# 鸢尾花分类任务对比实验

## 实验介绍

### 简介

优化器是神经网络训练过程中非常重要的结构，正是因为优化器的存在帮助初始化参数的网络一步步学习到了符合训练集数据特征的最优参数。本实验主要介绍如何使用MindSpore进行优化实验。共包含两个实验，分别为：

* 实验一：利用不同优化器求解函数极值点实验
* 实验二：鸢尾花数据在不同优化器下的分类实验

通过分析无优化器、SGD优化器、Momentum优化器、Adam优化器模型在求极值点和分类任务下的结果 ，得到不同优化器的作用。

参考mindspore/course仓库：

<https://gitee.com/mindspore/course/tree/master/optimizer>

### 实验目的

* 掌握MindSpore中优化器的使用及其作用。
* 了解如何使用MindSpore进行SGD、Momentum、Adam优化器实验。
* 了解不同优化器原理。

## 实验环境要求

ModelArts平台：MindSpore-1.3，也可在本地搭建python3.7.5和MindSpore1.3环境完成实验（本地环境训练会很慢，还是用ModelArts平台好一些）。

## 实验总体设计

本实验主要分为两个部分，第一部分重点在于理解各个优化器的原理，自己编写优化器，体会优化器的作用，主要设计思路如下：

构建Beale 公式及其导数：Beale 公式是一个经典的二元函数，它在三维空间中有一个复杂的曲面，方程最低点在(3,0.5)处。该部分首先要定义这个函数并定义其偏导数。这部分重点掌握如何将数学公式在python中实例化。

编写动量优化器并优化Beale 公式：通过编写动量优化器函数来优化上一步定义好的beale函数。该步骤重点掌握优化器函数是如何实现的和动量优化器的原理。

编写Adagrad优化器并优化Beale 公式：这步与上一步类似，重点掌握Adagrad优化器的原理。

第二部分主要通过构建全连接网络体会有无优化器的全连接网络的差别，从而深入理解优化器在神经网络模型构建中的作用，主要设计流程如下：

导入模块和数据集、数据预处理和构建神经网络这三步与第二章、第四章的设计相同。

无优化器模型预测：主要是初始化网络后，不经过训练，直接进行预测任务。

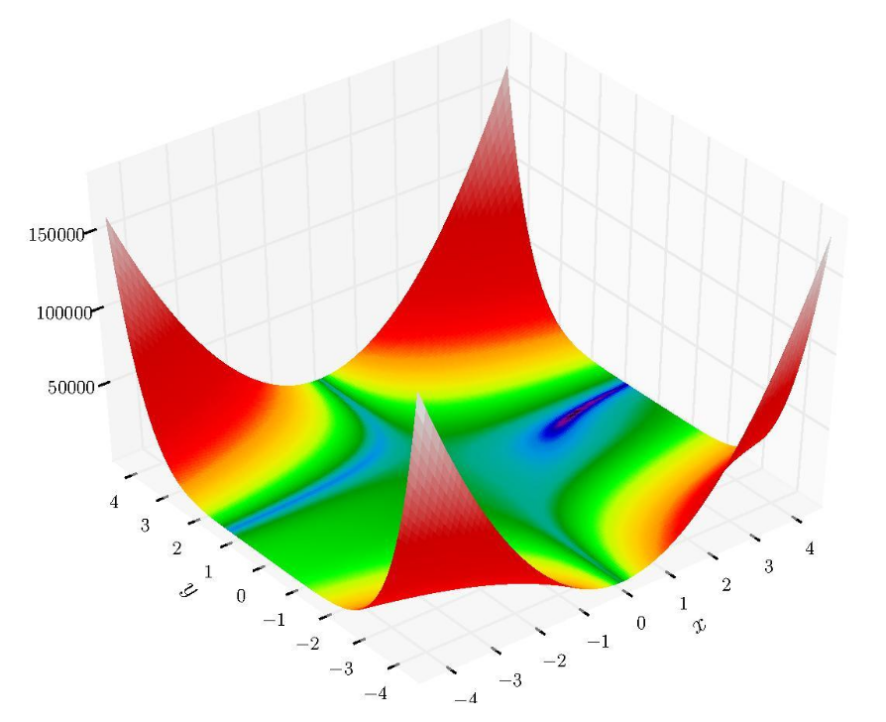
有优化器模型训练并预测：与通常训练完网络并进行预测的步骤相同。重点是结合上一步，体会优化器对于神经网络的作用。

