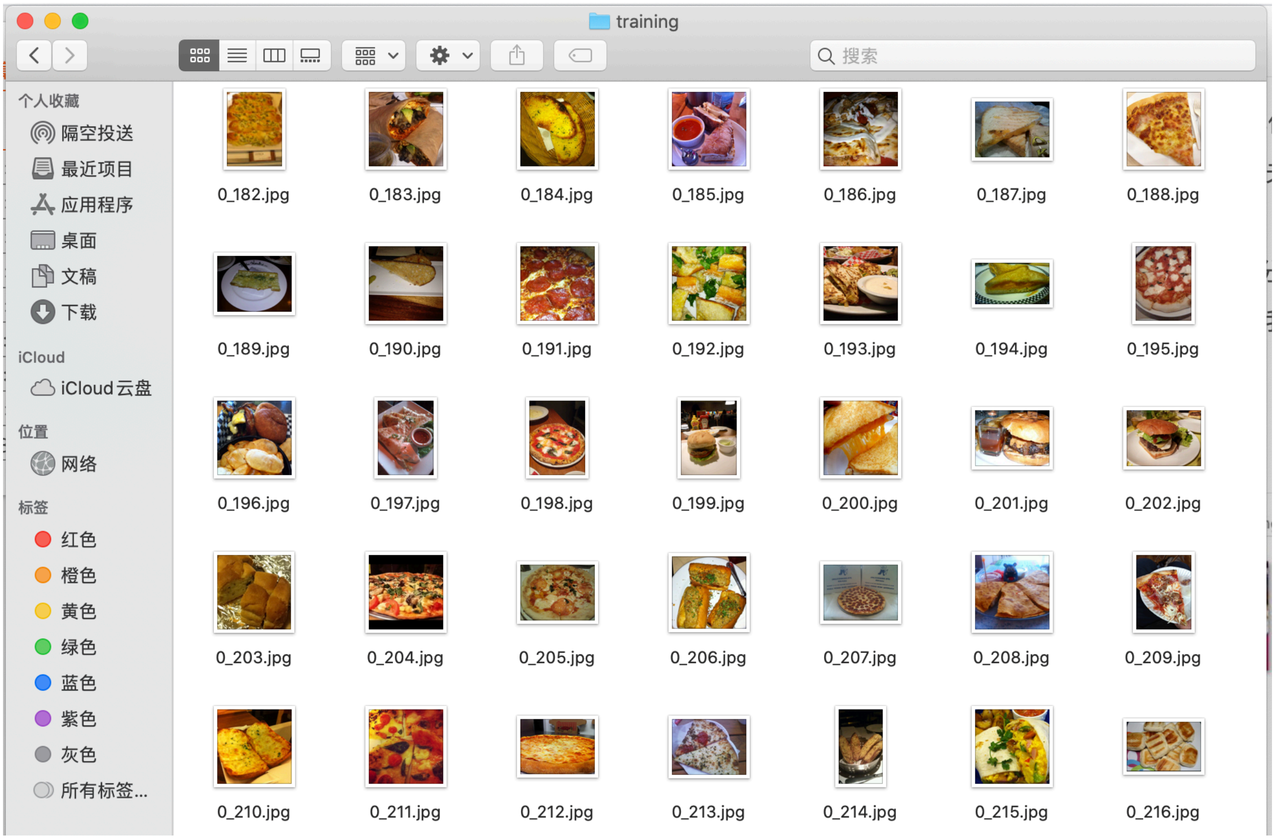
一、数据预处理

本次实验的主要任务是将图片分类，总共为11类，图片格式是为类别_图片序号，所以只要字符串截取一下就行。



主要用到DataSet和DataLoader。

Dataset是一个包装类，用来将数据包装为Dataset类，我们只需要自己写一个类然后继承Dataset类，然后传入DataLoader中，我们再使用DataLoader这个类来更加快捷的对数据进行操作。

