机器学习实验报告 (五)

2018/12/23

目录

1 实验环境

操作系统	win10
编程语言	python3
编程环境	Jupyter Notebook
报告编写	latex

2 问题描述与分析

问题描述：通过支持向量机、神经网络、*knn* 方法来测试 *MNIST* 手写数字训练集, 并对不同模型做性能对比。

2.1 多分类问题

一个样本属于且只属于多个类中的一个，一个样本只能属于一个类，不同类之间是互斥的。有两种方法，

（1）将多类问题分成 N 个二类分类问题，训练 N 个二类分类器，对第 i 个类来说，所有属于第 i 个类的样本为正样本，其他样本为负样本，每个二类分类器将属于 i 类的样本从其他类中分离出来。

（2）训练出 N(N-1) 个二类分类器，每个分类器区分一对类 (i,j)。

2.2 MNIST 数据集

MNIST 是一个简单的视觉计算数据集，其中包括类似如下的手写数字图片信息（60000 张训练集与 10000 张测试集）:

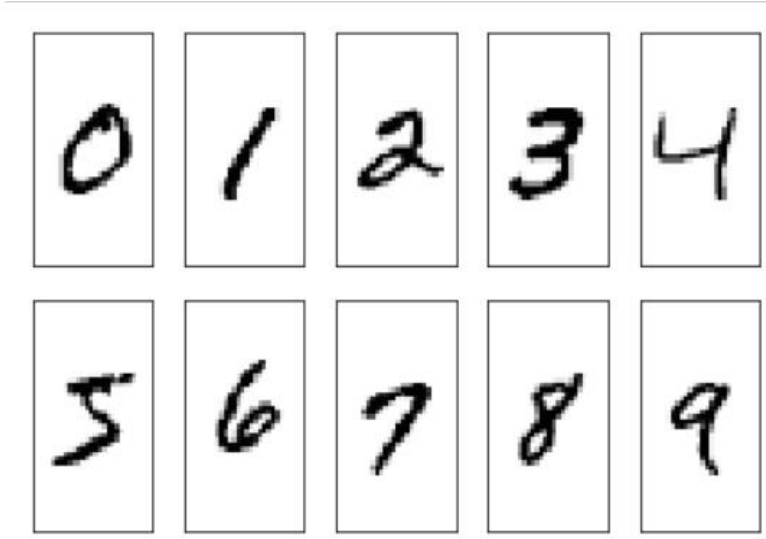

图 1　MNIST 数据集示例

每张图片都是 28*28 大小，除了包含的像素信息以外，每张图片还有一个标签标识所代表的数字。

2.3　支持向量机

支持向量机的基本模型是在特征空间上找到最佳的分离超平面使得训练集上正负样本间隔最大，一般有三种形式：硬间隔支持向量机、软间隔支持向量机、非线性支持向量机。其中非线性支持向量机采用核方法，不同的核方法所购造出来的支持向量机得性能也有很大的差别。

本次实验使用 LIBSVM（台湾大学林智仁教授等开发设计的一个简单、易于使用和快速有效的 SVM 模式识别与回归的软件包）使用不同核函数进行性能对比

2.4　神经网络

BP 神经网络，是 1986 年由 Rumelhart 和 McClelland 为首的科学家提出的概念，是一种按照误差逆向传播算法训练的多层前馈神经网络。BP

算法是一种有效的多层神经网络学习方法，通过向信号前向输出层传递，误差向输入层反馈，不断调节网络权重值，使得网络的最终输出与期望输出尽可能接近，以达到训练的目的。一般来说，一个多层神经网络由输入层，若干隐含层，输出层构成。

本次试验通过学习调用 tensorflow 库中神经网络来进行相应性能对比

2.5 K 近邻

KNN 是通过测量不同特征值之间的距离进行分类。主要思路为: 如果一个样本在特征空间中的 k 个最相似的样本中的大多数属于某一个类别，则该样本也属于这个类别，其中 K 通常是不大于 20 的整数。

