**同济大学计算机系**

**人工智能课程设计综合实验报告3**

****

**实验名称 Machine Learning**

**学 号**

**姓 名**

**专 业 计算机科学与技术**

**授课老师 武妍老师**

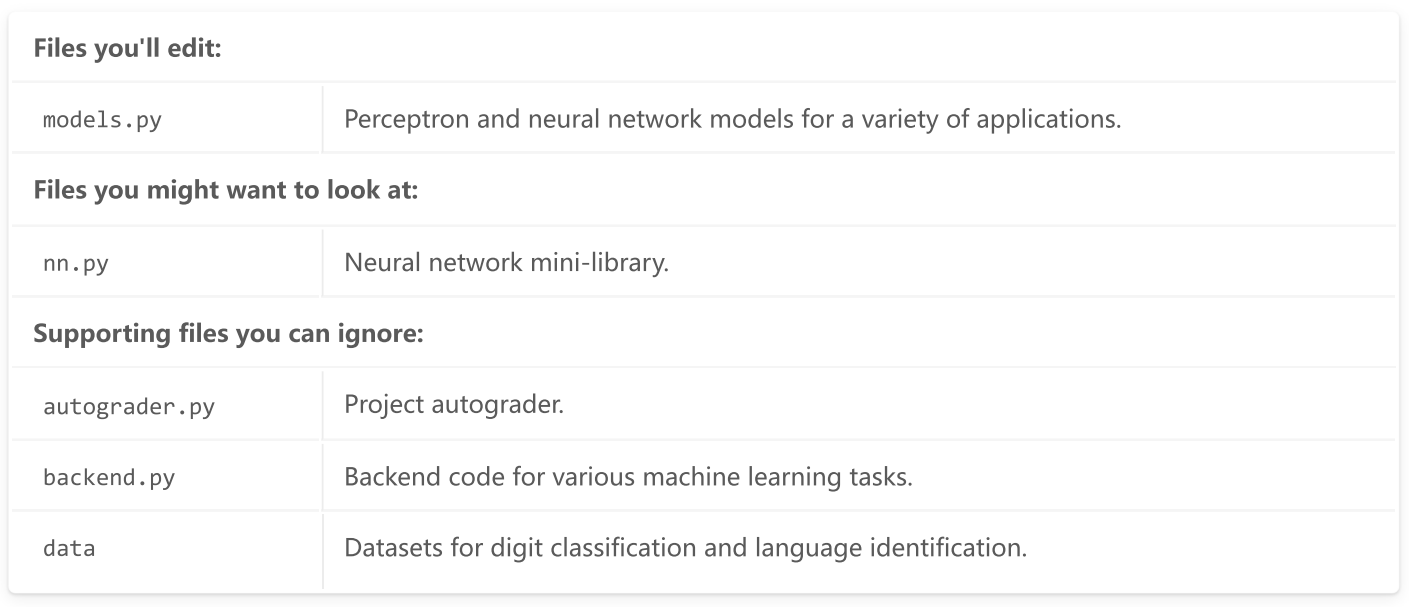
**完成日期 2024年6月6日**

1. 项目概述与分析

**1、项目的总体概述**

在这次的实验中，需要通过搭建不同的神经网络来完成一些不同的任务，包括一个二进制感知机，实现对已有结点的分类；模拟近似一个正弦函数；训练一个网络来识别手写的数字；训练一个网络来识别一个单词属于哪种语言。以上的任务都要求识别的准确率或者模拟的精度达到一定的程度。

