07 Machine Learning I

本节实验目标：练习感知器和（线性/非线性）回归

报告说明（重要）：本部分“机器学习”包含两周内容，本节课为第一周内容，请留意该文档目录中标有“报告应实现任务[分值]”的部分，请在实现后及时保存文件和运行结果截图，在下一周课程报告中一并提交。

Introduction

本堂课的代码文件包含：

你应该要编辑的文件:	
models.py	**适用于各种应用的感知器和神经网络模型**
你应该看但不要编辑的文件:	
nn.py	**神经网络迷你库**
你可以忽略的文件:	
autograder.py	Project autograder
backend.py	Backend code for various machine learning tasks
data	Datasets for digit classification and language identification
submission_autograder.py	Submission autograder (generates tokens for submission)

- Autograder会评判你的实现。请**不要** 更改代码中提供的任何函数或类的名称。
- **正确使用数据集**：你在此项目中的部分分数取决于你训练的模型在Autograder随附的测试集上的表现。我们不提供任何 API 供你直接访问测试集。

安装python包

对于这个项目，你需要安装以下两个库：

- numpy，提供对大型多维数组的支持 - installation instructions
- matplotlib，一个 2D 绘图库 - 安装说明

如果你有 conda 环境，你可以通过运行以下命令在命令行上安装这两个包：

```
conda activate [your environment name]
```

```
pip install numpy
```

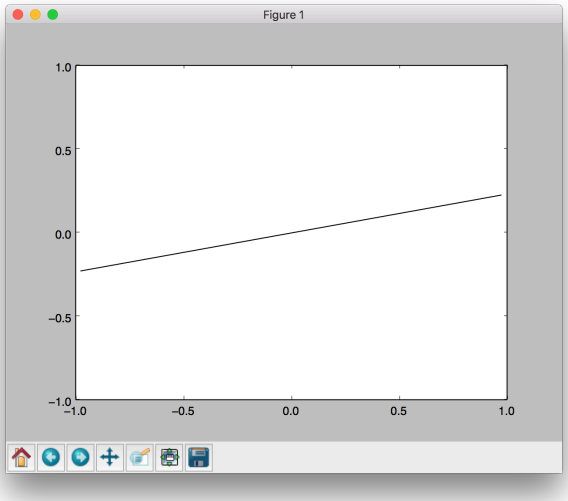
```
pip install matplotlib
```

你不会直接使用这些库，但它们是运行提供的代码和autograder所必需的。

要测试是否已安装所有内容，请运行：

```
python autograder.py --check-dependencies
```

如果 `numpy` 和 `matplotlib` 已正确安装，你应该会看到一个窗口弹出，其中一条线段在一个圆圈中旋转：



提供的nn.py和数据集

我们为你提供了一个神经网络迷你库（`nn.py`）和一组数据集（`backend.py`）。

`nn.py` 中的库定义了节点对象的集合。每个节点代表一个实数或实数矩阵。这个库对节点对象的操作进行了优化，比使用 Python 的内置类型（例如列表lists）更快。

以下是一些提供的节点类型：

- `nn.Constant` 表示浮点数的矩阵（二维数组）。它通常用于表示输入特征或目标输出/标签。该类型的实例将由 API 中的其他函数提供给你；你不需要直接构建它们。
- `nn.Parameter` 表示感知器或神经网络的可训练参数。
- `nn.DotProduct` 计算其输入之间的点积。

额外提供的功能：

- `nn.as_scalar` 可以从节点中提取 Python 浮点数。

在训练感知器或神经网络时，你将收到一个“数据集”对象。你可以通过调用 `dataset.iterate_once(batch_size)` 来检索一个批量的训练示例（batch_size需要自己定义）比如1,2,4...：

```
for x, y in dataset.iterate_once(batch_size):
...
```

