**《深度学习与大数据智能》实验报告一**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **年级、专业、班级** | | **20计算机科学与技术（卓越）01** | | **姓名** | | **黄昊** | **学号** | **20204205** |
| **实验题目** | **鸢尾花分类任务对比实验** | | | | | | | |
| **g实验时间** | **2023/05/06** | | **实验地点** | | **DS3401** | | | |
| **学年学期** | **2022-2023（2）** | | **实验性质** | | **□验证性 □设计性 ■综合性** | | | |
| 一、实验目的  1．熟悉实验环境。  2. 理解和掌握卷积神经网络的基本构成。  3. 理解不同优化器的原理及在卷积神经网络中的作用。 | | | | | | | | |
| 二、实验项目内容  1. 基本内容：  1）利用不同优化器求解Beale函数极值点  2) 构建基于鸢尾花数据集的分类卷积神经网络模型。  2. 进阶内容：  1）基于求函数极值点实验，使用不同优化器进行实验，并对实验结果进行分析和对比。  2）自己编写不同的优化器，实现鸢尾的分类,并与框架集成的优化器进行对比分析。 | | | | | | | | |
| 三、实验的过程和主要源程序  ①实验原理  A.Beale函数  Beale函数有一个复杂的曲面，其公式为：    其三维曲面和等高线图如下两图所示：      可以看出，该函数状况比较复杂，有多个局部最优解。因此，这个函数可以用来测试不同优化器的性能优劣，可以看哪些优化器有更好的跳出局部最优解和鞍点的能力。  B.SGD优化器  SGD优化器每次选出样本中的一部分（即mini-batch），利用普通的梯度下降法进行更新。梯度下降法的公式为：  其中g为原函数的导数，*η*为学习率。  SGD的一个问题是跳出局部最优解的能力较差。因此本实验还尝试了以下优化器：  C.Momentum优化器  动量梯度下降法的公式如下所示：    该公式的更新步长为*v*，可以看出更新步长对前一个迭代步骤的更新步长有所记忆（即*μv*项）。其借鉴了物理学的概念，对于局部最优解，其有一定的跳出局部最优解的能力。  D.AdaGrad优化器  AdaGrad的更新公式如下所示：    该公式引入了自适应学习率的机制。其中σ为超参数，ε为防止数值不稳定引入的极小值。观察公式，我们可以发现，在梯度较⼤的地⽅，学习率衰减更快，下⼀步迭代步⻓更小；梯度较小的地⽅，学习率衰减更慢。但随着迭代次数增加，学习率一定会越来越小。这与直观的初期迭代步长大，尽量搜到全局最优；后期稳定下来，方便找到全局最优的直觉符合。通过对步长的记录，AdaGrad实现了自适应学习率的机制。  E.Adam优化器  Adam优化器的更新公式如下：    该公式结合了动量的思想和自适应学习率的思想，拥有更好的跳出局部最优解的能力。其中*s*项控制学习率的变化，*v*项目控制迭代步长的变化，同时具有一定的“动量”。通过两种思想的结合，我们可以更好地搜到全局最优解。  ②实验思路  对于函数极值点的任务，我们手写了SGD，Momentum和AdaGrad优化器，并使用作为评价指标；对于鸢尾花分类的任务，我们首先对数据集以6：4的比例划分为训练集和测试集，使用了框架提供的Adam，SGD和Momentum优化器，并自己编写了Adam，SGD和Momentum优化器，同时搭建全连接神经网络和卷积神经网络，并使用框架提供的优化器和自己写的优化器，在两种网络上进行优化，以交叉熵作为损失函数，就训练集和测试集的准确率和acc进行比较。  ③源程序  A.函数极值点优化  import numpy as np  import matplotlib.pyplot as plt  import matplotlib.colors as anaplt\_cl  # 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  # 计算R2损失  def getR2Loss(lst):      bestVal = [3, 0.5]      bestY = beale(bestVal[0], bestVal[1])      ans = []      for pair in lst:          x1 = float(pair[0])          x2 = float(pair[1])          y = beale(x1, x2)          ans.append((y - bestY) \*\* 2)      return ans  # 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=anaplt\_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)  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  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)  sgd\_r2 = getR2Loss(x\_traj\_sgd)  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  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)  momentum\_r2 = getR2Loss(x\_traj\_momentum)  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).astype(float)     # 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])).astype(float)          r += dfdx \*\* 2          x\_traj.append(x\_traj[-1] - conf\_para['learning\_rate'] / (np.sqrt(r) + conf\_para['epsilon']) \* dfdx)      return x\_traj  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)  adagrad\_r2 = getR2Loss(x\_traj\_adag)  plt.plot(sgd\_r2)  plt.plot(adagrad\_r2)  plt.plot(momentum\_r2)  plt.legend(["SGD", "AdaGrad", "Momentum"])  endPoint = 30  plt.plot(sgd\_r2[:endPoint])  plt.plot(adagrad\_r2[:endPoint])  plt.plot(momentum\_r2[:endPoint])  plt.legend(["SGD", "AdaGrad", "Momentum"])  B.