积

def forward(x):
    return x * w  # 定义前向传播

def loss(y, y_pred):
    return (y - y_pred) ** 2  # 定义损失函数

print("predict (before training):", 4, forward(torch.tensor([4.])).item())

# 训练过程
for epoch in range(100):
    for x_val, y_val in zip(x_list, y_list):
        x = torch.tensor([x_val])  # 转换为张量
        y = torch.tensor([y_val])  # 转换为张量
        l = loss(y, forward(x))  # 计算损失
        l.backward()  # 反向传播计算梯度
        print("\tgrad:", x_val, y_val, w.grad.item())
        with torch.no_grad():  # 更新权重
            w -= 0.01 * w.grad
        w.grad.zero_()  # 清零梯度，为下一次迭代准备
    print("progress:", epoch, l.item())

print("predict (after training):", 4, forward(torch.tensor([4.])).item())
```

二、反向传播：对于 requires_grad 的理解，无法理解为什么老师给出的代码中 backward 会是直接对 w 求导，而不是其他的数值，即 backward 如何锁定自变量

解决：实际上，l 调用 backward 的时候，会对所有的 requires_grad=True 的张量求导，并分别存储在相应张量的.grad 中

三、线性回归：出现警告：UserWarning: size_average and reduce args will be deprecated, please use reduction='sum' instead, warnings. warn (warning.format(ret))

解决：**torch.nn.MSELoss(size_average=False)**其中括号内的参数已被弃用，新版本应该使用 reduction= ‘sum’，若为求平均则使用 reduction= ‘mean’

四、线性回归：为什么这里只能是 forward，而不能是其他函数名，且在后续中未出现 forward 函数，而是直接的由 model=LinearModel()，调用 model(x)，便可以调用 forward

解决：因为在继承的父类中，有一个__call__函数，会自动调用名为 forward()的函数，所以命名不可改变。且__call__函数会得到参数*args,为一个列表，在 model(x)中会把 x 带入到自动调用的 forward 中。

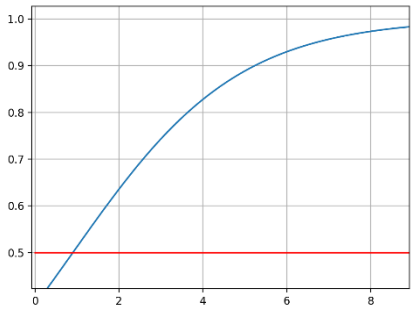
```
class LinearModel(torch.nn.Module):
    def __init__(self):
        super(LinearModel, self).__init__()
        self.linear = torch.nn.Linear(1, 1)

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

逻辑斯蒂回归：实际上是用来二分类的，利用了 sigmoid 进行非线性转化，loss 运用了 BCEloss 函数来计算，公式为

Loss Function for Binary Classification

$$loss = -(y \log \hat{y} + (1 - y) \log(1 - \hat{y}))$$



绘图时运用到 np.linspace(0, 10, 200)，表示的是生成一个 0 到 10 之间的两百个数据，构成一个列表，再用 x_t =torch.Tensor(x).view((a, b))，作用是得到一个 a*b 的矩阵,与 model 中的 linear 对应即可。含有 c= ‘r’ 的一处是用于生成一个参照线，常用于二分类中，

```
x = np.linspace(0, 10, 200)
x_t = torch.Tensor(x).view((200, 1))
y_t = model(x_t)
y = y_t.data.numpy()
plt.plot(x, y)
plt.plot([0, 10], [0.5, 0.5], c='r')
plt.grid()
plt.show()
```

同时更深入的进行了向量的运算，可以优化计算速度

$$\begin{bmatrix}\hat{y}^{(1)}\\ \vdots \\ \hat{y}^{(N)}\end{bmatrix} = \begin{bmatrix}\sigma(z^{(1)})\\ \vdots \\ \sigma(z^{(N)})\end{bmatrix} = \sigma\left(\begin{bmatrix}z^{(1)}\\ \vdots \\ z^{(N)}\end{bmatrix}\right)$$

$$z^{(1)} = \begin{bmatrix}x_1^{(1)} & \cdots & x_8^{(1)}\end{bmatrix}\begin{bmatrix}\omega_1\\ \vdots \\ \omega_8\end{bmatrix} + b$$

$$\vdots$$

$$z^{(N)} = \begin{bmatrix}x_1^{(N)} & \cdots & x_8^{(N)}\end{bmatrix}\begin{bmatrix}\omega_1\\ \vdots \\ \omega_8\end{bmatrix} + b$$

```
class Model(torch.nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.linear = torch.nn.Linear(8, 1)
        self.sigmoid = torch.nn.Sigmoid()

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

model = Model()
```

