**人工智能课程总结报告**

王湘博2015201979

这学期选修的胡鹤老师的人工智能课程，学习了时下非常热门的python的scikit-learn包的使用和机器学习框架TensorFlow的使用，但并非浅显停留于函数方法的调用，而是深入原理，理解其数学本质。虽然有些地方比较深奥难懂，但经过课后的资料查询，尽力理解，从这门课程中学到了很多干货。并且通过把课上笔记和课后资料查询整理后，几乎每次课后都写了一篇博客，以【机器学习】为系列连载到博客园网站中，以这种记博客的方法，及时梳理所学内容，勉励自己始终不松懈，不断学习。博客地址是<http://www.cnblogs.com/rucwxb/>。

**（一）回归分析、过拟合、分类**

一、Linear Regression

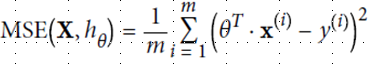
线性回归是相对简单的一种，表达式如下

http://images2017.cnblogs.com/blog/1214565/201710/1214565-20171023084213801-387430459.png

其中，θ0表示bias，其他可以看做weight，可以转换为如下形式

http://images2017.cnblogs.com/blog/1214565/201710/1214565-20171023084016957-1298670442.png

为了更好回归，定义损失函数，并尽量缩小这个函数值，使用MSE方法（mean square equal）



缩小方法采用梯度下降法，即不断地向现在站立的山坡往下走，走的速度就是学习速率η（learning rate），太小耗尽计算资源，太大走过了山谷。

（1）Normal Equation

http://images2017.cnblogs.com/blog/1214565/201710/1214565-20171023084526879-302485737.png

[复制代码](javascript:void(0);)

1 from sklearn.linear\_model import LinearRegression

2 import numpy as np

3 import matplotlib.pyplot as plt

4

5 # 数据集

6 X = 2\*np.random.rand(100, 1)

7 y = 4+3\*X+np.random.randn(100,1)

8

9 # X每个元素加1

10 X\_b = np.c\_[np.ones((100,1)), X]

11 theta\_best = np.linalg.inv(X\_b.T.dot(X\_b)).dot(X\_b.T).dot(y)

12

13 # 训练

14 lin\_reg = LinearRegression()

15 lin\_reg.fit(X, y)

16 print(lin\_reg.intercept\_, lin\_reg.coef\_)

17

18 # 测试数据

19 X\_new = np.array([[0],[2]])

20 X\_new\_b = np.c\_[np.ones((2,1)), X\_new]

21 y\_predict = X\_new\_b.dot(theta\_best)

22 print(y\_predict)

23

24 # 画图

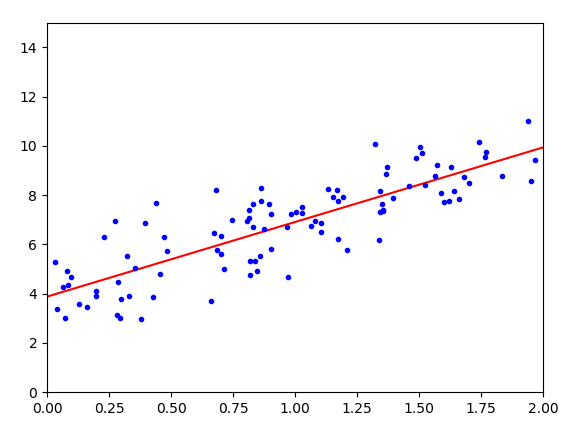
25 plt.plot(X\_new, y\_predict, "r-")

26 plt.plot(X, y, "b.")