鸢尾花分类  import torch  import torch.nn as nn  import pandas as pd  import numpy as np  import matplotlib.pyplot as plt  import torch.nn.functional as F  from torch.utils.data import TensorDataset  from torch.utils.data import DataLoader  from sklearn.model\_selection import train\_test\_split  # CONFIG  batch = 8  epochs = 250  # 注意这个要调，改成False就会跑CNN的  isMLP = True  KERNAL\_SIZE = 2  OUTPUT\_CHANNEL = 16  data = pd.read\_csv(r'iris.csv')  # 将鸢尾花种类由字母转换成数字  data['Species']=pd.factorize(data.Species)[0]  # 保存“萼片长度”，“萼片宽度”，“花瓣长度”，“花瓣宽度”  X = data.iloc[:,1:-1].values  # 保存鸢尾花种类  Y = data.Species.values  # CNN 需要(N, C, L)的数据  if isMLP == False:      X = X.reshape(X.data.shape[0], 1, X.data.shape[1])  # 划分训练集和测试集 默认6：4  train\_x,test\_x,train\_y,test\_y=train\_test\_split(X,Y)  # 将numpy数据转换成tensor  train\_x = torch.from\_numpy(train\_x).type(torch.FloatTensor)  train\_y = torch.from\_numpy(train\_y).type(torch.LongTensor)  test\_x = torch.from\_numpy(test\_x).type(torch.FloatTensor)  test\_y = torch.from\_numpy(test\_y).type(torch.LongTensor)  # 创建训练集  train\_ds = TensorDataset(train\_x,train\_y)  train\_dl = DataLoader(train\_ds,batch\_size=batch,shuffle=True)  # 创建测试集  test\_ds = TensorDataset(test\_x,test\_y)  test\_dl = DataLoader(test\_ds,batch\_size=batch)  class MLPModel(nn.Module):      def \_\_init\_\_(self):          super().\_\_init\_\_()          self.model = nn.Sequential(              nn.Linear(4, 32),              nn.ReLU(True),              nn.Linear(32, 32),              nn.ReLU(True),              nn.Linear(32, 3),          )      def forward(self,input):          x = self.model(input)          return x  class CNNModel(nn.Module):      def \_\_init\_\_(self):          super().\_\_init\_\_()          self.model = nn.Sequential(              nn.Conv1d(1, OUTPUT\_CHANNEL, kernel\_size=KERNAL\_SIZE),              nn.ReLU(True),              nn.MaxPool1d(kernel\_size=KERNAL\_SIZE),              nn.ReLU(),              nn.Dropout(p=0.5),              nn.ReLU(),              nn.Flatten(),              nn.LazyLinear(3),          )      def forward(self,input):          x = self.model(input)          return x    class SGD(torch.optim.Optimizer):      def \_\_init\_\_(self, params, lr=0.01):          defaults = dict(lr=lr)          super(SGD, self).\_\_init\_\_(params, defaults)      def step(self, closure = None):          loss = None          if closure is not None:              loss = closure()          for group in self.param\_groups:              lr = group['lr']              for p in group['params']:                  if p.grad is None:                      continue                  grad = p.grad.data                  p.data.add\_(grad, alpha=-lr)                  return loss    class AdamOptimizer(torch.optim.Optimizer):      def \_\_init\_\_(self, params, lr=0.01, betas=(0.9, 0.999), eps = 1e-8):          defaults = dict(lr=lr, betas=betas, eps = eps)          super(AdamOptimizer, self).