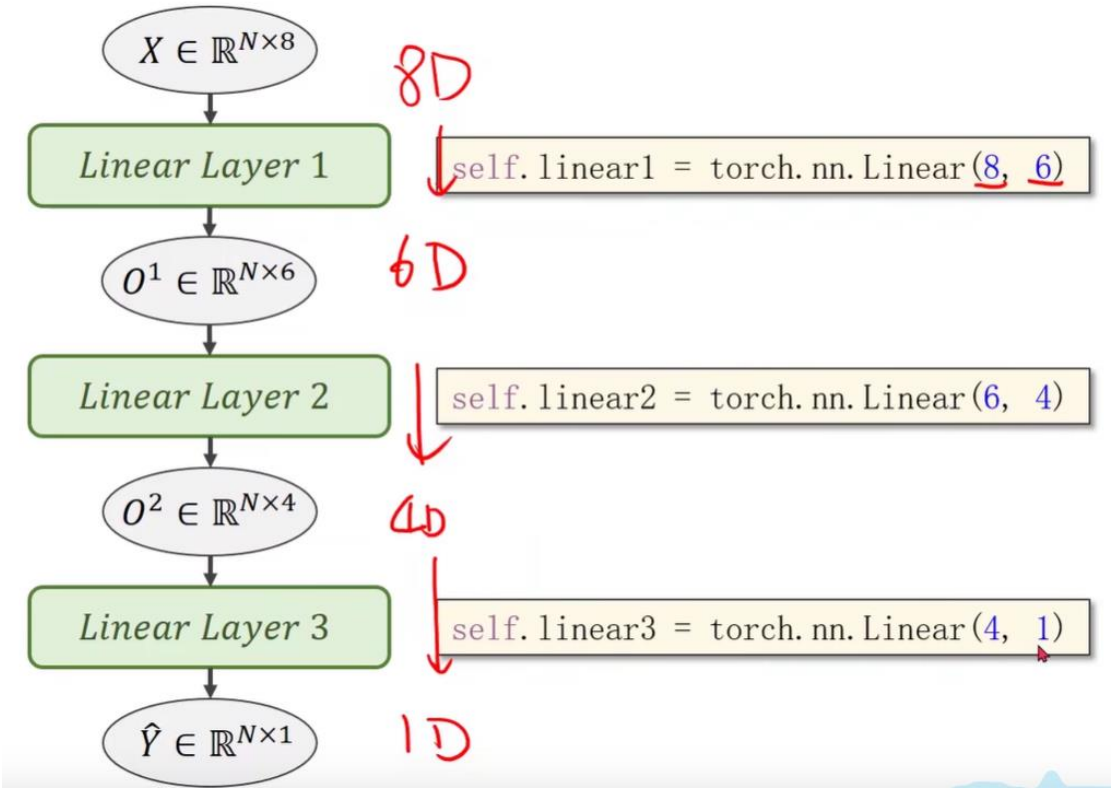
$$\begin{bmatrix}z^{(1)}\\ \vdots \\ z^{(N)}\end{bmatrix} = \begin{bmatrix}x_1^{(1)} & \cdots & x_8^{(1)}\\ \vdots & \ddots & \vdots \\ x_1^{(N)} & \cdots & x_8^{(N)}\end{bmatrix}\begin{bmatrix}\omega_1\\ \vdots \\ \omega_8\end{bmatrix} + \begin{bmatrix}b\\ \vdots \\ b\end{bmatrix}$$

$N \times 1$ $N \times 8$ 8×1 $N \times 1$

五、多维特征值的输入：每一层叠加起来，就是多层的神经网络了，每层的矩阵大小，大者可以让学习能力上升，但是过强的学习能力会学入训练集中的噪声，导致训练集的 loss 变小时，test 内若是计算了的 loss 却会变大，所以需要一定的把控。

```
class LogisticRegression(torch.nn.Module):
    def __init__(self):
        super(LogisticRegression, self).__init__()
        self.linear1 = torch.nn.Linear(8, 6)# 使得神经网络的复杂程度上升，提高性能
        self.linear2 = torch.nn.Linear(6, 4)
        self.linear3 = torch.nn.Linear(4, 1)
        self.activate = torch.nn.ReLU()# 两种非线性处理的方法
        self.sigmoid = torch.nn.Sigmoid()
```

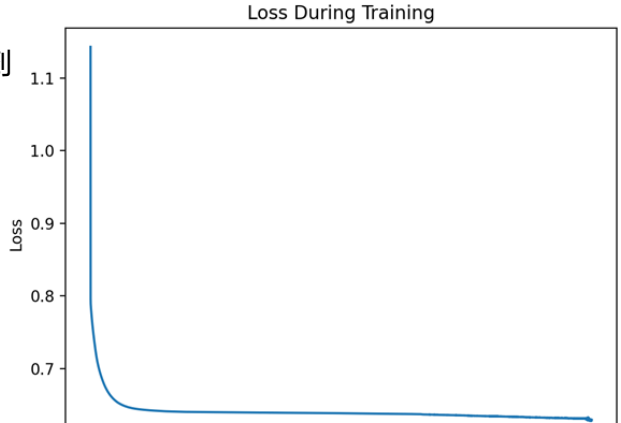
如下图，为维度转化的表示




```
xy = np.loadtxt('diabetes_data.csv', delimiter=',', dtype=np.float32)
x_data = torch.from_numpy(xy[:,:-1])
y_data = torch.from_numpy(xy[:, [-1]])
```