27 plt.axis([0,2,0,15])

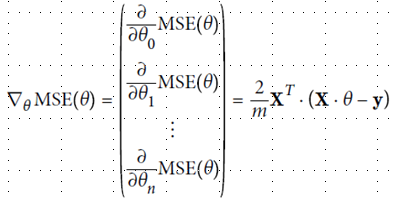
28 plt.show()

[复制代码](javascript:void(0);)



（2）Batch Gradient Descent

　　基本算是遍历了所有数据，不适用于数据规模大的数据



[复制代码](javascript:void(0);)

1 # BGD梯度下降

2 eta = 0.1

3 n\_iterations = 1000

4 m = 100

5 theta = np.random.randn(2,1)

6 for iteration in range(n\_iterations):

7 gradients = 2/m \* X\_b.T.dot(X\_b.dot(theta) - y)

8 theta = theta - eta\*gradients

9 print(theta)

[复制代码](javascript:void(0);)



可以看出，结果是差不多的

（3）Stochastic Gradient Descent

　　可以避免局部最优结果，但是会震来震去。为了防止这种震荡，让学习速率η不断减小（类似模拟退火）

[复制代码](javascript:void(0);)

# SGD梯度下降

m = 100

n\_epochs = 50

t0, t1 = 5, 50 # η初始值0.1

def learning\_schedule(t):

return t0 / (t + t1)

theta = np.random.randn(2,1) # random initialization

for epoch in range(n\_epochs):

for i in range(m):

random\_index = np.random.randint(m)

xi = X\_b[random\_index:random\_index+1]

yi = y[random\_index:random\_index+1]

gradients = 2 \* xi.T.dot(xi.dot(theta) - yi)

eta = learning\_schedule(epoch \* m + i)

theta = theta - eta \* gradients

print(theta)

# sklearn 提供了SGDRegressor的方法

from sklearn.linear\_model import SGDRegressor

sgd\_reg = SGDRegressor(max\_iter=50, penalty=None, eta0=0.1)

sgd\_reg.fit(X, y.ravel())

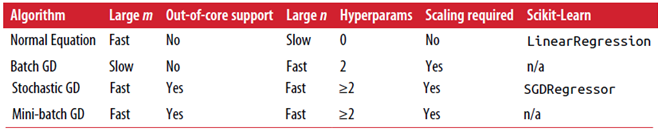
print(sgd\_reg.intercept\_, sgd\_reg.coef\_)

[复制代码](javascript:void(0);)

（4）Min-batch Gradient Descent

　　使用小批随机数据，结合SGD与BGD优点

以下是各种方法对比



二、Polynomial Regression

但有的时候，y本身是由x取平方所得，无法找出来一条合适的线性回归线来拟合数据，该怎么办呢？

我们可以尝试将x取平方，取3次方等方法，多加尝试

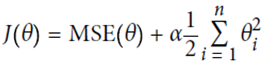
三、误差分析

四、防止过拟合

用惩罚系数（penalty），即正则项（regularize the model）

（1）岭回归ridge regression

　　控制参数自由度，减少模型复杂度。所控制的α=αhttp://images2017.cnblogs.com/blog/1214565/201710/1214565-20171023101157207-1443913606.png，越大控制结果越强

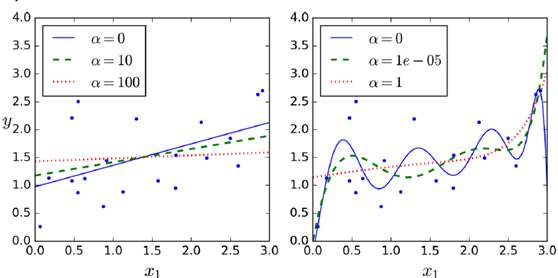


　优势：直接用公式可以计算出结果

1 from sklearn.linear\_model import Ridge

2 ridge\_reg = Ridge(alpha=1, solver="cholesky")

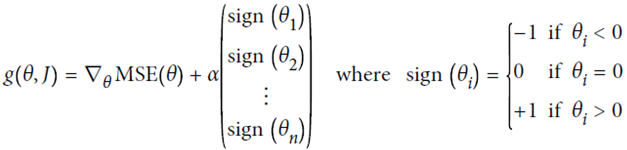
3 ridge\_reg.fit(X, y)