当我们集成了一个Dataset类之后，我们需要重写 **len** 方法，该方法提供了dataset的大小； **getitem** 方法， 该方法支持从 0 到 len(self)的索引。

```
def read_file(path, flag):
    """
    读取文件目录里的内容
    :param path: 文件夹位置
    :param flag: 1训练集或验证集 0测试集
    """
    image_dir = os.listdir(path)
    x = np.zeros((len(image_dir), 128, 128, 3), dtype=np.uint8)
    y = np.zeros(len(image_dir))

    for i, file in enumerate(image_dir):
```

```python
        img = cv2.imread(os.path.join(path, file))
        x[i, :, :, :] = cv2.resize(img, (128, 128)) # 将图片大小变为128*128
        if flag:
            y[i] = file.split('_')[0]

    if flag:
        return x, y
    else:
        return x
```

```python
class ImgDataset(Dataset):
    """
    实现对数据的封装
    """
    def __init__(self, x, y=None, transform=None):
        self.x = x
        self.y = y
        if y is not None:
            self.y = torch.LongTensor(y)
        self.transform = transform

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

    def __getitem__(self, index):
        res_x = self.x[index]
        if self.transform is not None:
            res_x = self.transform(res_x)
        if self.y is not None:
            res_y = self.y[index]
            return res_x, res_y
        else:
            return res_x


train_transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.RandomHorizontalFlip(), # 随机水平翻转图片
    transforms.RandomRotation(15), # 随机旋转图片15度
    transforms.ToTensor() # 将图片变为Tensor [H, W, C]-->[C, H, W]
])

test_transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.ToTensor()
])
```

二、模型建立

2.1 5层卷积层+5层池化层+3层全连接层（原模型）

本次实验采用CNN卷积神经网络，网络结构为5层卷积层+5层池化层+3层全连接层。

注意连到全连接层时，需要将tensor展开，从[n, 512, 4, 4]->[n, 512 * 4 * 4]，n为batch_size。

```
class Classifier1(nn.Module):
    """
    构建神经网络1：5层卷积+5层池化+3层全连接
    """
    def __init__(self):
        super(Classifier1, self).__init__()

        # torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride,
padding)
        # torch.nn.MaxPool2d(kernel_size, stride, padding)
        self.cnn = nn.Sequential(           # input: 3 * 128 * 128
            # 卷积层1
            nn.Conv2d(3, 64, 3, 1, 1),        # output: 64 * 128 * 128
            nn.BatchNorm2d(64), # 归一化处理，可以使每一个batch的分布都在高斯分布附近，
这样可以使用更大的学习率，加快训练速度
            nn.ReLU(),
            nn.MaxPool2d(2, 2, 0),            # output: 64 * 64 * 64

            # 卷积层2
            nn.Conv2d(64, 128, 3, 1, 1),      # output: 128 * 64 * 64
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(2, 2, 0),            # output: 128 * 32 * 32

             # 卷积层3
            nn.Conv2d(128, 256, 3, 1, 1),     # output: 256 * 32 * 32
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(2, 2, 0),            # output: 256 * 16 * 16

            # 卷积层4
            nn.Conv2d(256, 512, 3, 1, 1),     # output: 512 * 16 * 16
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.MaxPool2d(2, 2, 0),            # output: 512 * 8 * 8

            # 卷积层5
```

```
            nn.Conv2d(512, 512, 3, 1, 1),    # output: 512 * 8 * 8
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.MaxPool2d(2, 2, 0)            # output: 512 * 4 * 4
        )

        self.fc = nn.Sequential(
            nn.Linear(512 * 4 * 4, 1024), # 全连接层
            nn.ReLU(),
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Linear(512, 11)
        )

    def forward(self, x):
        cnn_out = self.cnn(x)
        flatten = cnn_out.view(cnn_out.size()[0], -1) # 将Tensor展开
        return self.fc(flatten)
```