**2、项目已有代码分析**

****

1. **需要完成的代码models.py**

PerceptronModel类：包含训练一个二进制感知机的方法的类；

RegressionModel类：实现用神经网络拟合近似正弦函数的方法；

DigitClassificationModel类：训练神经网络来实现手写数字的识别；

LanguageIDModel类：训练神经网络来识别一个单词属于哪种语言；

\_\_init\_\_函数：初始化参数，包括不同层神经网络的权重矩阵和偏移矩阵；

run函数：使用现有的参数来预测对应输入的输出；

get\_loss函数：返回现有参数预测的损失；

train函数：从数据集中取数据，使用参数来进行预测对应输出，计算损失值，并根据损失值调整参数矩阵的值，重复这个过程直到精度到达一定程度。

1. **需要参考的代码nn.py**

包含一些用于训练神经网络的库函数；

nn.Parameter代表一个可训练的感知器或神经网络的参数；

nn.DotProduct计算其输入的点积；

nn.as\_scalar可以从节点中提取一个Python浮点数；

nn.Add按元素对矩阵进行加法；

nn.AddBias将偏置向量添加到每个特征向量；

nn.Linear对输入应用线性变换（矩阵乘法）；

nn.ReLU应用逐元素的修正线性单元非线性函数；

nn.SoftmaxLoss计算批次的softmax损失，用于分类问题；

nn.gradients计算相对于提供的参数的损失梯度。

1. **需要参考的代码backend.py**

通过调用dataset.iterate\_once（batch\_size）来获取训练示例的批次；

dataset.iterate\_forever(batch\_size)生成一个无限的批次示例序列；

dataset.get\_validation\_accuracy()返回模型在验证集上的准确率；

**3、子问题概述及简要分析**

**Q1：Perceptron**

通过计算输入和参数的点积并以此计算出输出预测值，根据预测值和真实值的差别来调整参数，直到数据集中所有数都能够被正确分类。

**Q2：Non-linear Regression**

搭建一个神经网络，根据这个神经网络中的参数值来计算输入对应的输出值，根据预测值的损失来对参数进行更新，直到损失值低到一定程度，这表明拟合的精度到达了一定的高度。

**Q3：Digit Classification**

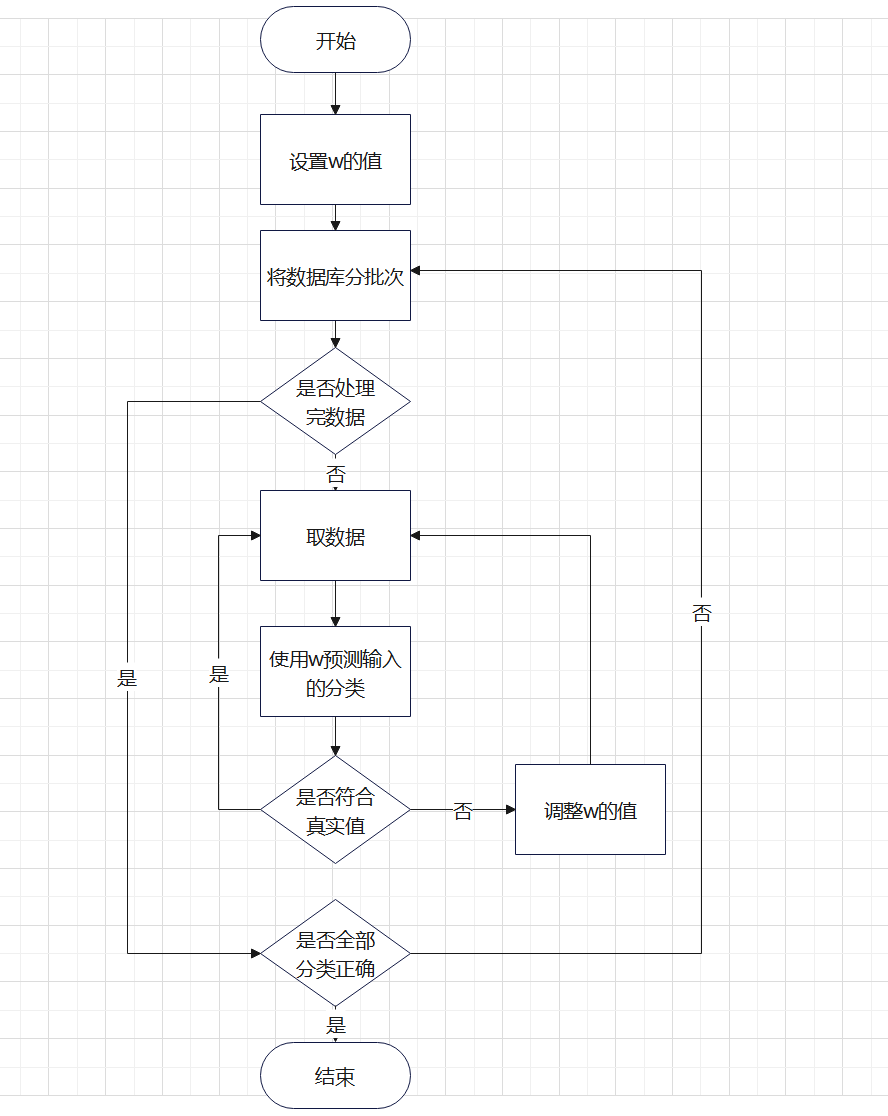
搭建一个神经网络，根据这个神经网络对输入的手写图片进行映射，映射到0-9的分类中，不断的根据损失值来调整参数的值，直到准确率到达一定的水准。