(2)Lasso Regression（least absolute shrinkage and selection operator regression）

正则化项同ridge regression不同，正则化的控制更强

http://images2017.cnblogs.com/blog/1214565/201710/1214565-20171023100504051-1145677898.png



1 # Lasso Regression

2 from sklearn.linear\_model import Lasso

3 lasso\_reg = Lasso(alpha=0.1)

4 lasso\_reg.fit(X, y)

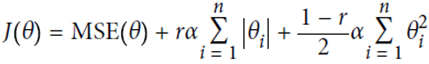
5 lasso\_reg.predict([[1.5]])

对于高阶degree regularize尤为明显，是一个sparse model，很多高阶参数项成为了0



 （3）Elastic Net（一般推荐使用）

 相当于ridge和lasso regression的结合，用超参数r来控制其平衡



1 # Elastic Net

2 from sklearn.linear\_model import ElasticNet

3 elastic\_net = ElasticNet(alpha=0.1, l1\_ratio=0.5)

4 elastic\_net.fit(X, y)

（4）Early Stopping

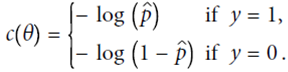
找到开始上升的点，从那里停止（整体找到，取最优的）

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

五、Logistic Regression（可用作分类）

http://images2017.cnblogs.com/blog/1214565/201710/1214565-20171023102106269-1536687157.pnghttp://images2017.cnblogs.com/blog/1214565/201710/1214565-20171023102119426-267487926.png（使用sigmod函数，y在（0,1）之间）

定义cost function，由于p在（0,1）之间，故最前面加一个符号，保证代价始终为正的。p值越大，整体cost越小，预测的越对



即http://images2017.cnblogs.com/blog/1214565/201710/1214565-20171023102438285-488682845.png

不存在解析解，故用偏导数计算

以Iris花的种类划分为例

[复制代码](javascript:void(0);)

1 import matplotlib.pyplot as plt

2 from sklearn import datasets

3 iris = datasets.load\_iris()

4 print(list(iris.keys()))

5 # ['DESCR', 'data', 'target', 'target\_names', 'feature\_names']

6 X = iris["data"][:, 3:] # petal width

7 y = (iris["target"] == 2).astype(np.int) # 1 if Iris-Virginica, else 0

8

9 from sklearn.linear\_model import LogisticRegression

10 log\_reg = LogisticRegression()

11 log\_reg.fit(X, y)

12 X\_new = np.linspace(0, 3, 1000).reshape(-1, 1)

13 # estimated probabilities for flowers with petal widths varying from 0 to 3 cm:

14 y\_proba = log\_reg.predict\_proba(X\_new)

15

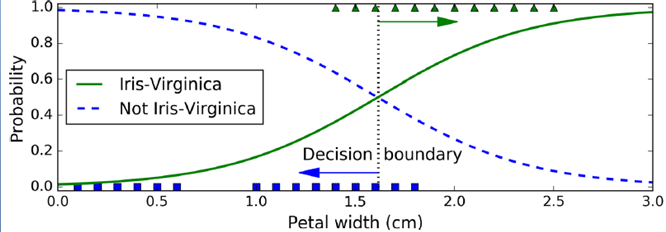
16 plt.plot(X\_new, y\_proba[:, 1], "g-", label="Iris-Virginica")

17 plt.plot(X\_new, y\_proba[:, 0], "b--", label="Not Iris-Virginica")

18 plt.show()

19 # + more Matplotlib code to make the image look pretty

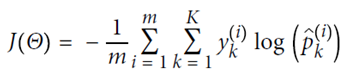
[复制代码](javascript:void(0);)



六、Softmax Regression

可以用做多分类

http://images2017.cnblogs.com/blog/1214565/201710/1214565-20171023130959332-2074370370.png

使用交叉熵

[复制代码](javascript:void(0);)

X = iris["data"][:, (2, 3)] # petal length, petal width

y = iris["target"]

softmax\_reg = LogisticRegression(multi\_class="multinomial",solver="lbfgs", C=10)

softmax\_reg.fit(X, y)

print(softmax\_reg.predict([[5, 2]]))