\_\_init\_\_(params, defaults)      def step(self, closure=None):          for group in self.param\_groups:              for p in group['params']:                  if p.grad is None:                      continue                  # 获取基本参数                  grad = p.grad.data                  lr = group['lr']                  eps = group['eps']                  beta1, beta2 = group["betas"]                  state = self.state[p]                  # 初始化参数                  if len(state) == 0:                      state['m'] = torch.zeros\_like(p.data)                      state['v'] = torch.zeros\_like(p.data)                  v = state['v']                  m = state['m']                  g\_t = p.grad.data                  m\_t = beta1 \* m + (1 - beta1) \* g\_t                  v\_t = beta2 \* v + (1 - beta2) \* g\_t \*\* 2                  m\_hat = m\_t / (1 - beta1)                  v\_hat = v\_t / (1 - beta2)                  p.data.add\_(m\_hat / (eps + torch.sqrt(v\_hat)), alpha = -lr)                  if closure is not None:                      return closure()  class MomentumOptimizer(torch.optim.Optimizer):      def \_\_init\_\_(self, params, lr=0.01, momentum=0.9):          defaults = dict(lr=lr, momentum=momentum)          # 调用父类构造函数          super(MomentumOptimizer, self).\_\_init\_\_(params, defaults)      def step(self, closure=None):          for group in self.param\_groups:              for p in group['params']:                  if p.grad is None:                      continue                  grad = p.grad.data                  lr = group['lr']                  momentum = group['momentum']                  state = self.state[p]                  if len(state) == 0:                      state['velocity'] = torch.zeros\_like(p.data)                  velocity = state['velocity']                  velocity.mul\_(momentum).add\_(grad)                  p.data.add\_(velocity, alpha = -lr)                  if closure is not None:                      return closure()  def accuracy(y\_pred,y\_true):      # torch.argmax将数字转换成真正的预测结果      y\_pred = torch.argmax(y\_pred,dim=1)      acc = (y\_pred == y\_true).float().mean()      return acc  def trainModel(model, loss\_fn, optim):      train\_loss=[]      train\_acc=[]      test\_loss=[]      test\_acc=[]      for epoch in range(epochs):          # 训练          for x,y in train\_dl:              model.train()              y\_pred = model(x)              loss = loss\_fn(y\_pred, y)              optim.zero\_grad()              loss.backward()              optim.step()          # 测试          with torch.no\_grad():              model.eval()              epoch\_accuracy = accuracy(model(train\_x),train\_y)       # 训练集准确率              epoch\_loss = loss\_fn(model(train\_x), train\_y).data      # 训练集loss              epoch\_test\_accuracy = accuracy(model(test\_x),test\_y)    # 测试集准确率              epoch\_test\_loss = loss\_fn(model(test\_x), test\_y).data   # 测试集loss              if epoch % 50 == 0:                  print('epoch: ',epoch,'train\_loss: ',round(epoch\_loss.item(),3),'train\_accuracy: ',round(epoch\_accuracy.item(),3),                      'test\_loss: ',round(epoch\_test\_loss.item(),3),'test\_accuracy: ',round(epoch\_test\_accuracy.item(),3)                      )              train\_loss.append(epoch\_loss)              train\_acc.append(epoch\_accuracy)              test\_loss.append(epoch\_test\_loss)              test\_acc.append(epoch\_test\_accuracy)      print('final test\_accuracy: ',epoch\_test\_accuracy)      return train\_loss, train\_acc, test\_loss, test\_acc, float(epoch\_test\_accuracy)  # 画图  def