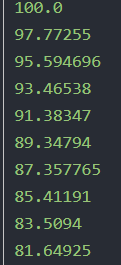
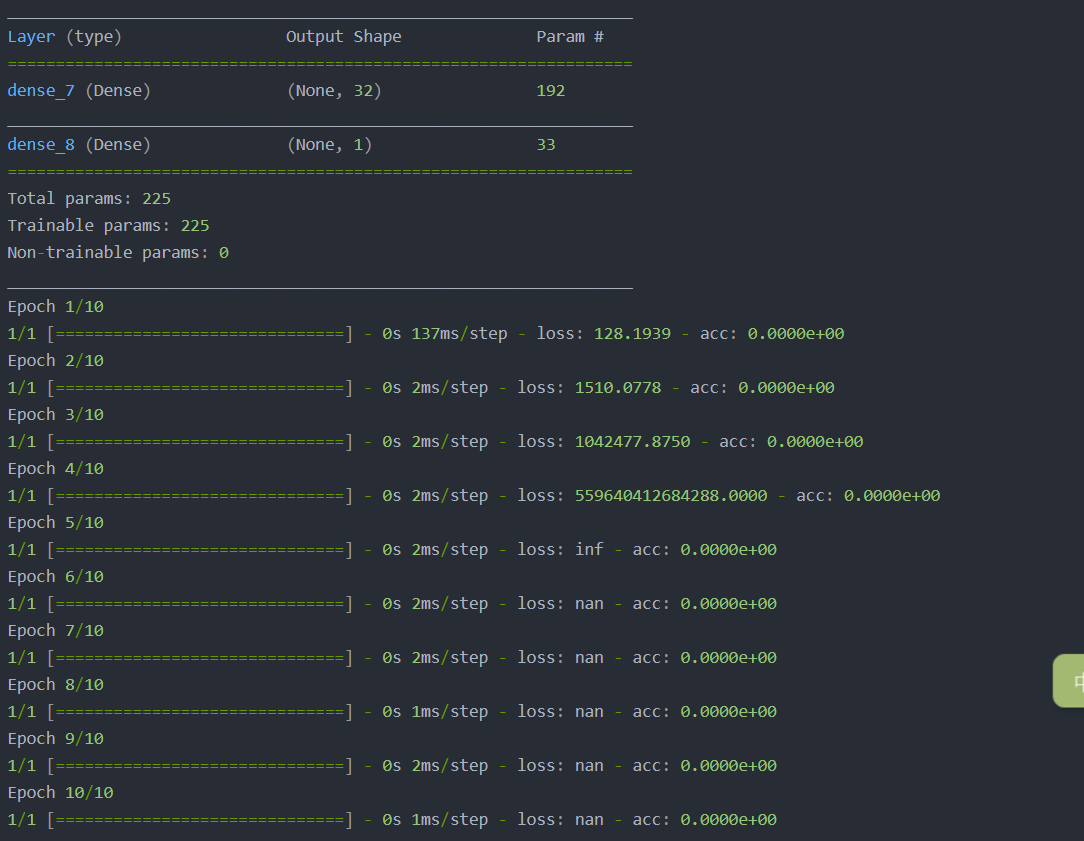
有关神经网络建模、强化学习与迁移学习

1、  
**import** tensorflow **as** tf  
**import** numpy **as** np  
x\_input = np.array([[1,2,3,4,5]]) *# 提供了一个虚拟数据集*y\_input = np.array([[10]])  
x = tf.placeholder(tf.float32,[**None**,5])*# 创建一个占位符*y = tf.placeholder(tf.float32,[**None**,1])  
W = tf.Variable(tf.zeros([5,1]))*# 使用一些变量对占位符进行操作*b = tf.Variable(tf.zeros([1]))  
y\_pred = tf.matmul(x,W)+b  
loss = tf.reduce\_sum(tf.pow((y - y\_pred),2)) *# 定义一个损失函数*train = tf.train.GradientDescentOptimizer(0.0001).minimize(loss) *# 指定优化器和想要最小化的变量*init = tf.global\_variables\_initializer() *# 初始化所有变量，创建一个名为init的变量*sess = tf.Session() *#创建一个回话，病运行10个周期训练数据*sess.run(init)  
**for** i **in** range(10):  
 feed\_dict = {x:x\_input, y:y\_input}  
 *#sess.run(train, feed\_dict = feed\_dict)* \_, loss\_value = sess.run([train,loss], feed\_dict = feed\_dict)  
 print(loss\_value)