**Q4：Language Identificaion**

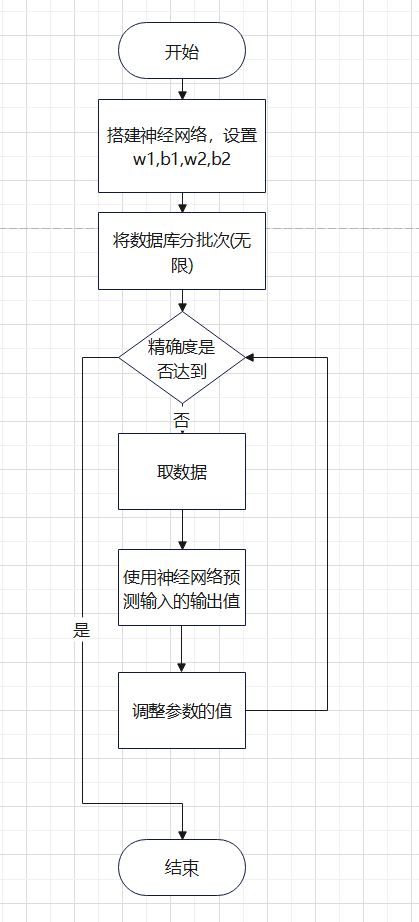
搭建一个神经网络，每读入一个字母，就使用输入层对其进行映射，并使用隐藏层对之前得到的映射结果进行映射，将二者之和作为新的映射结果，直到最后一个字母读入完毕，再使用输出层映射得到最后的预测结果，根据损失值不停调整参数值，直到精度到达一定的程度。

