深度学习本科课，2024 年春，作业 1

课程老师：黄雷 助教：周百川

（关于本次作业的任何疑问请在微信群聊或私聊助教提出，助教微信号：Reloisa）

DDL：4 月 2 日 24:00

上传链接：

https://bhpan.buaa.edu.cn/link/AAE05699C06E4A4002A831007FB72C99CD

Folder Name：深度学习本科课程 2024 年春季作业 1

Expires：2024-04-02 23:59

Pickup Code：dukp

1. 背景

1.1 作业介绍

诸如 TensorFlow 和 PyTorch 的深度学习框架使得训练复杂的神经网络架构变得非常容易。你甚至不需要真正理解它们是如何工作的！在这些框架的加持下，仅需几行代码你便可以定义一个复杂的神经网络架构并在你的数据集上训练。这些框架对于那些不想深入了解的人来说十分方便，但对于想要深入了解神经网络内部原理的人来说，他们的高层次设计掩盖了神经网络内部的优雅之处。对于初学者，我们推荐自己上手编写一个简单的神经网络，从网络架构，超参数，数据处理与训练优化几个角度，建立对神经网络的初步认知。因此，在这个作业中，你将使用 NumPy，自己搭建一个简单的全连接神经网络。

一个基本的神经网络由以下几个部分组成：

- 一个定义输入信息如何在模型内部传播的结构。这个结构定义了输入与输出之间的函数关系
- 一个损失函数
- 优化算法（此次作业使用 SGD，不过你可以自己尝试定义如 Adam 之类的优化算法）
- 超参数

你需要使用 NumPy 实现以下几个部分：

- 激活函数(ReLU, Sigmoid, TanH, etc.)的前向传播过程与反向传播过程
- 全连接层模块的前向传播与反向传播过程
- 模型的训练

本次作业我们使用神经网络在一个监督学习的任务上训练，具体来说，我们在 Iris 数据集上，训练一个简单的由全连接神经网络构成的分类器。

监督学习任务通常会给定一系列输入，与一系列输入的真实标签。我们希望通过训练模型学习输入与真实标签的函数关系。为了学习到这个函数关系，通常我们需要定义一个优化算法，迭代地更新模型的参数，通常，我们使用 **mini-batch gradient descent** 作为优化算法，利用通过损失函数计算得到的梯度进行参数更新。我们使用**反向传播**算法进行梯度计算。

为了利用反向传播计算梯度，我们先将输入前向传入网络，计算损失。在前向传入过程中，我们通常将 mini-batch（32, 64, 128…）个样本送入网络，得到我们预测的标签的概率分布，与真实的标签共同送入我们定义的损失函数中，计算得到一个损失值（通常是一个浮点数）。利用我们得到的损失，我们可以计算损失对网络中各个参数的梯度，从输出开始，利用链式法则，一路将梯度传播至网络最开始的参数中。这个过程便是反向传播过程。这种计算方法避免了梯度的重复计算，也是现代深度学习必不缺少的关键算法。Chris Olah 在这里

写了一个非常直观的解释，十分推荐阅读。3Blue1Brown 同样制作了关于反向传播的可视化解释，如果对反向传播不了解，这些资料可以帮助你数学上的直觉。**掌握反向传播对完成本次作业十分重要。**

