Part 1.

Section 1. 代码基本架构

代码架构上我积极参考了 PyTorch 的文档，实现了一个 torch-like 的深度学习框架，各个模块的接口都尽量向 torch 看齐，能比较方便地实现可伸缩易调整的网络结构：

- `nn` 模块是框架核心部分，包含了 `Linear` 、 `Sigmoid` 等基本网络结构的前后向传播逻辑。
- `optim` 模块和训练优化器相关，包含了最基本的优化器 `Optimizer` 等。
- `utils` 模块与训练数据加载有关，其中实现了 `DataLoader` 、 `BatchSampler` 等。
- `model` 模块是使用框架搭建的自定义模型，包含模型结构、训练逻辑、模型存取等。
- `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。

为了加速和方便计算表示，我使用了 `numpy` 库进行数学计算，并使用 `numpy.ndarray` 代替 `torch.Tensor` 进行向量化计算，使用 `numpy` 让性能不至于太低。

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。

Linear

```
class Linear(Module):
    def __init__(self, input_size, output_size):
        super(Linear, self).__init__()
        self.inputs = None
        self.params = {"W": None, "b": None}
        self.grads = {"W": None, "b": None}
        sqrt_k = np.sqrt(1 / input_size)
        self.params["W"] = np.random.uniform(-sqrt_k, sqrt_k, (input_size,
output_size))
        self.params["b"] = np.random.uniform(-sqrt_k, sqrt_k, (1, output_size))

    def forward(self, inputs):
        self.inputs = inputs
        return np.matmul(self.inputs, self.params["W"]) + self.params["b"]

    def backward(self, grads):
        self.grads["W"] = np.matmul(self.inputs.T, grads)
        self.grads["b"] = np.sum(grads, axis=0)
        return np.matmul(grads, self.params["W"].T)
```

线性层很大程度地参考了 PyTorch 文档的定义，但是矩阵计算采用 $y = xA + b$ 式（不同于文档提到的 $y = xA^T + b$），可以使用 `input_size` 和 `output_size` 指定每个 sample 的输入输出维度。在参数的初始化方面，参考 PyTorch 文档使用 $U(-\sqrt{k}, \sqrt{k})$，其中 $k = \frac{1}{\text{input\_size}}$。

这样初始化也较符合 Project1 文档和老师上课的说法，将初始参数调得比较小，能比较好达到收敛。

Softmax

考虑到 softmax 函数的计算过程需要在后续被复用，我把 softmax 的计算逻辑单独封装为 `softmax` 函数：

```
def softmax(input, dim):
    exp_logits = np.exp(input - np.max(input, axis=dim, keepdims=True))
    softmax_scores = exp_logits / np.sum(exp_logits, axis=dim, keepdims=True)
    return softmax_scores
```

注意到这里的 softmax 计算其实先对每个 sample 减去了分量中的最大值，然后再计算 exp，与课件上或 PyTorch 文档中的定义不同。这么做是为了避免因输入数值较大而出现上溢，而且这么做不会影响 softmax 计算的合理性，这点将在下一节证明。

所以 `Softmax` 的前向传播就是简单地直接调用 `softmax`，反向传播则利用保存下来的 `softmax_scores`：

```
class Softmax(Module):
    def __init__(self, dim):
        super(Softmax, self).__init__()
        self.dim = dim

    def forward(self, inputs):
        self.softmax_scores = softmax(inputs, dim=self.dim)
        return self.softmax_scores

    def backward(self, grads):
        sum = np.sum(grads * self.softmax_scores, axis=self.dim, keepdims=True)
        return self.softmax_scores * (grads - sum)
```

对 `ndarray` 使用 softmax 后，在指定维度上的分量将会在 $[0, 1)$ 范围内，并且这些分量和为 1 。

CrossEntropyLoss

`nn.CrossEntropyLoss` = `nn.LogSoftmax` + `nn.NLLLoss` 。

```
class CrossEntropyLoss(Module):
    def __init__(self):
        super(CrossEntropyLoss, self).__init__()

    def forward(self, predicts, labels):
        self.predicts = predicts
        self.labels = labels
        self.batch_size = predicts.shape[0]

        self.softmax_scores = softmax(predicts, dim=1)

        if predicts.shape == labels.shape:
            loss = -np.sum(labels * np.log(self.softmax_scores))
        else:
            e = labels[0]
```

```
            if not (np.isscalar(e) and np.issubsctype(np.asarray(e), np.integer)):
                raise ValueError
            loss = -np.sum(np.log(self.softmax_scores[np.arange(self.batch_size),
labels]))
        return loss / self.batch_size

    def backward(self):
        if self.predicts.shape == self.labels.shape:
            grads = self.softmax_scores - self.labels
        else:
            grads = self.softmax_scores.copy()
            grads[np.arange(self.batch_size), self.labels] -= 1
        return grads / self.batch_size
```