2.2 3层卷积层+3层池化层+3层全连接层（深度减半、参数量与原模型相当的模型）

由于作业说明中有该要求，所以还定义了另外两种网络结构。

Score - report.pdf

1. 請說明你實作的 CNN 模型，其模型架構、訓練參數量和準確率為何？(1%)
2. 請實作與第一題接近的參數量，但 CNN 深度（CNN 層數）減半的模型，並說明其模型架構、訓練參數量和準確率為何？(1%)
3. 請實作與第一題接近的參數量，簡單的 DNN 模型，同時也說明其模型架構、訓練參數和準確率為何？(1%)
4. 請說明由 1 ~ 3 題的實驗中你觀察到了什麼？(1%)
5. 請嘗試 data normalization 及 data augmentation，說明實作方法並且說明實行前後對準確率有什麼樣的影響？(1%)
6. 觀察答錯的圖片中，哪些 class 彼此間容易用混？[繪出 confusion matrix 分析](1%)

```
class Classifier2(nn.Module):
    """
    构建神经网络2: 3层卷积+3层池化+3层全连接
    """
    def __init__(self):
        super(Classifier2, self).__init__()
```

```
        # torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride, 
padding)
        # torch.nn.MaxPool2d(kernel_size, stride, padding)
        self.cnn = nn.Sequential(           # input: 3 * 128 * 128
            # 卷积层1
            nn.Conv2d(3, 64, 3, 1, 1),       # output: 64 * 128 * 128
            nn.BatchNorm2d(64), # 归一化处理
            nn.ReLU(),
            nn.MaxPool2d(4, 4, 0),           # output: 64 * 32 * 32

            # 卷积层2
            nn.Conv2d(64, 512, 3, 1, 1),     # output: 512 * 32 * 32
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.MaxPool2d(4, 4, 0),           # output: 512 * 8 * 8

             # 卷积层3
            nn.Conv2d(512, 512, 3, 1, 1),    # output: 512 * 8 * 8
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.MaxPool2d(2, 2, 0),           # output: 512 * 4 * 4
        )

        self.fc = nn.Sequential(
            nn.Linear(512 * 4 * 4, 1024), # 全连接层
            nn.ReLU(),
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Linear(512, 11)
        )

    def forward(self, x):
        cnn_out = self.cnn(x)
        flatten = cnn_out.view(cnn_out.size()[0], -1) # 将Tensor展开
        return self.fc(flatten)
```

2.3 5层卷积层+5层池化层+2层全连接层（简单DNN）

```
class Classifier3(nn.Module):
    """
    构建神经网络3：5层卷积+5层池化+2层全连接
    """
    def __init__(self):
        super(Classifier3, self).__init__()
```

```python
        # torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride, 
padding)
        # torch.nn.MaxPool2d(kernel_size, stride, padding)
        self.cnn = nn.Sequential(           # input: 3 * 128 * 128
            # 卷积层1
            nn.Conv2d(3, 64, 3, 1, 1),        # output: 64 * 128 * 128
            nn.BatchNorm2d(64), # 归一化处理
            nn.ReLU(),
            nn.MaxPool2d(2, 2, 0),            # output: 64 * 64 * 64

            # 卷积层2
            nn.Conv2d(64, 128, 3, 1, 1),      # output: 128 * 64 * 64
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(2, 2, 0),            # output: 128 * 32 * 32

             # 卷积层3
            nn.Conv2d(128, 256, 3, 1, 1),     # output: 256 * 32 * 32
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(2, 2, 0),            # output: 256 * 16 * 16

            # 卷积层4
            nn.Conv2d(256, 512, 3, 1, 1),     # output: 512 * 16 * 16
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.MaxPool2d(2, 2, 0),            # output: 512 * 8 * 8

            # 卷积层5
            nn.Conv2d(512, 512, 3, 1, 1),     # output: 512 * 8 * 8
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.MaxPool2d(2, 2, 0)             # output: 512 * 4 * 4
        )

        self.fc = nn.Sequential(
            nn.Linear(512 * 4 * 4, 1024), # 全连接层
            nn.ReLU(),
            nn.Linear(1024, 11)
        )

    def forward(self, x):
        cnn_out = self.cnn(x)
        flatten = cnn_out.view(cnn_out.size()[0], -1) # 将Tensor展开
        return self.fc(flatten)
```