六、糖尿病模型的示例：delimiter 表示分隔符，类似于 python 基础中的 split(‘,’)，dtype 表示数据类型，这里使用 np.float32，精度足够用了便可。

下文中的数据读取,x_data 一行中，[:, :-1]，类似于切片，中括号内以逗号分隔，前一个表示行，两处不填默认为从开始到结束，后面的则是从头到最后一个，左闭右开。y_data 一行中。[-1]表示只取最后一列，也就是 y_data 在这里只取了每一行的最后一列。两者的数据类型为张量。运用⑥中模型得到



七、加载数据集：Dataset 为父类，但是是一个抽象类，不能直接构造实例，在这里我们自己设立一个数据集 DataSet，并建立魔法方法__getitem__等（magic function），其中的 shape 是返回一个元组，其中两个数值分别是行数与列数，在这里也就用 len 来接收了行数。

```
class DataSet(Dataset):
    def __init__(self, filepath):
        xy = np.loadtxt(filepath, delimiter=',', dtype=np.float32)
        self.x_data = torch.from_numpy(xy[:, :-1])
        self.y_data = torch.from_numpy(xy[:, [-1]])
        self.len = xy.shape[0]

    def __getitem__(self, idx):
        return self.x_data[idx], self.y_data[idx]

    def __len__(self):
        return self.len
```

魔法方法的含义是不需要直接调用他们，他们会自动的在需要的时候运行，命名上又着双边下划线的特殊规定，其他的魔法方法还有：

__init__(self, [...]): 构造器，创建并初始化类的新实例时调用。

__del__(self): 析构器，实例被销毁时调用。

__str__(self): 当实例被转换为字符串时调用，如使用 str()函数或 print()函数。

__len__(self): 当使用 len()函数获取对象的长度时调用。

__getitem__(self, key): 使得实例可以使用 self[key]的方式进行索引。

__setitem__(self, key, value): 使得实例可以使用 self[key] = value 的方式设置值。

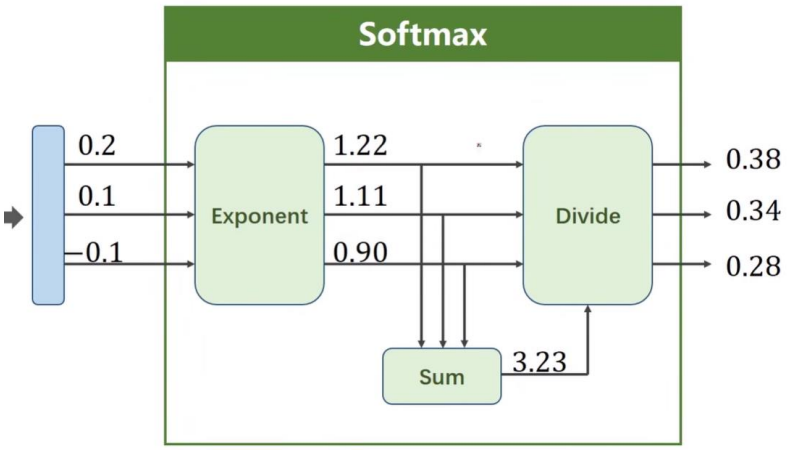
__add__(self, other): 定义加法行为。

__iter__(self): 返回迭代器，允许对象被迭代（比如在 for 循环中）

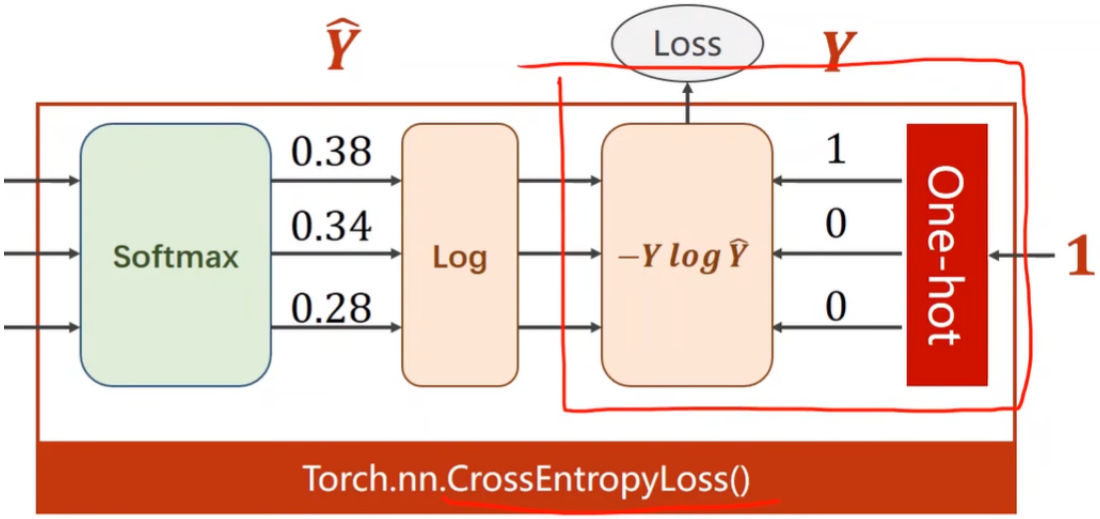
八、多分类问题的 loss 部分：不同于之前的二分类，这里最后的输出结果除去 0 与 1 会有多种，而不脱离的是概率。我们期望各种类别的可能加和为 1，于是之前的 sigmoid 不再适用，他的公式：无法保证这一点