# array([2])

print(softmax\_reg.predict\_proba([[5, 2]]))

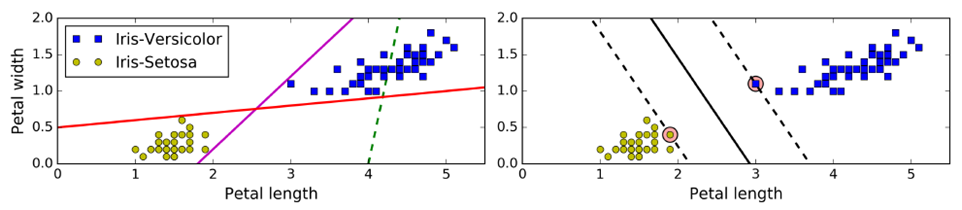
# array([[ 6.33134078e-07, 5.75276067e-02, 9.42471760e-01]])

[复制代码](javascript:void(0);)

**（二）支持向量机（SVM）**

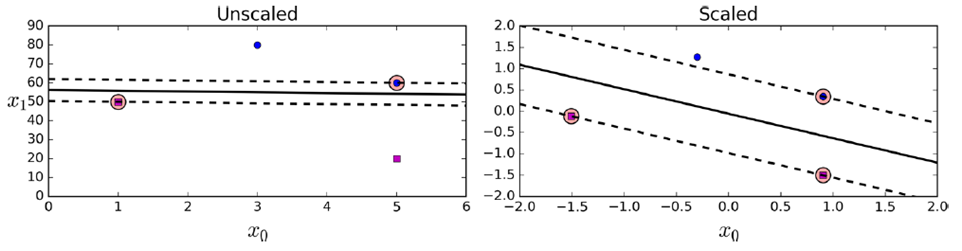
一、关于SVM

可以做线性分类、非线性分类、线性回归等，相比逻辑回归、线性回归、决策树等模型（非神经网络）功效最好

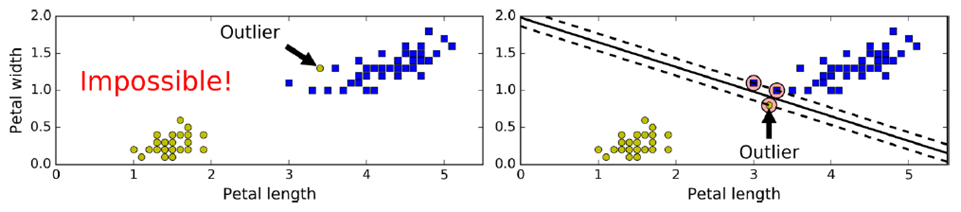


传统线性分类：选出两堆数据的质心，并做中垂线（准确性低）——上图左

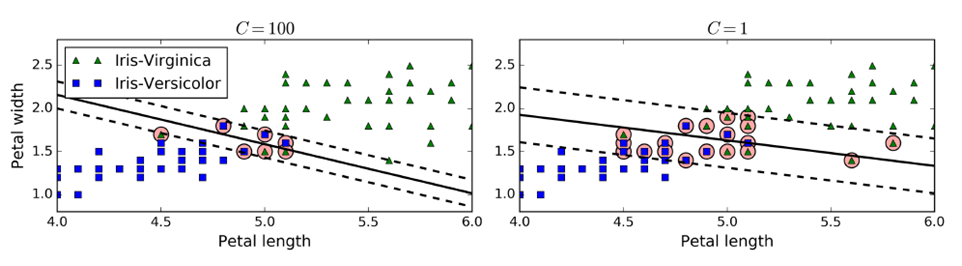
SVM：拟合的不是一条线，而是两条平行线，且这两条平行线宽度尽量大，主要关注距离车道近的边缘数据点（支撑向量support vector），即large margin classification——上图右