## 背景知识

Beale 公式：



表面如下图：



这个方程和这个曲面就是我们要优化的对象，优化的过程，就是我们随机给定曲面上的一个点，经过多次迭代。最终，找到曲面上最低的那个点。

## 实验过程

### 创建实验环境

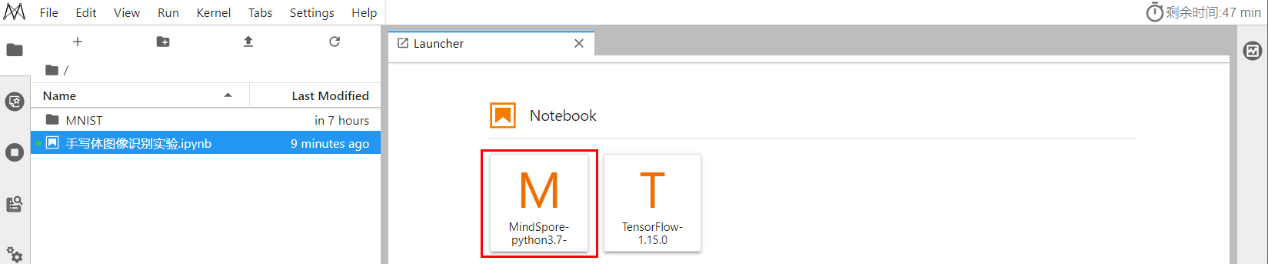
在开始本实验前，需要完成实验环境搭建工作。

进入ModelArts开发环境

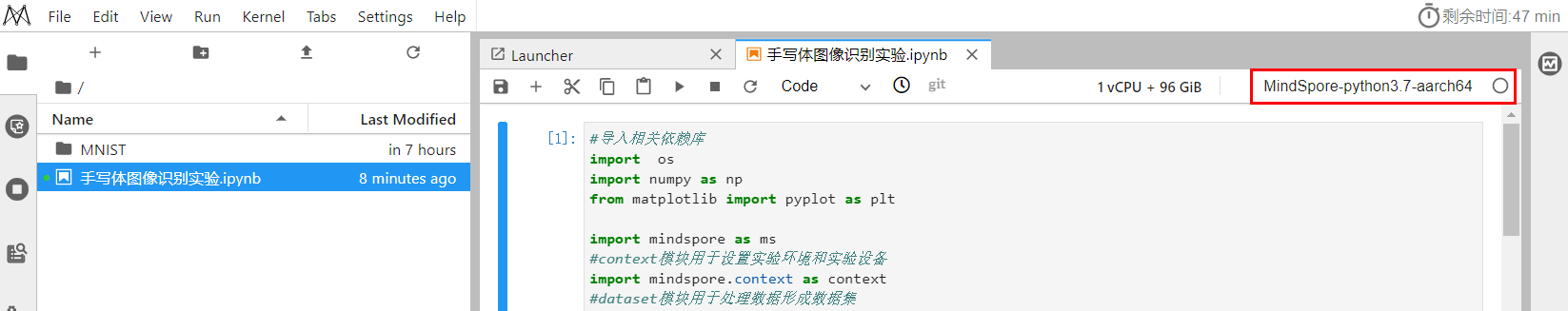
参考文末附录，创建ModelArts上的开发环境Notebook并进入。

打开Notebook

打开Notebook控制台后，新建或打开ipynb文件，选择MindSpore环境作为Kernel，即可开始编辑实验代码。



创建ipynb文件



打开ipynb文件

### 构建Beale 公式及其导数

导入相关模块

import numpy as np

import matplotlib.pyplot as plt

import matplotlib.colors as plt\_cl

构建相关函数

# 定义beale公式

def beale(x1,x2):

return (1.5-x1+x1\*x2)\*\*2+(2.25-x1+x1\*x2\*\*2)\*\*2+(2.625-x1+x1\*x2\*\*3)\*\*2

# 定义beale公式的偏导函数

def dbeale\_dx(x1, x2):

dfdx1 = 2\*(1.5-x1+x1\*x2)\*(x2-1)+2\*(2.25-x1+x1\*x2\*\*2)\*(x2\*\*2-1)+2\*(2.625-x1+x1\*x2\*\*3)\*(x2\*\*3-1)

dfdx2 = 2\*(1.5-x1+x1\*x2)\*x1+2\*(2.25-x1+x1\*x2\*\*2)\*(2\*x1\*x2)+2\*(2.625-x1+x1\*x2\*\*3)\*(3\*x1\*x2\*\*2)

return dfdx1, dfdx2

可视化beale方程

# 定义画图函数

def gd\_plot(x\_traj):

plt.rcParams['figure.figsize'] = [6, 6]

plt.contour(X1, X2, Y, levels=np.logspace(0, 6, 30),

norm=plt\_cl.LogNorm(), cmap=plt.cm.jet)

plt.title('2D Contour Plot of Beale function(Momentum)')

plt.xlabel('$x\_1$')

plt.ylabel('$x\_2$')

plt.axis('equal')

plt.plot(3, 0.5, 'k\*', markersize=10)

if x\_traj is not None:

x\_traj = np.array(x\_traj)

plt.plot(x\_traj[:, 0], x\_traj[:, 1], 'k-')

plt.show()

step\_x1, step\_x2 = 0.2, 0.2

X1, X2 = np.meshgrid(np.arange(-5, 5 + step\_x1, step\_x1),

np.arange(-5, 5 + step\_x2, step\_x2))

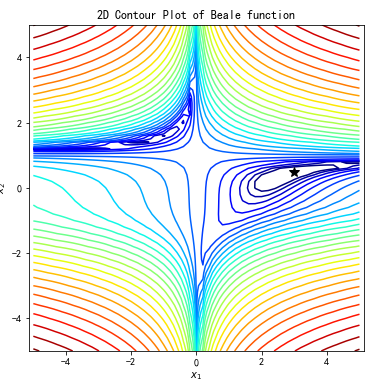
Y = beale(X1, X2)

print("目标结果 (x\_1, x\_2) = (3, 0.5)")

gd\_plot(None)

输出结果：

目标结果 (x\_1, x\_2) = (3, 0.5)



### 编写不使用优化器优化Beale 公式

无优化器训练不更新参数，不管训练多少次，模型的参数没有发生变化。模型效果与模型初始化参数关系较大。

构建无优化器函数

# 定义无优化器函数

def gd\_no(df\_dx, x0, conf\_para=None):

if conf\_para is None:

conf\_para = {}

conf\_para.setdefault('n\_iter', 1000) # 迭代次数

conf\_para.setdefault('learning\_rate', 0.001) # 设置学习率

x\_traj = []

x\_traj.append(x0)

v = np.zeros\_like(x0)

