同济大学

TONGJI UNIVERSITY

《人工智能原理课程设计》

实验报告

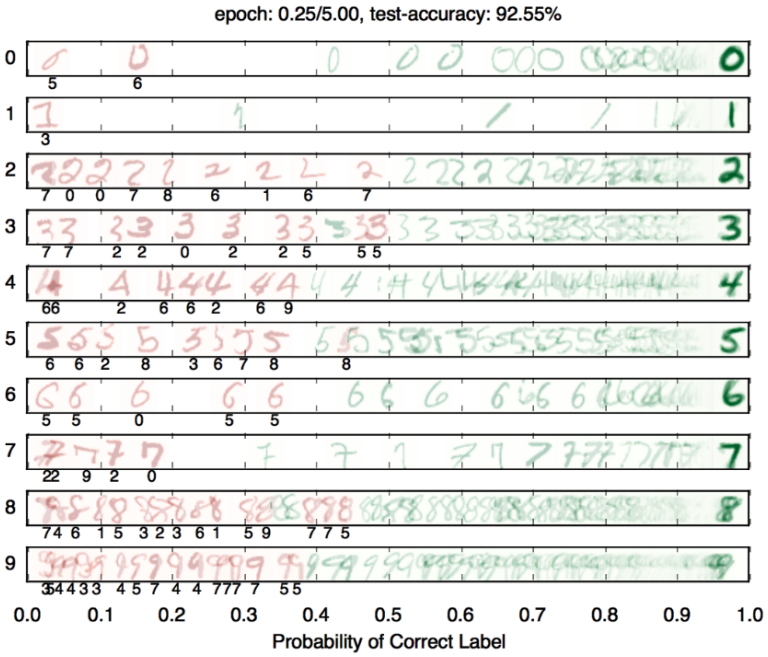
实验名称	Project3:machinelearning
学　　院	电子与信息工程学院
专　　业	计算机科学与技术
学生姓名	汪清濯
学　　号	2154057
任课教师	张红云/苗夺谦
日　　期	2023 年 6 月 3 日

目 录

目 录

1 实验介绍

本项目将介绍机器学习，我将需要构建一个神经网络来进行数字分类等操作。项目已经提供了一个神经网络迷你库（nn.py）和需要训练的数据集（backend.py）。我需要完成 models.py 中的四种模型。



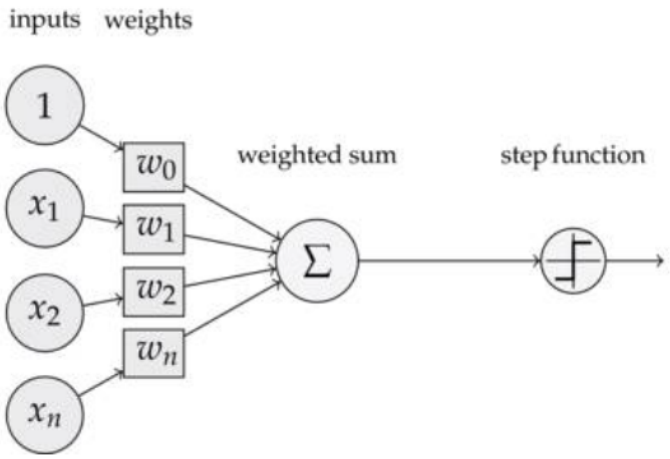
2 Perceptron——感知器

2.1 问题概述

2.1.1 直观描述

在这部分中，需要实现 models.py 中的 PerceptronModel 类，即感知器。感知器是组成神经网络的神经元，它可以被视为一种最简单形式的前馈神经网络，是一种二元线性分类器。在这里我们研究是最简单的神经网络问题，只有一个感知器的情况，而神经网络中则是由大量的感知器分层组合而成，问题更加复杂。一个感知器由输入权值、激活函数等组成。

下图是一个感知器



我们要实现的感知器的输出标签只有 1 和-1，这意味着数据集中的数据点（x,y）将有一个 1x1 的向量 y，我们需要将 x 和权重 w 的点积转化为一个实数与预测值比较。

2.1.2 已有代码的阅读和理解

nn.py 中定义了一些节点类，每个节点代表一个实数或者一个实数矩阵。同时对节点对象的操作进行了优化，比直接使用 Python 的内置 list 数据类型工作的更快。

本问题中，我们需要阅读 nn.py 中的 Parameter 类、Constant 类、DotProduct 类以及 as_scalar 函数

A. Parameter 类

```
1 class Parameter(DataNode)
```

该类是一个继承自 DataNode 的参数节点类，用于表示感知器或神经网络的可训练参数，并提供更新参数的方法。它有以下成员函数

a. **构造函数**

```
def __init__(self, *shape)
```

参数：*shape：形状参数，确保 shape 具有 2 个维度，并且每个维度都是正整数。

主要功能：它根据给定的形状，在指定范围内生成随机数据，并将其存储在 data 属性中。

b. update **函数**

```
def update(self, direction, multiplier)
```

参数：direction：必须是 Constant 节点类型，表示更新的方向。direction 的形状必须与参数节点的形状相匹配。

multiplier：必须是 Python 标量（整数或浮点数），用于缩放更新方向。

主要功能：该方法将参数节点的数据与乘积结果相加，以更新参数的值。然后，它检查更新后的参数是否包含 NaN 或无穷大的值，如果有，则引发异常。

B. Constant **类**

```
class Constant(DataNode)
```

该类是一个继承自 DataNode 的常量节点类，表示一个二维浮点矩阵。用于表示输入特征、输出标签和通过反向传播计算的梯度。它有以下成员函数

a. **构造函数**

```
def __init__(self, data)
```

参数：data：一个 numpy 数组，表示常量节点的数据。数据类型应为浮点类型。

主要功能：初始化常量节点。它接受一个参数 data，表示节点的数据。构造函数首先检查 data 是否是 numpy 数组类型，然后检查 data 的数据类型是否为浮点类型。然后，它调用父类 DataNode 的构造函数，将数据存储在 data 属性中。

C. DotProduct **类**

```
class DotProduct(FunctionNode)
```

该类是一个继承自 FunctionNode 的批量点积计算类，计算其两个输入的点积。它有以下成员函数。

a. _forward 函数

```
1  def _forward(*inputs)
```

参数：*inputs：在这个类中，第一个参数 inputs[0] 表示特征（features），它的形状是 (batch_size x num_features)；第二个参数 inputs[1] 表示权重（weights），它的形状是 (1 x num_features)。