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

开始使用另外的函数，softmax 能够实现输出值加和为 1，达到数据映射结果符合概率的一般规律的目的



交叉熵函数 torch.nn.CrossEntropyLoss，可以较好的解决上述问题，其中包含了 softmax，这也是为什么在这里我们不再对 Net 中的 forward 的最后返回值进行激活。



九、多分类问题里的部分函数：（训练集以 mnist 为例）

1）import transforms 后，Compose 函数用于构筑一个转换器，ToTensor 用于将图像转化为一个 Tensor 类型，Normalize 是用于标准化的，两个括号内的数字分别表示第一层通道（由于这里是灰度图所以只有一层，如果是彩色会有 RGB 三层）的均值和标准差，可以将让数据符合 N(0,1)的正态分布，这样的分布有利于机器的学习。

```
transformer = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
```

2）右图中，是在获取训练集和完成程序内对训练集的读取。from torch.utils.data import datasets 后，可用 MNIST 类，获得 mnist 的数据内容。train 表示是否为训练集；loader 中，shuffle 表示 Batch 打乱顺序后再创建 mini-Batch，让训练效果更好。batch_size 表示一份 mini-Batch 的数据量。num_workers 为以多少核进行运行。

```
train_data = datasets.MNIST(
    root='../datasets/mnist',
    train=True,
    transform=transformer,
    download=True,
)
train_loader = DataLoader(
    dataset=train_data,
    shuffle=True,
    batch_size=64,
    num_workers=4,
)
```

3）问题：右图中的亮色标出的 sum()函数，显示bool类型没有该函数，但是运行结果显示可以正常的调用。

```
def test():
    correct = 0
    total=0
    for data in test_loader:
        inputs, labels= data
        outputs = model(inputs)
        total += labels.size(0)
        i, predict = torch.max(outputs.data, dim=1)
        correct += (predict == labels).sum().item()
        print("AC rate: %", (100*correct/total))
```

因为这里，data 可以传递出两个 Tensor 量，分别是一组特征值组成的张量和一组标签构成的张量，由 Net 的实例 model 调用__call__函数调用 forward 函数，得到预测值，最后返回的是一组刚被线性处理完的张量（可以知道这个张量内的各项和其实是不为 1 的，因为最后进行的是线性变化，而没有经过 softmax 等函数的转化。但是后续 torch.max 也只是寻找最大值，softmax 也是一个单调函数，所以这里不需要进一步转化了。）

回到最开始的问题，bool 却有 sum 函数，因为这里的 predict 和 labels 都是 N*1 的张量，进行“==“的判别后也会生成一个张量，在对应的位置得到对应的 bool 值，True 或 False，也就是 0 和 1，接着调用 sum 对 0 和 1 进行加和是被允许的，返回值仍然是一个 Tensor，但只有单一元素了（就是总和），用 item 提取出来作为预测正确的次数。

label.size 与之前提到过的 shape 相似，他们返回的都是一个元组，有两个元素即张量的行数与列数，这里的 size(0)也就是调出了行数，作为预测的总次数，用作最后计算的分母。

4）Net 类：其中的 784 也就是 mnist 里面的每个 28*28 手写数字图像对应的像素个数，

```
class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.linear1 = torch.nn.Linear(784, 512)
        self.linear2 = torch.nn.Linear(512, 256)
        self.linear3 = torch.nn.Linear(256, 128)
        self.linear4 = torch.nn.Linear(128, 64)
        self.linear5 = torch.nn.Linear(64, 10)

    def forward(self, x):
        x = x.view(-1, 784)
        x = F.relu(self.linear1(x))
        x = F.relu(self.linear2(x))
        x = F.relu(self.linear3(x))
        x = F.relu(self.linear4(x))
        return self.linear5(x)
```