在 KNN 中，通过计算对象间距离来作为各个对象之间的非相似性指标，避免了对象之间的匹配问题同时，KNN 通过依据 k 个对象中占优的类别进行决策，而不是单一的对象类别决策。这两点就是 KNN 算法的优势。

KNN 算法的结果很大程度取决于 K 的选择与距离的度量。因此此次实验选择调用 sklearn 库对不同 k 的取值进行性能对比

3 具体过程

3.1 MNIST 数据集的导入

在导入所需要的库依赖之后，我们首先要做的就是讲 MNIST 数据集读入。通过参考网上相关资料，我在这定义了 $load_mnist\_train(path)$ 和 $load\_mnist\_test(path)$ 两个方法，分别用于加载训练集与测试集

```
# 加载数据
def load_mnist_train(path):
    labels_path = os.path.join(path,'train-labels.idx1-ubyte')
    images_path = os.path.join(path, 'train-images.idx3-ubyte')
    with open(labels_path, 'rb') as lbpath:
```

```
6           magic, n = struct.unpack('>II',lbpath.read(8))
7           labels = np.fromfile(lbpath, dtype=np.uint8)
8       with open(images_path, 'rb') as imgpath:
9           magic, num, rows, cols = struct.unpack('>IIII',
10          imgpath.read(16))
11          images = np.fromfile(imgpath,
12          dtype=np.uint8).reshape(len(labels), 784)
13      return images, labels
14
15  def load_mnist_test(path):
16      labels_path = os.path.join(path,'t10k-labels.idx1-ubyte')
17      images_path = os.path.join(path, 't10k-images.idx3-ubyte')
18      with open(labels_path, 'rb') as lbpath:
19          magic, n = struct.unpack('>II',lbpath.read(8))
20          labels = np.fromfile(lbpath, dtype=np.uint8)
21      with open(images_path, 'rb') as imgpath:
22          magic, num, rows, cols = struct.unpack('>IIII',
23          imgpath.read(16))
24          images = np.fromfile(imgpath,
25          dtype=np.uint8).reshape(len(labels), 784)
26      return images, labels
27
28  X_train,Y_train = load_mnist_train('E:\学习\大三\机器学习\MNIST')
29  X_test,Y_test = load_mnist_test('E:\学习\大三\机器学习\MNIST')
```

3.2 支持向量机

libSVM 支持多类分类问题，当有 k 个待分类问题时，libSVM 构建 k*（k-1）/2 种分类模型来进行分类，即：libSVM 采用一对一的方式来构建多

类分类器

LibSVM 在训练和预测过程中需要一系列参数来调整控制，下面给出一些主要的参数：

核函数 -t

0	linear(线性核)	u'*v
1	polynomial(多项式核)	$(gamma * u' * v + coef0)^degree$
2	radial basis function(RBF, 径向基核/高斯核)	$exp(-gamma * \|u - v\|^2)$
3	sigmoid(S 型核)	$tanh(gamma * u' * v + coef0)$
4	precomputed kernel(预计算核)	核矩阵存储在 $training\_set\_file$ 中

调整 SVM 或核函数中参数的选项：

- -d 调整核函数的 degree 参数，默认为 3
- -g 调整核函数的 gamma 参数，默认为 $1/num_features$
- -r 调整核函数的 coef0 参数，默认为 0
- -c 调整 C-SVC, epsilon-SVR 和 nu-SVR 中的 Cost 参数，默认为 1
- -n 调整 nu-SVC, one-class SVM 和 nu-SVR 中的错误率 nu 参数，默认为 0.5
- -p 调整 epsilon-SVR 的 loss function 中的 epsilon 参数，默认 0.1
- -m 调整内缓冲区大小, 以 MB 为单位，默认 100
- -e 调整终止判据，默认 0.001
- -wi 调整 C-SVC 中第 i 个特征的 Cost 参数