主要功能：静态方法，用于执行前向传播计算。它接受任意数量的输入参数，但在这个类中，只有两个输入参数。该方法首先检查输入参数的数量是否为 2，然后检查第一个输入参数的维度是否为 2，第二个输入参数的维度是否为 2，且第二个输入参数的第一维度是否为 1，以及两个输入参数的第二维度是否相等。最后，它执行批量点积计算，并返回结果。

b. _backward 函数

```
1  def _backward(gradient, *inputs)
```

参数：无调用

主要功能：静态方法，用于执行反向传播计算。在这个类中，该方法抛出了一个 NotImplementedError 异常，表示在这个作业中不需要通过 DotProduct 节点进行反向传播计算。

D. as_scalar 函数

```
1  def as_scalar(node)
```

参数：node: 节点对象，表示要转换为标量的节点。

主要功能：将一个节点的值转换为标准的 Python 数值并返回。这个函数只适用于只有一个元素的节点（例如，SquareLoss、SoftmaxLoss 以及批量大小为 1 的 DotProduct）。

2.1.3 解决问题的思路与方法

感知器只需要调整一个参数，即权重 w，模型已为我们提供。我们将输入 x 与 w 的点积作为预测值，并不断训练直到感知器收敛即可。

2.2 算法设计

2.2.1 算法功能

设计一个感知器模型类，用于对数据点进行分类。它具有初始化权重、计算感知器输出、获取预测类别和训练感知器等功能。感知器通过调整权重来进行训练，直到达到收敛条件。

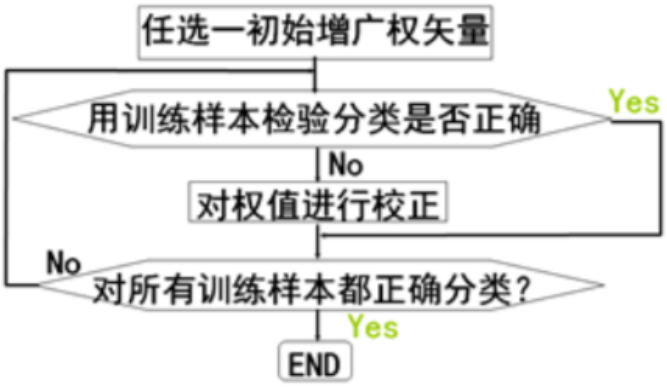
2.2.2 设计思路

run 方法直接返回输入 x 与权重 w 的点积。get_prediction 方法中，先使用 as_scalar 函数将点积转化为实数，因为感知器的输出标签只有 1 和-1，所以当该实数大于 0 时，返回 1，否则返回-1。训练时，反复循环数据集，对错误分类的实例进行更新，使用 Parameter 类的 update 方法更新权重 w。算法仅在预测错误时修改权值，若正确标签是 y=1，则修正权值为 w+x，若正确标签是 y=-1，则修正权值为 w-x。当循环完整个数据集而没有任何错误时，感知器收敛，模型训练完成。权值更新可表示如下：

$$weights \leftarrow weights + directions * multiplier$$

directions 是与参数节点形状相同的节点，multiplier 参数是 Python 标量

2.2.3 算法流程图



2.3 算法实现

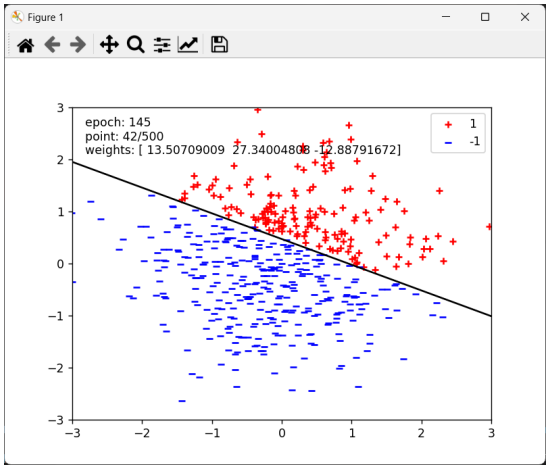
下面贴出代码。细节已在注释中标明

```
class PerceptronModel(object):
  def __init__(self, dimensions):
      # 初始化权重
      self.w = nn.Parameter(1, dimensions)

  def get_weights(self):
      return self.w

  def run(self, x):
      # 计算感知器的输出
      return nn.DotProduct(x, self.w)

  def get_prediction(self, x):
    if nn.as_scalar(self.run(x)) >= 0:
        return 1
```

```
    else:
        return -1

def train(self, dataset):
    # 批大小为 1
    batch_size = 1

    while 1:
        convergence = 1
        for x, y in dataset.iterate_once(batch_size):
            # 与预期不符
            if self.get_prediction(x) != nn.as_scalar(y):
                convergence = 0
                # 更新权值
                self.w.update(x, nn.as_scalar(y))
        # 若收敛则退出循环
        if convergence:
            break
```

2.4 实验结果

所有测试用例均已通过




```
Question q1
===========
*** q1) check_perceptron
Sanity checking perceptron...
Sanity checking perceptron weight updates...
Sanity checking complete. Now training perceptron
*** PASS: check_perceptron

### Question q1: 6/6 ###

Finished at 22:39:01

Provisional grades
==================
Question q1: 6/6
------------------
Total: 6/6
```

该问题的评测按照以下顺序：感知器的完整性检查——权重更新检查——感知器训练

感知器完整性检查主要检查方法实现是否正确，如 get_prediction 方法是否只返回 1 或-1 等。权值更新检查更新后的参数是否包含 NaN 或无穷大的值，如果有，则引发异常。感知器训练则检查感知器是否能在有限时间内训练为 100% 正确

3 Non-linear Regression——非线性回归

3.1 问题概述

3.1.1 直观描述

在部分中，需要实现 models.py 中的 RegressionModel 类。我将训练一个神经网络来近似 sin(x) 在 $[-2\pi, 2\pi]$ 间的取值。需要完成参数初始化、返回预测值方法、返回给定输入和目标输出的损失方法以及基于梯度和损失的训练模型方法。

3.1.2 已有代码的阅读和理解

本问题中，我们需要阅读 nn.py 中的 Add 类、AddBias 类、Linear 类、ReLU 类、SquareLoss 类以及 gradients 函数。

A. Add 类

```
1  class Add(FunctionNode)
```

该类是继承自 FunctionNode，用于执行矩阵的元素逐元素相加操作。它接受形状由的两个节点，并构造一个也具有形状由的节点。它具有以下成员函数。

a. _forward 函数

```
1  def _forward(*inputs)
```

参数：*inputs：输入参数，第一个参数 inputs[0] 表示矩阵 x，它的形状是 (batch_size x num_features)；第二个参数 inputs[1] 表示矩阵 y，它的形状与 x 相同。