参考 PyTorch 文档，输入的每个 sample 不需要分量求和为 1，因为文档提到 `nn.CrossEntropyLoss` 等价于先计算 `nn.LogSoftmax` 然后计算 `nn.NLLLoss`，也就是会先对输入进行 softmax，然后再计算交叉熵损失函数。所以在多分类问题中，若使用 `CrossEntropyLoss` 作为损失函数，则模型的输出层实际无需再添加 `Softmax` 层。

同时，这里实现的 `CrossEntropyLoss` 也支持两种 `target` 形式，可以是形如 `(batch_size)` 的一维数组，存储每个 sample 的目标分类索引，也可以是形如 `(batch_size, class_num)` 的矩阵，存储每个 sample 的各个类的分类概率，例如 one-hot 编码。

对于 `labels`（也即 `target`）是 `(batch_size)` 的一维数组时，代码中利用了 `numpy` 的高级索引——`self.softmax_scores[np.arange(self.batch_size), labels]` 简化代码编写，提高计算效率。

MSELoss

```
class MSELoss(Module):
    def __init__(self):
        super(MSELoss, self).__init__()
        self.predicts = None
        self.labels = None
        self.batch_size = None

    def forward(self, predicts, labels):
        self.predicts = predicts
        self.labels = labels
        self.batch_size = predicts.shape[0]
        loss = np.square((predicts - labels)) / 2
        return np.sum(loss) / self.batch_size

    def backward(self):
        loss_grad_predicts = self.predicts - self.labels
        return loss_grad_predicts / self.batch_size
```

比较简单的均方误差损失函数，默认计算平均损失，但是和 PyTorch 文档中的 $l_n = (x_n - y_n)^2$ 不同，这里采用课件上的 $l_n = \frac{1}{2}(x_n - y_n)^2$ 式。

Sigmoid

```
class Sigmoid(Module):
    def __init__(self):
        super(Sigmoid, self).__init__()
        self.outputs = None

    def forward(self, inputs):
        self.outputs = 1.0 / (1.0 + np.exp(-inputs))
        return self.outputs

    def backward(self, grads):
        outputs_grad_inputs = np.multiply(self.outputs, (1.0 - self.outputs))
        return np.multiply(grads, outputs_grad_inputs)
```

简单的 `Sigmoid` 实现，也叫做 `Logistic` 函数，作为激活层，也是二分类常用的函数。

ReLU

```
class ReLU(Module):
    def __init__(self):
        super(ReLU, self).__init__()
        self.inputs = None

    def forward(self, inputs):
        self.inputs = inputs
        return np.maximum(0, inputs)

    def backward(self, grads):
        outputs_grad_inputs = self.inputs > 0
        return np.multiply(grads, outputs_grad_inputs)
```

除了课件上提到的 `Sigmoid` 模块，我还实现了 `ReLU` 激活函数，考虑到 `Sigmoid` 容易面临梯度消失的问题，所以实际代码中我更多使用 `ReLU` 。但是 `ReLU` 也可能导致神经元死亡，对应部分参数不更新，训练模型时需要谨慎调整超参。

Optimizer

```
class Optimizer:
    def __init__(self, model, lr):
        self.model = model
        self.lr = lr

    def step(self):
        for layer in self.model.module_list:
            if hasattr(layer, "params") and isinstance(layer.params, dict):
                for key in layer.params.keys():
                    layer.params[key] -= self.lr * layer.grads[key]
```

```
    def zero_grad(self):
        for layer in self.model.module_list:
            if hasattr(layer, "grads") and isinstance(layer.grads, dict):
                for key in layer.grads.keys():
                    layer.grads[key].fill(0.0)
```

按照 PyTorch 的文档，`Optimizer` 应当传入一个可迭代的 `params` 声明要被优化（调整）的模型权重，但是这又涉及到 torch 中的子模块注册机制，严格实现会增加很多意义不大的工作量。考虑到实验中只用到了类似 `ModuleList` 这样的结构，而且其中的子模块都是 `nn` 中的基本结构。故可以直接把整个 `model` 传入 `Optimizer`，调整模型参数时直接遍历 `model.module_list` 即可。