# 没有迭代更新的操作，所以，坐标没有变化

for iter in range(1, conf\_para['n\_iter'] + 1):

x\_traj.append(x\_traj[-1])

return x\_traj

优化beale方程

# 初始化坐标

x0 = np.array([1.0, 1.5])

conf\_para\_no = {'n\_iter': 2000, 'learning\_rate': 0.005}

# 调用函数进行更新

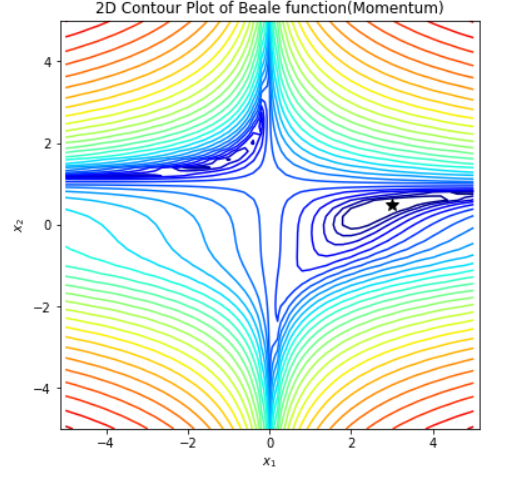
x\_traj\_no = gd\_no(dbeale\_dx, x0, conf\_para\_no)

print("无优化器求得极值点 (x\_1, x\_2) = (%s, %s)" % (x\_traj\_no[-1][0], x\_traj\_no[-1][1]))

gd\_plot(x\_traj\_no)

输出结果：

无优化器求得极值点 (x\_1, x\_2) = (1.0, 1.5)



### 编写SGD优化器并优化Beale 公式

梯度下降法：梯度下降（gradient descent）在机器学习中应用十分的广泛，是求解无约束优化问题最简单和最古老的方法之一。通过迭代，参数向梯度的反方向更新，直到收敛。



其中表示损失函数 J 关于参数W的梯度表示学习率；

缺点：

有可能会陷入局部最小值；

不会收敛，最终会一直在最小值附近波动，并不会达到最小值并停留在此；

下降速度慢；

选择合适的学习率比较困难；

在所有方向上统一的缩放梯度，不适用于稀疏数据；

构建SGD优化器

def gd\_sgd(df\_dx, x0, conf\_para=None):

if conf\_para is None:

conf\_para = {}

conf\_para.setdefault('n\_iter', 1000) # 迭代次数

conf\_para.setdefault('learning\_rate', 0.001) # 设置学习率

x\_traj = []

x\_traj.append(x0)

v = np.zeros\_like(x0)

# 利用梯度值对坐标进行更新

for iter in range(1, conf\_para['n\_iter'] + 1):

dfdx = np.array(df\_dx(x\_traj[-1][0], x\_traj[-1][1]))

v = - conf\_para['learning\_rate'] \* dfdx

x\_traj.append(x\_traj[-1] + v)

return x\_traj

优化beale方程

x0 = np.array([1.0, 1.5])

conf\_para\_sgd = {'n\_iter': 2000, 'learning\_rate': 0.005}

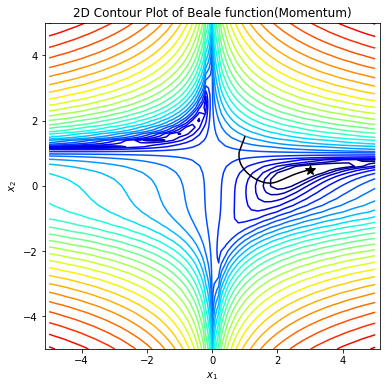
x\_traj\_sgd = gd\_sgd(dbeale\_dx, x0, conf\_para\_sgd)

print("SGD求得极值点 (x\_1, x\_2) = (%s, %s)" % (x\_traj\_sgd[-1][0], x\_traj\_sgd[-1][1]))

gd\_plot(x\_traj\_sgd)

输出结果：

SGD求得极值点 (x\_1, x\_2) = (2.981957257903412, 0.4954609564102124)



### 编写动量优化器并优化Beale 公式

Momentum：是动量优化法中的一种（Momentum、NAG），即使用动量(Momentum)的随机梯度下降法(SGD)，主要思想是引入一个积攒历史梯度信息的动量来加速SGD。其参数优化公式如下所示：



其中表示损失函数 J 关于参数W的梯度表示学习率；表示动量的大小，一般取值为0.9。

优点：

参考了历史梯度，增加了稳定性；

由于引入加速动量，加快收敛速度。

还有一定摆脱局部最优的能力。

构建Momentum优化器

def gd\_momentum(df\_dx, x0, conf\_para=None):

if conf\_para is None:

conf\_para = {}

conf\_para.setdefault('n\_iter', 1000) # 迭代次数

conf\_para.setdefault('learning\_rate', 0.001) # 设置学习率

conf\_para.setdefault('momentum', 0.9) # 设置动量参数

x\_traj = []

x\_traj.append(x0)

v = np.zeros\_like(x0)

# 套用动量优化器公式，对坐标值进行更新

for iter in range(1, conf\_para['n\_iter'] + 1):

dfdx = np.array(df\_dx(x\_traj[-1][0], x\_traj[-1][1]))

v = conf\_para['momentum'] \* v - conf\_para['learning\_rate'] \* dfdx

x\_traj.append(x\_traj[-1] + v)

return x\_traj

优化beale方程

x0 = np.array([1.0, 1.5])

conf\_para\_momentum = {'n\_iter': 500, 'learning\_rate': 0.005}

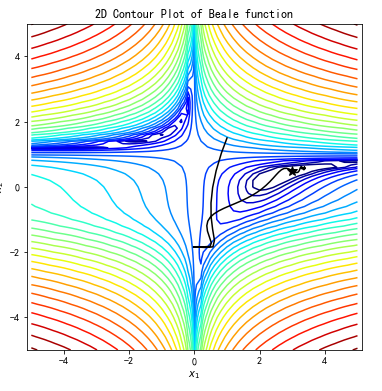
x\_traj\_momentum = gd\_momentum(dbeale\_dx, x0, conf\_para\_momentum)

print("Momentum求得极值点 (x\_1, x\_2) = (%s, %s)" % (x\_traj\_momentum[-1][0], x\_traj\_momentum[-1][1]))

gd\_plot(x\_traj\_momentum)

