MODEL

一、内容

之前的实现是通过编写大量代码并对一些相当大的代码块进行修改来构建模型。Model类将前向传播、反向传播的训练和验证过程封装，将模型本身变成一个对象，特别是希望做一些像保存和加载这个对象以用于未来预测任务。这将使用这个对象来减少一些更常见的代码行，使得更容易使用当前的代码库并构建新模型。

二、代码

一、 创建Layer_Input

因为希望在层的循环中执行这个操作，并需要知道每一层的前一层和后一层的内容，为保持代码的完整性，这里在第一个Dense层前面加入Layer_Input层，作为输入层，但没有与之相关的权重和偏差。输入层只包含训练数据，在循环中迭代层时，只将其用作第一层的“前一层”。

```
class Layer_Input:
      def __init__(self):
            pass

      def forward(self, input):
            self.output = input
```

二、修改Loss

```
class Loss:
      def __init__(self):
            pass

      # 在求loss时需要知参别些层里面有可以训练参数，可以正则化
```

```
    def save_trainable_layer(self, trainable_layer):
        self.trainable_layer = trainable_layer

    # 统一通过调用calculate方法计算损失
    def calculate(self, y_pred, y_ture, add_regular_loss=False):
        # 对于不同的损失函数，通过继承Loss父类，并实现不同的forward方法。
        data_loss = np.mean(self.forward(y_pred, y_ture))

        # 在加入正则代码后，可以求得正则损失
        # 注意之前版本调用regularization_loss(layer)
        # 但这个版本有了self.trainable_layer，可直接找到Dense层（有参数）
        regularization_loss = self.regularization_loss()
        if not add_regular_loss:
            # 在测试模型性能时只关心data_loss
            regularization_loss = 0
        # 注意，这里计算得到的loss不作为类属性储存，而是直接通过return返回
        return data_loss, regularization_loss

    def regularization_loss(self):
        # 默认为0
        regularization_loss = 0
        for layer in self.trainable_layer:
            # 如果存在L1的loss
            if layer.weight_L1 > 0:
                regularization_loss += layer.weight_L1 *
np.sum(np.abs(layer.weight))
            if layer.bias_L1 > 0:
                regularization_loss += layer.bias_L1 * np.sum(np.abs(layer.bias))
            # 如果存在L2的loss
            if layer.weight_L2 > 0:
                regularization_loss += layer.weight_L2 * np.sum(layer.weight ** 2)
            if layer.bias_L2 > 0:
                regularization_loss += layer.bias_L2 * np.sum(layer.bias ** 2)

        return regularization_loss
```

这里加入了save_trainable_layer和修改了regularization_loss方法、calculate方法。调用save_trainable_layer方法，创建self.trainable_layer属性，存放Dense层（有参数可以学习）。calculate方法在测试时通过add_regular_loss=False参数，输出测试集上的regularization_loss = 0，只关心data_loss。

三、修改Activation

```
class Activation_Softmax:
	def prediction(self, output):
		return np.argmax(output, axis=1, keepdims=True)

class Activation_Sigmoid:
	def prediction(self, output):
		# output > 0.5反回的是二进制值
		# 乘1变成数值
		return ( output > 0.5 ) * 1

class Activation_Linear:
	def prediction(self, output):
		return output
```

> 这里不修改ReLu，因为它不作为输出层。prediction方法输出预测类别（分类）或输出值（回归）。
>
> **注意：softmax中要保持形状，因为prediction以矩阵形式用来计算准确率**

四、修改Dropout

```
	def forward(self, input, drop_on=True):
		self.input = input
		# 按概率生成一个0、1矩阵
		# 因为1的概率只有rate这么大，就要除以rate偿损失值
		if not drop_on:
			# 如果关上dropout就输出等于输入
			self.output = self.input
			return

		self.mask = np.random.binomial(1, self.rate, size=self.input.shape) /
self.rate
		self.output = self.input * self.mask
```