主要功能：静态方法，用于执行前向传播计算。该方法首先检查输入参数的数量是否为 2，然后检查第一个输入参数的维度是否为 2，第二个输入参数的维度是否为 2，以及两个输入参数的形状是否相同。最后，它执行矩阵的元素逐元素相加操作，并返回结果。

b. _backward 函数

```
1  def _backward(gradient, *inputs)
```

参数：gradient：表示反向传播过程中的梯度。梯度是指损失函数对于函数输出的偏导数，用于衡量函数输出对于输入的敏感程度。它与 inputs[0] 具有相同的形状。

*inputs：与 _forward 相同

主要功能：静态方法，用于执行反向传播计算。该方法首先检查梯度参数的形状与第一个输入

参数的形状是否相同，然后返回一个包含梯度和梯度的列表。

B. AddBias 类

```
class AddBias(FunctionNode)
```

该类用于将偏置向量添加到特征矩阵的每个特征向量中。它有以下成员函数。

a. _forward 函数

```
def _forward(*inputs)
```

参数：*inputs：输入参数，第一个参数 inputs[0] 表示特征矩阵，它的形状是 (batch_size x num_features)；第二个参数 inputs[1] 表示偏置向量，它的形状是 (1 x num_features)。

主要功能：静态方法，用于执行前向传播计算。该方法首先检查输入参数的数量是否为 2，然后检查第一个输入参数的维度是否为 2，第二个输入参数的维度是否为 2，并且第二个输入参数的第一个维度应为 1。此外，它还检查两个输入参数的第二个维度是否相同。最后，它将每个特征向量加上对应的偏置向量，并返回结果。

b. _backward 函数

```
def _backward(gradient, *inputs)
```

参数：gradient：表示反向传播过程中的梯度。它与 inputs[0] 具有相同的形状。

*inputs: 与 _forward 相同

主要功能：静态方法，用于执行反向传播计算。在这个类中，该方法首先检查梯度参数的形状与第一个输入参数的形状是否相同，然后返回一个包含梯度和对梯度进行求和的偏置梯度的列表。

C. Linear 类

```
class Linear(FunctionNode)
```

该类对输入进行线性变换，即矩阵乘法操作。下面是一个矩阵乘法公式。

$$\mathbf{AB} = \begin{bmatrix} \vec{A}_0^T \\ \vec{A}_1^T \\ \cdots \\ \vec{A}_{m-1}^T \end{bmatrix} \mathbf{B} = \begin{bmatrix} \vec{A}_0^T\mathbf{B} \\ \vec{A}_1^T\mathbf{B} \\ \cdots \\ \vec{A}_{m-1}^T\mathbf{B} \end{bmatrix} \qquad \mathbf{AB} = \mathbf{A}\begin{bmatrix} \vec{B}_0 & \vec{B}_1 & \cdots & \vec{B}_{p-1} \end{bmatrix} = \begin{bmatrix} \mathbf{A}\vec{B}_0 & \mathbf{A}\vec{B}_1 & \cdots & \mathbf{A}\vec{B}_{p-1} \end{bmatrix}$$

该类有以下成员函数。

a. _forward **函数**

```
def _forward(*inputs)
```

参数：*inputs：输入参数，第一个参数 inputs[0] 表示特征矩阵，它的形状是 (batch_size x input_features)；第二个参数 inputs[1] 表示权重矩阵，它的形状是 (input_features x output_features)。

主要功能：静态方法，用于执行前向传播计算。该方法首先检查输入参数的数量是否为 2，然后检查第一个输入参数的维度是否为 2，第二个输入参数的维度是否为 2，并且第一个输入参数的第二个维度应与第二个输入参数的第一个维度相同。最后，它执行矩阵乘法操作，将特征矩阵与权重矩阵相乘，并返回结果。

b. _backward **函数**

```
def _backward(gradient, *inputs)
```

参数：gradient：表示反向传播过程中的梯度。它的第一维与 inputs[0] 的第一维相同，第二维与 inputs[1] 的第二维相同。

*inputs: 与 _forward 相同

主要功能：静态方法，用于执行反向传播计算。该方法首先检查梯度参数的形状与第一个和第二个输入参数的形状是否匹配。然后，它分别计算输入特征矩阵和权重矩阵对应的梯度，并返回一个包含这两个结果的列表。

D. ReLU **类**

```
class ReLU(FunctionNode)
```

该类是一个继承自 FunctionNode 的 ReLU（修正线性单元）非线性变换类。这种非线性将输入中的所有负条目替换为零。即 $Relu(x) = max(x, 0)$。该类有以下成员函数。

a. _forward **函数**

```
def _forward(*inputs)
```

参数：*inputs：输入参数。在这个类中，唯一的输入参数 inputs[0] 表示要应用 ReLU 的节点，它的形状是 (batch_size x num_features)。

主要功能：静态方法，用于执行前向传播计算。它接受一个输入参数，并检查输入参数的数量是否为 1 以及维度是否为 2。然后，它应用 ReLU 函数，将输入参数中的负值替换为零，并返回结果。

b. _backward 函数

```
def _backward(gradient, *inputs)
```

参数：gradient：表示反向传播过程中的梯度。它与 inputs[0] 具有相同的形状。

*inputs: 与 _forward 相同

主要功能：静态方法，用于执行反向传播计算。该方法首先检查梯度参数的形状与输入参数的形状是否匹配。接下来，它计算输入参数的梯度，即根据输入参数的值是否大于零来决定梯度的取值，大于零时梯度为 1，小于等于零时梯度为 0，最后将梯度乘以输入参数的梯度，并返回结果。

E. SquareLoss 类

```
class ReLU(FunctionNode)
```

该类计算用于回归问题的批处理平方损失。它具有以下成员函数。

a. _forward 函数

```
def _forward(*inputs)
```

参数：inpputs：输入参数，包括两个节点 inputs[0] 和 inputs[1]，分别表示损失函数的计算结果和目标值，它们的形状都是 (batch_size x dim)。

主要功能：静态方法，用于执行前向传播计算。它接受两个输入参数，并检查输入参数的数量是否为 2 以及维度是否为 2。然后，它计算每个位置上的 0.5 * (a[i,j] - b[i,j])**2，生成一个 (batch_size x dim) 的矩阵。接下来，它计算并返回该矩阵中所有元素的平均值。

b. _backward 函数

```
def _backward(gradient, *inputs)
```

参数：gradient：表示损失函数对输出的梯度，是一个标量。

*inputs: 与 _forward 相同

主要功能：静态方法，用于执行反向传播计算。该方法首先检查梯度参数是否为标量。然后，它根据输入参数计算梯度，并返回结果。计算方法是根据损失函数的导数公式，将梯度乘以 (inputs[0] - inputs[1]) / inputs[0].size 和 (inputs[1] - inputs[0]) / inputs[0].size。