运行结果：

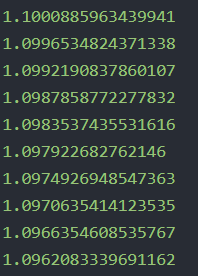


**from** keras.models **import** Sequential  
**from** keras.layers **import** Dense  
**import** numpy **as** np  
x\_input = np.array([[1,2,3,4,5]]) *# 提供一个虚拟数据集*y\_input = np.array([[10]])   
model = Sequential() *# 使用一个具有32个神经元的隐形层和一个神经元的输出层*model.add(Dense(units = 32,input\_dim = x\_input.shape[1]))  
model.add(Dense(units = 1))  
model .compile(loss = **'mse'**,optimizer = **'sgd'**, *#对模型进行编译，配置不同的设置，如损失函数、优化器和测度标准* metrics = [**'accuracy'**])  
model.summary() *#可以轻松输出显示的模型摘要*history = model.fit(x\_input,y\_input,epochs = 10,batch\_size = 32) *#直接训练模型，将结果保存*pred = model.predict(x\_input,batch\_size = 128)*#预测为函数可以在训练后使用*运行结果：

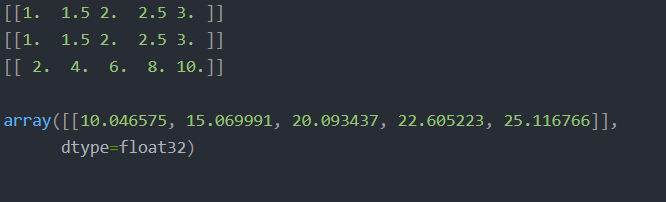


3.  
**import** torch  
batch\_size = 32 *#设定随机训练数据的大小*input\_shape = 5  
output\_shape = 10  
torch.set\_default\_tensor\_type(**'torch.cuda.FloatTensor'**) *#为了启动GPU,讲使用如下张量 这将保证所有的计算将使用附加的GPU***from** torch.autograd **import** Variable *# 用它来生成随机训练数据*x = Variable(torch.randn(batch\_size,input\_shape))  
y = Variable(torch.randn(batch\_size,output\_shape),requires\_grad = **False**)

*#使用一个简单的神经网络，其中一个有32个神经元的隐藏层和一个神经元的输出层 #使用.cuda()扩展保证模型在GPU上运行*model = torch.nn.Sequential(torch.nn.Linear(input\_shape,32),torch.nn.Linear(32,output\_shape)).cuda()  
loss\_function = torch.nn.MSELoss() *# 定义MSE损失函数*learning\_rate = 0.001*# 训练模型，迭代10次***for** i **in** range(10):  
 y\_pred = model(x)  
 loss = loss\_function(y\_pred,y)  
 print(loss.item())  
 *# 零梯度* model.zero\_grad()  
 loss.backward()  
  
 *# 更新权值* **for** param **in** model.parameters():  
 param.data -= learning\_rate \* param.grad.data