使用前，需要对数据集做一个scaling，以做出更好的决策边界（decision boundary）

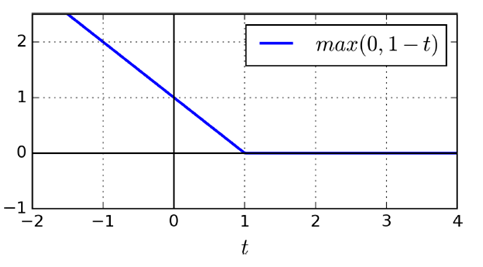


但需要容忍一些点跨越分割界限，提高泛化性，即softmax classification



在sklearn中，有一个超参数c，控制模型复杂度，c越大，容忍度越小，c越小，容忍度越高。c添加一个新的正则量，可以控制SVM泛化能力，防止过拟合。（一般使用gradsearch）

SVM特有损失函数Hinge Loss



二、LinearSVC(liblinear库，不支持kernel函数，但是相对简单，复杂度O(m\*n)）

同SVM特点吻合，仅考虑落在分类面附近和越过分类面到对方领域的向量，给于一个线性惩罚（l1），或者平方项（l2）

[复制代码](javascript:void(0);)

import numpy as np

from sklearn import datasets

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import StandardScaler

from sklearn.svm import LinearSVC

iris = datasets.load\_iris()

X = iris["data"][:,(2,3)]

y = (iris["target"]==2).astype(np.float64)

svm\_clf = Pipeline((

("scaler",StandardScaler()),

("Linear\_svc",LinearSVC(C=1,loss="hinge")),

))

svm\_clf.fit(X,y)

print(svm\_clf.predit([[5.5,1.7]]))

[复制代码](javascript:void(0);)

三、对于nonlinear数据的分类

　　有两种方法，构造高维特征，构造相似度特征

1. 使用高维空间特征（即kernel的思想），将数据平方、三次方。。映射到高维空间上

[复制代码](javascript:void(0);)

from sklearn.preprocessing import PolynomialFeatures

polynomial\_svm\_clf = Pipeline((

("poly\_features", PolynomialFeatures(degree=3)),

("scaler", StandardScaler()),

("svm\_clf", LinearSVC(C=10, loss="hinge"))

))

polynomial\_svm\_clf.fit(X, y)

[复制代码](javascript:void(0);)

这种kernel trick可以极大地简化模型，不需要显示的处理高维特征，可以计算出比较复杂的情况

但模型复杂度越强，过拟合风险越大

SVC（基于libsvm库，支持kernel函数，但是相对复杂，不能用太大规模数据，复杂度O(m^2 \*n)-O(m^3 \*n)）

 可以直接使用SVC（coef0：高次与低次权重）

[复制代码](javascript:void(0);)

from sklearn.svm import SVC

poly\_kernel\_svm\_clf = Pipeline((

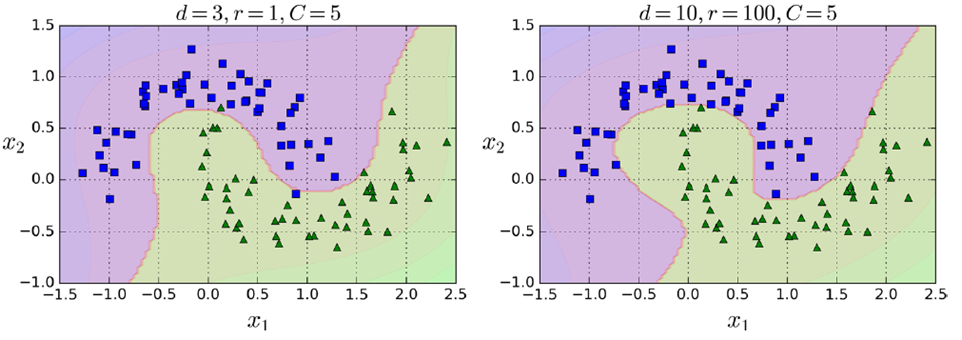
("scaler", StandardScaler()),

("svm\_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))

))