F. gradients 函数

```
def gradients(loss, parameters)
```

参数：loss: 损失函数节点，可以是 SquareLoss 或 SoftmaxLoss 类型的节点。这是待优化的损失函数，对它进行梯度计算。

parameters: 参数列表，包含待计算梯度的参数节点。这是要计算梯度的目标参数。

函数工作流程如下：

1. 首先，对输入的损失函数和参数进行断言检查，确保它们的类型和有效性。
2. 将 loss 节点标记为已使用，以防止重复使用。
3. 创建一个集合 nodes 和一个列表 tape，用于存储节点和反向传播的顺序。
4. 定义内部函数 visit，用于递归遍历节点的父节点，并将节点添加到 tape 列表中。
5. 使用 visit 函数遍历损失函数和参数节点，将它们添加到 tape 列表和 nodes 集合中。
6. 创建一个字典 grads，用于存储每个节点的梯度，并初始化损失函数节点的梯度为 1.0。
7. 从反向遍历的 tape 列表中，对每个节点调用其 _backward 方法，传递相应的梯度和父节点的数据，并获取父节点的梯度。
8. 将父节点的梯度添加到对应的参数节点的梯度中。
9. 最后，根据参数节点的梯度创建 Constant 对象，并将它们存储在列表中作为结果返回。

总结：这个函数用于计算损失函数相对于给定参数的梯度，它使用反向传播算法根据节点之间的依赖关系计算梯度，并返回一个包含梯度值的常数对象列表。

3.1.3 解决问题的思路与方法

该神经网络模型用于逼近从实数到实数的函数映射。所以我们可以让隐藏层数较少，这样训练的超参数也较少，同时让隐藏层大小足够大，能够在区间 $[-2\pi, 2\pi]$ 上合理精度地逼近 sin(x) 函数。

3.2 算法设计

3.2.1 算法功能

设计一个用于非线性回归的神经网络模型，通过定义模型结构和训练方法来逼近从实数到实数的函数映射，主要包括模型的初始化、运行模型、计算损失和训练方法。

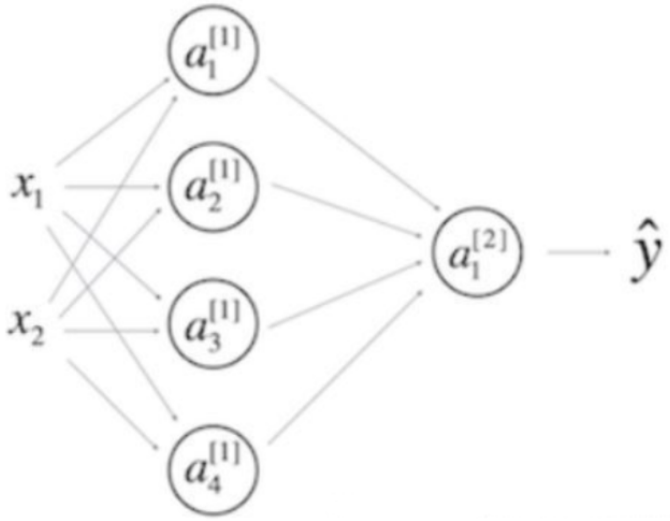
3.2.2 设计思路

我们先通过 nn.py 中提供的框架来构建一个简单的神经网络。一个简单的神经网络具有线性层，每个线性层执行线性操作（就和感知器一样）。线性层被非线性层分开，使网络可以近似一般的函数。我们用 ReLU 方法（上文的 ReLU 类）进行非线性操作。例如，一个简单的单隐藏层/两个线性层神经网络输出向量 f(x) 将由以下函数给出：

$$f(x) = relu(x * W_1 + b_1) * W_2 + b_2$$

我们需要训练的就是参数矩阵 W_1, W_2 和参数向量 b_1, b_2，它们在梯度下降的过程中学习。W_1 是一个 i x h 的矩阵，i 是输入向量 x 的维度，h 是隐藏层的大小。b_1 是一个 1 x h 的向量。我们可以自由选择隐藏层的大小，通常来说，隐藏层越大的网络能容纳更多数据，会使网络更强大，但同时也会使网络更难训练，或者导致数据过拟合。

含一个隐藏层的神经网络构造如下图所示：



我们可以添加更多的隐藏层来构建更深层次的网络，如两个隐藏层/三个线性层的神经网络

$$\hat{y} = f(x) = relu(relu(x * W_1 + b_1) * W_2 + b_2) * W_3 + b_3$$

也可以分解上述内容并明确指出两个隐藏层

$$h_1 = f_1(x) = relu(x * W_1 + b_1)$$
$$h_2 = f_2(h_1) = relu(h_1 * W_2 + b_2)$$
$$\hat{y} = f_3(h_2) = h_2 * W_3 + b_3$$

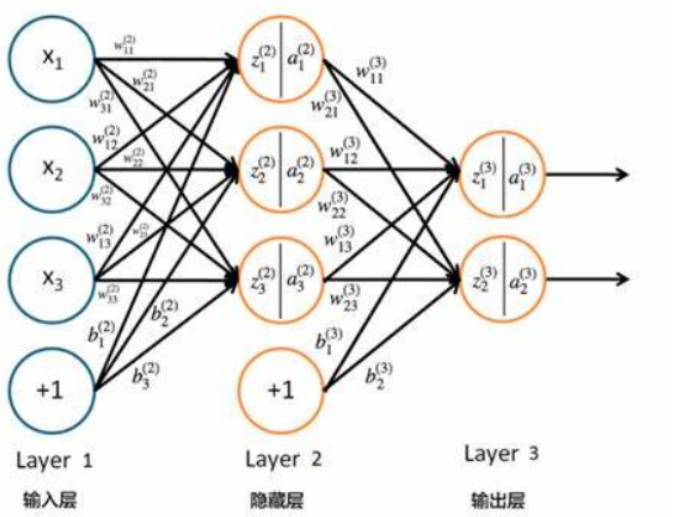
为了提高效率，我们需要一次性处理一批数据，这意味着输入 x 是一个 b x i 的矩阵，b 是批量大小。同时，我们应该选择一个良好的学习率，学习率是指导我们，在梯度下降法中，如何使用损失函数的梯度调整网络权重的超参数，超参数需要用学习率更新。

$$weight \leftarrow weight - learning\_rate * gradient$$

学习率太低可能导致模型训练太慢，太高可能导致损失函数发散到无穷大。首先应尝试不同的学习率，观察损失如何随时间减少。尝试不同的批次大小时，最佳学习率可能会随着批大小改变。

