2018年12月19日

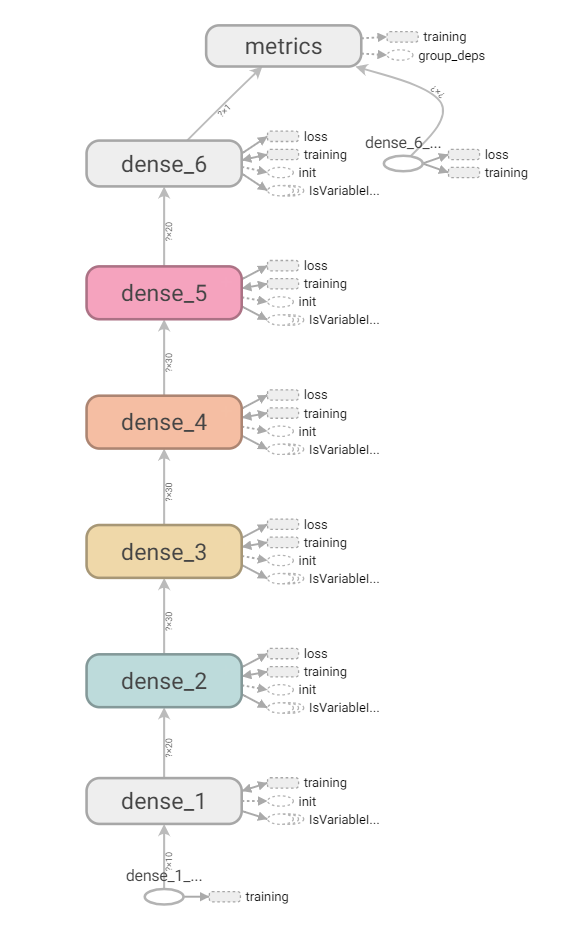
实验三任务说明：

实验内容如下：对给定数据集进行分类，并建立相应的分类器。分析分类结果指标，改变参数并比较不同的实验结果，给出你认为的最佳模型，并说明理由。

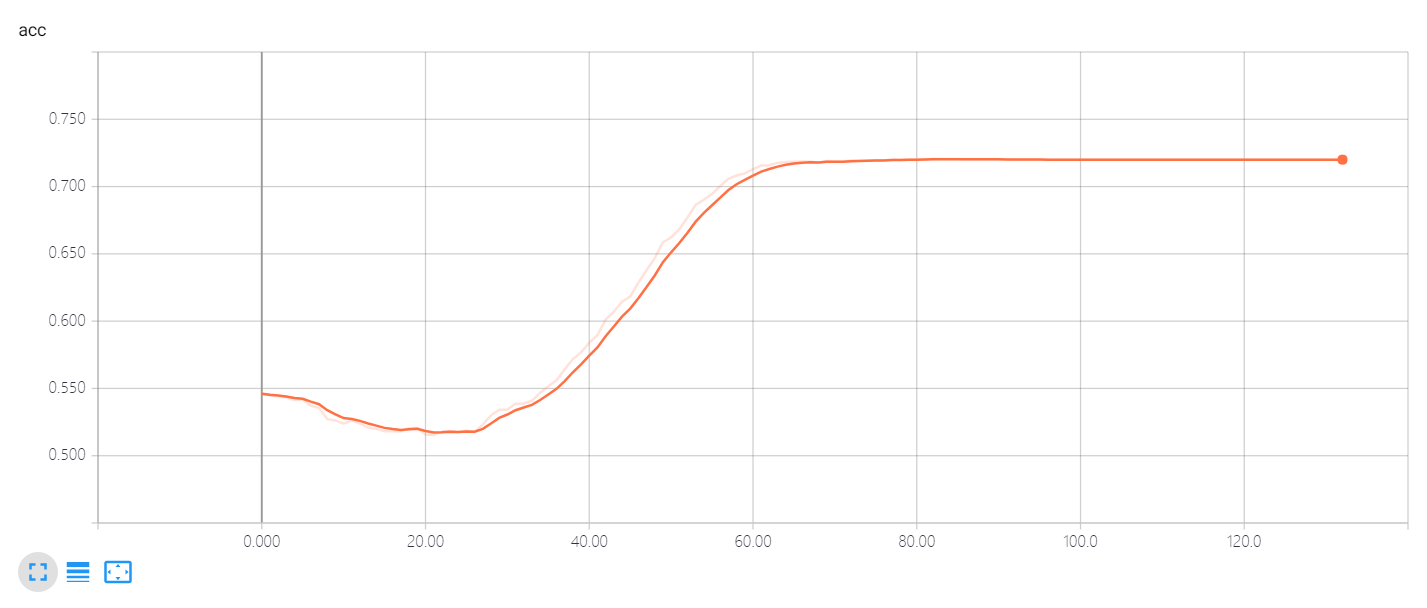
本次实验使用keras搭建二分类神经网络分类器.