> 在forward方法中加了on_off=True参数，当测试时前向传播不用dropout

五、创建Accuracy

父类

这个类根据预测结果计算准确率。

```
class Accuracy:
	# 计算准确率
	def calculate(self, prediction, y_true):
		# 获得比较结果
		comparision = self.compare(prediction, y_true)

		# 计算准确率
		accuracy = np.mean(comparision)

		return accuracy
```

子类1

```
class Accuracy_Regresion(Accuracy):
	def __init__(self):
		# 创建一个属性，保存精度
		# 因为对于Regresion，要自己先创建一个精度标准
		self.precision = None

	def compare(self, precision, y_true):
		if self.precision is None:
			self.precision = np.std(y_true) / 250
		return np.abs(precision - y_true) < self.precision
```

> **这里使用**np.std(y_true) / 250**作为精度标准。**来计算回归问题的准确率。

子类2

```
class Accuracy_Classification(Accuracy):
	def __init__(self):
		pass

	def compare(self, precision, y_true):
		# onehot编码
		if len(y_true.shape) == 2:
			# 改成单个类别
			y_true = np.argmax(y_true,axis=1) #此时是行向量，可能用keepdims=保持矩阵
		# 注意：prediction是一个矩阵，y_true是一个向量1xa
		# 当矩阵是ax1时，会错误产生广播
		# 非常重要，我以为是模型代码错了一天的bug，
		# 最后发现可能只是正确率证算错误了
```

```
            y_true = y_true.reshape(-1, 1)
            compare = (precision == y_true) * 1
            return compare
```