对于本问题，我设置一个隐藏层，并先设置隐藏层的大小为 400，根据题目要求，输入输出大小都为 1。同时初始化学习率，批大小和两组 W 与 b。学习率从低向高尝试，W 和 b 的大小按照上文中的规定定义。run 方法按照含有一个隐藏层的神经网络获取预测值，get_loss 方法通过 SquareLoss 获取损失。训练时，基于学习率和梯度更新超参数。矩阵乘法、偏置向量添加等都使用 nn.py 中提供的方法。

3.2.3 算法流程图



3.3 算法实现

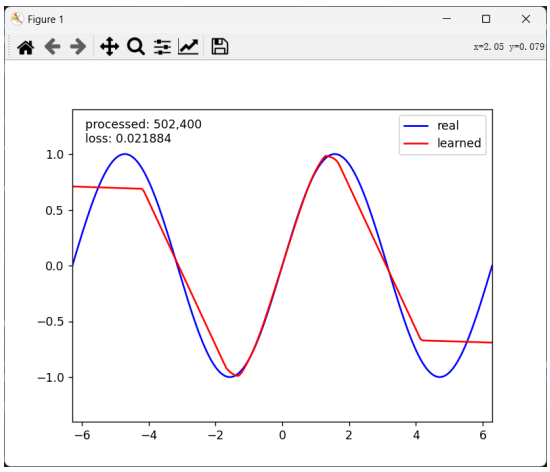
下面贴出代码，细节已在注释中标明

```python
class RegressionModel(object):
    def __init__(self):
        # Initialize your model parameters here
        "*** YOUR CODE HERE ***"
        self.hidden_layer_size = 400     # 隐藏层大小
        self.batch_size = 100            # 批量大小，dataset 的大小要能被 batch_size 整除
        self.learning_rate = 0.03        # 学习率
        self.input_size = 1              # 输入大小，sin 函数只有一个参数
        self.output_size = 1             # 输出大小

        # 设一个隐藏层，共两个线性层
        '''
        x : batch_size x input_size
        W1 : input_size x hidden_layer_size
        b1 : 1 x hidden_layer_size
        W2 : hidden_layer_size x output_size
        b2 : 1 x output_size
        '''
```

```
        self.W1 = nn.Parameter(self.input_size,self.hidden_layer_size)
        # b 的行为 1，列与要加的矩阵相同
        self.b1 = nn.Parameter(1,self.hidden_layer_size)
        self.W2 = nn.Parameter(self.hidden_layer_size,self.output_size)
        self.b2 = nn.Parameter(1,self.output_size)

    def run(self, x):
        # 隐藏层计算
        h1 = nn.ReLU(nn.AddBias(nn.Linear(x,self.W1),self.b1))
        # 预期值
        prediction = nn.AddBias(nn.Linear(h1,self.W2),self.b2)
        return prediction

    def get_loss(self, x, y):
        return nn.SquareLoss(self.run(x),y)

    def train(self, dataset):
        while 1:
            for x,y in dataset.iterate_once(self.batch_size):
                loss = self.get_loss(x,y)
                # 计算损失梯度
                grad_W1,grad_b1,grad_W2,grad_b2 =
                ↪  nn.gradients(loss,[self.W1,self.b1,self.W2,self.b2])
                # 更新参数
                self.W1.update(grad_W1,-self.learning_rate)
                self.b1.update(grad_b1,-self.learning_rate)
                self.W2.update(grad_W2,-self.learning_rate)
                self.b2.update(grad_b2,-self.learning_rate)
            # 平均损失小于 0.02 时，训练完成
            if nn.as_scalar(loss) < 0.02:
                break
```

3.4 实验结果

所有测试用例均已通过




```
Question q2
===========
*** q2) check_regression
Your final loss is: 0.018645
*** PASS: check_regression


### Question q2: 6/6 ###

Finished at 17:06:17

Provisional grades
==================
Question q2: 6/6
------------------
Total: 6/6
```

该问题测评按以下顺序：模型的完整性检查——权重更新检查——模型训练

完整性检查主要检查方法实现是否正确，各参数的大小是否合法等。权值更新检查更新后的参数是否包含 NaN 或无穷大的值。模型训练则检查模型能否在有限的时间内训练到预期的损失。为减少参数初始化的随机性产生的影响，我连续测评了十次该问题，均以满分通过。

4 Digit Classification——数字分类

4.1 问题概述

4.1.1 直观描述

在这部分中，需要实现 models.py 中的 DigitClassificationModel 类，训练网络以对 MNIST 数据集中的手写数字进行分类。每个数字的大小是 28x28 像素，它的值存储在一个 784 维的向量中。每一个输出标签都是一个 10 维向量，它的所有位置都是 0，除了对应的正确数字位置上是 1。

4.1.2 已有代码的阅读和理解

本问题中，我们需要阅读 nn.py 中的 SoftmaxLoss 类。

A. SoftmaxLoss 类

```
class SoftmaxLoss(FunctionNode)
```

该类用于计算分类问题的批处理 softmax 损失。它具有以下成员函数。

a. log_softmax 函数

```
def log_softmax(logits)
```

参数：logits: 一个形状为 (batch_size x num_classes) 的节点。每一行表示一个示例属于某个类别的得分。它是模型的输出，通过 softmax 函数进行处理，用于计算损失和梯度。

主要功能：该方法对 logits 进行 softmax 操作，并返回 softmax 后的对数概率。

b. _forward 函数

```
def _forward(*inputs)
```

参数：inputs：代表两个输入参数，logits 和 labels，分别是形状为 (batch_size x num_classes) 的节点。

主要功能：该方法中，先确保输入的维度和形状正确，然后计算 log-softmax 值，并通过交叉熵损失函数计算平均损失值，返回一个标量节点。

c. _backward 函数

```
def _backward(gradient, *inputs)
```

参数：garadient：反向传播中的梯度，是一个标量。

*inputs：同 _forward 函数

主要功能：该方法中，根据损失函数对 log-softmax 的求导公式，计算梯度并返回梯度列表。梯度列表中包含两个元素，分别是对 logits 和 labels 的梯度。

4.1.3 解决问题的思路和方法

该问题中，我们也设置一个隐藏层，这样需要的参数就与问题 2 相同，我们只需要根据题目要求改变输入输出向量的大小。并根据训练结果进行调参即可。

4.2 算法设计

4.2.1 算法功能

设计一个对手写数字进行分类的神经网络。使模型的识别准确率能达到 97% 以上。

4.2.2 设计思路

基本与问题 2 相同。设置一个隐藏层，并先设置隐藏层的大小为 200，根据题目要求，输入大小为 784，输出大小为 10。同时初始化学习率，批大小和两组 W 与 b。学习率从低向高尝试，W 和 b 的大小按照上文中的规定定义。run 方法按照含有一个隐藏层的神经网络获取预测值，get_loss 方法通过 SoftmaxLoss 获取损失。训练时，基于学习率和梯度更新超参数。矩阵乘法、偏置向量添加等都使用 nn.py 中提供的方法。判断训练是否完成时，使用 dataset.get_validation_accuracy() 方法看准确率是否超过 98%，若超过则训练结束，否则继续训练。