**import** mxnet **as** mx  
**import** numpy **as** np *#创建一些分配给GPU和CPU的简单虚拟数据*x\_input = mx.nd.empty((1,5),mx.gpu())  
x\_input[:] = np.array([[1,2,3,4,5]],np.float32)  
y\_input = mx.nd.empty((1,5),mx.cpu())  
y\_input[:] = np.array([[10,15,20,22.5,25]],np.float32)  
x\_input *# 可以很容易地复制和调整数据*w\_input = x\_input  
z\_input = x\_input.copyto(mx.cpu())  
x\_input += 1  
w\_input /= 2  
z\_input \*= 2  
print(x\_input.asnumpy()) *#输出显示*print(w\_input.asnumpy())  
print(z\_input.asnumpy())  
batch\_size = 1 *#创建一个迭代器*train\_iter = mx.io.NDArrayIter(x\_input,y\_input,batch\_size,  
 shuffle = **True**,data\_name = **'input'**,label\_name = **'target'**)  
X = mx.sym.Variable(**'input'**) *# 为模型创建符号*Y = mx.symbol.Variable(**'target'**)  
fc1 = mx.sym.FullyConnected(data =X,name = **'fc1'**, num\_hidden = 5)  
lin\_reg = mx.sym.LinearRegressionOutput(data = fc1, label = Y,name = **"lin\_reg"**)  
model = mx.mod.Module(symbol = lin\_reg, data\_names = [**'input'**],label\_names = [**'target'**]) *# 开始训练之前需要定义模型*model.fit(train\_iter,optimizer\_params={**'learning\_rate'**:0.01,**'momentum'**:0.9},num\_epoch = 100,  
 batch\_end\_callback = mx.callback.Speedometer(batch\_size,2))  
model.predict(train\_iter).asnumpy()



从互联网电影数据库（IMDB）获取50,000个流行电影影评作为数据集。这里将其分割为25,000个影评的训练集和25,000个影评的测试集。其中每个数据集都包含50%的好评和50%的差评。

**from** keras.datasets  
**import** imdb(train\_data,train\_labels),(test\_data,test\_labels) = imdb.load\_data(num\_words = 10000)  
*# 查看这两个数据*train\_data[0]train\_labels[0]  
*# 最长的句子只有10000个单词*max([max(sequence) **for** sequence **in** train\_data])  
word\_index = imdb.get\_word\_index()  
reverse\_word\_index = dict([(value,key) **for** (key,value) **in** word\_index.items()])  
decoded\_review = **''**.join([reverse\_word\_index.get(i-3,**'?'**) **for** i **in** train\_data[0]])  
*# print(decoded\_review)***import** numpy **as** np  
*# 将每个句子进行独热编码，形成0，1矩阵***def** vectorize\_sequences(sequences,dimension = 10000):  
 results = np.zeros((len(sequences),dimension))  
 **for** i,sequence **in** enumerate(sequences):  
 results[i,sequence] = 1.  
 **return** results  
  
x\_train = vectorize\_sequences(train\_data)  
x\_test = vectorize\_sequences(test\_data)  
  
*# 将标签也进行向量化*y\_train = np.asarray(train\_labels).astype(**'float32'**)  
y\_test = np.asarray(test\_labels).astype(**'float32'**)  
  
x\_train[0]  
array([0., 1., 1., ..., 0., 0., 0.])  
**from** keras **import** models  
**from** keras **import** layers  
  
model = models.Sequential()  
model.add(layers.Dense(16,activation = **'relu'**,input\_shape = (10000,)))  
model.add(layers.Dense(16,activation = **'relu'**))  
model.add(layers.Dense(1,activation = **'sigmoid'**))  
  
*# 损失函数为二元交叉熵,三个参数依次为：优化器，损失函数，评价指标*model.compile(optimizer = **'rmsprop'**,loss = **'binary\_crossentropy'**,metrics = [**'acc'**])  
  
*# 把训练样本分为训练集和交叉测试集*x\_val = x\_train[:10000]  
partial\_x\_train = x\_train[10000:]  
  
y\_val = y\_train[:10000]  
partial\_y\_train = y\_train[10000:]  
  
  
result = model.fit(partial\_x\_train,partial\_y\_train,epochs = 20,batch\_size = 512,validation\_data = (x\_val,y\_val))  
  