> 计算分类问题的准确率。**注意：不同形状的y_true计算会不同，现只接收onehot编码和行向量**

六、 创建Model

Model类，Model类将前向传播、反向传播的训练和验证过程封装，使整个过程实现方便。

实现

```
class Model():
      def __init__(self):
            # 这个属性用来存模型的每层结构
            self.layer = []
            # 先初始化为None，后面会在finalize中判断是否符合
softmax+categoricalCrossentropy或sigmiod+binaryCrossentropy
            self.softmax_categoricalCrossentropy = None
            self.sigmoid_binaryCrossentropy = None

      # 用来加入层结构
      def add(self, layer):
            self.layer.append(layer)

      # 用来设置损失loss的类型、优化器等
      # 在星号之后的所有参数都必须作为关键字参数传递，而不能作为位置参数传递
      def set(self, *, loss, optimizer, accuracy):
            self.loss = loss
            self.optimizer = optimizer
            self.accuracy = accuracy

      # 训练模型
      # epochs训练轮数
      # print_every每多少轮输出一次
      def train(self, X, y, *, epochs=1, print_every=1, vaildation_data=None):
            # 注意：vaildation_data需要输入一个元组，包括X、y
            for epoch in range(1, epochs+1):
                  # 前向传播
                  output = self.forward(X)
                  # 计算损失
                  data_loss, regularization_loss = self.loss.calculate(output, y,
add_regular_loss=True)
                  # 总loss
                  loss = data_loss + regularization_loss
                  # 计算预测值或预测类别
```

```
                prediction = self.output_layer.prediction(output)
                # 计算准确率
                accuracy = self.accuracy.calculate(prediction, y)

                # 反向传播
                self.backward(output, y)

                # 优化器进行优化
                self.optimizer.pre_update_param()
                for layer in self.trainable_layer:
                        self.optimizer.update_param(layer)
                self.optimizer.post_update_param()

                # 输出信息
                if not epoch % print_every:
                        print(f'epoch: {epoch}, ' +
                              f'acc: {accuracy:.3f}, ' +
                              f'loss: {loss:.3f} (' +
                              f'data_loss: {data_loss:.3f}, ' +
                              f'reg_loss: {regularization_loss:.3f}), ' +
                              f'lr: {self.optimizer.current_learning_rate}')

        if vaildation_data:
                X_val, y_val = vaildation_data
                # 输出层的输出
                output = self.forward(X_val, False)
                # 计算loss
                data_loss, regularization_loss = self.loss.calculate(output, y_val)
                loss = data_loss + regularization_loss
                # 预测类别或预测值
                prediction = self.output_layer.prediction(output)
                # 计算准确率
                accuracy = self.accuracy.calculate(prediction, y_val)
                # 测试输出
                print(f'validation, ' +
                        f'acc: {accuracy:.3f}, ' +
                        f'loss: {loss:.3f}')
                # plt.plot(X_val, y_val)
                # plt.plot(X_val, output)
                # plt.show()

## 在该方法内实现模型的定型
# 1.确定不同层之间的前后次序
# 2.确定Dense层
# 3.将Dense层传入loss对象中，以计算正则损失
# 4.判断是否符合softmax+categoricalCrossentropy或sigmiod+binaryCrossentropy
def finalize(self):
        # 创建输入层
        self.input_layer = Layer_Input()
        # 模型层数，不包括输入层、loss层
```

```
            layer_num = len(self.layer)
            # 存放Dense层（有参数可以学习）
            self.trainable_layer = []

            # 循环设置层间关系
            for i in range(layer_num):
                  if i == 0:
                        # 第一层Dense,它的前一层是input_layer
                        self.layer[i].pre = self.input_layer
                        self.layer[i].next = self.layer[i + 1]
                  elif i == layer_num-1:
                        # 最后一个Dense，它是后一层是loss
                        self.layer[i].pre = self.layer[i - 1]
                        self.layer[i].next = self.loss
                        # 在最后一层标记一下所用的输出层是什么Activation存在Model的属性中
                        self.output_layer = self.layer[i]
                  else:
                        self.layer[i].pre = self.layer[i-1]
                        self.layer[i].next = self.layer[i+1]

                  if hasattr(self.layer[i], 'weight'):
                        # 如果当前层有'weight'属性，说是当前层是Dense层
                        # 该层是可以训练的
                        self.trainable_layer.append(self.layer[i])

            # 把Dense层告诉loss对象
            self.loss.save_trainable_layer(self.trainable_layer)
            # 判断是否符合softmax+categoricalCrossentropy或sigmiod+binaryCrossentropy
            if isinstance(self.layer[-1], Activation_Softmax) and \
                        isinstance(self.loss, Loss_CategoricalCrossentropy):
                  self.softmax_categoricalCrossentropy =
Activation_Softmax_Loss_CategoricalCrossentropy()

            if isinstance(self.layer[-1], Activation_Sigmoid) and \
                        isinstance(self.loss, Loss_BinaryCrossentropy):
                  self.sigmoid_binaryCrossentropy =
Activation_Sigmoid_Loss_BinaryCrossentropy()

      # 前向传播
      # 该方法将在train方法中调用（训练过程将调用很多种方法，forward中是其中一个）
      def forward(self, input, dropout=True):
            self.input_layer.forward(input)
            for layer in self.layer:
                  if isinstance(layer,Dropout) and (not dropout):
                        layer.forward(layer.pre.output,dropout)
                  else:
                        layer.forward(layer.pre.output)

            # 这里的layer是最后一层的activation
            return layer.output
```

```
    def backward(self, output, y_true):
        if self.softmax_categoricalCrossentropy:
            self.softmax_categoricalCrossentropy.backward(output, y_true)
            # 最后一层是softmax,不调用backward求dinput,
            # 因为softmax_categoricalCrossentropy已经算好
            self.layer[-1].dinput = self.softmax_categoricalCrossentropy.dinput
            # 注意：这里循环不包含最后一层（softmax）
            for layer in reversed(self.layer[:-1]):
                layer.backward(layer.next.dinput)
            return
        if self.sigmoid_binaryCrossentropy:
            self.sigmoid_binaryCrossentropy.backward(output, y_true)
            # 最后一层是sigmoid,不调用backward求dinput,
            # 因为softmax_categoricalCrossentropy已经算好
            self.layer[-1].dinput = self.sigmoid_binaryCrossentropy.dinput
            # 注意：这里循环不包含最后一层（softmax）
            for layer in reversed(self.layer[:-1]):
                layer.backward(layer.next.dinput)
            return

        self.loss.backward(output, y_true)
        # 注意：这里用的不是self.trainable_layer
        for layer in reversed(self.layer):
            layer.backward(layer.next.dinput)
```