4.3 算法实现

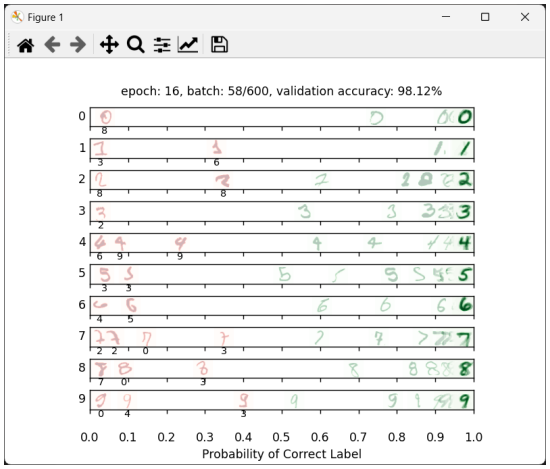
下面贴出代码，细节已在注释中标明。

```
class DigitClassificationModel(object):
    def __init__(self):
        # Initialize your model parameters here
        "*** YOUR CODE HERE ***"
        self.hidden_layer_size = 200     # 隐藏层大小
        self.batch_size = 100            # 批量大小，dataset 的大小要能被 batch_size 整除
        self.learning_rate = 0.5         # 学习率
        self.input_size = 784            # 输入大小，28*28
        self.output_size = 10            # 输出大小
        # 设一个隐藏层，共两个线性层
        '''
        x : batch_size x input_size
        W1 : input_size x hidden_layer_size
        b1 : 1 x hidden_layer_size
        W2 : hidden_layer_size x output_size
        '''
```

```
        '''
        b2 : 1 x output_size
        '''
        self.W1 = nn.Parameter(self.input_size, self.hidden_layer_size)
        # b 的行为 1，列与要加的矩阵相同
        self.b1 = nn.Parameter(1, self.hidden_layer_size)
        self.W2 = nn.Parameter(self.hidden_layer_size, self.output_size)
        self.b2 = nn.Parameter(1, self.output_size)

    def run(self, x):
        # 隐藏层计算
        h1 = nn.ReLU(nn.AddBias(nn.Linear(x, self.W1), self.b1))
        # 预期值
        prediction = nn.AddBias(nn.Linear(h1, self.W2), self.b2)
        return prediction

    def get_loss(self, x, y):
        return nn.SoftmaxLoss(self.run(x),y)

    def train(self, dataset):

        while 1:
            for x, y in dataset.iterate_once(self.batch_size):
                loss = self.get_loss(x, y)
                # 计算损失梯度
                grad_W1, grad_b1, grad_W2, grad_b2 = nn.gradients(
                    loss, [self.W1, self.b1, self.W2, self.b2])
                # 更新参数
                self.W1.update(grad_W1, -self.learning_rate)
                self.b1.update(grad_b1, -self.learning_rate)
                self.W2.update(grad_W2, -self.learning_rate)
                self.b2.update(grad_b2, -self.learning_rate)
            # 若准确率高于 98% 则退出
            if dataset.get_validation_accuracy() > 0.98:
                break
```

4.4 实验结果

所有测试用例均已通过




```
Question q3
===========
*** q3) check_digit_classification
Your final test set accuracy is: 97.820000%
*** PASS: check_digit_classification

### Question q3: 6/6 ###

Finished at 18:11:21

Provisional grades
==================
Question q3: 6/6
------------------
Total: 6/6
```

该问题测评按以下顺序：模型的完整性检查——权重更新检查——模型训练

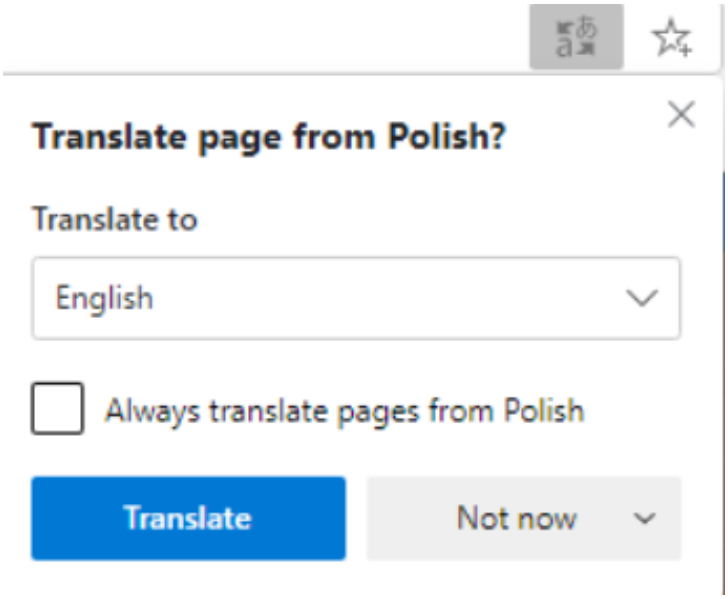
完整性检查主要检查方法实现是否正确，各参数的大小是否合法等。权值更新检查更新后的参数是否包含 NaN 或无穷大的值。模型训练则检查模型能否在有限的时间内训练到预期的准确度。为减少参数初始化的随机性产生的影响，我连续测评了十次该问题，均以满分通过。

5 Language Identification——语言识别

5.1 问题概述

5.1.1 直观描述

在这部分中，需要实现 models.py 中的 LanguageIDModel 类。语言识别任务是在给定一段文本的情况下找出文本是用什么语言编写的。例如，浏览器可能能够检测到用户是否访问过外语页面并提供翻译。这是 Edge 的一个示例（它使用神经网络来实现此功能）：



在本问题中，我们将构建一个小型神经网络模型，一次识别一个单词的语言。我们的数据集由五种语言的单词组成，如下表所示：

单词	语言
discussed	English
eternidad	Spanish
itseänne	Finnish
paleis	Dutch
mieszkać	Polish

5.1.2 解决问题的思路和方法

不同单词的长度不同，所以我们的模型需要有可以处理可变长度输入的设计，单词长度 = 隐藏层数量 +1。每个隐藏层的函数共用同一神经网络，所以可以让所有隐藏层共用一个权重参数 W_{hidden}，因此，我们只需要训练三个参数，输入权重 W_x，隐藏权重 W_{hidden}，输出权重 W_o。