例如，让我们从感知器训练数据中提取大小为 1 的批次（即单个训练示例）：

```
>>> batch_size = 1
>>> for x, y in dataset.iterate_once(batch_size):
...     print(x)
...     print(y)
...     break
...
<Constant shape=1x3 at 0x11a8856a0>
<Constant shape=1x1 at 0x11a89efd0>
```

输入的特征 `x` 和正确标签 `y` 以 `nn.Constant` 节点的形式提供。 `x` 的形状（shape）是 `batch_size x num_features`，y 的形状是 `batch_size x num_outputs`。

以下是一个计算 `x` 与自身的点积的示例，第一步中 `x` 作为节点，第二步输出的 `x` 是该结点对应的 Python 浮点数值。

```
>>> nn.DotProduct(x, x)
<DotProduct shape=1x1 at 0x11a89edd8>
>>> nn.as_scalar(nn.DotProduct(x, x))
1.9756581717465536
```

Question 1: Perceptron 报告应实现任务[1分]

- **说明：该小题截图只需截 python autograder.py -q q1 的运行结果，获得评分器给的全部8分，这道题才算得它占的1分。**

在开始这题之前，请确保你已经安装了 `numpy` 和 `matplotlib`！

在这一部分中，你将实现一个二进制感知器。你的**任务是在 `models.py` 中完成 `PerceptronModel` 类的实现。**

对于感知器，输出标签将是 +1 或 -1，这意味着数据集中的数据点 `(x, y)` 将具有一个表示 `y` 的 `nn.Constant` 节点， `y` 取值为 +1 或 -1。你可以查看"**7_Q1二进制感知器补充知识.pdf**"中的Binary Perceptron的 +1/-1 感知器示例及其算法（跟参考教材和课件上的+1/0感知器形式上有点区别，但本质上是一样的）.

我们已经将感知器权重 `self.w` 初始化为 (1 × *dimensions*) 参数节点（测试数据会指定dimensions，不用管）。提供的代码将在需要时在 `x` 内包含一个偏差特征(bias feature)，因此你不需要单独的偏差参数。

你的任务是：

- 实现 `run(self, x)` 方法。这应该计算存储的权重向量和给定输入的点积，返回一个 `nn.DotProduct` 对象。
- 实现 `get_prediction(self, x)`，如果上述点积为非负数，则返回 1，否则返回-1。你应该使用 `nn.as_scalar` 将标量(scalar) `Node` 转换为 Python 浮点数。
- 实现 `train(self)` 方法。这应该反复循环数据集并对错误分类的样本进行更新。使用 `nn.Parameter` 类的 `update` 方法更新权重。当整个数据集的一次训练完成且没有任何错误时，已经达到 100% 的训练准确率，训练可以终止。

在给定的代码中，改变参数值的唯一方法是调用 `parameter.update(direction, multiplier)`，它会更新权重：[*weights* ← *weights* + *direction*·*multiplier*]，其中 `direction` 是与参数形状相同的 `Node`， `multiplier` 是 Python 标量(scalar)，parameter指的是一个参数变量，如w = nn.Parameter(1, dimension)。