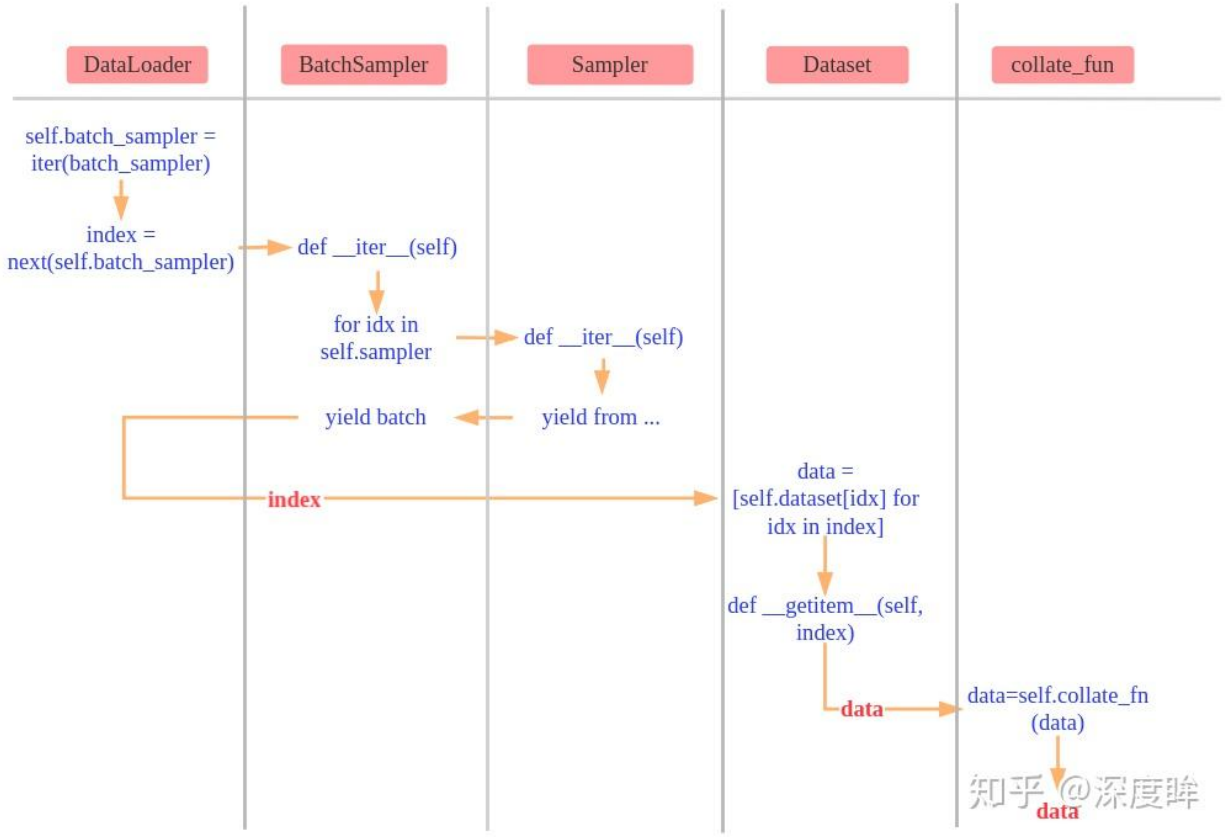
`step()` 方法应在模型反向传播后调用，其遍历各个子模块的参数，根据学习率和各层的梯度进行参数调整。`zero_grad()` 方法直接遍历各层模块并清零存储的梯度。

DataLoader

为了方便训练代码的编写，我还实现了一个 `DataLoader`，使用的方式和 torch 中的相近，能够根据 `Dataset` 和 `Sampler` 对给出的数据集进行采样、整合，并提供一种迭代的方式使用数据。

```
class DataLoader:
    def __init__(
        self, dataset, batch_size=1, shuffle=False, collate_fn=None, drop_last=False
    ):
        sampler = RandomSampler(dataset) if shuffle else SequentialSampler(dataset)
        batch_sampler = BatchSampler(sampler, batch_size, drop_last)
        if collate_fn is None:
            collate_fn = default_collate

        self.dataset = dataset
        self.batch_size = batch_size
        self.drop_last = drop_last
        self.sampler = sampler
        self.batch_sampler = batch_sampler
        self.batch_sampler_iter = None
        self.collate_fn = collate_fn

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

    def __next__(self):
        indices = next(self.batch_sampler_iter)
        data = [self.dataset[idx] for idx in indices]
        stacked = self.collate_fn(data)
        return stacked

    def __iter__(self):
        self.batch_sampler_iter = iter(self.batch_sampler)
        return self
```