Train on 15000 samples, validate on 10000 samples  
Epoch 1/20  
15000/15000 [==============================] - 5s 338us/step - loss: 0.5326 - acc: 0.7917 - val\_loss: 0.4064 - val\_acc: 0.8700  
Epoch 2/20  
15000/15000 [==============================] - 4s 235us/step - loss: 0.3258 - acc: 0.8987 - val\_loss: 0.3154 - val\_acc: 0.8851  
Epoch 3/20  
15000/15000 [==============================] - 3s 232us/step - loss: 0.2357 - acc: 0.9245 - val\_loss: 0.2827 - val\_acc: 0.8899  
Epoch 4/20  
15000/15000 [==============================] - 3s 233us/step - loss: 0.1866 - acc: 0.9397 - val\_loss: 0.2862 - val\_acc: 0.8838  
Epoch 5/20  
15000/15000 [==============================] - 4s 235us/step - loss: 0.1502 - acc: 0.9521 - val\_loss: 0.2768 - val\_acc: 0.8887  
Epoch 6/20  
15000/15000 [==============================] - 4s 236us/step - loss: 0.1256 - acc: 0.9615 - val\_loss: 0.3118 - val\_acc: 0.8800  
Epoch 7/20  
15000/15000 [==============================] - 3s 233us/step - loss: 0.1045 - acc: 0.9687 - val\_loss: 0.3120 - val\_acc: 0.8837  
Epoch 8/20  
15000/15000 [==============================] - 4s 234us/step - loss: 0.0882 - acc: 0.9730 - val\_loss: 0.3211 - val\_acc: 0.8817  
Epoch 9/20  
15000/15000 [==============================] - 3s 232us/step - loss: 0.0736 - acc: 0.9792 - val\_loss: 0.3548 - val\_acc: 0.8819  
Epoch 10/20  
15000/15000 [==============================] - 4s 240us/step - loss: 0.0595 - acc: 0.9851 - val\_loss: 0.3971 - val\_acc: 0.8744  
Epoch 11/20  
15000/15000 [==============================] - 4s 236us/step - loss: 0.0508 - acc: 0.9863 - val\_loss: 0.3904 - val\_acc: 0.8782  
Epoch 12/20  
15000/15000 [==============================] - 4s 235us/step - loss: 0.0396 - acc: 0.9912 - val\_loss: 0.4199 - val\_acc: 0.8742  
Epoch 13/20  
15000/15000 [==============================] - 4s 235us/step - loss: 0.0335 - acc: 0.9933 - val\_loss: 0.4490 - val\_acc: 0.8725  
Epoch 14/20  
15000/15000 [==============================] - 4s 244us/step - loss: 0.0278 - acc: 0.9943 - val\_loss: 0.4748 - val\_acc: 0.8734  
Epoch 15/20  
15000/15000 [==============================] - 4s 235us/step - loss: 0.0239 - acc: 0.9947 - val\_loss: 0.5067 - val\_acc: 0.8704  
Epoch 16/20  
15000/15000 [==============================] - 4s 235us/step - loss: 0.0164 - acc: 0.9979 - val\_loss: 0.5940 - val\_acc: 0.8555  
Epoch 17/20  
15000/15000 [==============================] - 4s 234us/step - loss: 0.0124 - acc: 0.9988 - val\_loss: 0.5650 - val\_acc: 0.8676  
Epoch 18/20  
15000/15000 [==============================] - 4s 235us/step - loss: 0.0112 - acc: 0.9988 - val\_loss: 0.6037 - val\_acc: 0.8653  
Epoch 19/20  
15000/15000 [==============================] - 4s 242us/step - loss: 0.0089 - acc: 0.9992 - val\_loss: 0.6369 - val\_acc: 0.8646  
Epoch 20/20  
15000/15000 [==============================] - 4s 253us/step - loss: 0.0078 - acc: 0.9982 - val\_loss: 0.6863 - val\_acc: 0.8683  
*# 绘制训练损失和验证损失***import** matplotlib.pyplot **as** plt  
history\_dict = history.history  
loss\_values = history\_dict[**'loss'**]  
val\_loss\_values = history\_dict[**'val\_loss'**]  
  