最后返回的为 10，因为一共 10 个数字。下面的 forward()函数中，x.view()起到一个转换的作用，将 inputs（实际上是经过前面提到的 transform 转换的图像，转换后得到的是 1*28*28 的张量）重塑为一个 size 为（1， 784）的张量，view 中参数 784 表示我们希望第二维度（就是列数，dim=1）的大小为 784，-1 表示自动计算原来张量的元素个数，然后计算重塑后张量的第一维度的大小为原来元素个数除以 784 得到第一维度的大小。这样的自动计算可以简洁代码。后面的非线性运算 relu 可以加快运算效率，最后返回的是一个经过线性运算的数值，因为 train 中的 CrossEntrophyLoss 会自动进行 log-softmax，而 test 中在 3）已作出解释。

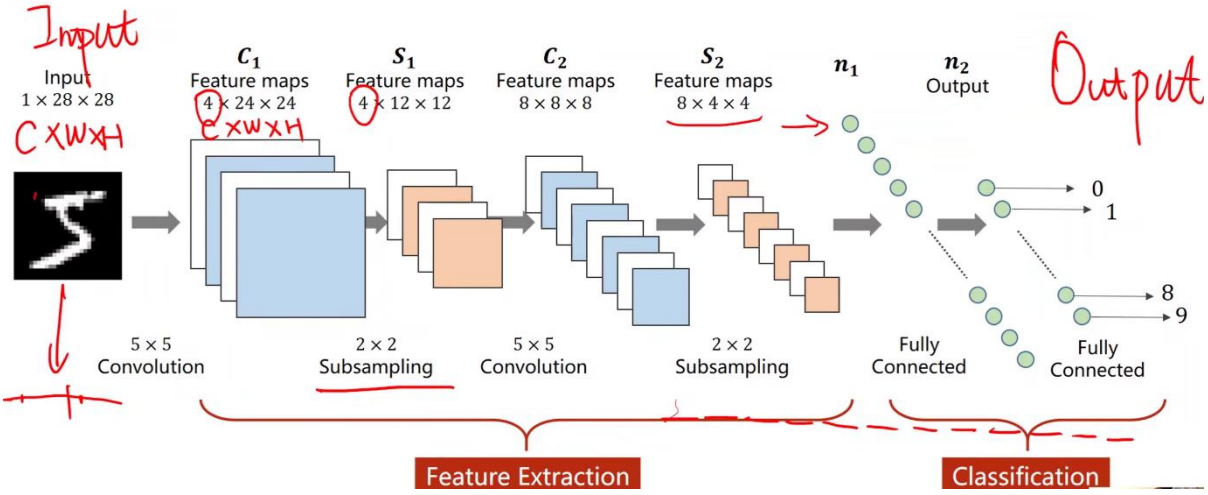
5）图示为损失函数和优化器的构建，以及实例的创建，这里主要讲 momentum，原理是利用上一轮的 grad 对这一轮产生一定的影响，其比率为 0.5，即上一轮权重的变化的 0.5 倍会在这一轮参与计算。作用是帮助跨过鞍点，克服局部最优的问题，也可以加快学习的效率。SGD 表示随机梯度下降，momentum 便是加速梯度下降。

```
model = Net()
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
```