为了更好地解释这部分代码，这里引用知乎上的一张流程图：



`BatchSampler` 是批量采样器，迭代这个批量采样器会每次返回一批的索引，我们利用这些索引从 `dataset` 中获取一批的 sample，以一个 `list` 的形式传入 `collate_fun` 进行整理，整理后返回一个可以输入网络的 `ndarray`。默认整理函数参考了 PyTorch 的源代码，利用递归可以处理 `ndarray`、`dict`、`list` 形式的 sample：

```
def default_collate(batch):
    elem_type = type(batch[0])
    if elem_type.__module__ == "numpy":
        return np.stack(batch, 0)
    elif isinstance(batch[0], collections.Mapping):
        return {key: default_collate([d[key] for d in batch]) for key in batch[0]}
    elif isinstance(batch[0], collections.Sequence):
        transposed = zip(*batch)
        return [default_collate(samples) for samples in transposed]
    else:
        raise NotImplementedError
```

`collate_fun` 把 batch_size 个形如 $(dim_0, dim_1, \cdots, dim_k)$ 的 `ndarray` sample 整理成一个形如 $(\text{batch\_size}, dim_0, \cdots, dim_k)$ 的 `ndarray` batch。而 `Sampler` 提供对整个数据集所有样本的一个采样索引序列，每次 `BatchSampler` 从 `Sampler` 中取 batch_size 个索引，利用这些索引去 `dataset` 中取对应的 sample。所以 `dataset` 只需要实现 `__getitem__` 和 `__len__` 方法即可。

Others

在这里简单介绍一下自定义的网络结构 `ANN`：

```
class ANN(Module):
    def __init__(self, layer_sizes):
        super(ANN, self).__init__()
        self.module_list = []
        for i in range(len(layer_sizes) - 2):
            self.module_list.append(Linear(layer_sizes[i], layer_sizes[i + 1]))
            self.module_list.append(ReLU())
        self.module_list.append(Linear(layer_sizes[-2], layer_sizes[-1]))

    def forward(self, inputs):
        for layer in self.module_list:
            inputs = layer(inputs)
        return inputs

    def backward(self, grads):
        for layer in reversed(self.module_list):
            grads = layer.backward(grads)
```

结合传入的参数 `layer_sizes`，利用之前的代码可以比较容易地搭建可调整的结构，前向传播是将前一层的输出作为下一层的输入，反向传播是将后面的梯度输入到前一层。整个结构顺序传播，主要使用类似 `nn.ModuleList` 的方式搭建。

Section 2. 对反向传播算法的理解

前馈神经网络具有很强的拟合能力，根据通用近似定理，只要隐藏层神经元的数量足够，它可以以任意的精度来近似任何一个定义在实数空间 $\mathbb{R}^D$ 中的有界闭集函数。而学习隐藏层参数一般使用梯度下降法，计算损失函数对参数的偏导数，若通过链式法则逐一对每个参数求偏导效率低，所以训练网络时常使用反向传播算法计算梯度。

第 l 层的误差项（损失关于该层输出的偏导数 $\delta^{(l)} = \frac{\partial \mathcal{L}}{\partial z^{(l)}}$）可以通过第 $l+1$ 层的误差项计算得到，也就是误差可以反向传播。

注：以下推导中涉及的矩阵求导全部使用 **分子布局**。

Linear

线性层是代码中的核心模块，也是实验 Part1 中神经网络的主要组成部分。在代码中我使用表达式 $z^{(l)} = a^{(l-1)}W^{(l)} + b^{(l)}$（其中 $a^{(l-1)}$ 是一个行向量，来自前一层），假设该层的输出向量长度 M_l，令 $\mathcal{L}$ 是最终的损失函数，根据链式法则有：

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} = \frac{\partial \mathcal{L}}{\partial z^{(l)}} \cdot \frac{\partial z^{(l)}}{\partial w_{ij}^{(l)}}$$

$$\frac{\partial \mathcal{L}}{\partial b^{(l)}} = \frac{\partial \mathcal{L}}{\partial z^{(l)}} \cdot \frac{\partial z^{(l)}}{\partial b^{(l)}}$$

