Part 2.

Section 1. 代码基本结构

本部分的实验中我使用了两种模型，分别是 LeNet 和 ResNet18 。代码文件结构如下：

- `models` 模块中有 `LeNet.py` 和 `ResNet.py` 两个文件，这两个文件中分别使用 `torch` 框架搭建了 LeNet 和 ResNet18 的结构。
- `init` 模块中主要包含初始化时使用的函数，包括设置随机种子、日志设置、数据预处理等。
- `dataset` 模块中实现了 `CharDataset` 作为数据集。
- `classifier` 模块中实现了类 `CNNClassifier` ，这个类有 `fit` 、`evaluate` 、`predict` 等方法。
- `fit_char.ipynb` 是主程序，包括加载数据集、训练模型、测试模型等。
- `data` 文件夹下保存着处理好的 `npz` 数据文件，`record` 保存每次训练产生的日志，`save` 保存最佳的模型参数。

下面挑选代码中最重要的一些类进行分析，为节省字数，代码中所有的注释都被删去，具体注释详见源码。

LeNet

```
class LeNet(nn.Module):
    def __init__(self, in_channels, out_features):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, 6, kernel_size=5)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv3 = nn.Conv2d(6, 16, kernel_size=5)
        self.pool4 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv5 = nn.Conv2d(16, 120, kernel_size=5)
        self.fc6 = nn.Linear(120, 84)
        self.fc7 = nn.Linear(84, out_features)

    def forward(self, x):
        x = self.pool2(F.relu(self.conv1(x)))
        x = self.pool4(F.relu(self.conv3(x)))
        x = F.relu(self.conv5(x))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc6(x))
        y = self.fc7(x)
        return y
```

原论文上的 `LeNet` $S2$ 池化层到 $C3$ 卷积层的使用了一个连接表来描述 $C3$ 层和前一层特征的依赖关系。考虑到电脑上有 `cuda` 可以加速计算（算力相对充足），代码中我没有使用连接表，而是让 $C3$ 层的每个卷积核都对上一层所有 channel 的特征进行卷积，从而简化了代码的编写。

ResBlock

```
class ResBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride):
        super(ResBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, 3, stride, 1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, 3, 1, 1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = None
        if stride != 1 or in_channels != out_channels:
            self.downsample = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, 1, stride, bias=False),
                nn.BatchNorm2d(out_channels),
            )

    def forward(self, x):
        identity = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        if self.downsample is not None:
            identity = self.downsample(x)
        out += identity
        out = self.relu(out)
        return out
```

`ResBlock` 是残差网络的基本结构，其恒等映射的思想是残差网络的核心。这部分代码对输入进行卷积和批量正则化，最后则再通过直连边加上原输入 `identity` ，最后经过 `relu` 输出，也就是期望模块去拟合 $h(x)-x$ 的部分。

考虑到如果输入和输出的特征通道数不同，则需要利用 1×1 的卷积对原输入进行下采样，使得与最后的张量形状相同，可以通过直连边相加。

ResNet18

```
class ResNet18(nn.Module):
    def __init__(self, in_channels, num_classes):
        super(ResNet18, self).__init__()
        self.in_channels = 64
        self.conv1 = nn.Conv2d(in_channels, 64, 7, 2, 3, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self._make_layer(64, 2, stride=1)
        self.layer2 = self._make_layer(128, 2, stride=2)
        self.layer3 = self._make_layer(256, 2, stride=2)
```

```
        self.layer4 = self._make_layer(512, 2, stride=2)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512, num_classes)

    def _make_layer(self, out_channels, blocks, stride):
        layers = []
        layers.append(ResBlock(self.in_channels, out_channels, stride))
        self.in_channels = out_channels
        for _ in range(1, blocks):
            layers.append(ResBlock(self.in_channels, out_channels, stride=1))
        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

`ResNet18` 利用上面提到的 `ResBlock` 进行搭建，残差网络就是将很多和残差块（残差单元）串联起来的比较深的网络。利用类中的 `_make_layer` 方法，在初始化的时候搭建了四层残差层，每一层中有 2 个残差块，每一个残差块中有两个卷积层，再加上网络首的卷积层和网络尾的全连接层，网络的深度达到了 18 。

CharDataset

```
class CharDataset(Dataset):
    def __init__(self, x, y, transform=None):
        self.x = x
        self.y = y
        self.transform = transform