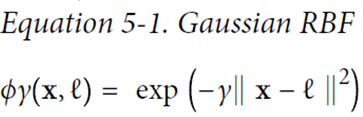
poly\_kernel\_svm\_clf.fit(X, y)

[复制代码](javascript:void(0);)



2. 添加相似度特征（similarity features）

例如，下图分别创造x1,x2两点的高斯分布，再创建新的坐标系统，计算高斯距离（Gaussian RBF Kernel径向基函数）



gamma（γ）控制高斯曲线形状胖瘦，数据点之间的距离发挥更强作用

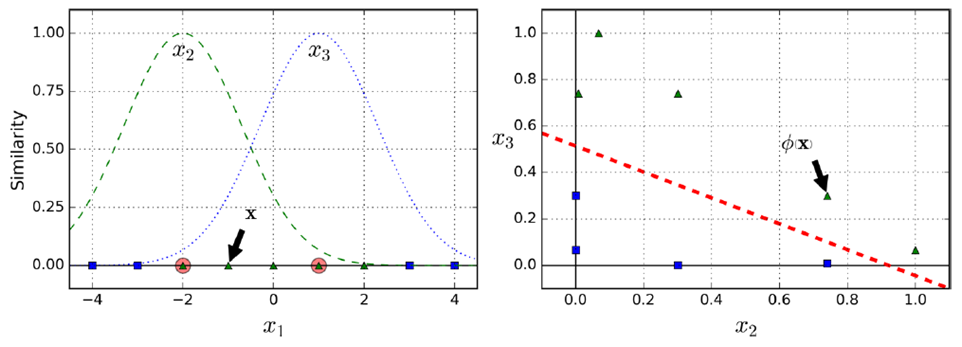
rbf\_kernel\_svm\_clf = Pipeline((

("scaler", StandardScaler()),

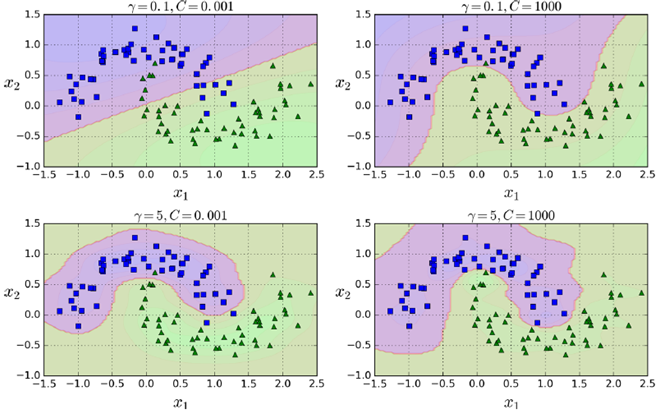
("svm\_clf", SVC(kernel="rbf", gamma=5, C=0.001))

))

rbf\_kernel\_svm\_clf.fit(X, y)



如下是不同gamma和C的取值影响



 SGDClassifier(支持海量数据，时间复杂度O(m\*n))

四、SVM Regression（SVM回归）

尽量让所用instance都fit到车道上，车道宽度使用超参数http://images2017.cnblogs.com/blog/1214565/201710/1214565-20171030091021152-1879505612.png控制，越大越宽

使用LinearSVR

from sklearn.svm import LinearSVR

svm\_reg = LinearSVR(epsilon=1.5)

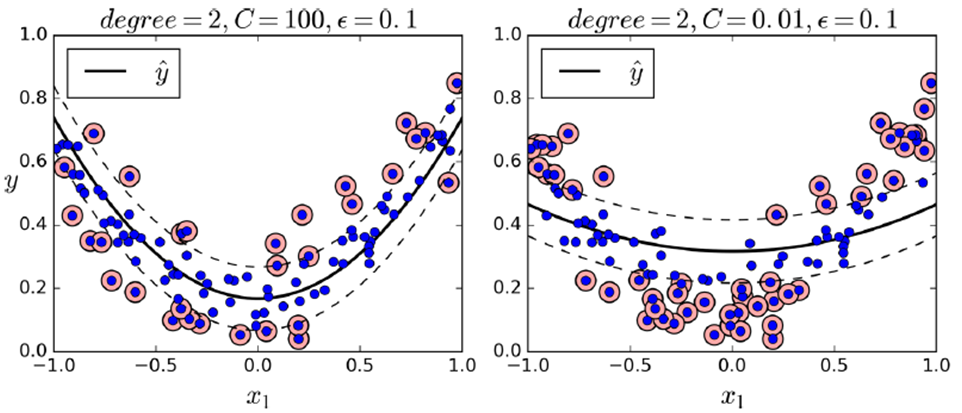
svm\_reg.fit(X, y)