要测试你的实现，请运行:

```
python autograder.py -q q1
```

注意：如果实现正确，autograder最多需要 20 秒左右的时间来运行。 如果autograder运行很慢，你的代码可能有错误。

神经网络小技巧

构建神经网络

在本堂课中，你将使用 `nn.py` 中提供的框架来构建神经网络来解决各种机器学习问题。一个简单的神经网络有层(layers)，每一层都执行线性操作（linear operation, 就像感知器一样）。层由非线性*(non-linearity)* 分隔，这允许网络模拟一些通用函数。我们将使用 ReLU 操作来实现非线性，定义为 $relu(x) = \max(x, 0)$。例如，用于将输入行向量 $\mathbf{x}$ 映射到输出向量 $\mathbf{f(x)}$ 的简单两层神经网络将由以下函数给出：$\mathbf{f(x)} = relu(\mathbf{x} \cdot \mathbf{W_1} + \mathbf{b_1}) \cdot \mathbf{W_2} + \mathbf{b_2}$，其中我们在梯度下降期间学习参数矩阵 $\mathbf{W_1}$ 和 $\mathbf{W_2}$ 以及参数向量 $\mathbf{b}_1$和 $\mathbf{b}_2$ 。 $\mathbf{W_1}$ 将是一个 $i \times h$ 矩阵，其中 i 是我们输入向量 $\mathbf{x}$ 的维度，h 是隐藏层大小 *(hidden layer size)*。 $\mathbf{b}_1$ 是一个大小为 h 的向量。我们可以自由选择任何我们想要的隐藏层大小值（我们只需要确保其他矩阵和向量的维度一致，以便我们可以执行操作）。使用更大的隐藏层大小值通常会使网络更强大（能够适应更多的训练数据），但会使网络更难训练（因为它为我们需要学习的所有矩阵和向量添加了更多参数），或者可能导致过度拟合训练数据。

我们还可以通过添加更多层来创建更深的网络，例如三层网络：$\mathbf{f(x)} = relu(relu(\mathbf{x} \cdot \mathbf{W_1} + \mathbf{b_1}) \cdot \mathbf{W_2} + \mathbf{b_2}) \cdot \mathbf{W_3} + \mathbf{b_3}$

批处理

为了提高效率，你将需要一次处理整批数据，而不是一次处理一个样本。 这意味着你将看到一批表示为 $b \times i$ 大小的矩阵 X 的 b 个输入，而不是大小为 i 的单个输入行向量 x。 我们提供了一个线性回归示例来演示如何在批处理设置中实现一个线性层。

随机性

你的神经网络的参数会被随机初始化，一些任务中的数据会以打乱(shuffled)的顺序呈现。 由于这种随机性，即使使用强大的架构，你仍然可能偶尔会失败一些任务——这是局部最优的问题！ 不过，这种情况应该很少发生——如果在测试代码时，autograder连续两次在一个问题上失败，则应该探索其他架构。

实用技巧

设计神经网络可能需要反复试验。以下是一些可能帮助到你的提示：

- Be systematic 系统化。记录你尝试过的每个架构、超参数（层大小、学习率等）是什么，以及对应的性能结果是什么。当你尝试更多的东西时，你可以开始看到一些参数可能在这个任务中更重要。如果你在代码中发现错误，请务必删除记录中因该错误而无效的过去结果。
- Start with a shallow network 从浅层网络开始（只有两层）。更深的网络具有成倍增加的超参数组合，即使一个错误也可能会破坏你的模型性能。使用小型网络找到好的学习率和层大小；之后，你可以考虑添加更多类似大小的层。
- If your learning rate is wrong, none of your other hyperparameter choices matter 如果你的学习率是错误的，那么你的其他超参数选择都无关紧要了。作为验证，你可以从研究论文中获取最先进的模型，并更改学习率，使其表现不比随机更好。学习率太低会导致模型学习太慢，学习率太高可能会导致损失发散到无穷大。首先尝试不同的学习率，同时观察损失如何随着时间的推移而减少。
- Smaller batches require lower learning rates 较小的批次需要较低的学习率。在尝试不同的批大小时，请注意最佳学习率可能会因批量大小而异。
- Refrain from making the network too wide (hidden layer sizes too large) 避免让网络太宽（隐藏层尺寸太大）如果你持续让网络更宽，精度最终会逐渐下降，计算时间会随着层尺寸的二次方增加——你很可能会因为训练太慢而放弃。
- 如果你的模型返回 Infinity 或 NaN，则说明对于你当前的架构，你的学习率可能太高。
- 超参数的推荐值：
  - Hidden layer sizes 隐藏层大小：10 到 400 之间。
  - Batch size 批量大小：介于 1 和数据集大小之间。对于 Q2，我们要求数据集的总大小可以被批量大小整除。
  - Learning rate 学习率：0.001 到 1.0 之间。
  - Number of hidden layers 隐藏层数：1 到 3 之间。