epochs = range(1,len(loss\_values) + 1 )  
  
plt.plot(epochs,loss\_values,**'bo'**,label = **'Training loss'**)  
plt.plot(epochs,val\_loss\_values,**'b'**,label = **'Validation loss'**)  
  
plt.title(**'Training and validation loss'**)  
plt.xlabel(**'Epochs'**)  
plt.ylabel(**'Loss'**)  
plt.legend()  
  
plt.show()

# 绘制训练精度和验证精度

plt.clf() # 清空图像

acc = history\_dict['acc']

val\_acc = history\_dict['val\_acc']

plt.plot(epochs,acc,'bo',label = 'Training acc')

plt.plot(epochs,val\_acc,'b',label = 'Validation acc')

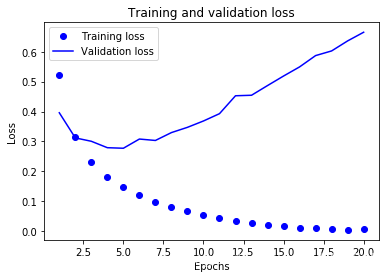
plt.title('Training and validation acc')

plt.xlabel('Epochs')

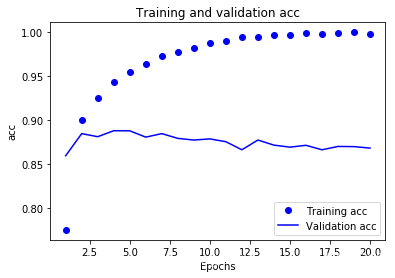
plt.ylabel('acc')

plt.legend()

plt.show()



*# 绘制训练精度和验证精度*plt.clf() *# 清空图像*acc = history\_dict[**'acc'**]  
val\_acc = history\_dict[**'val\_acc'**]  
  
plt.plot(epochs,acc,**'bo'**,label = **'Training acc'**)  
plt.plot(epochs,val\_acc,**'b'**,label = **'Validation acc'**)  
  
plt.title(**'Training and validation acc'**)  
plt.xlabel(**'Epochs'**)  
plt.ylabel(**'acc'**)  
plt.legend()  
  
plt.show()



*# 绘制训练损失和验证损失***import** matplotlib.pyplot **as** plt  
history\_dict = history.history  
loss\_values = history\_dict[**'loss'**]  
val\_loss\_values = history\_dict[**'val\_loss'**]  
  
epochs = range(1,len(loss\_values) + 1 )  
  
plt.plot(epochs,loss\_values,**'bo'**,label = **'Training loss'**)  
plt.plot(epochs,val\_loss\_values,**'b'**,label = **'Validation loss'**)  
  
plt.title(**'Training and validation loss'**)  
plt.xlabel(**'Epochs'**)  
plt.ylabel(**'Loss'**)  
plt.legend()  
  
plt.show()  
*# 绘制训练精度和验证精度*plt.clf() *# 清空图像*acc = history\_dict[**'acc'**]  
val\_acc = history\_dict[**'val\_acc'**]  
  
plt.plot(epochs,acc,**'bo'**,label = **'Training acc'**)  
plt.plot(epochs,val\_acc,**'b'**,label = **'Validation acc'**)  
  
plt.title(**'Training and validation acc'**)  
plt.xlabel(**'Epochs'**)  
plt.ylabel(**'acc'**)  
plt.legend()  
  
plt.show()  
  