 使用SVR

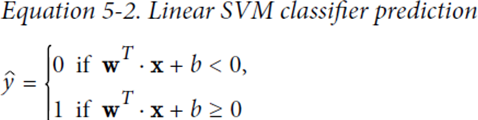
from sklearn.svm import SVR

svm\_poly\_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)

svm\_poly\_reg.fit(X, y)



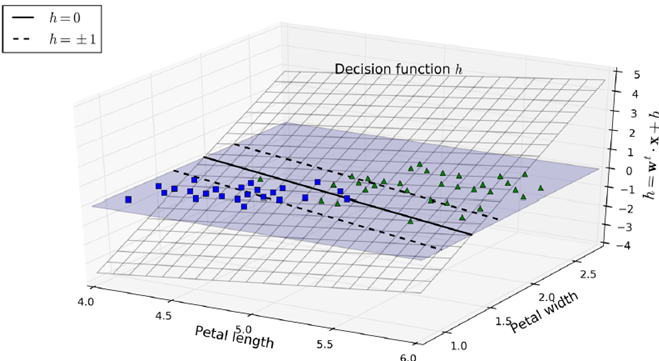
 五、数学原理：



w通过控制h倾斜的角度，控制车道的宽度，越小越宽，并且使得违反分类的数据点更少

1. hard margin linear SVM

优化目标：http://images2017.cnblogs.com/blog/1214565/201710/1214565-20171030092128918-2105925709.png，并且保证http://images2017.cnblogs.com/blog/1214565/201710/1214565-20171030092147136-1416388018.png



2. soft margin linear SVM

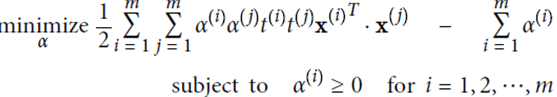
增加一个新的松弛变量（slack variable）http://images2017.cnblogs.com/blog/1214565/201710/1214565-20171030092922230-1468836139.png，起正则化作用

优化目标：http://images2017.cnblogs.com/blog/1214565/201710/1214565-20171030092758449-2029674071.png，并且保证http://images2017.cnblogs.com/blog/1214565/201710/1214565-20171030092814324-881401317.png

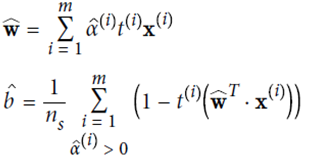
放宽条件，即使有个别实例违反条件，也惩罚不大

3. LinearSVM

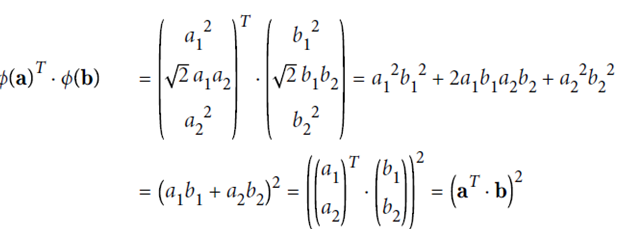
使用拉格朗日乘子法进行计算，α是松弛项后的结果



计算结果：http://images2017.cnblogs.com/blog/1214565/201710/1214565-20171030094942152-564412812.png取平均值