调整算法功能的选项：

- -b 是否估算正确概率, 取值 0 - 1，默认为 0

- -h 是否使用收缩启发式算法 (shrinking heuristics), 取值 0 - 1，默认为 0
- -v 交叉校验
- -q 静默模式

在具体测试时，我们先将 svm 及 svmutil 相应内容导入，并准备好数据

```
import sys
path = 'D:\LIBSVM\libsvm-3.22\python'
sys.path.append(path)
from svmutil import *
from svm  import *

y_train_svm = Y_train.tolist()
x_train_svm=[]
index = range(1,785)
for i in X_train:
    x_train_svm.append(dict(zip(index, i)))

y_test_svm = Y_test.tolist()
x_test_svm=[]
index = range(1,785)
for i in X_test:
    x_test_svm.append(dict(zip(index, i)))
```

我们将测试核函数分别采取线性核、多项式核、RBF 核时的性能。

编写 $test\_svm(nums, typet)$ 函数，便于测试在不同训练集下时间与准确率的对比，其中 $nums$ 为训练样本数，$typet$ 为核函数的选择（0 代表线性核，1 代表多项式核，2 代表 RBF 核，3 代表 S 型核），返回值分别为准确度，训练时间，总运行时间

```
def test_svm(nums,typet):
    start =time.clock()
    prob  = svm_problem(y_train_svm[:nums], x_train_svm[:nums])
    param = svm_parameter('-t%d-c4-m1024'%typet)
    model = svm_train(prob, param)
    end1 = time.clock()
    print('trainingTime: %s Seconds'%(end1-start))
    p_label, p_acc, p_val = svm_predict(y_test_svm[:1000],
                                x_test_svm[:1000], model)
    end = time.clock()
    print('Running time: %s Seconds'%(end-start))
    return p_acc[0],end1-start,end-start
```

编写 $check\_time(types)$ 函数，便于之后的性能比对

```
def check_time(types):
    ac=[]
    train_time=[]
    total_time=[]
    for i in range(100):
        a,b,c = test_svm((i+1)*100,types)
        ac.append(a)
        train_time.append(b)
        total_time.append(c)
    return ac,train_time,total_time
```

测试线性核函数与多项式核函数：

```
ac,train_time,total_time = check_time(0)
ac1,train_time1,total_time1 = check_time(1)
```