5.2 算法设计

5.2.1 算法功能

设计一个语言识别的神经网络，使模型准确率能达到 81% 以上。

5.2.2 设计思路

数据集将为每个字符提供一个单独的输入：$x_0, x_1, ..., x_{L-1}$，L 是单词的长度。我们需要一个初始函数（像之前的网络一样），接受输入 x_0 并输出矩阵 h_1，大小为 batch_size x hidden_layer_size。

$$h_1 = f_{initial}(x_0)$$

接下来，将上一步的输出和单词中的下一个字母组合在一起，生成前两个字母的向量标签。为此，我们应用一个子网 f，该子网接受当前字母和上一次的隐藏状态，并输出隐藏状态，f 可表示如下：

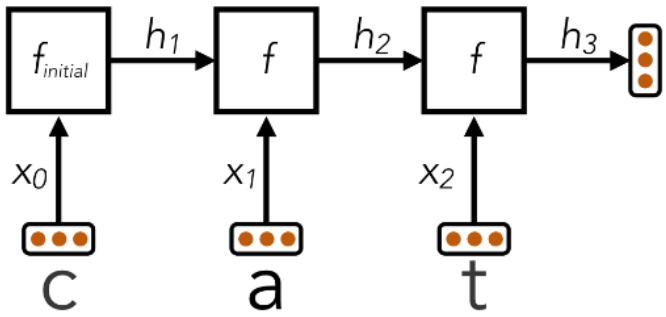
$$h_2 = f(h_1, x_1)$$

依次类推，继续处理输入单词的所有字母

$$h_3 = f(h_2, x_2)$$

...

在这些计算中，函数 f 是同一块神经网络，使用相同的可训练参数，其中与之前隐藏状态做矩阵乘法的参数，我们定义它为 W_{hidden}。与输入字母做矩阵乘法的参数，我们定义它为 W_x，该参数也是 $f_{initial}$ 的参数。该技术称为递归神经网络（RNN），如下图所示

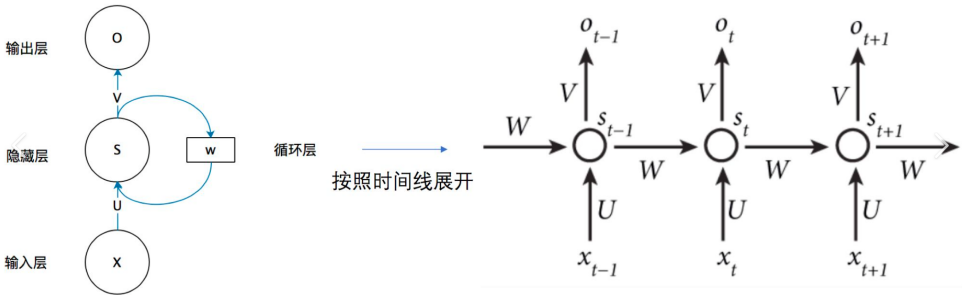


RNN 处理完输入全长后，将任意长度的输入词编码为固定大小的向量 h_L，提供给输出转换层，由输出层的参数 W_o 与 h_L 计算预测值。

隐藏层数量为 L-1，设置隐藏层大小为 100，根据题目要求，输入大小为 47，输出大小为 5。初

始化学习率、批大小和三个权重参数。学习率从低向高尝试，权重参数的大小按上文规定定义。run 方法按照 RNN 神经网络获取预测值，get_loss 方法通过 SoftmaxLoss 获取损失。训练时，基于学习率和梯度更新超参数。矩阵乘法、加法等都使用 nn.py 中提供的方法。判断训练是否完成时，使用 dataset.get_validation_accuracy() 方法看准确率是否超过 83%，若超过则训练结束，否则继续训练。

5.2.3 算法流程图



5.3 算法实现

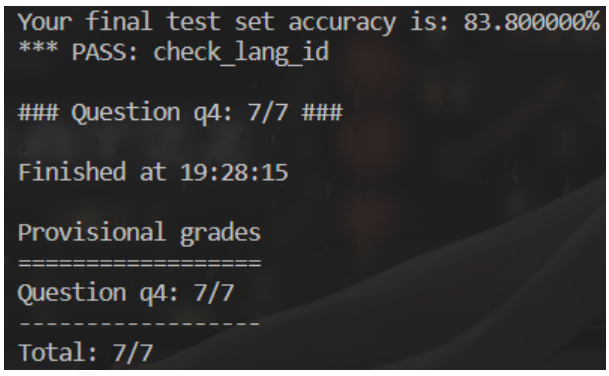
下面贴出代码，细节已在注释中标明。

```
class LanguageIDModel(object):
    def __init__(self):
        self.num_chars = 47
        self.languages = ["English", "Spanish", "Finnish", "Dutch", "Polish"]
        # Initialize your model parameters here
        "*** YOUR CODE HERE ***"
        self.hidden_layer_size = 100                # 隐藏层大小
        self.batch_size = 100                       # 批量大小，dataset 的大小要能被 batch_size 整除
        self.learning_rate = 0.45                   # 学习率
        self.input_size = self.num_chars            # 输入大小，47
        self.output_size = 5                        # 输出大小

        '''
        x : batch_size x input_size
        W_hidden : hidden_layer_size x hidden_layer_size
        Wx : input_size x hidden_layer_size
        Wo : hidden_layer_size x output_size
        '''
        self.W_hidden = nn.Parameter(self.hidden_layer_size, self.hidden_layer_size)
        self.Wx = nn.Parameter(self.input_size, self.hidden_layer_size)
        self.Wo = nn.Parameter(self.hidden_layer_size, self.output_size)

        def run(self, xs):
            # print(len(xs))
            hi = nn.Linear(xs[0],self.Wx)
            h_hidden = nn.ReLU(hi)
```

```python
        # 神经网络模拟
        for i in range(1,len(xs)):
            # xs[i]*Wx + h_hidden*W_hidden
            h_hidden =
            ↪  nn.ReLU(nn.Add(nn.Linear(xs[i],self.Wx),nn.Linear(h_hidden,self.W_hidden)))
        prediction = nn.Linear(h_hidden,self.Wo)
        return prediction

    def get_loss(self, xs, y):
        return nn.SoftmaxLoss(self.run(xs),y)

    def train(self, dataset):
        while 1:
            for xs, y in dataset.iterate_once(self.batch_size):
                loss = self.get_loss(xs, y)
                # 计算损失梯度
                grad_Wx, grad_W_hidden, grad_Wo = nn.gradients(
                    loss, [self.Wx, self.W_hidden, self.Wo])
                # 更新参数
                self.Wx.update(grad_Wx, -self.learning_rate)
                self.W_hidden.update(grad_W_hidden, -self.learning_rate)
                self.Wo.update(grad_Wo, -self.learning_rate)

            # 若准确率高于 83% 则退出
            if dataset.get_validation_accuracy() > 0.83:
                break
```