首先计算 $\frac{\partial z^{(l)}}{\partial w_{ij}^{(l)}}$，因为 $z^{(l)} = a^{(l-1)}W^{(l)} + b^{(l)}$，也就是 $z_j^{(l)} = \sum_i a_i^{(l-1)} \cdot w_{ij}^{(l)}$，所以：

$$
\begin{aligned}
\frac{\partial z^{(l)}}{\partial w_{ij}^{(l)}} &= \left[\frac{\partial z_1^{(l)}}{\partial w_{ij}^{(l)}}, \cdots, \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}}, \cdots, \frac{\partial z_{M_l}^{(l)}}{\partial w_{ij}^{(l)}}\right]^T \\
&= \left[0, \cdots, \frac{\partial(\sum_i a_i^{(l-1)} \cdot w_{ij}^{(l)})}{\partial w_{ij}^{(l)}}, \cdots, 0\right]^T \\
&= [0, \cdots, a_i^{(l-1)}, \cdots, 0]^T
\end{aligned}
$$

注意其中 $a_i^{(l-1)}$ 是第 j 个分量值。

然后计算 $\frac{\partial z^{(l)}}{\partial b^{(l)}}$，因为 $z^{(l)} = a^{(l-1)}W^{(l)} + b^{(l)}$，则显然有：

$$
\frac{\partial z^{(l)}}{\partial b^{(l)}} = I_{M_l} \quad \in \mathbb{R}^{M_l \times M_l}
$$

为 $M_l \times M_l$ 的单位矩阵。

根据 $z^{(l+1)} = a^{(l)}W^{(l+1)} + b^{(l+1)}$ 有：

$$
\frac{\partial z^{(l+1)}}{\partial a^{(l)}} = (W^{(l+1)})^T \quad \in \mathbb{R}^{M_{l+1} \times M_l}
$$

$a^{(l)}$ 代表第 l 层的输出 $z^{(l)}$ 经过激活函数后的活性值，将会输入到下一个线性层。根据 $a^{(l)} = f_l(z^{(l)})$（其中 $f_l(\cdot)$ 为按位计算函数），则有：

$$
\begin{aligned}
\frac{\partial a^{(l)}}{\partial z^{(l)}} &= \frac{\partial f_l(z^{(l)})}{\partial z^{(l)}} \\
&= \mathrm{diag}(f_l'(z^{(l)})) \quad \in \mathbb{R}^{M_l \times M_l}
\end{aligned}
$$

最后计算 $\delta^{(l)} = \frac{\partial \mathcal{L}}{\partial z^{(l)}}$，因为 $z^{(l)} = a^{(l-1)}W^{(l)} + b^{(l)}$，根据链式法则有：

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial z^{(l)}} &= \frac{\partial \mathcal{L}}{\partial a^{(l)}} \cdot \frac{\partial a^{(l)}}{\partial z^{(l)}} \\
&= \left(\frac{\partial \mathcal{L}}{\partial z^{(l+1)}} \cdot \frac{\partial z^{(l+1)}}{\partial a^{(l)}}\right) \cdot \frac{\partial a^{(l)}}{\partial z^{(l)}} \\
&= \left(\delta^{(l+1)} \cdot (W^{(l+1)})^T\right) \cdot \mathrm{diag}(f_l'(y^{(l)}))
\end{aligned}
$$

因此 $\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}}$ 就可以被表示为：

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} &= \delta^{(l)} \cdot \frac{\partial z}{\partial w_{ij}^{(l)}} \\
&= [\delta_1^{(l)}, \cdots, \delta_j^{(l)}, \cdots, \delta_{M_l}^{(l)}] \cdot [0, \cdots, a_i^{(l-1)}, \cdots, 0]^T \\
&= a_i^{(l-1)} \cdot \delta_j^{(l)}
\end{aligned}
$$

其中 $a_i^{(l-1)} \cdot \delta_j^{(l)}$ 相当于向量 $a^{(l-1)}$ 和向量 $\delta^{(l)}$ 的外积的第 i, j 个元素，故上式可以改写为：

$$
\left[\frac{\partial \mathcal{L}}{\partial W^{(l)}}\right]_{ij} = \left[(a^{(l-1)})^T \cdot \delta^{(l)}\right]_{ij}
$$