将结果整理为折线图如下：

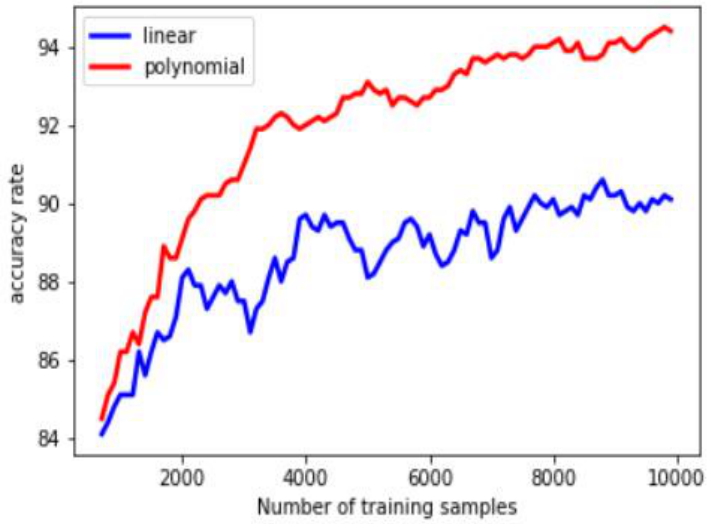

图 1　svm 线性核与多项式核准确率对比

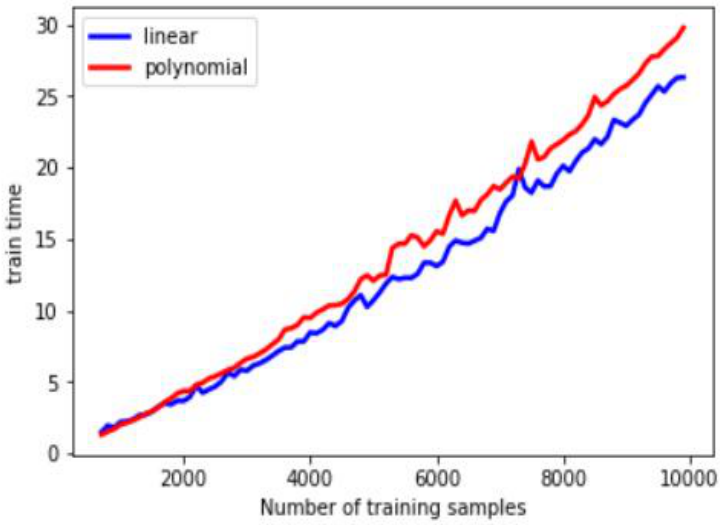

图 1　svm 线性核与多项式核训练时间对比

可以很清晰的看出，在此样例集中，多项式核的效果比线性核的效果要好，相反的，训练时间也比线性核长一些。在训练 2000 张左右的准确率就达到了 90% 左右，学习速度很快

通常而言，RBF 核是比较好的选择。这个核函数将样本非线性地映射到一个更高维的空间，与线性核不同，它能够处理分类标注和属性的非线性关系。测试 RBF 核

```
start =time.clock()
prob_3  = svm_problem(y_train_svm[:10000], x_train_svm[:10000])
param_3 = svm_parameter('-t␣2␣-c␣4␣-b␣1')
model_3 = svm_train(prob_3, param_3)
p_label_3, p_acc_3, p_val_3 = svm_predict(y_test_svm[:1000],
                  x_test_svm[:1000], model_3)
end = time.clock()
print('Running␣time:␣%s␣Seconds'%(end-start))
```

得到如下结果

Model supports probability estimates, but disabled in predicton.

Accuracy = 12.6% (126/1000) (classification)

Running time: 941.7631710461183 Seconds

我们发现 RBF 核似乎在本次训练中的效果并不好。通过查询资料发现，特征维数比价大的时候，很可能 RBF 核的性能要差与线性核与多项式核。

此外，SVM 模型有两个非常重要的参数 C 与 gamma。其中 C 是惩罚系数，即对误差的宽容度。c 越高，说明越不能容忍出现误差, 容易过拟合。C 越小，容易欠拟合。C 过大或过小，泛化能力变差。gamma 是选择 RBF 函数作为 kernel 后，该函数自带的一个参数。隐含地决定了数据映射到新的特征空间后的分布，gamma 越大，支持向量越少，gamma 值越小，支持向量越多。支持向量的个数影响训练与预测的速度。

3.3 神经网络

TensorFlow 是 Google 开源的一款人工智能学习系统，是一种计算图模型，即用图的形式来表示运算过程的一种模型。Tensor 是 Tensorflow 中

最重要的数据结构，用来表示 Tensorflow 程序中的所有数据，Tensor 理解成 N 维矩阵（N 维数组）。其中零维张量表示的是一个标量，也就是一个数；一维张量表示的是一个向量，也可以看作是一个一维数组；二维张量表示的是一个矩阵；同理，N 维张量也就是 N 维矩阵。

我们将通过 *tensorflow* 来实现多层神经网络。首先导入数据

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
# 自动获取MNIST的数据集
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