> 以下是该类中方法的作用简述：

add()，用来加入层结构。

set()，用来设置损失loss的类型、优化器、准确率等。

forward()，前向传播，在finalize方法调用。

backward()，反向传播，在finalize方法调用。

finalize()，在该方法内实现模型的定型

train()，训练模型，在该方法中调其他方法实现训练，其中有vaildation_data参数，用于测试。

实例1

```
# 生成数据共1000个点
X, y = sine_data()
X_test = X[::2]
y_test = y[::2]
X = X[1::2]
y = y[1::2]
```

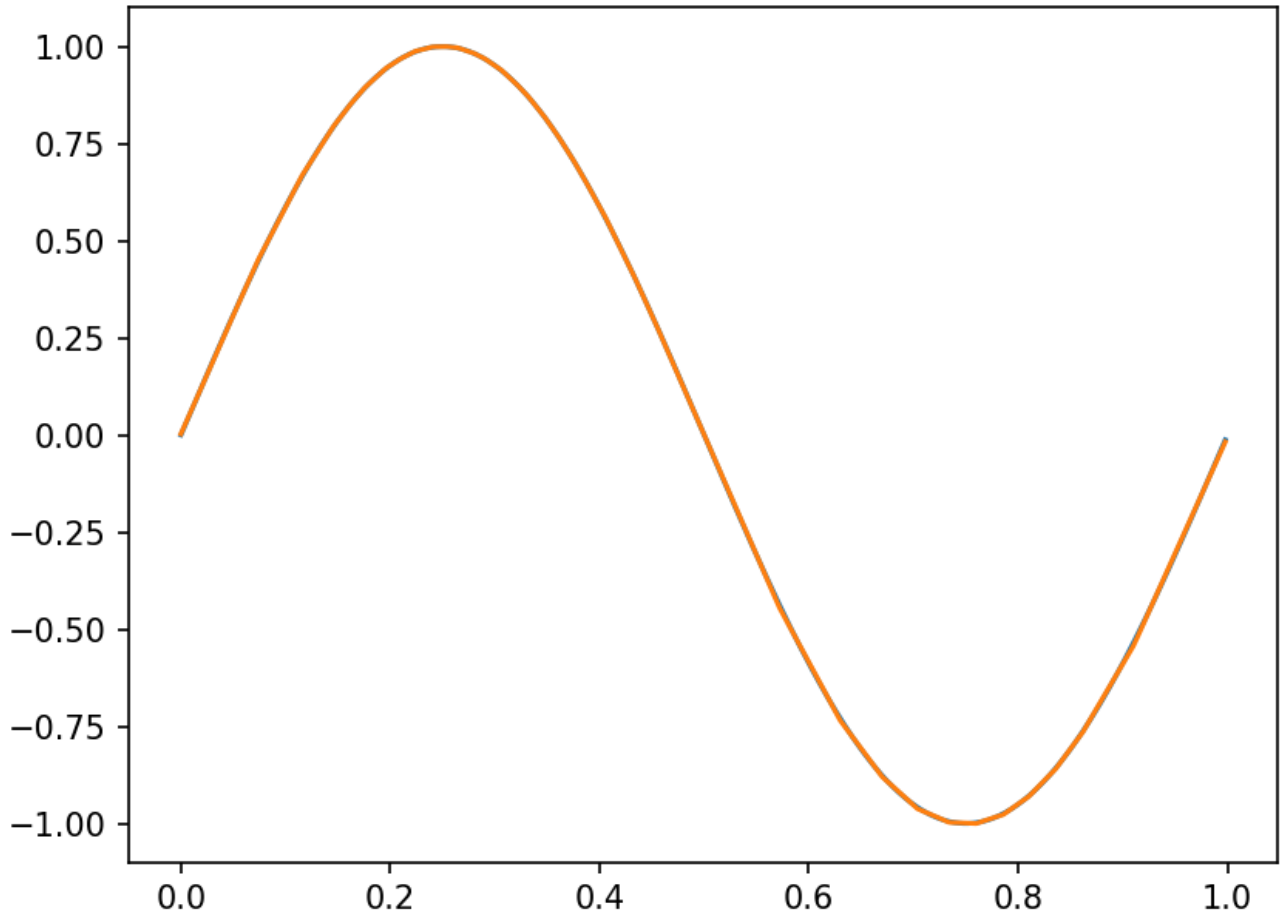
```
model = Model()
model.add(Layer_Dense(1,64))
model.add((Activation_ReLu()))
model.add(Layer_Dense(64,64))
model.add(Activation_ReLu())
model.add(Layer_Dense(64,1))
model.add(Activation_Linear())

# 计得加()号，loss=Loss_MeanSquaredError是不行的，
# 这样只调用了对象的属性
model.set(loss=Loss_MeanSquaredError(),
                optimizer=Optimizer_Adam(learning_rate=0.005, decay=1e-3),
                accuracy=Accuracy_Regresion())

model.finalize()

model.train(X, y, epochs=10000, print_every=100)
```

```
epoch: 9500, acc: 0.896, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000), lr: 0.0004762358319839844
epoch: 9600, acc: 0.892, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000), lr: 0.0004717426172280403 6
epoch: 9700, acc: 0.878, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000), lr: 0.0004673333956444527 5
epoch: 9800, acc: 0.894, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000), lr: 0.0004630058338735068 7
epoch: 9900, acc: 0.890, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000), lr: 0.0004587576841912101 6
epoch: 10000, acc: 0.892, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000), lr: 0.0004545867806164196 4
validation, acc: 0.888, loss: 0.000
```