  
Epoch 1/4  
25000/25000 [==============================] - 12s 478us/step - loss: 0.4383 - acc: 0.8220  
Epoch 2/4  
25000/25000 [==============================] - 4s 172us/step - loss: 0.2485 - acc: 0.9127  
Epoch 3/4  
25000/25000 [==============================] - 4s 153us/step - loss: 0.1965 - acc: 0.9304  
Epoch 4/4  
25000/25000 [==============================] - 4s 147us/step - loss: 0.1659 - acc: 0.9391  
25000/25000 [==============================] - 6s 259us/step  
[0.30140374687194826, 0.88092]

原始数据集预处理为张量传入神经网络。单词序列编码为二值向量或者其它形式；

一系列带有relu激活函数的Dense layer能解决广泛的问题，包括情感分类，后续会常用到的；

二值分类问题（输出两个类别）中，最后的一个Dense layer带有一个sigmoid激活函数和一个单元：网络输出是0到1之间的标量，代表概率值；

二分类问题中有sigmoid标量输出的，损失函数选择binary\_crossentropy损失函数；

rmsprop优化器对于大部分深度学习模型来说是足够好的选择；

随着在训练集上表现越来越好，神经网络模型开始过拟合，在新数据上表现越来越差。关注验证集上的监控指标

3.新闻分类：多分类问题

在本小节，你将学习构建神经网络，把路透社新闻分为互不相交的46类主题。很明显，这个问题是多分类问题，并且每个数据点都只归为一类，那么该问题属于单标签、多分类；如果每个数据点可以属于多个分类，那么你面对的将是多标签、多分类问题。

路透社新闻数据集是由路透社1986年发布的短新闻和对应主题的集合，它常被用作文本分类的练手数据集。该数据集有46个不同的新闻主题，在训练集中每个主题包含至少10个新闻。

**from** keras.datasets **import** reuters  
(train\_data,train\_labels),(test\_data,test\_labels) = reuters.load\_data(num\_words = 10000)  
len(train\_data)  
train\_data

word\_index = reuters.get\_word\_index()  
*# 将单词和索引对调位置*reverse\_word\_index = dict([(value,key) **for** (key,value) **in** word\_index.items()])  
decoded\_newswire = **''**.join([reverse\_word\_index.get(i-3,**'?'**) **for** i **in** train\_data[0]])  
decoded\_newswire  
**import** numpy **as** np  
  
*# 将训练数据向量化,one-hot编码***def** vectorize\_sequences(sequences,dimension = 10000):  
 results = np.zeros((len(sequences),dimension))  
 **for** i,sequence **in** enumerate(sequences):  
 results[i,sequence] = 1  
 **return** results  
  
x\_train = vectorize\_sequences(train\_data)  
x\_test = vectorize\_sequences(test\_data)  
  
*# 对标签也进行向量化，因为有46个标签，所以进行one-hot编码***def** to\_one\_hot(labels,dimension = 46):  
 results = np.zeros((len(labels),dimension))  
 **for** i,label **in** enumerate(labels):  
 results[i,label] = 1  
 **return** results  
  
one\_hot\_train\_labels = to\_one\_hot(train\_labels)  
one\_hot\_test\_labels = to\_one\_hot(test\_labels)  
print(one\_hot\_train\_labels[0])  
  
*# 注意，keras可以用内置的方法实现这个操作***from** keras.utils.np\_utils **import** to\_categorical  
one\_hot\_train\_labels = to\_categorical(train\_labels)  
one\_hot\_test\_labels = to\_categorical(test\_labels)  
print(one\_hot\_train\_labels[0])

# 注意，keras可以用内置的方法实现这个操作

from keras.utils.np\_utils import to\_categorical

one\_hot\_train\_labels = to\_categorical(train\_labels)

one\_hot\_test\_labels = to\_categorical(test\_labels)

print(one\_hot\_train\_labels[0])