5.4 实验结果

所有测试用例均已通过



该问题测评按以下顺序：模型的完整性检查——权重更新检查——模型训练

完整性检查主要检查方法实现是否正确，各参数的大小是否合法等。权值更新检查更新后的参数是否包含 NaN 或无穷大的值。模型训练则检查模型能否在有限的时间内训练到预期的准确度。

选取一组用例查看测评机如何评测

```
epoch 4 iteration 172 validation-accuracy 84.6%
  rides      English ( 76.3%)              |en 76%|es 16%|fi  1%|nl  7%|pl  0%
  reverend   English ( 17.9%) Pred: Dutch  |en 18%|es  0%|fi  0%|nl 81%|pl  1%
  cleared    English ( 97.8%)              |en 98%|es  0%|fi  0%|nl  1%|pl  0%
  mirada     Spanish ( 79.1%)              |en  1%|es 79%|fi  2%|nl  1%|pl 17%
  octubre    Spanish ( 36.9%)              |en 33%|es 37%|fi  3%|nl 16%|pl 11%
  conociste  Spanish ( 80.4%)              |en  2%|es 80%|fi  1%|nl 15%|pl  2%
  jätit      Finnish ( 94.7%)              |en  2%|es  0%|fi 95%|nl  3%|pl  0%
  silmänsä   Finnish (100.0%)              |en  0%|es  0%|fi100%|nl  0%|pl  0%
  mutten     Finnish ( 49.2%)              |en  8%|es  0%|fi 49%|nl 42%|pl  0%
  intiem     Dutch   ( 60.7%)              |en 17%|es  1%|fi  1%|nl 61%|pl 20%
  mijzelf    Dutch   ( 81.3%)              |en 12%|es  0%|fi  0%|nl 81%|pl  6%
  weggegaan  Dutch   ( 98.0%)              |en  1%|es  0%|fi  1%|nl 98%|pl  0%
  zachowywać Polish  (100.0%)              |en  0%|es  0%|fi  0%|nl  0%|pl100%
  resztą     Polish  ( 99.5%)              |en  0%|es  0%|fi  0%|nl  0%|pl100%
  wierzą     Polish  (100.0%)              |en  0%|es  0%|fi  0%|nl  0%|pl100%
```

可以看出，评测机通过对每个单词的预测准确率综合计算，从而得出模型的准确率。为减少参数初始化的随机性产生的影响，我连续测评了十次该问题，均以满分通过。

6 总结与分析

6.1 感知器模型

算法理解：感知器是一种最简单的神经网络模型，用于二分类任务。它通过不断迭代训练，调整权重和偏置值，找到一个能够将数据分割成两个类别的超平面。

优点：感知器算法具有简单、易于实现和理解的特点。它对于线性可分的问题能够收敛到一个全局最优解。

缺点：感知器只能解决线性可分问题，对于非线性可分的问题无法收敛。此外，感知器对初始权重和偏置值的选择比较敏感。

适用场景：感知器适用于简单的线性分类问题，例如二分类的逻辑门问题。

6.2 非线性回归模型

算法理解：非线性回归模型是一个多层神经网络模型，用于拟合非线性函数。它包含一个或多个隐藏层，使用非线性的激活函数（如 ReLU）。

优点：非线性回归模型能够拟合复杂的非线性关系，具有较强的表达能力。它可以逼近任意函数，并能够在训练集上获得较低的损失。

缺点：非线性回归模型的训练复杂度较高，需要较长的训练时间和大量的数据。如果模型过于复杂，容易发生过拟合现象。

适用场景：非线性回归模型适用于需要拟合复杂非线性关系的问题，例如图像识别、自然语言处理等。

6.3 手写数字识别模型

算法理解：手写数字识别模型是一个多层感知器（MLP）的分类器，用于将手写数字图像分类为 0 到 9 的数字。

优点：手写数字识别模型能够有效地识别手写数字图像，具有较高的准确率。MLP 模型具有较强的表达能力，能够处理高维输入数据。

缺点：手写数字识别模型的训练时间较长，特别是在大规模数据集上。需要选择合适的网络结构和参数来避免过拟合。

适用场景：手写数字识别模型适用于需要对手写数字进行自动识别的应用，例如邮件排序、数字验证码识别等。

6.4 单词分类模型

算法理解：单词语言分类模型是一个使用循环神经网络（RNN）的分类器，用于将输入的单词序列分类为不同的语言。

优点：单词语言分类模型能够捕捉单词序列中的上下文信息，具有较好的语言特征提取能力。RNN 模型适用于处理序列数据。

缺点：单词语言分类模型的训练复杂度较高，需要较长的训练时间和大量的数据。RNN 模型在处理长序列时容易出现梯度消失或梯度爆炸问题。

适用场景：单词语言分类模型适用于文本分类、语言识别等需要处理序列数据的任务。

6.5 调参心得

在调参过程中，以下是一些关键参数的调整方法：

1. 学习率（Learning Rate）：学习率决定了模型在每次迭代中更新权重和偏置值的幅度。通常从一个较小的学习率开始，然后逐渐增大学习率，以获得更好的收敛性和稳定性。如果学习率过大，可能导致模型无法收敛；如果学习率过小，训练过程可能会很慢。

2. 权重和偏置值（Weights and Biases）：权重和偏置值是模型的参数，可以通过随机初始化或预训练模型进行初始化。在训练过程中，可以使用不同的权重初始化策略，并观察其对模型性能的影响。另外，正则化技术（如 L1 或 L2 正则化）可以应用于权重以避免过拟合。

3. 隐藏层大小（Hidden Layer Size）：隐藏层的大小决定了模型的复杂性和表达能力。增加隐藏层的大小可以增加模型的容量，使其能够更好地拟合训练数据。然而，过大的隐藏层可能导致过拟合，而过小的隐藏层可能限制了模型的表达能力。

4. 批量大小（Batch Size）：批量大小决定了在每次参数更新中使用的样本数量。较大的批量大小可以加速训练过程，但也可能导致内存消耗增加。较小的批量大小可以提供更稳定的梯度估计，但训练过程可能更慢。

总之，调参是一个迭代的过程，需要根据具体问题和数据集的特点进行调整。通过尝试不同的参数组合，结合观察模型在验证集上的表现，可以找到最佳的参数配置，以获得最好的模型性能。同时，注意避免过拟合和欠拟合问题，以及控制训练时间和计算资源的消耗。

感谢老师和助教们一学期的教导与陪伴，这门课将我带入了人工智能的大门，让我收获颇丰。祝愿老师和助教们前程似锦，也希望我校、我国未来的人工智能发展蒸蒸日上。