---

nn.py的详细说明(Q2所需)

这是 `nn.py` 中可用节点的完整列表：

- `nn.Constant` 表示浮点数的矩阵（二维数组）。它通常用于表示输入特征或目标输出/标签。该类型的实例将由 API 中的其他函数提供给你；你不需要直接构建它们。
- `nn.Parameter` 表示感知器或神经网络的可训练参数。所有参数必须是二维的。
  - 用法：`nn.Parameter(n, m)` 构造一个形状为 $n \times m$ 的参数
- `nn.Add` 矩阵相加.
  - 用法：`nn.Add(x, y)` 接受两个形状为 *batch_size* x *num_features* 的节点，并构造一个形状也为 *batch_size* x *num_features* 的节点。
- `nn.AddBias` 为每个特征向量添加一个偏置向量。
  - 用法：`nn.AddBias(features, bias)` 接受形状为 *batch_size* x *num_features* 的 `features` 和形状为 1 x *num_features*的 `bias`，构造一个形状为 *batch_size* x *num_features* 的节点。
- `nn.Linear` 对输入应用线性变换（矩阵乘法）。
  - 用法：`nn.Linear(features, weights)` 接受形状为*batch_size* x *num_input_features* 的 `features` 和形状为 *num_input_features* x *num_output_features*的 `weights`，构造一个形状为*batch_size* x *num_output_features* 的节点。

- `nn.ReLU` 应用element-wise Rectified Linear Unit nonlinearity $relu(x) = max(x, 0)$。这种nonlinearity将其输入中的所有负项替换为零。
  - 用法：`nn.ReLU(features)`，它返回一个与`input`形状相同的节点。
- `nn.SquareLoss` 计算批量平方损失，用于回归问题
  - 用法：`nn.SquareLoss(a, b)`，其中`a`和`b`都具有形状 *batch_size* × *num_outputs*。
- `nn.SoftmaxLoss` 计算批量 softmax 损失，用于分类问题。
  - 用法：`nn.SoftmaxLoss(logits, labels)`，其中`logits`和`labels`的形状都是 *batch_size* × *num_classes*。术语“logits”是指模型产生的分数，其中每个条目可以是任意实数。但是，标签必须是非负数，并且每行总和为 1。确保不要交换参数的顺序！
- 请勿将`nn.DotProduct`用于感知器以外的任何模型。

`nn.py` 中提供了以下方法：

- `nn.gradients` 根据提供的参数计算损失的梯度。
  - 用法：`nn.gradients(loss, [parameter_1, parameter_2, ..., parameter_n])` 将返回一个列表`[gradient_1, gradient_2, ..., gradient_n]`，其中每个元素都是一个`nn.Constant`包含损失相对于对应参数的梯度。
- `nn.as_scalar` 可以从损失节点(a loss node)中提取 Python 浮点数。 这对于确定何时停止训练很有用。
  - 用法：`nn.as_scalar(node)`，其中`node`要么是损失节点，要么具有形状`(1,1)`。

提供的数据集还有两种附加方法：

- `dataset.iterate_forever(batch_size)` 产生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性。 这对于确定何时停止训练很有用。

Example: Linear Regression

作为神经网络框架如何工作的示例，让我们将一条线拟合到一组数据点。 我们将从使用函数 $y = 7x_0 + 8x_1 + 3$ 构建的四个训练数据点开始。 在批处理形式中，我们的数据是：

X = [00 | 01 | 10 | 11]
Y = [3 | 11 | 10 | 18]

假设数据以`nn.Constant`节点的形式提供给我们：

```
>>> x
<Constant shape=4x2 at 0x10a30fe80>
>>> y
<Constant shape=4x1 at 0x10a30fef0>
```

让我们构建并训练一个形式为 $f(x) = x_0 \cdot m_0 + x_1 \cdot m_1 + b$ 的模型。 如果操作正确，我们应该能够学习到 $m_0 = 7$、$m_1 = 8$ 和 $b = 3$。