建立了6层神经网络

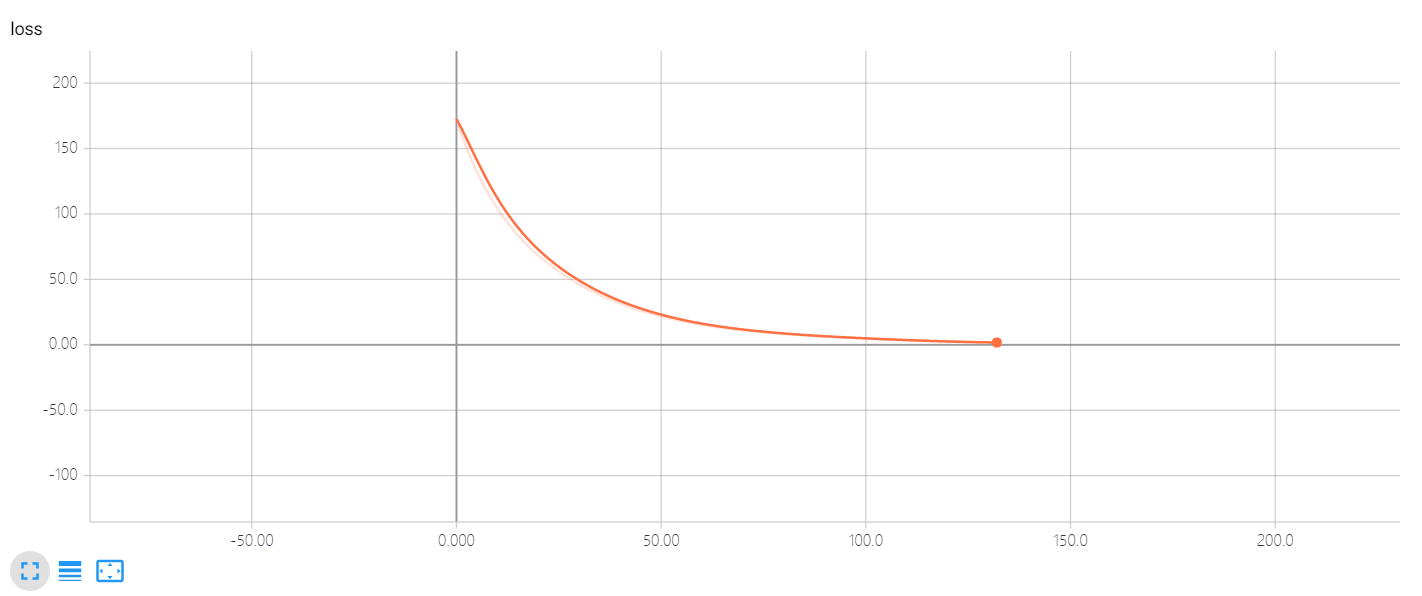
如下图所示:



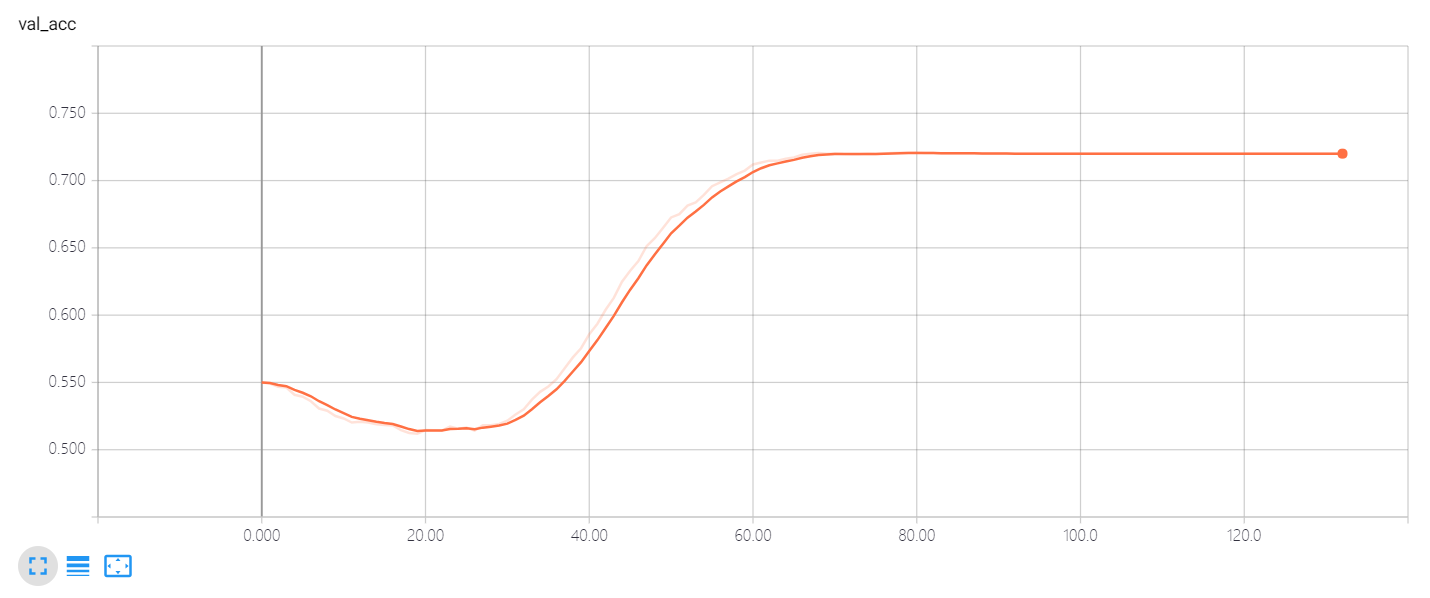
实验在训练集准确率:



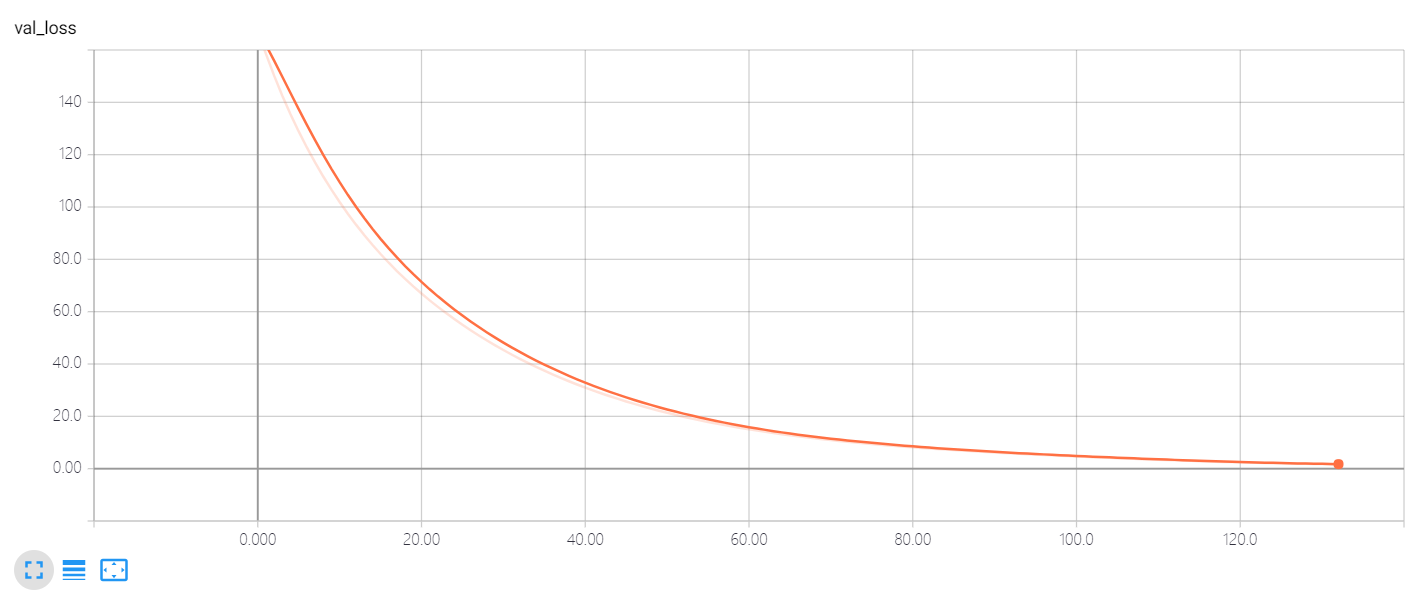
实验在训练集损失值:



实验在验证集准确率:



实验在验证集上损失值:



为了避免过拟合,我们通过观察验证集的损失函数作为我们训练的依据.我们的目标就是使损失函数最小.

我们将神经网络输出与类别进行交叉熵计算,作为损失函数,函数形式如下所示,

其中a为标签,y为神经网络输出值,如上图所示,在迭代120次后,验证集的损失值达到最小为1.6. 同时在验证集的准确率为72%并且不再发生变化.

通过改变神经网络每一层的神经元数量,以及神经网络的层数,发现层数越高准确率越高,但是存在瓶颈. 神经元的数量并不太大影响.

通过给每一层增加正则化方法提高了准确率.

为了避免过拟合,使用了earlyStopping方法,在验证集分类准确度下降时,终止在训练集上训练.

实验代码如下:

神经网络代码:

*from* keras *import* models  
*from* keras.layers *import* Dense  
*from* keras.optimizers *import* Adam  
*from* keras.regularizers *import* l2, l1  
  
  
*class* Model():  
 *def \_\_init\_\_*(self,*input\_dim*=10):  
 self.input\_dim=*input\_dim* self.model=self.model()  
 # Now compile the network.  
 optimizer = Adam(lr=1e-5, decay=1e-6)  
 metrics = ['accuracy']  
 self.model.compile(loss='binary\_crossentropy', optimizer=optimizer, metrics=metrics)  
  
 *def* model(self):  
 model = models.Sequential()  
 model.add(Dense(20, activation='relu', input\_dim=self.input\_dim))  
 model.add(Dense(30, activation='relu',  
 kernel\_regularizer=l2(0.01),  
 activity\_regularizer=l1(0.01),  
 bias\_regularizer=l2(0.001)))  
 model.add(Dense(30, activation='relu',  
 kernel\_regularizer=l2(0.01),  
 activity\_regularizer=l1(0.01),  
 bias\_regularizer=l2(0.001)))  
 model.add(Dense(30, activation='relu',  
 kernel\_regularizer=l2(0.01),  
 activity\_regularizer=l1(0.01),  
 bias\_regularizer=l2(0.001)))  
 model.add(Dense(20, activation='relu',  
 kernel\_regularizer=l2(0.01),  
 activity\_regularizer=l1(0.01),  
 bias\_regularizer=l2(0.001)))  
 model.add(Dense(1, activation='sigmoid'))  
 *return* model

数据获取代码(在实验开始时,将数据保存为csv格式):

*import* csv  
*from* time *import* sleep  
*import* numpy *as* np  
*import* random  
*import* pandas *as* pd  
*import* threading  
*class* threadsafe\_iterator:  
 *def \_\_init\_\_*(self, *iterator*):  
 self.iterator = *iterator* self.lock = threading.Lock()  
 *def \_\_iter\_\_*(self):  
 *return* self  
 *def \_\_next\_\_*(self):  
 *with* self.lock:  
 *return* next(self.iterator)  
*def* threadsafe\_generator(*func*):  
 """Decorator"""  
 *def* gen(*\*a*, *\*\*kw*):  
 *return* threadsafe\_iterator(func(\**a*, \*\**kw*))  
 *return* gen  
  