输出结果：

Momentum求得极值点 (x\_1, x\_2) = (3.0004236971560485, 0.5001057348328146)



### 编写自适应优化器并优化Beale 公式

自适应学习率优化算法主要有：AdaGrad算法，RMSProp算法，Adam算法以及AdaDelta算法。

AdaGrad

AdaGrad的基本思想是对每个变量用不同的学习率。这个学习率在一开始比较大，用于快速梯度下降。随着优化过程的进行，对于已经下降很多的变量，则减缓学习率，对于还没怎么下降的变量，则保持一个较大的学习率。其参数优化公式如下所示：



其中表示损失函数 J 关于参数W的梯度；表示学习率，一般取值0.01；是一个很小的数，防止分母为0；表示了前t 步参数梯度的平方累加。把沿路的Gradient的平方根，作为Regularizer。分母作为Regularizer项的工作机制如下：

训练前期，梯度较小，使得Regularizer项很大，放大梯度。[激励阶段]

训练后期，梯度较大，使得Regularizer项很小，缩小梯度。[惩罚阶段]

优点：

在数据分布稀疏的场景，能更好利用稀疏梯度的信息，比标准的SGD算法更有效地收敛；

对每个变量用不同的学习率，对输入参数学习率的依赖小，容易调节参数；

缺点：

主要缺陷来自分母项的对梯度平方不断累积，随之时间步地增加，分母项越来越大，最终导致学习率收缩到太小无法进行有效更新；

编写adagrad优化器

def gd\_adagrad(df\_dx, x0, conf\_para=None):

if conf\_para is None:

conf\_para = {}

conf\_para.setdefault('n\_iter', 1000) # 迭代次数

conf\_para.setdefault('learning\_rate', 0.001) # 学习率

conf\_para.setdefault('epsilon', 1e-7)

x\_traj = []

x\_traj.append(x0)

r = np.zeros\_like(x0)

# 套用adagrad优化器公式，对参数进行更新

for iter in range(1, conf\_para['n\_iter'] + 1):

dfdx = np.array(df\_dx(x\_traj[-1][0], x\_traj[-1][1]))

r += dfdx \*\* 2

x\_traj.append(x\_traj[-1] - conf\_para['learning\_rate'] / (np.sqrt(r) + conf\_para['epsilon']) \* dfdx)

return x\_traj

优化beale方程

x0 = np.array([1.0, 1.5])

conf\_para\_adag = {'n\_iter': 500, 'learning\_rate': 2}

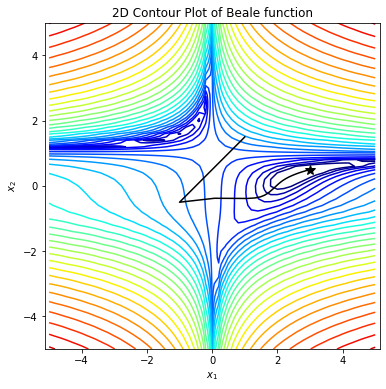
x\_traj\_adag = gd\_adagrad(dbeale\_dx, x0, conf\_para\_adag)

print("Adagrad求得极值点 (x\_1, x\_2) = (%s, %s)" % (x\_traj\_adag[-1][0], x\_traj\_adag[-1][1]))

gd\_plot(x\_traj\_adag)

输出结果：

Adagrad求得极值点 (x\_1, x\_2) = (2.9993173156940776, 0.49982846432011524)



### 思考题

* 思考题1：分析实验结果，对比各分类器不同。
* 思考题2：代码np.logspace的作用是什么？

### 导入模块、读取鸢尾花数据集并预处理、构建神经网络

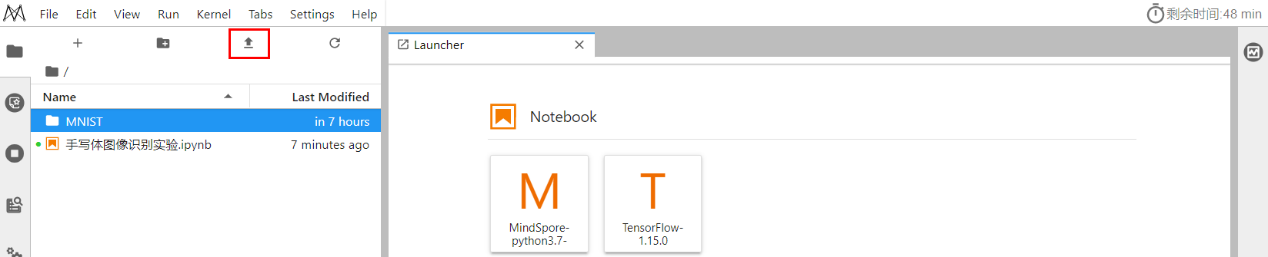
该案例主要内容是进行鸢尾花数据集标签的识别，鸢尾花数据集共收集了三类鸢尾花，即Setosa鸢尾花、Versicolour鸢尾花和Virginica鸢尾花，每一类鸢尾花收集了50条样本记录，共计150条。

注：

数据集的上传

以上传MNIST手写体字符的数据集为例：在MNIST文件夹下建立train和test两个文件夹，train中存放train-labels-idx1-ubyte和train-images-idx3-ubyte文件，test中存放t10k-labels-idx1-ubyte和t10k-images-idx3-ubyte文件。