首先，我们创建可训练的参数。 在矩阵形式中，它们是：

$M = [m_0 m_1]$ 和 $B = [b]$

对应于以下代码：

```
m = nn.Parameter(2, 1)
b = nn.Parameter(1, 1)
```

print出来看的话：

```
>>> m
<Parameter shape=2x1 at 0x112b8b208>
>>> b
<Parameter shape=1x1 at 0x112b8beb8>
```

接下来，我们计算模型对 y 的预测:

```
xm = nn.Linear(x, m)
predicted_y = nn.AddBias(xm, b)
```

我们的目标是让预测的 y 值与提供的数据相匹配。 在线性回归中，我们通过最小化平方损失来做到这一点：$\mathcal{L} = \frac{1}{2N}\sum_{(x,y)}(y - f(x))^2$

我们构造一个损失节点:

```
loss = nn.SquareLoss(predicted_y, y)
```

本代码框架提供了一种方法，**该方法将返回相对于参数的损失梯度**:

```
grad_wrt_m, grad_wrt_b = nn.gradients(loss, [m, b])
```

把这些结点（nodes）打印出来:

```
>>> xm
<Linear shape=4x1 at 0x11a869588>
>>> predicted_y
<AddBias shape=4x1 at 0x11c23aa90>
>>> loss
<SquareLoss shape=() at 0x11c23a240>
>>> grad_wrt_m
<Constant shape=2x1 at 0x11a8cb160>
>>> grad_wrt_b
<Constant shape=1x1 at 0x11a8cb588>
```

然后我们可以使用 `update` 方法来更新我们的参数。 以下是 `m` 的更新，假设我们已经根据我们选择的合适的学习率初始化了一个 `multiplier` 变量:

```
m.update(grad_wrt_m, multiplier)
```

如果我们还包括对 `b` 的更新并添加一个循环来重复执行梯度更新，我们将拥有线性回归的完整训练过程。

Question 2: Non-linear Regression 报告应实现任务[1分]

- **说明：该小题截图只需截 python autograder.py -q q2的运行结果，获得评分器给的全部8分，这道题才算得它占的1分。**

对于这个问题，你将训练一个神经网络在 $[-2\pi, 2\pi]$上逼近 $\sin(x)$。

你需要在 `models.py` 中完成 `RegressionModel` 类的实现。对于这个问题，一个相对简单的架构就足够了。使用 `nn.SquareLoss` 作为你的损失。

你的任务是:

- 使用任何需要的初始化(提示：定义网络参数，比如w1=nn.Parameter(xxx,xxx))去实现 `RegressionModel.__init__`
- 实现 `RegressionModel.run` ，它返回表示模型预测结果的 *batch_size* × 1 节点（提示：用参数表示网络层，注意应用激活函数）。
- 实现 `RegressionModel.get_loss` ，它返回给定输入和目标输出的损失。
- 实现 `RegressionModel.train`，它应该使用基于梯度的更新来训练你的模型。

此任务只有一个数据集拆分（即，只有训练数据，没有验证数据或测试集）。如果在数据集中的所有样例中平均损失为 0.02 或更低，你的实现将获得满分。你可以使用训练损失(training loss)来确定何时停止训练（使用 `nn.as_scalar` 将损失节点转换为 Python 数字）。请注意，模型应该需要几分钟的时间来训练。

```
python autograder.py -q q2
```

本节课实现内容保存说明 - 为提交第三次上机报告做准备

- **第8周上机课结束后，一并提交本部分”机器学习“的报告，该报告包含本节课的Q1,Q2的代码和运行截图，请按Q1,Q2大标题下的说明进行截图，保存。**
- **本节课实现的代码文件也请保存，尤其是实现了的 models.py 文件，你将需要把其中实现的代码加到第8周的models.py中跟第8周的任务实现一并进行提交。**