1. 流程图

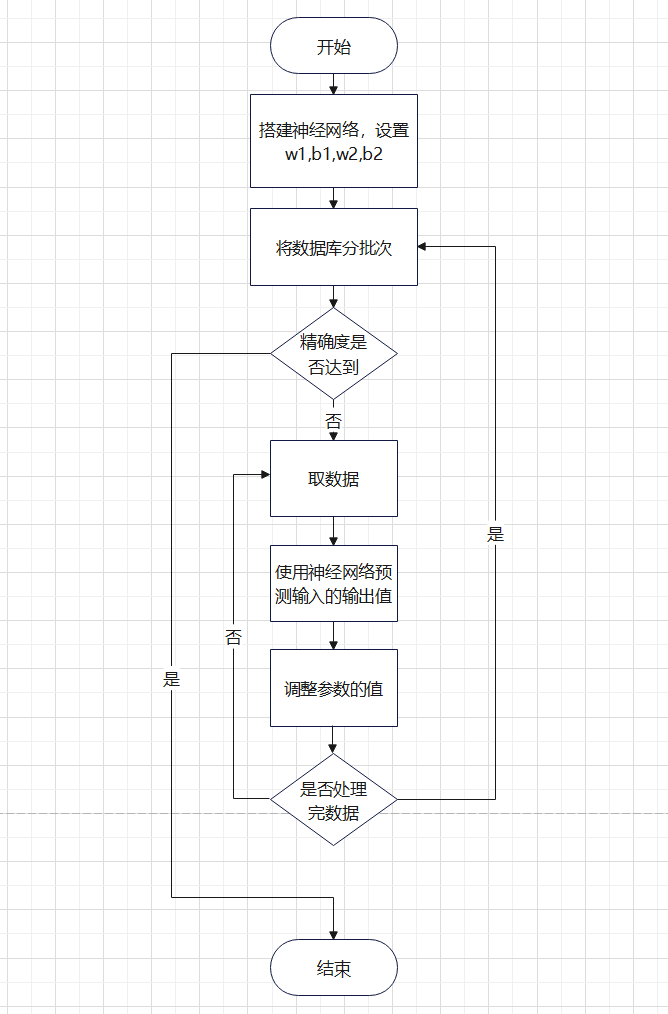
**Q1：Perceptron**



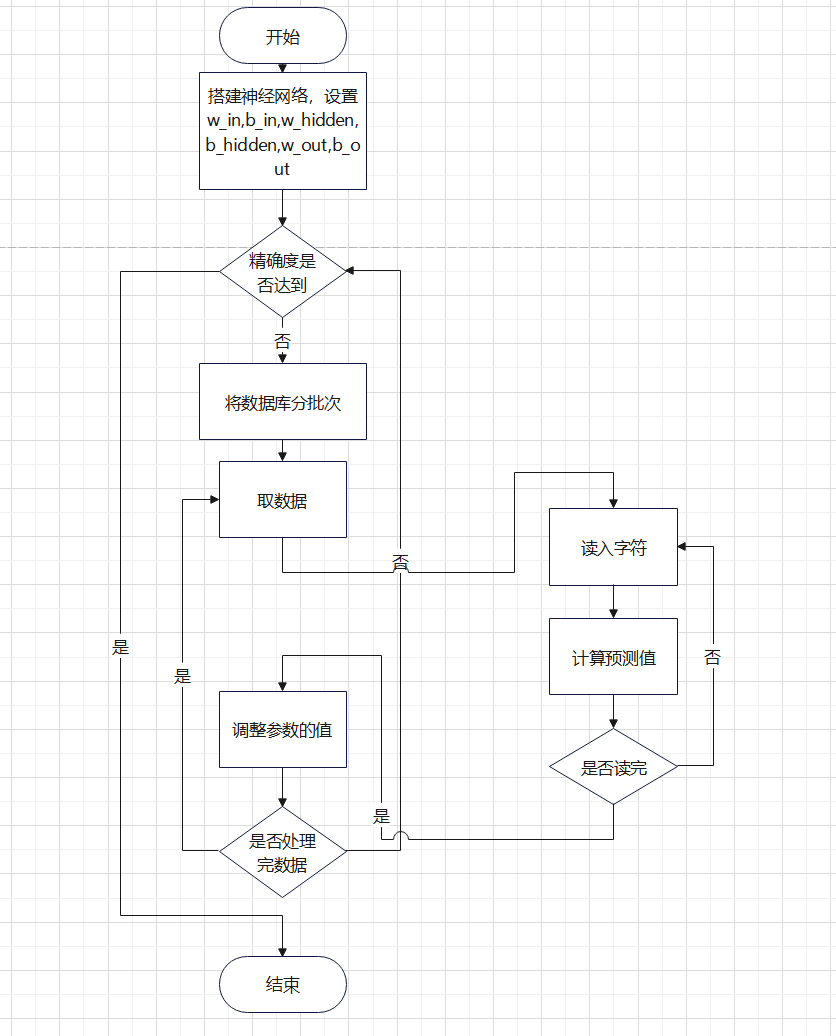
**Q2：Non-linear Regression**



**Q3：Digit Classification**



**Q4：Language Identificaion**



1. 算法设计与实现

**Q1：Perceptron**

初始化参数矩阵，计算输入和参数矩阵的点积并据此对输入进行分类，判断分类的预测值与真实值是否相同，若不相同则调整参数值，直到数据集中所有的数据都能够被正确的分类，结束。

class PerceptronModel(object):

    def \_\_init\_\_(self, dimensions):

        """

        Initialize a new Perceptron instance.

        A perceptron classifies data points as either belonging to a particular

        class (+1) or not (-1). `dimensions` is the dimensionality of the data.

        For example, dimensions=2 would mean that the perceptron must classify

        2D points.

        """

        self.w = nn.Parameter(1, dimensions)

    def get\_weights(self):

        """

        Return a Parameter instance with the current weights of the perceptron.

        """

        return self.w

    def run(self, x):

        """

        Calculates the score assigned by the perceptron to a data point x.

        Inputs:

            x: a node with shape (1 x dimensions)

        Returns: a node containing a single number (the score)

        """

        "\*\*\* YOUR CODE HERE \*\*\*"

        return nn.DotProduct(x, self.w)

    def get\_prediction(self, x):

        """

        Calculates the predicted class for a single data point `x`.

        Returns: 1 or -1

        """

        "\*\*\* YOUR CODE HERE \*\*\*"

        return 1 if nn.as\_scalar(nn.DotProduct(self.w, x)) >= 0 else -1

    def train(self, dataset):

        """

        Train the perceptron until convergence.

        """

        "\*\*\* YOUR CODE HERE \*\*\*"

        while True:

            DONE = True

            for x, y in dataset.iterate\_once(1):

                if nn.as\_scalar(y) != self.get\_prediction(x):

                    DONE = False

                    self.w.update(x, nn.as\_scalar(y))

            if DONE:

                break

权重矩阵的大小设置时已经给定的，同时已经规定batchs的值为1，即每次处理一个数据，因此在该题目中无需自己去搭建神经网络，只需要不停的更新参数的值直到符合要求即可。

**Q2：Non-linear Regression**

搭建神经网络，包含一个隐藏层，确定神经网络的深度和大小，使用该神经网络对输入进行映射，根据损失值来进行参数的调整，直到近似的精度到达一定的程度，结束。

class RegressionModel(object):

    """

    A neural network model for approximating a function that maps from real

    numbers to real numbers. The network should be sufficiently large to be able

    to approximate sin(x) on the interval [-2pi, 2pi] to reasonable precision.

    """

    def \_\_init\_\_(self):

        # Initialize your model parameters here

        "\*\*\* YOUR CODE HERE \*\*\*"

        self.batch\_size = 50

        self.w0 = nn.Parameter(1, 80)

        self.b0 = nn.Parameter(1, 80)

        self.w1 = nn.Parameter(80, 1)

        self.b1 = nn.Parameter(1, 1)

        self.alpha = 0.005

    def run(self, x):

        """

        Runs the model for a batch of examples.