对回归问题，效果很好。

实例2

```
nnfs.init()#默认随机种子为0，每次运行同样的数据
X, y = spiral_data(samples=1000, classes=2)
X_test, y_test = spiral_data(samples=100, classes=2)

model = Model()
model.add(Layer_Dense(2,64,weight_L2=5e-4,bias_L2=5e-4))#,weight_L2=5e-4,bias_L2=5e-4
model.add(Activation_ReLu())

model.add(Layer_Dense(64,1))
model.add(Activation_Sigmoid())
model.set(loss=Loss_BinaryCrossentropy(),
                optimizer=Optimizer_Adam(decay=5e-7),
                accuracy=Accuracy_Classification())

model.finalize()

model.train(X,y,vaildation_data=(X_test,y_test),epochs=10000,print_every=100)
```

```
epoch: 9500, acc: 0.974, loss: 0.205 (data_loss: 0.129, reg_loss: 0.076), lr: 0.000995272951118662
epoch: 9600, acc: 0.974, loss: 0.204 (data_loss: 0.128, reg_loss: 0.076), lr: 0.0009952234251708924
epoch: 9700, acc: 0.974, loss: 0.202 (data_loss: 0.127, reg_loss: 0.075), lr: 0.000995173904151816
epoch: 9800, acc: 0.974, loss: 0.201 (data_loss: 0.126, reg_loss: 0.075), lr: 0.0009951243880606966
epoch: 9900, acc: 0.974, loss: 0.200 (data_loss: 0.125, reg_loss: 0.075), lr: 0.0009950748768967994
epoch: 10000, acc: 0.974, loss: 0.198 (data_loss: 0.124, reg_loss: 0.074), lr: 0.0009950253706593885
validation, acc: 0.975, loss: 0.140
```