将MNIST数据集文件上传至环境项目内，如下图所示（这只是一个示例，此实验数据集并不是这个）：



数据上传

数据集包括4个属性，分别为花萼的长、花萼的宽、花瓣的长和花瓣的宽。对花瓣我们可能比较熟悉，花萼是什么呢？花萼是花冠外面的绿色被叶，在花尚未开放时，保护着花蕾。四个属性的单位都是cm，属于数值变量，1个标签，表示鸢尾花的分类。

导入模块和定义变量

用到的框架主要包括：MindSpore1.3，专用于搭建整体的网络框架，这里主要用于神经网络的搭建。另外 os也被引入，它提供了丰富的函数，主要用来处理文件和目录。

实验代码：

import csv

import os

import time

import numpy as np

from easydict import EasyDict as edict

from matplotlib import pyplot as plt

import mindspore

from mindspore import nn

from mindspore import context

from mindspore import dataset

from mindspore.train.callback import TimeMonitor, LossMonitor

from mindspore import Tensor

from mindspore.train import Model

from mindspore.train.callback import ModelCheckpoint, CheckpointConfig

context.set\_context(mode=context.GRAPH\_MODE, device\_target="Ascend")

# 设定运行模式为静态图模式，并且运行设备为昇腾芯片

# 变量定义

cfg = edict({

'data\_size': 150,

'train\_size': 120, # 训练集大小

'test\_size': 30 , # 测试集大小

'feature\_number': 4, # 输入特征数

'num\_class': 3, # 分类类别

'batch\_size': 30,

'data\_dir': './iris.data',

'save\_checkpoint\_steps': 5, # 多少步保存一次模型

'keep\_checkpoint\_max': 1, # 最多保存多少个模型

'out\_dir\_no\_opt': './model\_iris/no\_opt', # 保存模型路径，无优化器模型

'out\_dir\_sgd': './model\_iris/sgd', # 保存模型路径,SGD优化器模型

'out\_dir\_momentum': './model\_iris/momentum', # 保存模型路径，momentum模型

'out\_dir\_adam': './model\_iris/adam', # 保存模型路径，adam优化器模型

'output\_prefix': "checkpoint\_fashion\_forward" # 保存模型文件名

})

读取数据并预处理

# 读取鸢尾花数据集，本数据集共有150个带标签的数据

with open(cfg.data\_dir) as csv\_file:

data = list(csv.reader(csv\_file, delimiter=','))

# 共150条数据，将数据集的4个属性作为自变量X。将数据集的3个类别映射为{0, 1，2}，作为因变量Y

label\_map = {'setosa': 0,'versicolor': 1,'virginica':2 }

# 分别获取数据中的特征值X和标签值Y

X = np.array([[float(x) for x in s[:-1]] for s in data[:cfg.data\_size]], np.float32)

Y = np.array([label\_map[s[-1]] for s in data[:cfg.data\_size]], np.int32)

# 将数据集分为训练集120条，测试集30条。

train\_idx = np.random.choice(cfg.data\_size, cfg.train\_size, replace=False)

test\_idx = np.array(list(set(range(cfg.data\_size)) - set(train\_idx)))

X\_train, Y\_train = X[train\_idx], Y[train\_idx]

X\_test, Y\_test = X[test\_idx], Y[test\_idx]

# 使用MindSpore GeneratorDataset接口将numpy.ndarray类型的数据转换为Dataset

def gen\_data(X\_train, Y\_train, epoch\_size):

# 生成训练集

XY\_train = list(zip(X\_train, Y\_train))

ds\_train = dataset.GeneratorDataset(XY\_train, ['x', 'y'])

# 设定数据集大小

# 打乱操作并设定batchsize

ds\_train = ds\_train.shuffle(buffer\_size=cfg.train\_size).batch(cfg.batch\_size, drop\_remainder=True)

# 生成测试集

XY\_test = list(zip(X\_test, Y\_test))

ds\_test = dataset.GeneratorDataset(XY\_test, ['x', 'y'])

# 设定数据集大小

# 打乱操作并设定batchsize

ds\_test = ds\_test.shuffle(buffer\_size=cfg.test\_size).batch(cfg.test\_size, drop\_remainder=True)

return ds\_train, ds\_test

定义训练、评估、测试函数

# 训练函数

def train(network, net\_opt, ds\_train, prefix, directory, print\_times):

# 定义网络损失函数

net\_loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction="mean")

# 定义模型

model = Model(network, loss\_fn=net\_loss, optimizer=net\_opt, metrics={"acc"})

# 定义损失值指标

loss\_cb = LossMonitor(per\_print\_times=int(cfg.train\_size / cfg.batch\_size))

# 设置checkpoint

config\_ck = CheckpointConfig(save\_checkpoint\_steps=cfg.save\_checkpoint\_steps,

keep\_checkpoint\_max=cfg.keep\_checkpoint\_max)

ckpoint\_cb = ModelCheckpoint(prefix=prefix, directory=directory, config=config\_ck)

print("============== Starting Training ==============")

# 训练模型

model.train(epoch\_size, ds\_train, callbacks=[ckpoint\_cb, loss\_cb], dataset\_sink\_mode=False)

return model

class\_names=['setosa', 'versicolor', 'virginica']

# 评估预测函数

def eval\_predict(model, ds\_test):

# 使用测试集评估模型，打印总体准确率

metric = model.eval(ds\_test)

print(metric)

# 预测

test\_ = ds\_test.create\_dict\_iterator().\_\_next\_\_()

test = Tensor(test\_['x'], mindspore.float32)

predictions = model.predict(test)

predictions = predictions.asnumpy()

true\_label = test\_['y'].asnumpy()

for i in range(10):

p\_np = predictions[i, :]

pre\_label = np.argmax(p\_np)

print('第' + str(i) + '个sample预测结果：', class\_names[pre\_label], ' 真实结果：', class\_names[true\_label[i]])