        Inputs:

            x: a node with shape (batch\_size x 1)

        Returns:

            A node with shape (batch\_size x 1) containing predicted y-values

        """

        "\*\*\* YOUR CODE HERE \*\*\*"

        xw1 = nn.Linear(x, self.w0)

        r1 = nn.ReLU(nn.AddBias(xw1, self.b0))

        xw2 = nn.Linear(r1, self.w1)

        return nn.AddBias(xw2, self.b1)

    def get\_loss(self, x, y):

        """

        Computes the loss for a batch of examples.

        Inputs:

            x: a node with shape (batch\_size x 1)

            y: a node with shape (batch\_size x 1), containing the true y-values

                to be used for training

        Returns: a loss node

        """

        "\*\*\* YOUR CODE HERE \*\*\*"

        return nn.SquareLoss(self.run(x), y)

    def train(self, dataset):

        """

        Trains the model.

        """

        "\*\*\* YOUR CODE HERE \*\*\*"

        while True:

            for x, y in dataset.iterate\_once(self.batch\_size):

                loss = self.get\_loss(x, y)

                grad = nn.gradients(loss, [self.w0, self.w1, self.b0, self.b1])

                self.w0.update(grad[0], -self.alpha)

                self.w1.update(grad[1], -self.alpha)

                self.b0.update(grad[2], -self.alpha)

                self.b1.update(grad[3], -self.alpha)

            if nn.as\_scalar(self.get\_loss(nn.Constant(dataset.x), nn.Constant(dataset.y))) < 0.01:

                return

**Q3：Digit Classification**

搭建神经网络，确定神经网络的深度和隐藏层的大小，使用该神经网络对输入结点进行映射，识别是哪一个数字，根据损失值调整参数，直到识别的精度到达一定的程度，结束。

class DigitClassificationModel(object):

    """

    A model for handwritten digit classification using the MNIST dataset.

    Each handwritten digit is a 28x28 pixel grayscale image, which is flattened

    into a 784-dimensional vector for the purposes of this model. Each entry in

    the vector is a floating point number between 0 and 1.

    The goal is to sort each digit into one of 10 classes (number 0 through 9).

    (See RegressionModel for more information about the APIs of different

    methods here. We recommend that you implement the RegressionModel before

    working on this part of the project.)

    """

    def \_\_init\_\_(self):

        # Initialize your model parameters here

        "\*\*\* YOUR CODE HERE \*\*\*"

        self.batch\_size = 5

        self.w0 = nn.Parameter(784, 100)

        self.b0 = nn.Parameter(1, 100)

        self.w1 = nn.Parameter(100, 10)

        self.b1 = nn.Parameter(1, 10)

        self.multiplier = -0.004

    def run(self, x):

        """

        Runs the model for a batch of examples.

        Your model should predict a node with shape (batch\_size x 10),

        containing scores. Higher scores correspond to greater probability of

        the image belonging to a particular class.

        Inputs:

            x: a node with shape (batch\_size x 784)

        Output:

            A node with shape (batch\_size x 10) containing predicted scores

                (also called logits)

        """

        "\*\*\* YOUR CODE HERE \*\*\*"

        w1x = nn.Linear(x, self.w0)

        w1x\_plus\_b1 = nn.AddBias(w1x, self.b0)

        r1 = nn.ReLU(w1x\_plus\_b1)

        w2x = nn.Linear(r1, self.w1)

        w2x\_plus\_b2 = nn.AddBias(w2x, self.b1)

        return w2x\_plus\_b2

    def get\_loss(self, x, y):

        """

        Computes the loss for a batch of examples.

        The correct labels `y` are represented as a node with shape

        (batch\_size x 10). Each row is a one-hot vector encoding the correct

        digit class (0-9).

        Inputs:

            x: a node with shape (batch\_size x 784)

            y: a node with shape (batch\_size x 10)

        Returns: a loss node

        """

        "\*\*\* YOUR CODE HERE \*\*\*"

        return nn.SoftmaxLoss(self.run(x), y)

    def train(self, dataset):

        """

        Trains the model.

        """

        "\*\*\* YOUR CODE HERE \*\*\*"

        while True:

            for x, y in dataset.iterate\_once(self.batch\_size):

                loss = self.get\_loss(x, y)

                grad = nn.gradients(loss, [self.w0, self.w1, self.b0, self.b1])

                self.w0.update(grad[0], self.multiplier)

                self.w1.update(grad[1], self.multiplier)

                self.b0.update(grad[2], self.multiplier)

                self.b1.update(grad[3], self.multiplier)

            if dataset.get\_validation\_accuracy() >= 0.974:

                return

**Q4：Language Identificaion**

搭建神经网络，包含三个部分，输入层，隐藏层，输出层，不断的读入字符直到单词被读取完，在读取的过程中反复的使用输入层和隐藏层进行映射得到新的结果，在读入完毕之后将得到的结点使用输出层映射输出，根据损失值调整参数，直到准确率达到一定程度，结束。

class LanguageIDModel(object):

    """

    A model for language identification at a single-word granularity.