以下是三层网络实现代码

```
inputSize  = 784
outputSize = 10
hiddenSize = 89
batchSize  = 1
trainCycle = 60000

# 输入层
inputLayer = tf.placeholder(tf.float32, shape=[None, inputSize])

# 隐藏层
hiddenWeight = tf.Variable(tf.truncated_normal([inputSize,
                          hiddenSize], mean=0, stddev=0.1))
hiddenBias   = tf.Variable(tf.truncated_normal([hiddenSize]))
hiddenLayer  = tf.add(tf.matmul(inputLayer, hiddenWeight),
                                          hiddenBias)
hiddenLayer  = tf.nn.sigmoid(hiddenLayer)

# 输出层
```

```
19 outputWeight = tf.Variable(tf.truncated_normal([hiddenSize,
20                           outputSize], mean=0, stddev=0.1))
21 outputBias   = tf.Variable(tf.truncated_normal([outputSize],
22                                    mean=0, stddev=0.1))
23 outputLayer  = tf.add(tf.matmul(hiddenLayer,
24                                     outputWeight), outputBias)
25 outputLayer  = tf.nn.sigmoid(outputLayer)
26 # 标签
27 outputLabel = tf.placeholder(tf.float32, shape=[None, outputSize])
28 # 损失函数
29 loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
30                          labels=outputLabel, logits=outputLayer))
31 # 优化器
32 optimizer = tf.train.AdamOptimizer()
33 # 训练目标
34 target = optimizer.minimize(loss)
35 # 训练
36 accuracy_need=[]
37
38 with tf.Session() as sess:
39     sess.run(tf.global_variables_initializer())
40     for i in range(trainCycle):
41         batch = mnist.train.next_batch(batchSize)
42         sess.run(target,
43         feed_dict={inputLayer: batch[0], outputLabel: batch[1]})
44         if i%(100)==0 and i!=0:
45             corrected = tf.equal(tf.argmax(outputLabel, 1),
46             tf.argmax(outputLayer, 1))
47             accuracy  = tf.reduce_mean(tf.cast(corrected,
```

```
48                                  tf.float32))
49              accuracyValue = sess.run(accuracy,
50              feed_dict={inputLayer: mnist.test.images,
51               outputLabel: mnist.test.labels})
52              print("第",i,"张","accuracy␣on␣test␣set:", accuracyValue)
53              accuracy_need.append(accuracyValue)
54      # 测试
55      corrected = tf.equal(tf.argmax(outputLabel, 1),
56       tf.argmax(outputLayer, 1))
57      accuracy  = tf.reduce_mean(tf.cast(corrected,
58      tf.float32))
59      accuracyValue = sess.run(accuracy,
60      feed_dict={inputLayer: mnist.test.images,
61      outputLabel: mnist.test.labels})
62      print("最终␣accuracy␣on␣test␣set:", accuracyValue)
63      accuracy_need.append(accuracyValue)
64      sess.close()
```

我们分别测试两层、三层、四层网络，得到如下对比图：

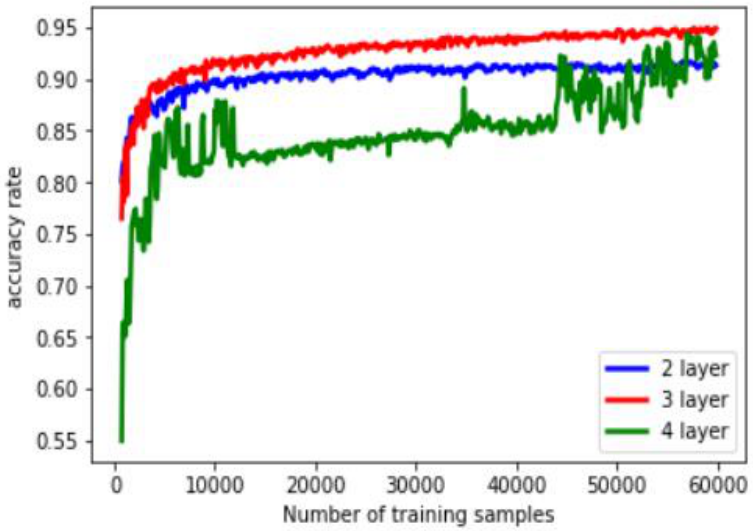

图 1　bp 神经网络层次不同准确度对比

可以看出，在此次试验中，三层网络的效果好于两层网络与四层网络，方便起见，我们此次神经网络模型就选择该三层模型。经过进一步统计，训练 60000 轮总用时 46.688996714275504 s，准确率可以达到 0.9389。

3.4　KNN

在 scikit-learn 中，与近邻法这一大类相关的类库都在 sklearn.neighbors 包之中。KNN 分类树的类是 KNeighborsClassifier，KNN 回归树的类是 KNeighborsRegressor。除此之外，还有 KNN 的扩展，即限定半径最近邻分类树的类 RadiusNeighborsClassifier 和限定半径最近邻回归树的类 RadiusNeighborsRegressor，以及最近质心分类算法 NearestCentroid。