十一、卷积神经网络：

1）通道数：理解为图是由几个层合成的，电脑中彩色图由 RGB 三色也就是三个图层合成，那么通道数（Channel,用 C 表示）也就是 3

2）卷积的过程：大致的流程如下，W 和 H 表示图像的宽和高；其中的 subsampling 表示下采样，可以达到减少计算量的目的。

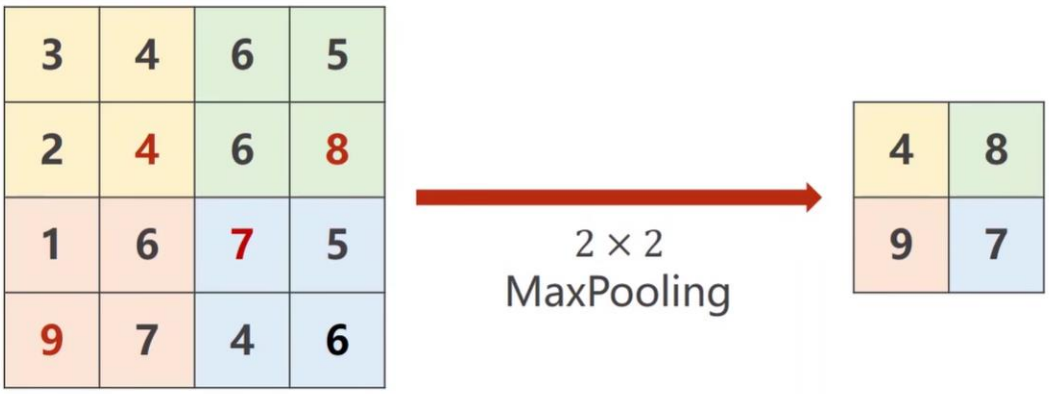


3）神经网络的相关函数：其中的 Conv2d 表示一个卷积核，大小为 5*5，前两个数据分别表示输入的通道数和希望的输出通道数。

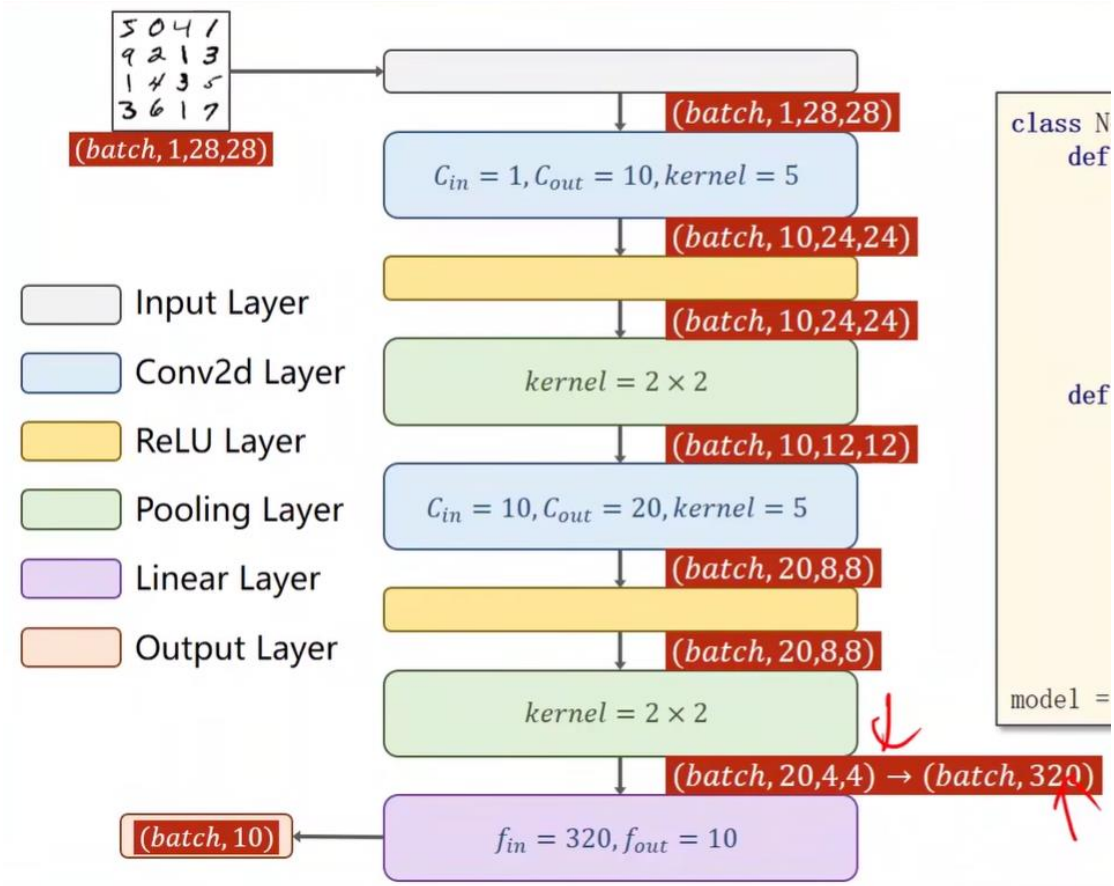
```
class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = torch.nn.Conv2d(1, 10,kernel_size=5)
        self.conv2 = torch.nn.Conv2d(10,20,kernel_size=5)
        self.pooling=torch.nn.MaxPool2d(2)
        self.func = torch.nn.Linear(320,10)

    def forward(self,x):
        batch_size=x.size(0)
        x=F.relu(self.pooling(self.conv1(x)))
        x = F.relu(self.pooling(self.conv2(x)))
        x=x.view(batch_size, -1)
        x=self.func(x)
        return x
```

MaxPool2d 表示池化，括号内的参数代表着一个 2*2 的窗口，会将张量分为若干个 2*2 的部分，然后取其中的最大值以原先的相对位置拼接为新生成张量。（如下图的运算逻辑）