drawFig(train\_loss, train\_acc, test\_loss, test\_acc, path, final\_test\_acc):      plt.cla()      plt.plot(range(1,epochs+1),train\_loss,label='train\_loss')      plt.plot(range(1,epochs+1),test\_loss,label='test\_loss')      plt.plot(range(1,epochs+1),train\_acc,label='train\_acc')      plt.plot(range(1,epochs+1),test\_acc,label='test\_acc')      plt.legend()      plt.title(path + "  final\_testacc: " + str(final\_test\_acc))      plt.savefig(path)  ## Config  LR = 0.0001  if isMLP:      ## MLP + SGD\_Write      model = MLPModel()      loss\_fn = nn.CrossEntropyLoss()      optim\_adam = SGD(model.parameters(), lr = LR)      train\_loss, train\_acc, test\_loss, test\_acc, final\_test\_acc= trainModel(model, loss\_fn, optim\_adam)      drawFig(train\_loss, train\_acc, test\_loss, test\_acc, "MLP\_SGD\_Write", final\_test\_acc)      ## MLP + Adam\_Write      model = MLPModel()      loss\_fn = nn.CrossEntropyLoss()      optim\_adam = AdamOptimizer(model.parameters(), lr = LR)      train\_loss, train\_acc, test\_loss, test\_acc, final\_test\_acc= trainModel(model, loss\_fn, optim\_adam)      drawFig(train\_loss, train\_acc, test\_loss, test\_acc, "MLP\_Adam\_Write", final\_test\_acc)      ## MLP + Momentum\_Write      model = MLPModel()      loss\_fn = nn.CrossEntropyLoss()      optim\_adam = MomentumOptimizer(model.parameters(), lr = LR)      train\_loss, train\_acc, test\_loss, test\_acc, final\_test\_acc= trainModel(model, loss\_fn, optim\_adam)      drawFig(train\_loss, train\_acc, test\_loss, test\_acc, "MLP\_Momentum\_Write", final\_test\_acc)      ## MLP + Adam      model = MLPModel()      loss\_fn = nn.CrossEntropyLoss()      optim\_adam = torch.optim.Adam(model.parameters(),lr=LR)      train\_loss, train\_acc, test\_loss, test\_acc, final\_test\_acc= trainModel(model, loss\_fn, optim\_adam)      drawFig(train\_loss, train\_acc, test\_loss, test\_acc, "MLP\_Adam", final\_test\_acc)      ## MLP + SGD      model = MLPModel()      loss\_fn = nn.CrossEntropyLoss()      optim\_adam = torch.optim.SGD(model.parameters(),lr = LR)      train\_loss, train\_acc, test\_loss, test\_acc, final\_test\_acc= trainModel(model, loss\_fn, optim\_adam)      drawFig(train\_loss, train\_acc, test\_loss, test\_acc, "MLP\_SGD", final\_test\_acc)      ## MLP + Momentum      model = MLPModel()      loss\_fn = nn.CrossEntropyLoss()      optim\_adam = torch.optim.SGD(params=model.parameters(),lr=LR, momentum=0.9)      train\_loss, train\_acc, test\_loss, test\_acc, final\_test\_acc= trainModel(model, loss\_fn, optim\_adam)      drawFig(train\_loss, train\_acc, test\_loss, test\_acc, "MLP\_Momentum", final\_test\_acc)  else:      ## CNN + SGD\_Write      model = CNNModel()      loss\_fn = nn.CrossEntropyLoss()      optim\_adam = SGD(model.parameters(), lr = LR)      train\_loss, train\_acc, test\_loss, test\_acc, final\_test\_acc= trainModel(model, loss\_fn, optim\_adam)      drawFig(train\_loss, train\_acc, test\_loss, test\_acc, "CNN\_SGD\_Write", final\_test\_acc)      ## CNN + Adam\_Write      model = CNNModel()      loss\_fn = nn.CrossEntropyLoss()      optim\_adam = AdamOptimizer(model.parameters(), lr = LR)      train\_loss, train\_acc, test\_loss, test\_acc, final\_test\_acc= trainModel(model, loss\_fn, optim\_adam)      