### 无优化器模型训练并预测

epoch\_size = 20 # 训练20次

print('------------------无优化器--------------------------')

# 数据

ds\_train, ds\_test = gen\_data(X\_train, Y\_train, epoch\_size) # 生成训练集和测试集

# 定义网络并训练

network = nn.Dense(cfg.feature\_number, cfg.num\_class) # 定义一个全连接网络层，输入特征为4，输出类别为3

model = train(network, None, ds\_train, "checkpoint\_no\_opt", cfg.out\_dir\_no\_opt, print\_times=4) # 用训练集训练网络，设置网络结构，模型名称，保存路径, print\_times

# 评估预测

eval\_predict(model, ds\_test) # 用测试集进行预测

输出结果：

------------------无优化器--------------------------

============== Starting Training ==============

epoch: 1 step: 4, loss is 1.1123106

epoch: 2 step: 4, loss is 1.1202087

epoch: 3 step: 4, loss is 1.1081936

epoch: 4 step: 4, loss is 1.1139123

epoch: 5 step: 4, loss is 1.1177686

epoch: 6 step: 4, loss is 1.1094363

epoch: 7 step: 4, loss is 1.1057143

epoch: 8 step: 4, loss is 1.1137706

epoch: 9 step: 4, loss is 1.1225764

epoch: 10 step: 4, loss is 1.1169411

epoch: 11 step: 4, loss is 1.115051

epoch: 12 step: 4, loss is 1.1098907

epoch: 13 step: 4, loss is 1.1069646

epoch: 14 step: 4, loss is 1.1149268

epoch: 15 step: 4, loss is 1.1180775

epoch: 16 step: 4, loss is 1.1170397

epoch: 17 step: 4, loss is 1.1150074

epoch: 18 step: 4, loss is 1.1237894

epoch: 19 step: 4, loss is 1.1233115

epoch: 20 step: 4, loss is 1.116143

{'acc': 0.43333333333333335}

第0个sample预测结果： versicolor 真实结果： virginica

第1个sample预测结果： versicolor 真实结果： versicolor

第2个sample预测结果： versicolor 真实结果： virginica

第3个sample预测结果： versicolor 真实结果： setosa

第4个sample预测结果： versicolor 真实结果： versicolor

第5个sample预测结果： versicolor 真实结果： setosa

第6个sample预测结果： setosa 真实结果： virginica

第7个sample预测结果： versicolor 真实结果： versicolor

第8个sample预测结果： versicolor 真实结果： setosa

第9个sample预测结果： versicolor 真实结果： virginica

### SGD优化器模型训练并预测

# ---------------------------------------------------SGD-------------------------------------

epoch\_size = 300 # 训练300次

lr = 0.01

print('-------------------SGD优化器-----------------------')

# 数据

ds\_train, ds\_test = gen\_data(X\_train, Y\_train, epoch\_size) # 生成训练集和测试集

# 定义网络并训练、测试、预测

network = nn.Dense(cfg.feature\_number, cfg.num\_class) # 定义一个全连接网络层，输入特征为4，输出类别为3

net\_opt = nn.SGD(network.trainable\_params(), lr) # 用SGD优化器进行优化

model = train(network, net\_opt, ds\_train, "checkpoint\_sgd", cfg.out\_dir\_sgd, 40) # 用训练集训练网络，设置网络结构，优化器，模型名称，保存路径, print\_times

# 评估预测

eval\_predict(model, ds\_test) # 用测试集进行预测

输出结果：

-------------------SGD优化器-----------------------

============== Starting Training ==============

epoch: 1 step: 4, loss is 1.0863036

epoch: 2 step: 4, loss is 1.0471957

epoch: 3 step: 4, loss is 1.0281278

epoch: 4 step: 4, loss is 1.0003911

epoch: 5 step: 4, loss is 1.0010878

epoch: 6 step: 4, loss is 0.9402846

epoch: 7 step: 4, loss is 0.936496

epoch: 8 step: 4, loss is 0.9070135

epoch: 9 step: 4, loss is 0.8746185

epoch: 10 step: 4, loss is 0.8588983

epoch: 11 step: 4, loss is 0.8791269

epoch: 12 step: 4, loss is 0.8409094

epoch: 13 step: 4, loss is 0.8171078

epoch: 14 step: 4, loss is 0.78627276

...

{'acc': 1.0}

第0个sample预测结果： versicolor 真实结果： versicolor

第1个sample预测结果： virginica 真实结果： virginica

第2个sample预测结果： virginica 真实结果： virginica

第3个sample预测结果： setosa 真实结果： setosa

第4个sample预测结果： virginica 真实结果： virginica

第5个sample预测结果： versicolor 真实结果： versicolor

第6个sample预测结果： versicolor 真实结果： versicolor

第7个sample预测结果： setosa 真实结果： setosa

第8个sample预测结果： virginica 真实结果： virginica

第9个sample预测结果： setosa 真实结果： setosa

### Momentum优化器模型训练并预测

epoch\_size = 20 # 训练20次

lr = 0.01 # 学习率为0.01

print('-------------------Momentum优化器-----------------------')

# 数据

ds\_train, ds\_test = gen\_data(X\_train, Y\_train, epoch\_size) # 生成训练集和测试集

# 定义网络并训练

network = nn.Dense(cfg.feature\_number, cfg.num\_class) # 定义一个全连接网络层，输入特征为4，输出类别为3

net\_opt = nn.Momentum(network.trainable\_params(), lr, 0.9) # 用 momentum 优化器进行优化，学习率为0.01，动量大小为0.9

model = train(network, net\_opt, ds\_train, "checkpoint\_momentum", cfg.out\_dir\_momentum, 4) # 用训练集训练网络，设置网络结构，优化器，模型名称，保存路径, print\_times