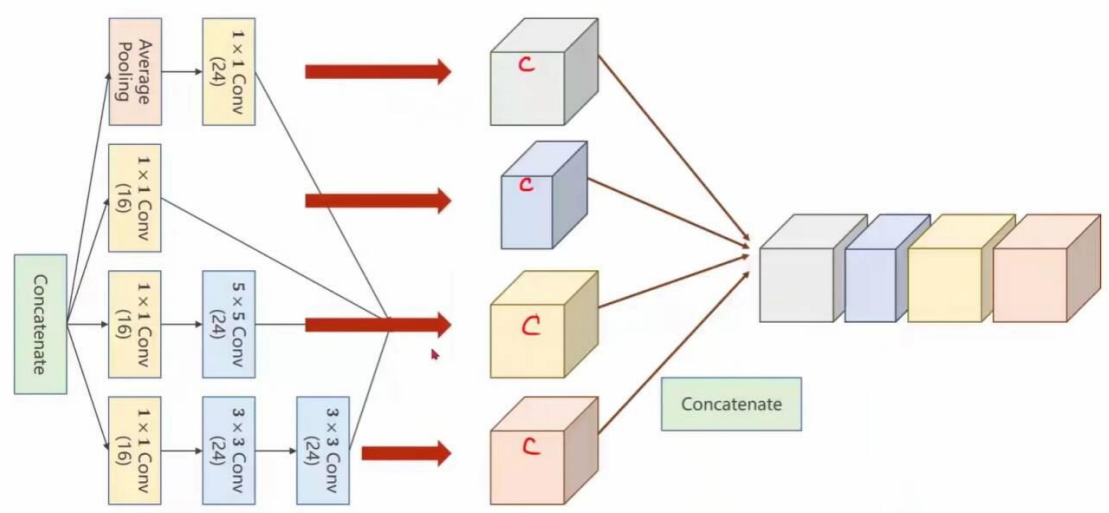


在这里的每一个最开始单通道图都会被加工为 20 个通道，最后一份 mini-batch 中也就是有 batch_size 个 20*4*4，通过 x.view 的展开来得到一个 batch_size 行，不改变总数据量前提下的若干列组的张量，来符合 Linear 函数的需要。



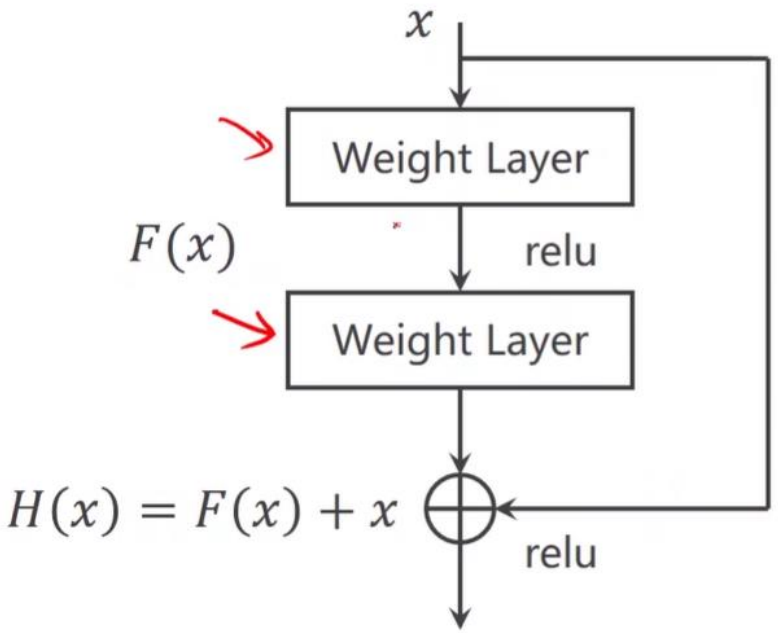
十二、卷积神经网络（深入）

由于超参数不好选择，例如前文提到的 kernel_size，以及是否进行下取样等；我们并不知道参数所取的数值和部分操作到最后是否真的利于训练，于是可以对卷积神经网络进行多个方案的卷积，最后使用 torch.cat 函数在 dim=1 的方向（通道方向）上进行合并，得到一个多种卷积方式拼接起来的结果。



十三、梯度消失：

由于 backward 的过程中多个绝对值小于 1 的导数值相乘，得到的最后结果被计算机作为 0 处理，就会导致类似于鞍点的麻烦，增加学习耗时。可以利用右图的思路，在正常的进行了权重层的卷积运算与 relu 的非线性计算后，原先的量直接再进行一次加和，可以有效的避免最后导数值为 0，因为有 1 的加