    (See RegressionModel for more information about the APIs of different

    methods here. We recommend that you implement the RegressionModel before

    working on this part of the project.)

    """

    def \_\_init\_\_(self):

        # Our dataset contains words from five different languages, and the

        # combined alphabets of the five languages contain a total of 47 unique

        # characters.

        # You can refer to self.num\_chars or len(self.languages) in your code

        self.num\_chars = 47

        self.languages = ["English", "Spanish", "Finnish", "Dutch", "Polish"]

        # Initialize your model parameters here

        "\*\*\* YOUR CODE HERE \*\*\*"

        self.batch\_size = 10

        self.alpha = 0.01

        self.w0 = nn.Parameter(self.num\_chars, 300)

        self.b0 = nn.Parameter(1, 300)

        self.w1 = nn.Parameter(300, 300)

        self.b1 = nn.Parameter(1, 300)

        self.wf = nn.Parameter(300, 5)

        self.bf = nn.Parameter(1, 5)

    def run(self, xs):

        "\*\*\* YOUR CODE HERE \*\*\*"

        h = nn.ReLU(nn.AddBias(nn.Linear(xs[0], self.w0), self.b0))

        z = h

        for i, x in enumerate(xs[1:]):

            z = nn.Add(nn.ReLU(nn.AddBias(nn.Linear(z, self.w1), self.b1)),

                       nn.ReLU(nn.AddBias(nn.Linear(x, self.w0), self.b0)))

            z = nn.Add(nn.ReLU(nn.Linear(z, self.w1)), nn.ReLU(nn.Linear(x, self.w0)))

        return nn.AddBias(nn.Linear(z, self.wf), self.bf)

    def get\_loss(self, xs, y):

        """

        Computes the loss for a batch of examples.

        The correct labels `y` are represented as a node with shape

        (batch\_size x 5). Each row is a one-hot vector encoding the correct

        language.

        Inputs:

            xs: a list with L elements (one per character), where each element

                is a node with shape (batch\_size x self.num\_chars)

            y: a node with shape (batch\_size x 5)

        Returns: a loss node

        """

        "\*\*\* YOUR CODE HERE \*\*\*"

        return nn.SoftmaxLoss(self.run(xs), y)

    def train(self, dataset):

        """

        Trains the model.

        """

        "\*\*\* YOUR CODE HERE \*\*\*"

        while True:

            for x, y in dataset.iterate\_once(self.batch\_size):

                loss = self.get\_loss(x, y)

                grad = nn.gradients(loss, [self.w0, self.w1, self.wf, self.b0, self.b1, self.bf])

                self.w0.update(grad[0], -self.alpha)

                self.w1.update(grad[1], -self.alpha)

                self.wf.update(grad[2], -self.alpha)

                self.b0.update(grad[3], -self.alpha)

                self.b1.update(grad[4], -self.alpha)

                self.bf.update(grad[5], -self.alpha)

            if dataset.get\_validation\_accuracy() >= 0.9:

                return

                value = self.expectimaxSelect(legalState, numAgents, layer + 1)

                if maxVal < value[0]:

                    maxVal = value[0]

                    maxAction = action

                avgVal += value[0]

            avgVal /= actionCount  # 如果是最小结点，则该结点的值为子结点值的平均值

            if layer % numAgents == 0:

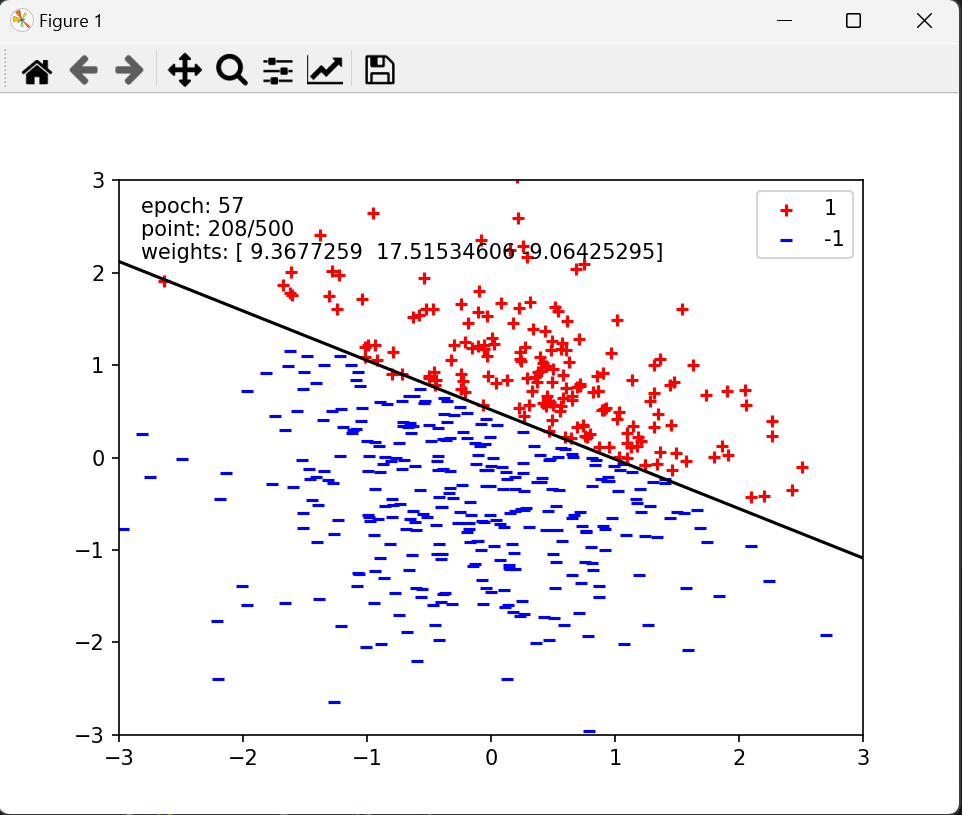
                return [maxVal, maxAction]

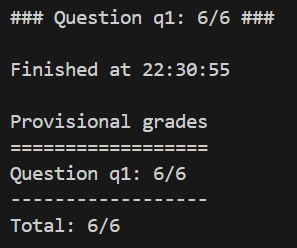
            else:

                return [avgVal, actionChoice]