三、模型训练

本模型采用交叉熵作为损失函数，Adam为优化器，总共训练了30epoch。

```
def train_model(train_loader, val_loader, train_len, val_len):
    """
    模型训练
    """
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    # 构建神经网络1：5层卷积+5层池化+3层全连接
    # model = Classifier1().to(device)

    # 构建神经网络2：3层卷积+3层池化+3层全连接
    model = Classifier2().to(device)

    # 构建神经网络3：5层卷积+5层池化+2层全连接
    # model = Classifier3().to(device)

    loss = nn.CrossEntropyLoss() # 使用交叉熵损失函数
    optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
    epochs = 30

    for epoch in range(epochs):
        epoch_start_time = time.time()
        train_acc = 0.0
        train_loss = 0.0
        val_acc = 0.0
        val_loss = 0.0

        # 保证BN层(Batch Normalization)用每一批数据的均值和方差，而对于Dropout层，随机
取一部分网络连接来训练更新参数
        model.train()
        for i, data in enumerate(train_loader):
            optimizer.zero_grad() # 清空梯度，否则会一直累加
            train_pred = model(data[0].to(device)) # data[0]: x data[1]: y
            batch_loss = loss(train_pred, data[1].to(device))
            batch_loss.backward()
            optimizer.step() # 更新参数

            # .data表示将Variable中的Tensor取出来
            # train_pred是(50, 11)的数据，np.argmax()返回最大值的索引，axis=1则是对行
进行，返回的索引正好就对应了标签，然后和y真实标签比较，则可得到分类正确的数量
            train_acc += np.sum(np.argmax(train_pred.cpu().data.numpy(),
axis=1) == data[1].numpy())
            train_loss += batch_loss.item()


        # 保证BN用全部训练数据的均值和方差，而对于Dropout层，利用到了所有网络连接
```

```
        model.eval()
        with torch.no_grad():
            for i, data in enumerate(val_loader):
                val_pred = model(data[0].to(device))
                batch_loss = loss(val_pred, data[1].to(device))

                val_acc += np.sum(np.argmax(val_pred.cpu().data.numpy(),
axis=1) == data[1].numpy())
                val_loss += batch_loss.item()


        print('[%03d/%03d] %2.2f sec(s) Train Acc: %3.6f Loss: %3.6f | Val Acc:
%3.6f loss: %3.6f' % \
             (epoch + 1, epochs, time.time() - epoch_start_time, \
              train_acc / train_len, train_loss / train_len, val_acc /
val_len,
              val_loss / val_len))

    return model
```

模型1训练结果：

```
[010/030] 54.66 sec(s) Train Acc: 0.627002 Loss: 0.021825 | Val Acc: 0.586297 loss: 0.025076
[011/030] 54.71 sec(s) Train Acc: 0.644740 Loss: 0.020693 | Val Acc: 0.562974 loss: 0.026767
[012/030] 54.91 sec(s) Train Acc: 0.662376 Loss: 0.019601 | Val Acc: 0.593003 loss: 0.024504
[013/030] 54.73 sec(s) Train Acc: 0.683560 Loss: 0.018413 | Val Acc: 0.617493 loss: 0.024406
[014/030] 54.72 sec(s) Train Acc: 0.701297 Loss: 0.017314 | Val Acc: 0.641983 loss: 0.021712
[015/030] 54.71 sec(s) Train Acc: 0.718123 Loss: 0.016390 | Val Acc: 0.611079 loss: 0.024078
[016/030] 54.80 sec(s) Train Acc: 0.727144 Loss: 0.015577 | Val Acc: 0.559475 loss: 0.028857
[017/030] 54.71 sec(s) Train Acc: 0.743057 Loss: 0.014897 | Val Acc: 0.612536 loss: 0.025222
[018/030] 54.77 sec(s) Train Acc: 0.754713 Loss: 0.014000 | Val Acc: 0.609621 loss: 0.027278
[019/030] 54.85 sec(s) Train Acc: 0.768498 Loss: 0.013130 | Val Acc: 0.604082 loss: 0.027779
[020/030] 54.76 sec(s) Train Acc: 0.785425 Loss: 0.012617 | Val Acc: 0.626822 loss: 0.026143
[021/030] 54.83 sec(s) Train Acc: 0.795054 Loss: 0.011745 | Val Acc: 0.637609 loss: 0.027243
[022/030] 54.70 sec(s) Train Acc: 0.814413 Loss: 0.011022 | Val Acc: 0.635277 loss: 0.025165
[023/030] 54.90 sec(s) Train Acc: 0.819785 Loss: 0.010286 | Val Acc: 0.659475 loss: 0.023837
[024/030] 54.80 sec(s) Train Acc: 0.830529 Loss: 0.009601 | Val Acc: 0.647813 loss: 0.026540
[025/030] 54.87 sec(s) Train Acc: 0.847963 Loss: 0.008765 | Val Acc: 0.664723 loss: 0.024129
[026/030] 54.78 sec(s) Train Acc: 0.858200 Loss: 0.008099 | Val Acc: 0.658601 loss: 0.024330
[027/030] 54.90 sec(s) Train Acc: 0.852118 Loss: 0.008258 | Val Acc: 0.646647 loss: 0.027492
[028/030] 54.89 sec(s) Train Acc: 0.872897 Loss: 0.007291 | Val Acc: 0.662391 loss: 0.026988
[029/030] 54.82 sec(s) Train Acc: 0.884857 Loss: 0.006622 | Val Acc: 0.656560 loss: 0.027906
[030/030] 54.83 sec(s) Train Acc: 0.892358 Loss: 0.006213 | Val Acc: 0.677551 loss: 0.027831
The number of parameters is %d 12833803
```