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

    def __len__(self):
        return self.x.shape[0]
```

注意到 `CharDataset` 在初始化的时候传入了一个 `transform` 用于对样本进行变换处理，在代码中我直接调用了 `torchvision.transforms` 中的工具类，如使用 `Resize` 进行图片大小放缩，使用 `ToTensor` 将 `ndarray` 转化为张量并对像素值进行归一化。

由于原图都是 28×28 的灰度图，而 LeNet 的输入需要的是 32×32 的图片，ResNet18 通常的输入是 224×224 ，所以输入网络前这里利用到了 `Resize` 将图像进行放缩。

Section 2. 网络结构设计理解

LeNet

LeNet 的结构直接按照参考资料搭建：

1. 用 $\text{in\_channel} \times 6$ 的大小为 5×5 的卷积核，输出 6 通道的 28×28 大小的特征。
2. 经过 2×2 ，步长为 2 的最大池化层后，输出 6 通道的 14×14 的特征。
3. 用 6×16 的大小为 5×5 的卷积核，得到 16 通道 10×10 大小的特征。
4. 用 2×2，步长为 2 的最大池化层后，输出 16 通道的 5×5 的特征。
5. 用 16×120 的大小为 5×5 的卷积核，得到 120 通道 1×1 大小的特征。
6. 此时，将特征图拉平，则有120个像素点，经过输入神经元个数为 120 ，输出神经元个数为 84 的全连接层后，输出的长度变为 84 。
7. 再经过一个全连接层的计算，最终得到了长度为 out_features 的输出结果。

卷积神经网络一般由卷积层、汇聚层和全连接层构成，其核心思想就是使用离散卷积来代替全连接。一方面每一个卷积层都只和前一层的某局部窗口内的神经元连接，相比于全连接网络，连接数量大大减少，称为 **局部连接** 。另一方面，一个卷积核会对输入的所有通道的特征图进行卷积，可以理解为一个卷积核只捕捉输入数据中的一种特定局部特征，称为 **权重共享** 。

对于卷积核较小的层会关注小范围特征，卷积核较大的层会关注大范围特征，如果要提取出多种特征，就需要设置多个不同的卷积核。每次特征图输入下一层进行卷积，实际上都是在对多个通道的特征图进行特征融合。通过大小不同的卷积和的特征映射之后，能比较好地捕捉图的特征。

注意到 CNN 中还用到了一个称为池化层（汇聚层）的结构，其作用是进行特征选择，降低特征数量，从而减少参数数量。

卷积层虽然可以显著减少网络中连接的数量，但特征映射组中的神经元个数并没有显著减少。如果后面接一个分类器，分类器的输入维数依然很高，很容易出现过拟合。为了解决这个问题，可以在卷积层之后加上一个汇聚层，从而降低特征维数，避免过拟合。

一般来说常见的汇聚层有：最大汇聚、平均汇聚。可以将汇聚层理解为一种下采样。

ResNet18

ResNet18 同样按照参考资料中的描述搭建，可以看到一个残差单元就是两个 3×3 的卷积的结合加上直连边，具体的 ResNet18 通过堆积残差单元加深网络：

1. 包含了一个步长为 2 ，大小为 7×7 的卷积层，卷积层的输出通道数为 64 ，卷积层的输出经过批量归一化、`relu` 激活函数后，连接一个步长为 2 的 3×3 的最大汇聚层。
2. 两个残差单元，`output_channels=64` ，特征尺寸不变。
3. 两个残差单元，`output_channels=128` ，特征尺寸减半。

4. 两个残差单元，`output_channels=256`，特征尺寸减半。
5. 两个残差单元，`output_channels=512`，特征尺寸减半。
6. 包含了一个全局平均汇聚层，将特征图变为 1×1 的大小，最终经过全连接层计算出最后的输出。

注意到代码中使用了 `BatchNorm2d` 这个模块，也就是逐层归一化，通过对神经网络某些层的数据进行一个归一化，可以使网络更加容易训练：

1. 加速训练：Batch Normalization 可以加速训练过程，因为它有助于克服内部协变量偏移（Internal Covariate Shift）问题。内部协变量偏移是指神经网络在训练过程中每一层输入分布的改变，这可能导致训练变得非常困难。

   Batch Normalization通过规范化每一层的输入分布，有助于保持它们的均值接近0和方差接近1，从而加速了梯度下降的收敛（平滑优化地形）。
2. 提高模型稳定性：Batch Normalization 可以增加模型的稳定性，使模型对超参数的选择不太敏感。这使得模型更容易训练，避免了某些训练过程中出现的数值问题。
3. 减少过拟合：Batch Normalization 具有正则化的效果，可以减少过拟合。它在每个小批次中对每个特征通道进行标准化，类似于在训练中应用丢弃（Dropout）。
4. 允许使用更高的学习率：由于Batch Normalization 有稳定性和加速训练的效果，通常可以使用更高的学习率，从而加快模型的收敛速度。

ResNet18 中最核心的地方就是残差单元。根据参考资料的描述，假如我们希望用神经网络近似一个函数 $h(x)$，可以把函数拆分成两个部分 $h(x) = x + (h(x) - x)$，虽然可以证明神经网络对于 $h(x)$ 和 $h(x) - x$ 两个函数都可以完成比较好的拟合，但是在实际的训练中可以发现后者更加容易被神经网络学习。因此在网络适当的加入残差模块可以更好的拟合也可以更快的收敛。

Section 3. 神经网络训练

网络结构

两种网络的结构在上一节已经有具体的介绍了，此处不赘述。同理于上一个 Part，模型最后没有经过 `Softmax` 是因为损失函数模块 `CrossEntropyLoss` 中已经包含了 `softmax` 计算了。

网络训练

对于 LeNet 的训练，经过多次尝试我确定了以下参数进行训练：