总的来说，完成一次神经网络的训练分为三步：

- 将输入前向传入至网络，得到输出
- 计算损失
- 反向传播进行梯度更新

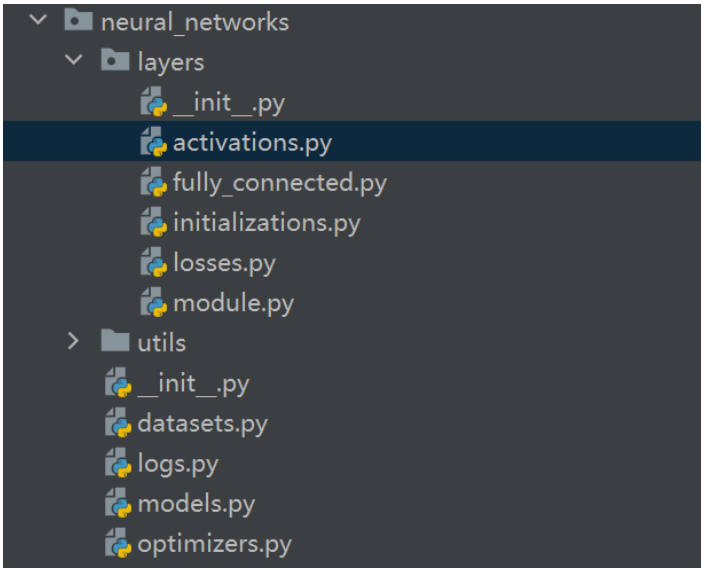
1.2 批量计算(Batching)

训练神经网络时，通常我们需要大量的数据，为了节省计算时间与资源，我们必须学会利用并行运算的优势，这意味着我们必须学会将运算向量化。NumPy 对向量化运算进行了专门的优化。比如说，当我们求 ReLU 函数的梯度时，我们不应使用 for 循环遍历每一个元素，判断这个元素是否大于 0。相反，我们应当学会将 index 过程向量化，例如 Z[Z > 0]，我们便可以直接找出所有大于 0 的元素。

在本次作业中，**所有你推得的数学表达式都应使用向量运算，不可使用循环遍历。**

2. 神经网络框架

我们已经提供了模块化的代码，如下图所示:



代码的不同模块展示的是不同的部分，如 activations, losses, fully_connected，这三个文件是你需要实现的地方，其中 models 部分需要你填写部分训练代码，但当你把前三个模块实现以后，这应该很简单。

其中，fully_connected.py 中的 FullyConnected 模块是最核心的模块，它包含以下几个属性:

- parameters: 该属性包含了全连接层必要的参数（权重 weights 与偏置 bias）
- cache: 该属性用于存储反向传播中需要的在前向传播中计算得到的值
- gradients: 每个参数所对应的梯度
- activations: 激活函数
- n_in: 输入张量的维度

- n_out: 输出张量的维度

全连接层的前向传播与反向传播由这两个函数定义:

- forward: 这个函数需要输入数据，或中间输出 X 作为输入。该部分通过 Affine 运算和全连接层的 W 和 b 结合(XW + b)，经过激活函数，得到最终值 Z，作为函数输出。在此过程中，你需要判断哪些值应该被存入 cache 中，以便在计算反向传播时访问这些值。

```
def forward(self, X: np.ndarray) -> np.ndarray:
    """前向传播：输入张量 X 是一个二维张量，与权重 W 进行矩阵乘法，加上偏置 bias 后，通过激活函数，得到输出。
    把所有反向传播中可能用到的张量或参数存入 cache 中。

    Parameters
    ----------
    X   输入张量的 shape (batch_size, input_dim)

    Returns
    -------
    输出张量  (batch_size, output_dim)
    """
    # initialize layer parameters if they have not been initialized
    if self.n_in is None:
        self._init_parameters(X.shape)

    ### BEGIN YOUR CODE ###

    # perform an affine transformation and activation
    z = np.dot(X, self.parameters["W"]) + self.parameters["b"]
    out = self.activation(z)

    # store information necessary for backprop in `self.cache`
    self.cache['z'] = z
    self.cache['X'] = X
    ### END YOUR CODE ###

    return out
```

- backward: 这个函数需要损失 Loss 对前向传播的输出 out 的梯度作为输入，利用在前向传播中保留的数据，进行反向传播计算，该函数返回的是损失 Loss 对本层输入的梯度。

```
def backward(self, dLdY: np.ndarray) -> np.ndarray:
        """反向传播实现。
        需要计算损失 Loss 对三个参数的梯度
            1. 权重  W
```

2. bias b

3. 本层的输入张量 Z

Parameters

---

dLdY　损失 Loss 对张量本层输出值的梯度（和输出张量的 shape 相同，(batch_size, output_dim)）

Returns

---

derivative of the loss with respect to the input of this layer

shape (batch_size, input_dim)