模型2训练结果：

```
[014/030] 40.21 sec(s) Train Acc: 0.747010 Loss: 0.014500 | Val Acc: 0.624781 loss: 0.025084
[015/030] 40.27 sec(s) Train Acc: 0.758869 Loss: 0.013803 | Val Acc: 0.607580 loss: 0.026644
[016/030] 40.28 sec(s) Train Acc: 0.768295 Loss: 0.013270 | Val Acc: 0.627697 loss: 0.024334
[017/030] 40.48 sec(s) Train Acc: 0.779343 Loss: 0.012502 | Val Acc: 0.595627 loss: 0.028033
[018/030] 40.37 sec(s) Train Acc: 0.799412 Loss: 0.011550 | Val Acc: 0.633819 loss: 0.025587
[019/030] 40.41 sec(s) Train Acc: 0.803365 Loss: 0.011134 | Val Acc: 0.602624 loss: 0.029757
[020/030] 40.39 sec(s) Train Acc: 0.820799 Loss: 0.010180 | Val Acc: 0.628571 loss: 0.027461
[021/030] 40.51 sec(s) Train Acc: 0.835901 Loss: 0.009303 | Val Acc: 0.651603 loss: 0.025309
[022/030] 40.47 sec(s) Train Acc: 0.844010 Loss: 0.009189 | Val Acc: 0.636152 loss: 0.028527
[023/030] 40.37 sec(s) Train Acc: 0.855666 Loss: 0.008249 | Val Acc: 0.649854 loss: 0.028465
[024/030] 40.50 sec(s) Train Acc: 0.864180 Loss: 0.007730 | Val Acc: 0.644315 loss: 0.029388
[025/030] 39.94 sec(s) Train Acc: 0.875025 Loss: 0.007277 | Val Acc: 0.669096 loss: 0.027035
[026/030] 39.45 sec(s) Train Acc: 0.886783 Loss: 0.006643 | Val Acc: 0.658017 loss: 0.028691
[027/030] 39.44 sec(s) Train Acc: 0.885465 Loss: 0.006581 | Val Acc: 0.669971 loss: 0.028205
[028/030] 40.91 sec(s) Train Acc: 0.896412 Loss: 0.006182 | Val Acc: 0.612536 loss: 0.035063
[029/030] 40.88 sec(s) Train Acc: 0.906142 Loss: 0.005413 | Val Acc: 0.620991 loss: 0.037128
[030/030] 41.07 sec(s) Train Acc: 0.906750 Loss: 0.005297 | Val Acc: 0.557726 loss: 0.048450
The number of parameters is %d 11579275
```

模型3训练结果：

```
[012/030] 53.89 sec(s) Train Acc: 0.664200 Loss: 0.019411 | Val Acc: 0.608746 loss: 0.023207
[013/030] 53.83 sec(s) Train Acc: 0.674843 Loss: 0.018914 | Val Acc: 0.544023 loss: 0.028995
[014/030] 53.91 sec(s) Train Acc: 0.695824 Loss: 0.017538 | Val Acc: 0.607289 loss: 0.024364
[015/030] 53.93 sec(s) Train Acc: 0.713562 Loss: 0.016707 | Val Acc: 0.579883 loss: 0.025975
[016/030] 53.84 sec(s) Train Acc: 0.725927 Loss: 0.015458 | Val Acc: 0.610496 loss: 0.023919
[017/030] 53.87 sec(s) Train Acc: 0.742550 Loss: 0.014876 | Val Acc: 0.469096 loss: 0.040474
[018/030] 53.83 sec(s) Train Acc: 0.752281 Loss: 0.014188 | Val Acc: 0.650437 loss: 0.023731
[019/030] 53.88 sec(s) Train Acc: 0.776100 Loss: 0.013214 | Val Acc: 0.649271 loss: 0.023670
[020/030] 54.11 sec(s) Train Acc: 0.785729 Loss: 0.012312 | Val Acc: 0.584257 loss: 0.028670
[021/030] 53.83 sec(s) Train Acc: 0.795966 Loss: 0.011760 | Val Acc: 0.655394 loss: 0.022604
[022/030] 54.01 sec(s) Train Acc: 0.813197 Loss: 0.010704 | Val Acc: 0.675219 loss: 0.022637
[023/030] 54.19 sec(s) Train Acc: 0.828806 Loss: 0.009897 | Val Acc: 0.688921 loss: 0.022180
[024/030] 53.99 sec(s) Train Acc: 0.840564 Loss: 0.009020 | Val Acc: 0.698542 loss: 0.021448
[025/030] 54.02 sec(s) Train Acc: 0.855666 Loss: 0.008428 | Val Acc: 0.682799 loss: 0.022019
[026/030] 54.08 sec(s) Train Acc: 0.863268 Loss: 0.007971 | Val Acc: 0.647813 loss: 0.026025
[027/030] 54.06 sec(s) Train Acc: 0.863471 Loss: 0.007775 | Val Acc: 0.601166 loss: 0.031481
[028/030] 54.14 sec(s) Train Acc: 0.874012 Loss: 0.007183 | Val Acc: 0.654227 loss: 0.027954
[029/030] 54.04 sec(s) Train Acc: 0.886580 Loss: 0.006387 | Val Acc: 0.634694 loss: 0.030733
[030/030] 53.91 sec(s) Train Acc: 0.902493 Loss: 0.005659 | Val Acc: 0.664431 loss: 0.029027
The number of parameters is %d 12314635
```