因此，$\mathcal{L}$ 关于第 l 层权重 $W^{(l)}$ 的梯度为：

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = (a^{(l-1)})^T \cdot \delta^{(l)} \quad \in \mathbb{R}^{M_{l-1} \times M_l}$$

同理可得，$\mathcal{L}$ 关于第 l 层偏置量 $b^{(l)}$ 的梯度为：

$$\frac{\partial \mathcal{L}}{\partial b^{(l)}} = \delta^{(l)} \quad \in \mathbb{R}^{M_l}$$

实际代码中一层神经元的输入并不是一个行向量，而是一个 $(\text{batch\_size},\ dim_{in})$ 的矩阵，但是梯度的表达式也几乎相同，在代码中使用 `numpy` 库的向量化运算可以简洁地表达。

Softmax

上一节曾提到，`Softmax` 在代码中的实现方式和原公式有所不同，注意到：

$$\frac{e^{x_i+c}}{\sum e^{x_i+c}} = \frac{e^c \cdot e^{x_i}}{e^c \cdot \sum e^{x_i}} = \frac{e^{x_i}}{\sum e^{x_i}}$$

所以加或减一个常数不会影响 `Softmax` 的结果。

虽然在代码中使用表达式 $e^{x_i - \max x} / \sum e^{x_i - \max x}$，但是这里不妨使用原始公式进行推导，因为实际上加减一个常数并不影响导数的形式。假设输入的 x 是行向量，输出的 y 也是行向量，长度均为 M，在 **分子布局** 下有：

$$\frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_M} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_M} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_M}{\partial x_1} & \frac{\partial y_M}{\partial x_2} & \cdots & \frac{\partial y_M}{\partial x_M} \end{bmatrix}$$

对于 $\frac{\partial y_k}{\partial x_k}$ 来说有：

$$\begin{aligned} \frac{\partial y_k}{\partial x_k} &= \frac{\partial \frac{e^{x_k}}{\sum_i e^{x_i}}}{\partial x_k} \\ &= \frac{\left(\frac{\partial e^{x_k}}{\partial x_k} \cdot \sum_i e^{x_i} - e^{x_k} \frac{\partial \sum_i e^{x_i}}{\partial x_k} \right)}{(\sum_i e^{x_i})^2} \\ &= \frac{e^{x_k}}{\sum_i e^{x_i}} \cdot \left(1 - \frac{e^{x_k}}{\sum_i e^{x_i}} \right) \\ &= y_k \cdot (1 - y_k) \end{aligned}$$

对于 $\frac{\partial y_j}{\partial x_k}$ 其中 $j \neq k$ 来说有：

$$\begin{aligned} \frac{\partial y_j}{\partial x_k} &= \frac{\partial \frac{e^{x_j}}{\sum_i e^{x_i}}}{\partial x_k} \\ &= \frac{-e^{x_j} e^{x_k}}{(\sum_i e^{x_i})^2} \\ &= -\frac{e^{x_j}}{\sum_i e^{x_i}} \cdot \frac{e^{x_k}}{\sum_i e^{x_i}} \\ &= -y_j \cdot y_k \end{aligned}$$

在反向传播算法中，$\delta^{(l)}$ 是由后一层传入的误差项，也就是损失关于本层输出的偏导数：

$$\delta^{(l)} = \left[\frac{\partial \mathcal{L}}{\partial y_1}, \cdots, \frac{\partial \mathcal{L}}{\partial y_M}\right]$$

根据链式法则有：

$$\delta^{(l-1)} = \frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y} \cdot \frac{\partial y}{\partial x} = \delta^{(l)} \cdot \frac{\partial y}{\partial x}$$

也就是有：

$$\begin{aligned}
\delta^{(l-1)} &= \delta^{(l)} \cdot \frac{\partial y}{\partial x} \\
&= \left[\frac{\partial \mathcal{L}}{\partial y_1}, \cdots, \frac{\partial \mathcal{L}}{\partial y_k}, \cdots, \frac{\partial \mathcal{L}}{\partial y_M}\right] \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_M} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_M} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_M}{\partial x_1} & \frac{\partial y_M}{\partial x_2} & \cdots & \frac{\partial y_M}{\partial x_M} \end{bmatrix} \\
&= \left[\frac{\partial \mathcal{L}}{\partial x_1}, \cdots, \frac{\partial \mathcal{L}}{\partial x_k}, \cdots, \frac{\partial \mathcal{L}}{\partial x_M}\right]
\end{aligned}$$