# 评估预测

eval\_predict(model, ds\_test) # 用测试集进行预测

输出结果：

-------------------Momentum优化器-----------------------

============== Starting Training ==============

epoch: 1 step: 4, loss is 1.0810095

epoch: 2 step: 4, loss is 0.94718075

epoch: 3 step: 4, loss is 0.8274425

epoch: 4 step: 4, loss is 0.66691095

epoch: 5 step: 4, loss is 0.6318491

epoch: 6 step: 4, loss is 0.57523024

epoch: 7 step: 4, loss is 0.56330514

epoch: 8 step: 4, loss is 0.481749

epoch: 9 step: 4, loss is 0.39138663

epoch: 10 step: 4, loss is 0.4436511

epoch: 11 step: 4, loss is 0.4150259

epoch: 12 step: 4, loss is 0.4283092

epoch: 13 step: 4, loss is 0.41189855

epoch: 14 step: 4, loss is 0.38341784

epoch: 15 step: 4, loss is 0.38927388

epoch: 16 step: 4, loss is 0.39323956

epoch: 17 step: 4, loss is 0.38287503

epoch: 18 step: 4, loss is 0.43862677

epoch: 19 step: 4, loss is 0.3592914

epoch: 20 step: 4, loss is 0.35841298

{'acc': 0.9666666666666667}

第0个sample预测结果： versicolor 真实结果： versicolor

第1个sample预测结果： versicolor 真实结果： versicolor

第2个sample预测结果： virginica 真实结果： virginica

第3个sample预测结果： versicolor 真实结果： versicolor

第4个sample预测结果： versicolor 真实结果： versicolor

第5个sample预测结果： virginica 真实结果： versicolor

第6个sample预测结果： setosa 真实结果： setosa

第7个sample预测结果： versicolor 真实结果： versicolor

第8个sample预测结果： virginica 真实结果： virginica

第9个sample预测结果： virginica 真实结果： virginica

### Adam优化器模型训练并预测

epoch\_size = 15 # 训练15次

lr = 0.1 # 学习率为0.1, 动态学习率

print('------------------Adam优化器--------------------------')

# 数据

ds\_train, ds\_test = gen\_data(X\_train, Y\_train, epoch\_size) # 生成训练集和测试集

# 定义网络并训练

network = nn.Dense(cfg.feature\_number, cfg.num\_class) # 定义一个全连接网络层，输入特征为4，输出类别为3

net\_opt = nn.Adam(network.trainable\_params(), learning\_rate=lr) # 用 Adam 优化器进行优化，学习率为0.1

model = train(network, net\_opt, ds\_train, "checkpoint\_adam", cfg.out\_dir\_adam, 4) # 用训练集训练网络，设置网络结构，优化器，模型名称，保存路径, print\_times

# 评估预测

eval\_predict(model, ds\_test)

输出结果：

-----------------Adam优化器--------------------------

============== Starting Training ==============

epoch: 1 step: 4, loss is 0.85976017

epoch: 2 step: 4, loss is 0.6710789

epoch: 3 step: 4, loss is 0.5583926

epoch: 4 step: 4, loss is 0.5004689

epoch: 5 step: 4, loss is 0.37697986

epoch: 6 step: 4, loss is 0.39567876

epoch: 7 step: 4, loss is 0.3641772

epoch: 8 step: 4, loss is 0.31769836

epoch: 9 step: 4, loss is 0.33615974

epoch: 10 step: 4, loss is 0.36870608

epoch: 11 step: 4, loss is 0.31535652

epoch: 12 step: 4, loss is 0.29351452

epoch: 13 step: 4, loss is 0.27576455

epoch: 14 step: 4, loss is 0.24675885

epoch: 15 step: 4, loss is 0.31923586

{'acc': 1.0}

{'acc': 0.9666666666666667}

第0个sample预测结果： setosa 真实结果： setosa

第1个sample预测结果： versicolor 真实结果： versicolor

第2个sample预测结果： versicolor 真实结果： versicolor

第3个sample预测结果： versicolor 真实结果： versicolor

第4个sample预测结果： setosa 真实结果： setosa

第5个sample预测结果： virginica 真实结果： virginica

第6个sample预测结果： versicolor 真实结果： versicolor

第7个sample预测结果： virginica 真实结果： virginica

第8个sample预测结果： setosa 真实结果： setosa

第9个sample预测结果： setosa 真实结果： setosa

### 查看保存的模型

os.listdir('./model\_iris/adam') # 查看保存的模型

输出结果：

['checkpoint\_adam-graph.meta', 'checkpoint\_adam-15\_4.ckpt']

### 停止实验环境

试验完成之后请及时停止实验环境，避免资源浪费，如下图：



停止实验环境

## 结果分析

从无优化器、SGD、momentum、adam优化器的loss上看：

* 无优化器训练loss基本没有发生变化，测试结果效果差。多运行几次发现结果偏差太大，学员可以自己尝试。
* SGD优化器loss下降速度很慢，而且在接近收敛处loss下降非常缓慢。增大学习率，减少迭代次数，会出现收敛到局部最优解的情况，学员可以自己尝试。
* Momentum优化器loss下降速度较快，充分说明Momentum优化器改进了SGD收敛速度慢的问题。改变参数，比较不同学习率和迭代次数的结果，会发现该优化器稳定性很强，学习率容易选择。相比于SGD优化器容易调参。
* adam优化器loss下降速度最快快，只需要15 epoch就可以达到收敛。改变模型学习率多跑几次，会发现，adam优化器可以适应不用的学习率，容易调节参数。