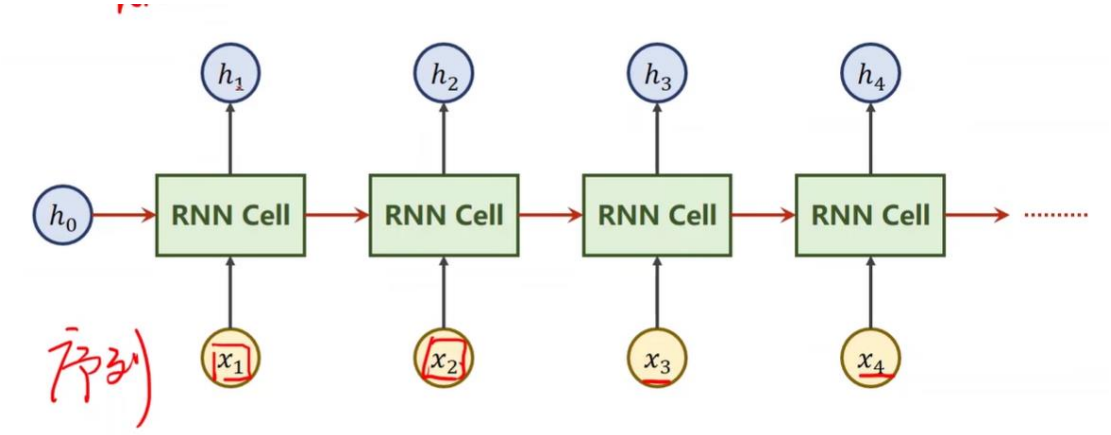
持。在使用代码实现时，需要注意将运算后的量与原先的 x 可以加和，要做到不改变它的 C、W、H。代码实现如下，可以看到 conv1 和 2 对应的参数中通道的输入输出完全一致，padding 也弥补了 kernel_size=3 会带来的 W 和 H 的变化问题。

```
class ResidualBlock(torch.nn.Module):
    def __init__(self, channels):
        super(ResidualBlock, self).__init__()
        self.channels = channels
        self.conv1 = torch.nn.Conv2d(channels, channels, kernel_size=3, padding=1)
        self.conv2 = torch.nn.Conv2d(channels, channels, kernel_size=3, padding=1)

    def forward(self,x):
        y=F.relu(self.conv1(x))
        y=self.conv2(y)
        return F.relu(x+y)
```

十四、循环神经网络：

基本的运行原理如图，需要指出的是这些 h_n 实际上是传递给了下一次的 RNN Cell，以此达到一种循环利用的感觉；传入的{x_n}是一个序列



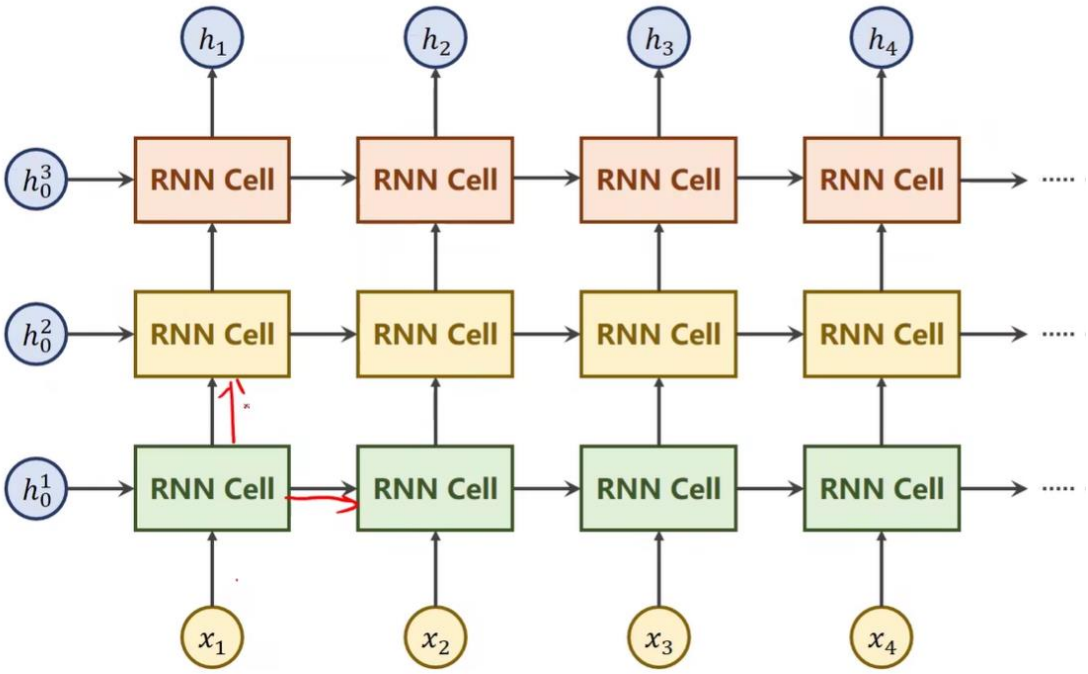
代码部分：其中的 input_size 是每一个 x 的维度，hidden_size 是每一个 h 的维度，

```python
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers=1):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size)
        out, _ = self.rnn(x, h0)
        out = self.fc(out[:, -1, :])

        return out
```

num_layer 表示的是层数，多层则会达到以下的效果。会带来泛化能力的提高，也会带来训练时间的加长以及潜在的梯度消失或爆炸问题，因为层数过多。



h0 为最初导入的 h 初始值，在这里 torch.zeros 表示生成一个括号内参数那样大小的零构成的张量赋给 h0。self.rnn(x,h0)返回的第一个值是（batch_size, seqlen, hidden_size）的张量，相应位置由相应的 h_n 构成。