*class* DataSet():  
 *def* get\_memory(self,*train\_test*):  
 data\_set = pd.read\_csv(f"data/{*train\_test*}\_data.csv").values# 加载数据集  
 X = data\_set[:, 0:10].astype(np.int16) # 分割为10个输入变量  
 Y = data\_set[:, 10].astype(np.int16)  
 *return* X,Y  
 @threadsafe\_generator  
 *def* generator(self, *batch\_size*, *train\_test*):  
 data\_set = pd.read\_csv(f"data/{*train\_test*}\_data.csv").values# 加载数据集  
 X = data\_set[:, 0:10].astype(np.int16) # 分割为10个输入变量  
 Y = data\_set[:, 10].astype(np.int16)  
 print("Creating %s generator with %d samples." % (*train\_test*, len(data\_set)))  
  
 *while* 1:  
 X, y = [], []  
 # Generate batch\_size samples.  
 *for* \_ *in* range(*batch\_size*):  
 sample = random.choice(data\_set)  
 X.append(sample[0:10].astype(np.int16))  
 y.append(sample[10].astype(np.int16))  
 *yield* np.array(X), np.array(y)  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 ds=DataSet()  
 *for* x,y *in* ds.generator(32,'train'):  
 print(x)  
 print(y)  
 sleep(5)

程序入口如下所示:

*from* keras.callbacks *import* TensorBoard, ModelCheckpoint, EarlyStopping, CSVLogger  
*from* models *import* Model  
*from* data *import* DataSet  
*import* time  
*import* os.path  
*import* os  
  
*def* train(saved\_model=*None*,  
 *load\_to\_memory*=*False*,  
 *batch\_size*=32,  
 *nb\_epoch*=100):  
 # Helper: Save the model.  
 *if not* os.path.exists(os.path.join('data', 'checkpoints')):  
 os.makedirs(os.path.join('data', 'checkpoints'))  
 *if not* os.path.exists(os.path.join('data', 'logs')):  
 os.makedirs(os.path.join('data', 'logs'))  
 *if not* os.path.exists(os.path.join('data', 'checkpoints')):  
 os.makedirs(os.path.join('data', 'checkpoints'))  
 check\_pointer = ModelCheckpoint(  
 filepath=os.path.join('data', 'checkpoints','val\_loss-{val\_loss:.3f}\_val\_acc-{val\_acc:3f}.hdf5'),  
 verbose=1,  
 save\_best\_only=*True*)  
 # Helper: TensorBoard  
 tb = TensorBoard(log\_dir=os.path.join('data', 'logs'))  
 # Helper: Stop when we stop learning.  
 early\_stopper = EarlyStopping(patience=2)  
 # Helper: Save results.  
 timestamp = time.time()  
 csv\_logger = CSVLogger(os.path.join('data', 'logs','training-' + str(timestamp) + '.log'))  
  
 # Get the data and process it.  
  
 data = DataSet()  
 # X,Y,X\_test,Y\_test,generator,val\_generator=None  
 *if load\_to\_memory*:  
 # Get data.  
 X, Y = data.get\_memory('train')  
 X\_test, Y\_test = data.get\_memory('test')  
 *else*:  
 # Get generators.  
 generator = data.generator(*batch\_size*, 'train')  
 val\_generator = data.generator(*batch\_size*, 'test')  
 # Get the model.  
 model=Model(10).model  
 # Fit!  
 *if load\_to\_memory*:  
 # Use standard fit.  
 model.fit(  
 X,  
 Y,  
 batch\_size=*batch\_size*,  
 validation\_data=(X\_test, Y\_test),  
 verbose=1,  
 callbacks=[tb, early\_stopper, csv\_logger, check\_pointer],  
 epochs=*nb\_epoch*)  
 *else*:  
 # Use fit generator.  
 steps\_per\_epoch = 3500 // *batch\_size* model.fit\_generator(  
 generator=generator,  
 steps\_per\_epoch=steps\_per\_epoch,  
 verbose=1,  
 callbacks=[tb, early\_stopper, csv\_logger, check\_pointer],  
 validation\_data=val\_generator,  
 validation\_steps=40,  
 workers=10,  
 epochs=*nb\_epoch*)  
*def* main():  
 saved\_model = *None* load\_to\_memory = *True* # pre-load the sequences into memory  
 batch\_size = 32  
 nb\_epoch = 1000 # 一般使用early\_stopper = EarlyStopping(patience=10)提前终止运行  
 train(saved\_model=saved\_model,  
 load\_to\_memory=load\_to\_memory,  
 batch\_size=batch\_size,  
 nb\_epoch=nb\_epoch)  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 main()