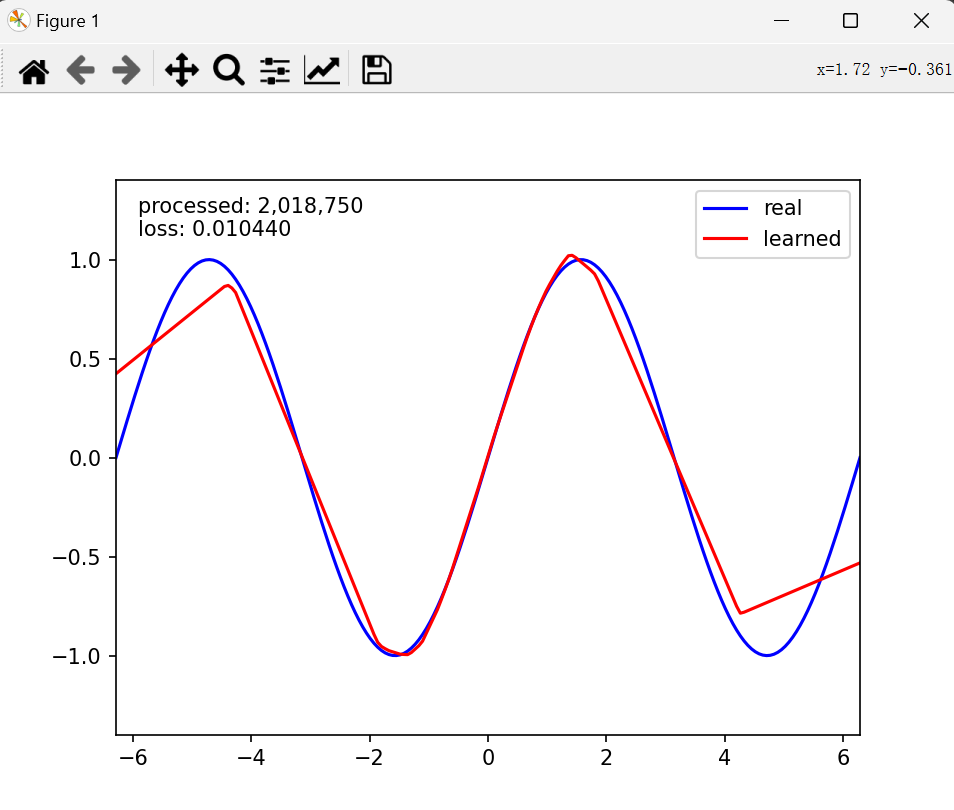
1. 结果展示

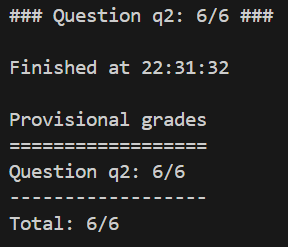
**Q1：Perceptron**

****

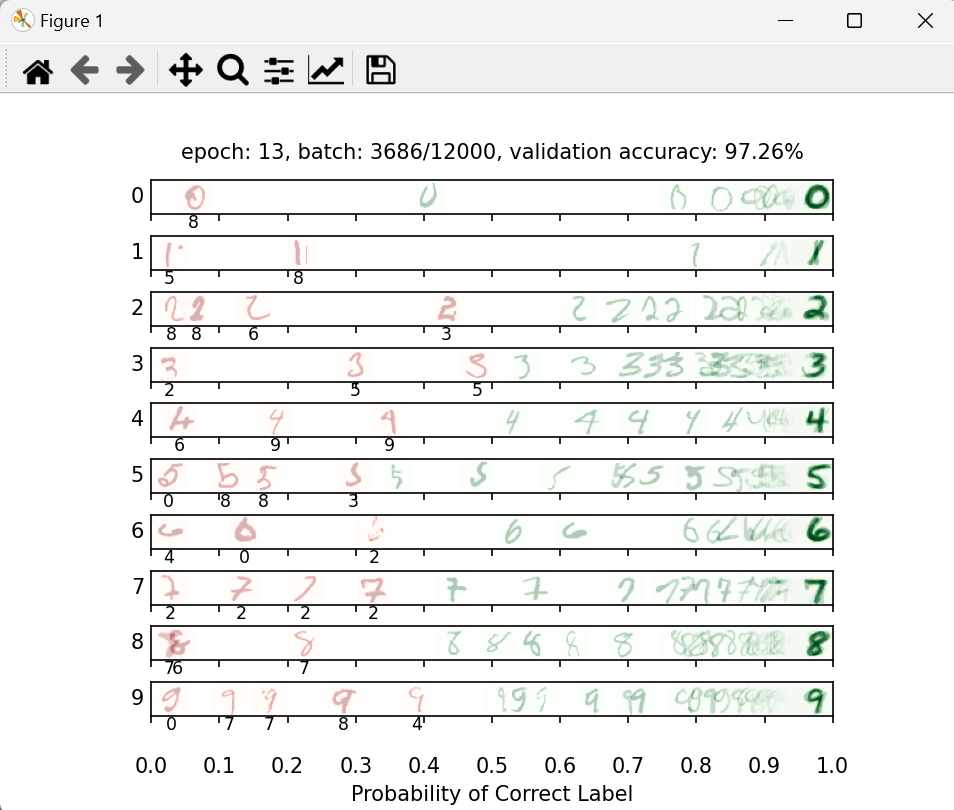
****

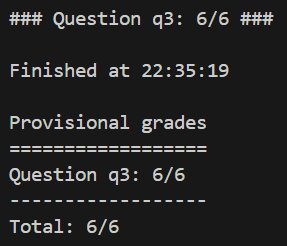
**Q2：Non-linear Regression**

****

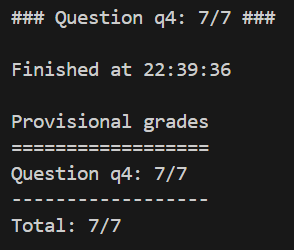
****

**Q3：Digit Classification**

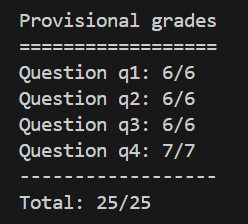
****

****

**Q4：Language Identificaion**



**Total Result**



1. 总结与分析

**1、遇到的困难及解决方法**

在调试过程中遇到的主要困难是神经网络的搭建，尤其是在搭建最后一个语言识别的时候，其明显区别于前两道题的结构，同时由于题目没有给出一些参数和网络搭建的建议，这道题花费了我很多时间去尝试，但是在尝试的过程中我逐渐理解了循环神经网络的原理，最终搭建了一个三层了网络，并选择了合适的参数，完成了这道题目。

**2、项目总结与个人收获**

通过这次项目，我算是比较好地完成了项目要求任务。在Perceptron项目中理解了感知机的基本原理和二进制分类问题；学会了使用感知机模型来解决简单的分类任务；掌握了参数更新和训练过程中的梯度计算。在Non-linear Regression项目中掌握了如何使用神经网络来拟合和预测函数值。在Digit Classification项目中了解图像分类问题和数字识别的基本概念和简单方法，在Language Identificaion项目中理解了文本分类问题和语言识别的基本原理。

在这次的实验中，我学习并理解了机器学习的基本概念，了解了神经网络的结果，并能够搭建简单的神经网络来解决一些简单的问题，明白了参数的选择对于机器学习的意义所在。