```
{
    "batch_size": 32,
    "data_path": "./data/data.npz",
    "epoches": 100,
    "hash_id": "8a7e8d55",
    "learning_rate": 0.003,
    "mode": "train_and_test",
    "model": "LeNet",
    "random_seed": 42,
    "raw_data_path": "./data/raw",
    "record_path": "./record",
    "save_path": "./save\\LeNet_best.ckpt"
}
```

对于灰度图片，其像素值在 $[0, 255]$ ，之前提到我在 `CharDataset` 初始化时传入一个 `transform` 将其归一化到 $[0, 1]$ 范围，而且经过多次比较发现，归一化后的训练效果往往更好。使用上述参数训练后，模型在验证集上的正确率达到了 97.9% 。

基于上面的预训练，我接着使用如下参数进行训练：

```
{
    "batch_size": 64,
    "data_path": "./data/data.npz",
    "epoches": 100,
    "hash_id": "301efa36",
    "learning_rate": 0.001,
    "mode": "train_and_test",
    "model": "LeNet",
    "random_seed": 42,
    "raw_data_path": "./data/raw",
    "record_path": "./record",
    "save_path": "./save\\LeNet_best.ckpt"
}
```

也就是将 `batch_size` 减小，减小了 `batch_size` 后模型权重的更新频率更高，通常也能引入更多的随机性。

基于上面的预训练，最后我将 `batch_size` 增大，提高训练的稳定性：

```
{
    "batch_size": 64,
    "data_path": "./data/data.npz",
    "epoches": 100,
    "hash_id": "301efa36",
    "learning_rate": 0.001,
    "mode": "train_and_test",
    "model": "LeNet",
    "random_seed": 42,
    "raw_data_path": "./data/raw",
    "record_path": "./record",
    "save_path": "./save\\LeNet_best.ckpt"
}
```

经过这次训练，模型最终在验证集上的正确率到达了 98.38% 。

对于 ResNet18 的训练，我使用了如下的参数，其中实验后发现学习率较小时能有比较好的效果，学习率过大比较容易出现神经元死亡：

```
{
    "batch_size": 64,
    "data_path": "./data/data.npz",
    "epoches": 80,
    "hash_id": "bb0e4dce",
    "learning_rate": 5e-05,
    "mode": "train",
    "model": "ResNet18",
    "random_seed": 42,
    "raw_data_path": "./data/raw",
    "record_path": "./record",
    "save_path": "./save\\ResNet18_best.ckpt"
}
```


经过训练后，模型最终在验证集上的正确率到达了 99.4% 。

参考资料

- PyTorch documentation — PyTorch 2.1 documentation
- nndl/nndl.github.io: 《神经网络与深度学习》 邱锡鹏著 Neural Network and Deep Learning