drawFig(train\_loss, train\_acc, test\_loss, test\_acc, "CNN\_Adam\_Write", final\_test\_acc)      ## CNN + Momentum\_Write      model = CNNModel()      loss\_fn = nn.CrossEntropyLoss()      optim\_adam = MomentumOptimizer(model.parameters(), lr = LR)      train\_loss, train\_acc, test\_loss, test\_acc, final\_test\_acc= trainModel(model, loss\_fn, optim\_adam)      drawFig(train\_loss, train\_acc, test\_loss, test\_acc, "CNN\_Momentum\_Write", final\_test\_acc)      ## CNN + Adam      model = CNNModel()      loss\_fn = nn.CrossEntropyLoss()      optim\_adam = torch.optim.Adam(model.parameters(),lr=LR)      train\_loss, train\_acc, test\_loss, test\_acc, final\_test\_acc= trainModel(model, loss\_fn, optim\_adam)      drawFig(train\_loss, train\_acc, test\_loss, test\_acc, "CNN\_Adam", final\_test\_acc)      ## CNN + SGD      model = CNNModel()      loss\_fn = nn.CrossEntropyLoss()      optim\_adam = torch.optim.SGD(model.parameters(),lr = LR)      train\_loss, train\_acc, test\_loss, test\_acc, final\_test\_acc= trainModel(model, loss\_fn, optim\_adam)      drawFig(train\_loss, train\_acc, test\_loss, test\_acc, "CNN\_SGD", final\_test\_acc)      ## CNN + Momentum      model = CNNModel()      loss\_fn = nn.CrossEntropyLoss()      optim\_adam = torch.optim.SGD(params=model.parameters(),lr=LR, momentum=0.9)      train\_loss, train\_acc, test\_loss, test\_acc, final\_test\_acc= trainModel(model, loss\_fn, optim\_adam)      drawFig(train\_loss, train\_acc, test\_loss, test\_acc, "CNN\_Momentum", final\_test\_acc) | | | | | | | | |
| 四、实验结果及分析和（或）源程序调试过程、实验总结与体会  1、实验运行结果截图  A.求函数极值点  初始点为(-1,2)的SGD，Momentum和AdaGrad的迭代情况和R2下降情况：          B.鸢尾花分类  全连接神经网络下，自己编写的优化器的迭代过程：        全连接神经网络，使用框架给的优化器的结果：        卷积神经网络下使用自己编写的优化器：        卷积神经网络，使用框架提供的优化器：        2、程序编写过程中遇到的问题及解决思路、方法等  实验中未碰到比较大的问题，由于第一个实验和第二个实验分别在最优化技术和机器学习课程中做过类似的任务（梯度下降法和使用logistic分类器对鸢尾花分类），因此上手起来也比较快。只是这次实验需要使用深度学习的框架，一些操作还不是很熟悉。这个问题通过充分查阅网上资料和官方文档就可以解决。  3、实验总结和体会  ①Beale函数极值点问题中，不同优化器之间的分析和对比  SGD的优化过程如下所示：    Momentum的优化过程如下所示：    Adagrad优化器的的优化过程如下所示：    可以看出，三个优化器都找到了最优解(3,0.5)，其中SGD最为稳定，AdaGrad次之，最后是Momentum。这里看起来SGD最好：迭代过程稳定，还找到了全局最优解。但是，我们的初始点选在了(1.0,1.5)，如果我们换成(-1,2)后，结果会有改变：其中SGD被困在了局部最优解(-2.363529592037631, 1.355159044091442)，Momentum和AdaGrad仍然能找到最优解，过程如下所示：        最后，我们来看看R2的变化：    以(-1,2)为起点进行迭代，我们可以发现，AdaGrad下降最快且最稳定，SGD下降相对较慢但也稳定，而Momentum最不稳定。可以看出，AdaGrad和SGD的迭代过程比Momentum稳定，之前提过SGD没有到最优解，但我们的R2由于选的是函数取值（*z=f(x1,x2)*），在局部最优点的取值和全局最优点的取值相近，因此最终的R2也相近。  ②鸢尾花数据集的卷积神经网络结果分析及不同优化器效果对比  考虑到鸢尾花的数据特征以及实验要求，我们选择了全连接神经网络和卷积神经网络。本次卷积神经网络，我选择了1\*2的卷积核（因为输入大小本身也不大，不可能选择更大的），同时因为实验要求，我重写了框架给定的优化器，并和框架给的优化器进行了比较。最后，我们从以下角度讨论结果：  A.卷积神经网络和全连接神经网络结果的对比  卷积神经网络在图像识别中运用广泛，但在本实验中的整体效果并不好。卷积神经网络下，仅有动量优化器在测试集上的表现良好，最终也能达到100%的正确率，但Adam和SGD的效果很差。其中最高的为自己写的Adam优化器，最终在测试集的准确率上有74%的效果（值得注意的是中间也出现过接近100%的效果，猜测是过拟合导致最终效果变差），其他的基本在40-50%上下，最差的是自己写的SGD优化器，几乎没有优化效果。  而全连接网络的效果表现良好  B.不同优化器之间在同一模型下的优化效果对比  C.相同优化器之间在不同模型下的优化效果对比  D.自己编写的优化器与框架给定的优化器的优化效果对比  **最终总结与体会：** | | | | | | | | |

实验报告填写说明：

1、第一、二部分由老师提供；

2、第三部分填写源程序和算法，源程序要符合程序编写风格（缩进、注释等）；

3、第四部分主要填写程序结果（截图）、解决问题的方法、总结和体会等；

4、报告规范：包含报告页眉、报告的排版、内容是否填写，命名是否规范等。

5、源程序和实验报告命名：学号姓名序号.py 学号姓名序号.docx，例如学号20181234的张三同学，**他的第一次实验命名为：20181234张三1.py和20181234张三1.docx**

**6、向蓝墨云提交两个文件，一个实验报告，一个程序文件（代码合并）**