显然，对于 $\frac{\partial \mathcal{L}}{\partial x_k}$ 有：

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial x_k} &= \sum_{j=1}^{M} \left(\frac{\partial \mathcal{L}}{\partial y_j} \cdot \frac{\partial y_j}{\partial x_k}\right) \\
&= -\sum_{j=1, j\neq k}^{M} \frac{\partial \mathcal{L}}{\partial y_j} \cdot y_j \cdot y_k + \frac{\partial \mathcal{L}}{\partial y_k} \cdot y_k \cdot (1 - y_k) \\
&= y_k \cdot \left(\sum_{j=1}^{M} \frac{\partial \mathcal{L}}{\partial y_j} \cdot (-y_j) + \frac{\partial \mathcal{L}}{\partial y_k}\right) \\
&= y_k \cdot \left(\frac{\partial \mathcal{L}}{\partial y_k} - \sum_{j=1}^{M} \left(\frac{\partial \mathcal{L}}{\partial y_j} \cdot y_j\right)\right)
\end{aligned}$$

CrossEntropyLoss

交叉熵损失函数也是一个很重要的损失函数，在多分类任务中不可或缺。老师在上课时提到，多分类任务也可以使用 `MSELoss`，但是使用 `CrossEntropyLoss` 是“标准答案”，因为 `MSELoss` 并没有那么适合分类问题。**多分类问题为何建议用交叉熵损失函数**，经过思考和查找资料，我认为：

- 交叉熵的输入通常是类别概率分布，代表模型对各个分类的后验概率，衡量的是两个概率分布之间的差异（极大似然）。而均方误差的输入则是一个高维空间中的点位置，衡量的是模型预测点和真实点之间的距离。
- 交叉熵只关心 $\hat{y}_p$ 是否更趋近 1 了，而均方误差除了与 $\hat{y}_p$ 有关，从取到最小值的方向看来，还会尽量希望剩下的预测概率相等。

  在大多分类问题中，我们对类别与类别之间的关系（相似、相近等）是难以量化的，所以一般标签都是 one-hot 。在这个前提下，假如标签为 $[1,\ 0,\ 0]$，均方误差会认为 $[0.8,\ 0.1,\ 0.1]$ 比 $[0.8,\ 0.15,\ 0.05]$ 更好，也就是平均比有倾向性更好。

  但实际上这是有悖常识的，例如在分类 `["cat", "tiger", "pig"]` 中，如果正确标签是 `"cat"`，预测结果中 `"tiger"` 通常会比 `"pig"` 有更高的概率，因为 `"tiger"` 的特征和 `"cat"` 更接近。

- 在一个极端的二分类例子中，令 $\hat{y}_i = \sigma(\sum w_i x_i + b)$，$\sigma$ 是 `Sigmoid`。如果正确标签是 $[0,\ 1]$，但预测结果是 $[1,\ 0]$，使用均方误差作为损失函数 $L = \frac{1}{2n}\sum(y_i - \hat{y}_i)^2$，则有：

  $$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial(\sum w_i x_i + b)} \cdot x_i = -(y_i - \hat{y}_i)(\hat{y}_i \cdot (1 - \hat{y}_i))x_i$$

  代入数值后发现梯度为 0，这显然是不合理的。

- 邱锡鹏老师的《神经网络与深度学习》中，**3.6 损失函数对比** 中也有提及。

因此，在多分类问题中更多时候是使用 `CrossEntropyLoss` 作为损失函数。

在代码的实现中，我按照 `PyTorch` 文档的说明和课件的提示，将 softmax 和交叉熵损失的计算都放在 `CrossEntropyLoss` 模块中，所以在推导该模块的反向传播的时候，可以利用 `Softmax` 的求导结果。假设输入 `CrossEntropyLoss` 的 x 是一个行向量，先经过 `softmax` 函数计算出行向量 $\hat{y}$，然后与正确结果 y 计算出交叉熵损失，为了方便不妨假设这里 y 是 one-hot 编码的行向量。