损失 Loss 对张量本层输出值的梯度 和输入张量的 shape 相同，(batch_size, input_dim)）

```
"""
### BEGIN YOUR CODE ###

# unpack the cache
z, X = self.cache['z'], self.cache['X']
# compute the gradients of the loss w.r.t. all parameters as well as the
# input of the layer

dLdZ = self.activation.backward(dLdY)
dX = dLdZ.dot(self.parameters['W'].T)

# store the gradients in `self.gradients`
# the gradient for self.parameters["W"] should be stored in
# self.gradients["W"], etc.
self.gradients['W'] = X.T.dot(dLdZ)
self.gradients['b'] = np.sum(dLdZ, axis=0)
### END YOUR CODE ###

return dX
```

每个来自 activations.py 的 Activation 类也是类似的布局（略简单些）

```
class Identity(Activation):
    name = "identity"
    """激活函数：f(z) = z"""

    def __init__(self):
        super().__init__()

    def forward(self, Z: np.ndarray) -> np.ndarray:
        """f(z) = z 的前向传播实现
```

```
        Parameters
        ----------
        Z   传入激活函数前的张量

        Returns
        -------
        经过激活函数后的输出值，在 Identity 函数的情况下，返回传入前的张量 Z
        """
        return Z

    def backward(self, dY: np.ndarray) -> np.ndarray:
        """f(z) = z 的反向传播实现

        Parameters
        ----------
        dY   损失 Loss 对张量激活值 f(Z) 的梯度（和输入张量 Z 的 shape 相同）

        Returns
        -------
        损失 Loss 对张量 Z 的梯度
        """
        return dY
```

2.1 Debug

我们提供了一个名为 check_gradients.ipynb 的 Python Notebook，你可以使用这个 notebook 去 debug 你的代码实现。具体来说，在这里面我们将对比你的代码实现求得的梯度，与利用数值分析求得的梯度。

$$\frac{\partial f}{\partial x_k}(a) \approx \frac{f(\ldots, a_k + \epsilon, \ldots) - f(\ldots, a_k - \epsilon, \ldots)}{2\epsilon}$$

如果你的梯度求导是正确的，那么误差应该非常小（通常不大于10^{-7}）

3. 代码实现

接下来的部分需要你自己编写代码完成。

这个部分你需要 1) 推导梯度的数学表达式 2) 写出对应代码

注意：该部分的表达式都应该被向量化，也就是说不应该依赖循环遍历每个元素。

3.1 激活函数（2 分）

首先，你将实现 activations.py 中的激活函数，本次作业只要求实现 ReLU 激活函数，其他的部分不要求完成。

$$\sigma_{\mathrm{ReLU}}(\gamma)=\begin{cases}0 & \gamma<0,\\ \gamma & \text{otherwise.}\end{cases}$$

注意，这个函数应该直接对一个向量输入进行操作，不要对单个元素遍历操作。

实现步骤：

1. 求出损失 Loss 对输入值的梯度的数学表达式，这个表达式应该包含损失 Loss 对输出激活值的梯度 dLdY，与**批量输入值 Z，** $Z \in R^{N\times M}$。
2. 在 activations.py 中实现前向传播与反向传播的过程。**不要循环遍历每个训练样本，用向量化表达式实现。**

3.2 全连接层（4 分）

现在，你将实现本次作业的核心部分，全连接层。找到 fully_connected.py，开始代码实现前，请认真阅读每个代码块提供的文档。

如果你不确定你的实现是否正确，可以参照 2.1 Debug 部分，检查梯度是否正确。

实现步骤:

1. 求出损失 Loss 对权重 W 和偏置 b 的梯度的数学表达式，$\frac{\partial L}{\partial b}$，$\frac{\partial L}{\partial W}$。你也需要求出 Loss 对输入 X 的梯度$\frac{\partial L}{\partial X}$，这个梯度会被传到下游层中当做 backward 函数的输入。记住，你必须使用向量化的运算，不要使用循环遍历。
2. 实现前向传播与反向传播。实现前向传播时，首先利用_init_parameters 和输入 X 的 shape 初始化 parameters，cache，gradients。之后，利用激活函数，和权重 W 以及偏置 b 求得输出值。$Y=\sigma(XW+b)$。实现反向传播时，backward 函数将 dLdY，也就是损失 Loss 对输出值 Y 的梯度，作为输入，你将利用这个输入求得损失 Loss 对权重 W 的梯度 dW, 对偏置 b 的梯度 db，对输入 X 的梯度 dX。**（HINT：梯度应当和原本的张量在形状上保持一致。经验上来讲，在求矩阵乘法的梯度时，如 dX, dW，如果你能保证张量和梯度形状的一致，那么你的实现就是正确的）**