实例3

```
nnfs.init()#默认随机种子为0，每次运行同样的数据
X, y = spiral_data(samples=1000, classes=3)
X_test, y_test = spiral_data(samples=100, classes=3)


model = Model()
model.add(Layer_Dense(2,512,weight_L2=5e-4,bias_L2=5e-4))#,weight_L2=5e-4,bias_L2=5e-4
model.add(Activation_ReLu())


model.add(Dropout(0.1))
model.add(Layer_Dense(512,3))
model.add(Activation_Softmax())
model.set(loss=Loss_CategoricalCrossentropy(),
                optimizer=Optimizer_Adam(learning_rate=0.05, decay=5e-5),
                accuracy=Accuracy_Classification())
```

```
model.finalize()

model.train(X,y,vaildation_data=(X_test,y_test),epochs=10000,print_every=100)
```

```
epoch: 9600, acc: 0.870, loss: 0.414 (data_loss: 0.356, reg_loss: 0.058), lr: 0.033784925166390756
epoch: 9700, acc: 0.865, loss: 0.429 (data_loss: 0.369, reg_loss: 0.060), lr: 0.03367116737937304
epoch: 9800, acc: 0.866, loss: 0.446 (data_loss: 0.389, reg_loss: 0.058), lr: 0.033558173093056816
epoch: 9900, acc: 0.860, loss: 0.437 (data_loss: 0.381, reg_loss: 0.056), lr: 0.0334459346466437
epoch: 10000, acc: 0.865, loss: 0.420 (data_loss: 0.366, reg_loss: 0.054), lr: 0.03333444448148271
这里没有dropout
validation, acc: 0.857, loss: 0.391
```

```
epoch: 9900, acc: 0.861, loss: 0.436 (data_loss: 0.389, reg_loss: 0.046),
lr: 0.0334459346466437
epoch: 10000, acc: 0.880, loss: 0.394 (data_loss: 0.347, reg_loss: 0.047),
lr: 0.03333444448148271
validation, acc: 0.867, loss: 0.379
```

对于3分类的问题，相同的参数设置训练结果与书上结果相似，但并没有体现出dropout层的优势。

实现4

```
X, y = spiral_data(samples=1000, classes=3)
X_test, y_test = spiral_data(samples=100, classes=3)
# print(X[:5])
# print(X_test[:5])

model = Model()
model.add(Layer_Dense(2,64,weight_L2=5e-4,bias_L2=5e-4))#,weight_L2=5e-4,bias_L2=5e-4
model.add(Activation_ReLu())

model.add(Layer_Dense(64,3))
model.add(Activation_Softmax())
model.set(loss=Loss_CategoricalCrossentropy(),
          optimizer=Optimizer_Adam(decay=5e-7),
          accuracy=Accuracy_Classification())
```

```
epoch: 9500, acc: 0.901, loss: 0.417 (data_loss: 0.319, reg_loss: 0.098), lr: 0.000995272951118662
epoch: 9600, acc: 0.903, loss: 0.415 (data_loss: 0.317, reg_loss: 0.097), lr: 0.0009952234251708924
epoch: 9700, acc: 0.903, loss: 0.413 (data_loss: 0.315, reg_loss: 0.097), lr: 0.000995173904151816
epoch: 9800, acc: 0.903, loss: 0.411 (data_loss: 0.314, reg_loss: 0.097), lr: 0.0009951243880606966
epoch: 9900, acc: 0.903, loss: 0.409 (data_loss: 0.313, reg_loss: 0.096), lr: 0.0009950748768967994
epoch: 10000, acc: 0.903, loss: 0.407 (data_loss: 0.311, reg_loss: 0.096), lr: 0.0009950253706593885
validation, acc: 0.880, loss: 0.354
```

同样是3分类问题，用更少的神经元并且不加dropout效果更好。所以对于解决一个复杂问题，是选一个简单的结构不加dropout，还选一个复杂的结构加上dropout，要根据实际性况而定。