## 实验总结

本实验目的是使用MindSpore对比不同优化器算法，通过比较不同优化器实验结果，得到如下结论：

* 无优化器条件训练，不更新参数，训练loss基本不变。模型与初始化参数关系大，随机性较强，效果差，不适合应用；
* 有优化器条件下训练，更新模型参数。训练loss改变；
* SGD优化器收敛速度慢，参数不容易调节；
* Momentum优化器是SGD加入历史梯度动量，增加了稳定性，收敛速度有明显提高，参数较SGD容易调节；
* Adam优化器是 RMSprop + Momentum。可以适应不同学习率，可以自动为不同参数调节学习率。稳定性高，参数容易调节；

## 创新设计

1. 编写代码实现Adam优化器，并用该优化器优化Beale方程。

# 附录：ModelArts开发环境搭建

* ModelArts平台：Mindspore-1.3

进入ModelArts

在[华为云](https://www.huaweicloud.com/)主页搜索Modelarts，点击“AI开发平台ModelArts”中的“进入控制台”。

图形用户界面, 文本, 应用程序

描述已自动生成

选择训练作业

选择“北京四”地区，在左侧下拉框中点击“开发环境”中的“Notebook”：

电脑萤幕的截图

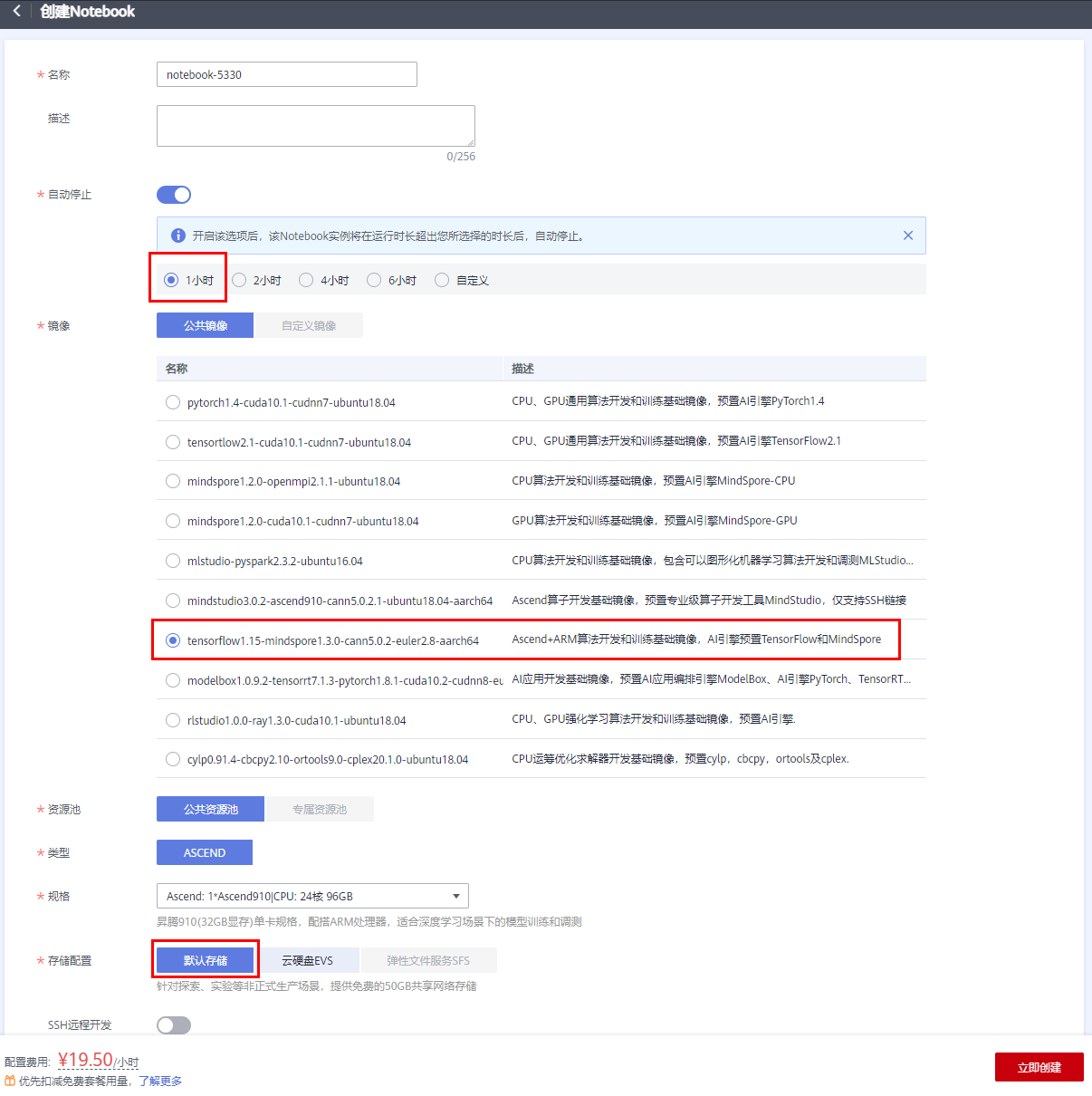
描述已自动生成

创建Notebook

点击创建按钮来创建一个新的Notebook，选择如下配置：

* 名称：自定义。
* 工作环境：Ascend+ARM算法开发和训练基础镜像。
* 存储配置：默认存储。

点击“下一步”，确认规格如下后选择提交：



启动Notebook进入开发环境

当Notebook状态变为“运行中”时，点击右侧“打开”按钮打开Notebook。打开后选择右侧“MindSpore-python3.7-aarch64”按钮，进入Notebook环境：