3.3 SoftMax（2 分）

接下来实现 SoftMax 部分，因为我们需要构建一个分类器，我们要求输出必须是一个概率分布，为了满足概率分布的性质，我们使用 SoftMax 函数将每个元素都限制在[0,1]中，并保证他们的和为 1。**这部分不要求使用向量化实现，你可以使用循环遍历。**

SoftMax 函数以一个 k 维的向量 s 作为输入$(s_1,s_2,s_3,\ldots,s_k)$，返回一个标准化的值，具体地，

$$\sigma_i=\frac{e^{s_i}}{\sum_{j=1}^{k}e^{s_j}},$$

但在具体实现时，为了保证数值的稳定，我们通常采用以下实现方式：

$$\sigma_i = \frac{e^{s_i - m}}{\sum_{j=1}^{k} e^{s_j - m}},$$

在这里，$m = max_j^k s_j$

实现步骤:

1. 求得 SoftMax 函数输入与输出的雅可比矩阵。**你不需要在这一步中使用向量化操作，也就是说循环遍历在这一部分是允许的。**但我们推荐你找出向量化操作的方法，因为这样非常高效。
2. 在 activations.py 中找到 SoftMax 类，实现前向传播与反向传播。

3.4 CrossEntropyLoss（1 分）

对于多分类问题，我们使用 cross entropy 作为损失，具体来说

$$L = -y \ln(\hat{y}),$$

在这里，y 是经过是真实标签的 one-hot 编码，$\hat{y}$是经过模型输出经过 softmax 后的概率分布。在批量数据中，cross entropy 可以表示为：

$$J = -\frac{1}{m}\left(\sum_{i=1}^{m} Y_i \ln(\hat{Y}_i)\right).$$

实现步骤：

1. 求得损失 Loss 对模型输出（未经过 SoftMax）的梯度，**必须使用向量化的运算，不能使用循环遍历。**
2. 实现 CrossEntropyLoss 的前向传播与反向传播。前向传播的输入是模型输出的 logits(未经过 SoftMax)与真实的 one-hot 编码 Y。前向传播时 logits 应先经过 SoftMax 得到$\hat{Y}$,与真实标签 Y 一起求得损失。反向传播时，**不要循环遍历每一个输入样本**。

3.5 两层神经网络（1 分）

现在你已经完成了大部分工作，你已经准备好训练一个两层的神经网络了。我们利用 Iris 数据集，每个输入样本有四个特征，三个可能的类别。我们提供了几个重要的实现帮助你训练你的神经网络：

1. 在 datasets.py 中，我们实现了 dataloader，这个 loader 帮助我们处理批量的数据
2. SGD 优化器，在 optimizers.py 中，我们实现了一个简单的 SGD 优化器，你可以传入动量比率作为超参数。
3. 权重初始化，在 initializations.py 中，我们实现了不同的权重初始化方法，本次作业默认使用 XavierUniform 作为初始化方法。
4. 一个 logger，在 logs.py 中，记载训练时的数据。

实现步骤：

1. 在 models.py 中，实现神经网络的前向传播与反向传播。

2. 在 train_ffnn.py 中，定义你自己的超参数，你需要从学习率，以及隐藏层的维度这两个方面，选择五组不同的超参数，并汇报他们对应的结果，并解释说明为什么选择特定的参数会造成不同的结果。将你的结果与解释写在 report.md 中

4. 作业提交

遵循 README.md 中的指示，提取所有代码实现，生成 submission.md，将 hw1_student 整个文件夹上传，按照格式周百川_20373349_hw1_student 这个格式上传。