首先计算交叉熵的导数，交叉熵的公式是 $\mathcal{L} = -\sum_{c=1}^{M} y_c \log \hat{y}_c$，其导数：

$$\frac{\partial \mathcal{L}}{\partial \hat{y}_c} = -\frac{y_c}{\hat{y}_c}$$

将这个式子带入上面计算出来的 $\frac{\partial \mathcal{L}}{\partial x_k}$ 中，可以得到：

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial x_k} &= \hat{y}_k \cdot \left(\frac{\partial \mathcal{L}}{\partial \hat{y}_k} - \sum_{j=1}^{M}\left(\frac{\partial \mathcal{L}}{\partial \hat{y}_j} \cdot \hat{y}_j\right)\right) \\ &= -y_k + \hat{y}_k \cdot \left(\sum_{j=1}^{M} y_j\right)\end{aligned}$$

由于 y 是 one-hot 编码（或者是概率分布），所以应当有 $\sum_{j=1}^{M} y_j = 1$，则：

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial x_k} &= -y_k + \hat{y}_k \cdot \left(\sum_{j=1}^{M} y_j\right) \\ &= \hat{y}_k - y_k \\ &= \frac{e^{x_k}}{\sum_i e^{x_i}} - y_k\end{aligned}$$

直接用向量表示就是：

$$\frac{\partial \mathcal{L}}{\partial x} = \hat{y} - y$$

这样就得出了损失关于 `CrossEntropyLoss` 输入的偏导数了。

Section 3. 神经网络训练

拟合 $\sin$ 和汉字分类上，我都将数据集按照 `train:valid = 9:1` 的比例划分训练集和验证集进行训练。由于拟合 $\sin$ 的过程比较简单，而且其训练集数据比较丰富，所以下面的内容都是针对汉字分类的训练过程。

网络结构

汉字分类在模型结构上使用了 `Linear` 和 `ReLU` 进行搭建，具体网络结构按照如下顺序连接：

1. `Linear(28 * 28, 1024)`
2. `ReLU()`
3. `Linear(1024, 2048)`
4. `ReLU()`
5. `Linear(2048, 512)`
6. `ReLU()`
7. `Linear(512, 12)`

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。对于 28×28 的输入图片，拉平成向量后长度就是 784 ，为了能从样本中充分地学习特征，线性层神经元数量至少需要上千的规模。经过探索，我认为上述的网络规模是一个比较合适的规模，一方面，再增加规模也很难提高模型在验证集上的正确率，而且训练的效率大打折扣；另一方面，减小模型的规模容易导致模型学习图像的特征不够充分。

网络训练

经过多次尝试后，确定使用如下训练参数进行训练，同时不使用 scheduler 进行学习率调整。

```
{
    "batch_size": 32,
    "data_path": "./data/char/data.npz",
    "epoches": 80,
    "hash_id": "7cb67d02",
    "learning_rate": 0.03,
    "mode": "train_and_test",
    "random_seed": 42,
    "raw_data_path": "./data/char/train_raw",
    "record_path": "./record/char",
    "save_path": "./save/char/best_model.pkl"
}
```

注意对于灰度图片，其像素值在 $[0, 255]$ ，这里我在 `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 92.3% 。

基于上面的预训练，我接着使用如下参数进行训练，同样不使用 scheduler 调整学习率。

```
{
    "batch_size": 16,
    "data_path": "./data/char/data.npz",
    "epoches": 80,
    "hash_id": "3981de82",
    "learning_rate": 0.03,
    "mode": "train_and_test",
    "random_seed": 42,
    "raw_data_path": "./data/char/train_raw",
    "record_path": "./record/char",
    "save_path": "./save/char/best_model.pkl"
}
```


与上一次训练参数的主要差别就是减小了 `batch_size` ，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。使用上述参数训练后，模型在验证集上的正确率达到了 92.95% 。

最后我调小了学习率，调大了 `batch_size` 希望让模型更稳定地学习，但是最终在验证集上地正确率没有提高。个人认为一方面数据集数据量比较小，另一方面受限于线性层过于简单，难以像 CNN 那样学习多种层次的特征。故若想进一步提升正确率，我认为可以对数据进行增强（增加数据、旋转图片、图片添加噪声等），同时使用更好的模型结构，如CNN等。

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
- 温故知新——前向传播算法和反向传播算法（BP算法）及其推导 - 知乎 (zhihu.com)
- 带你从零掌握迭代器及构建最简 DataLoader - 知乎 (zhihu.com)
- DataLoader原理解析 (最简单版本实现) - 知乎 (zhihu.com)
- PyTorch36.DataLoader源代码剖析 - 知乎 (zhihu.com)
- 直观理解为什么分类问题用交叉熵损失而不用均方误差损失?-腾讯云开发者社区-腾讯云 (tencent.com)