下面我们借助 *sklearn.neighbors* 来测试 KNN 方法。首先导入相关依赖

```
from sklearn.neighbors import KNeighborsClassifier
```

编写如下代码，查看当 k 不同取值时，准确率的变化

```
acc_rate = []
```

```
t_time = []
for i in range(10):
    start =time.clock()
    kNN_classifier_3 = KNeighborsClassifier(n_neighbors=i+1)
    kNN_classifier_3.fit(X_train, Y_train)
    end1 =time.clock()
    t_time.append(end1-start)
#     print("当k=",i+1,"时, 训练所用时间",end1-start,"s")
    y_predict_3 = kNN_classifier_3.predict(X_test[:1000])
    end2 =time.clock()
    acc_rate.append(sum(y_predict_3 ==
                Y_test[:1000])/len(Y_test[:1000]))
```

整理得到如下结果

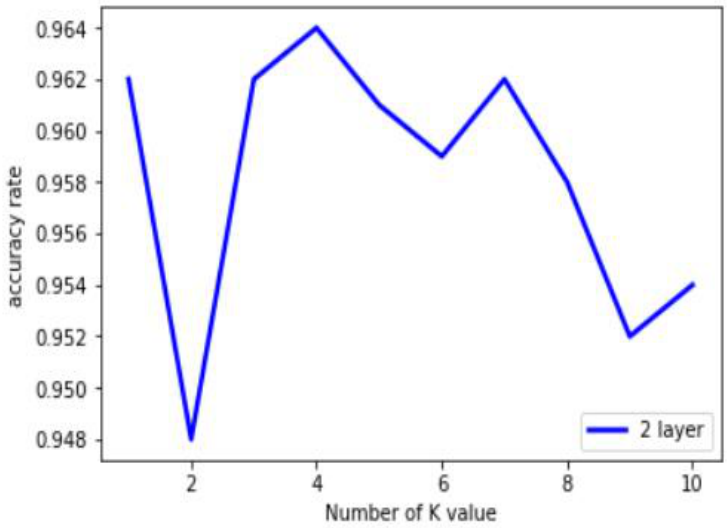

图 1　K 取值不同准确度对比

可以发现，不同的 k 值差别不大，在 0.955 左右波动，当 k=4 时，可以取到最大准确率。

3.5 综合对比

在此我们对带有多项式核的 svm，三层神经网络，以及 k=4 时的 KNN 方法进行综合比较。在取不同训练样本数目时，三种模型准确率

准确率 模型 / 样本数	多项式核 svm	三层网络 (1 轮)	KNN(K=4)	三层网络 (10 轮)
100	0.521	0.245	0.595	0.758
200	0.647	0.453	0.657	0.805
500	0.782	0.702	0.754	0.879
1000	0.854	0.697	0.824	0.898
2000	0.886	0.794	0.863	0.918
5000	0.928	0.871	0.908	0.936
10000	0.944	0.893	0.913	0.953
20000	0.955	0.91	0.934	0.964

可以看到，在样本量较小时，三层网络在训练次数少时的准确率并不理想，而 svm 及 KNN 均能达到 0.5 以上的水平，但三层网络训练十轮便能达到 0.75 的水平。随着样本数量的提高，不同模型的准确度均有提高，就一般来说，多层神经网络最能到达的极限最高。

从训练时间的角度，KNN 属于 Lazy learning，只有到了需要决策时才会利用已有数据进行决策，需要的存储空间比较大，决策过程比较慢；而神经网络与 SVM 属于 Eager learning ，先利用训练数据进行训练得到一个目标函数，待需要时就只利用训练好的函数进行决策，因此耗费训练时间较长，可是它的决策时间基本为 0。

没有最优的算法, 最优只取决于实际问题。模型的效果一方面取决于真实的数据结构, 另一方面也取决于模型的参数的选择。

4 总结与收获

通过这次实验，我对 Lazy learning 与 Eager learning 的方式有了更加直观的了解，学习了 tensorflow、sklearn 和 libsvm 的简单使用。认识到了工具的重要性，以及模型参数的重要性