四、模型测试

```
def predict_model(test_loader, model):
    """
    模型预测
    """
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    model.eval()
    result = []
    with torch.no_grad():
        for i, data in enumerate(test_loader):
            test_pred = model(data.to(device))
            test_label = np.argmax(test_pred.cpu().data.numpy(), axis=1)
            for y in test_label:
                result.append(y)
    return result

def write_file(result):
    with open('result.csv', mode='w') as f:
```

```
        f.write('Id,Category\n')
        for i, label in enumerate(result):
            f.write('{},{}\n'.format(i, label))
```

部分API解析

torch.nn.Conv2d：

```
torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0,
dilation=1, groups=1, bias=True)
```

- in_channels(`int`) – 输入信号的通道
- out_channels(`int`) – 卷积产生的通道
- kerner_size(`int` or `tuple`) - 卷积核的尺寸
- stride(`int` or `tuple`, `optional`) - 卷积步长
- padding(`int` or `tuple`, `optional`) - 输入的每一条边补充0的层数
- dilation(`int` or `tuple`, `optional`) – 卷积核元素之间的间距
- groups(`int`, `optional`) – 从输入通道到输出通道的阻塞连接数
- bias(`bool`, `optional`) - 如果`bias=True`，添加偏置

torch.nn.BatchNorm2d：

在卷积神经网络的卷积层之后总会添加BatchNorm2d进行数据的归一化处理，这使得数据在进行Relu之前不会因为数据过大而导致网络性能的不稳定。

$$y = \frac{x - mean(x)}{\sqrt{Var(x)} + \epsilon} * \gamma + \beta$$

```
torch.nn.BatchNorm2d(num_features, eps=1e-05, momentum=0.1, affine=True)
```

- num_features： 来自期望输入的特征数，该期望输入的大小为`C来自输入大小(N,C,H,W)`
- eps： 为保证数值稳定性（分母不能趋近或取0）,给分母加上的值。默认为1e-5。
- momentum： 动态均值和动态方差所使用的动量。默认为0.1。
- affine： 一个布尔值，当设为true，给该层添加可学习的仿射变换参数。

torch.nn.MaxPool2d：

```
torch.nn.MaxPool2d(kernel_size, stride=None, padding=0, dilation=1,
return_indices=False, ceil_mode=False)
```

- kernel_size(`int` or `tuple`) - max pooling的窗口大小
- stride(`int` or `tuple`, `optional`) - max pooling的窗口移动的步长。默认值是`kernel_size`
- padding(`int` or `tuple`, `optional`) - 输入的每一条边补充0的层数
- dilation(`int` or `tuple`, `optional`) – 一个控制窗口中元素步幅的参数
- return_indices - 如果等于`True`，会返回输出最大值的序号，对于上采样操作会有帮助
- ceil_mode - 如果等于`True`，计算输出信号大小的时候，会使用向上取整，代替默认的向下取整的操作

torch.nn.Linear：

对输入数据做线性变换$y = Ax + b$，即全连接层。

```
torch.nn.Linear(in_features, out_features, bias=True)
```

- in_features - 每个输入样本的大小
- out_features - 每个输出样本的大小
- bias - 若设置为False，这层不会学习偏置。默认值：True