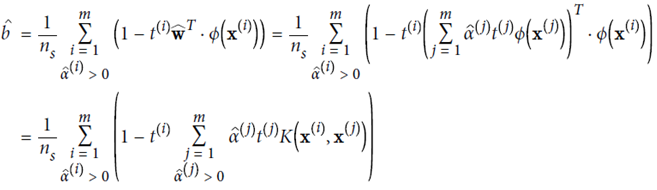
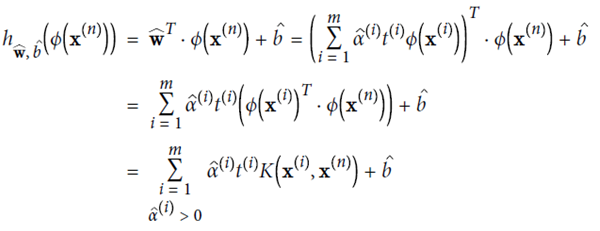


KernelizedSVM

由于

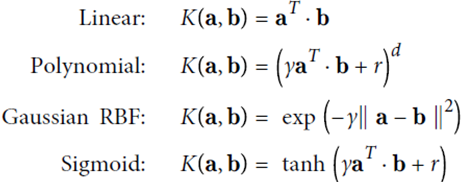
 故可先在低位空间里做点积计算，再映射到高维空间中。

下列公式表示，在高维空间计算可用kernel trick方式，直接在低维上面计算



http://images2017.cnblogs.com/blog/1214565/201710/1214565-20171030095810636-678808842.png

几个常见的kernal及其function



**（三）TensorFlow学习**

首先，何谓tensor？即高维向量，例如矩阵是二维，tensor是更广义意义上的n维向量（有type+shape）

TensorFlow执行过程为定义图，其中定义子节点，计算时只计算所需节点所依赖的节点，是一种高效且适应大规模的数据计算，方便分布式设计，对于复杂神经网络的计算，可将其拆开到其他核中同时计算。

Theano——torch———caffe（尤其是图像处理）——deeplearning5j——H20——MXNet，TensorFlow

运行环境

下载docker

打开docker quickstart terminal

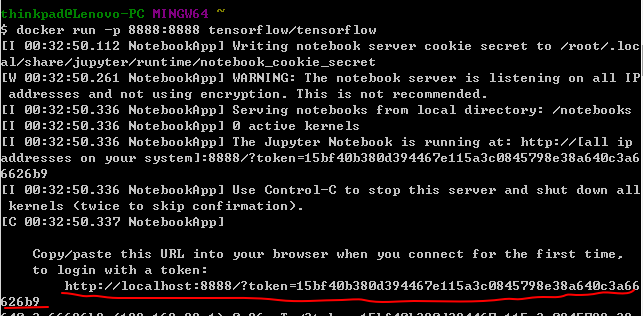


标红地方显示该docker虚拟机IP地址（即之后的localhost）

docker tensorflow/tensorflow　　//自动找到TensorFlow容器并下载

docker images　　//浏览当前容器

docker run -p 8888:8888 tensorflow/tensorflow　　//在8888端口运行



会出现一个token，复制该链接并替换掉localhost，既可以打开TensorFlow的一个编写器，jupyter

大体雏形

[复制代码](javascript:void(0);)

#python导入

import tensorflow as tf

#定义变量（节点）

x = tf.Variable(3, name="x")

y = tf.Variable(4, name="y")

f = x\*x\*y + y + 2

#定义session

sess = tf.Session()

#为已经定义的节点赋值

sess.run(x.initializer)

sess.run(y.initializer)

#运行session

result = sess.run(f)

print(result) #42

#释放空间

sess.close

[复制代码](javascript:void(0);)

还有一个更简洁的一种定义并运行session方法

[复制代码](javascript:void(0);)

# a better way

with tf.Session() as sess:

x.initializer.run()

y.initializer.run()

#即evaluate，求解f的值

result = f.eval()

[复制代码](javascript:void(0);)

初始化的两行也可以写作

init = tf.global\_variables\_initializer()

init.run()

而session可以改作sess=tf.InteractiveSession()运行起来更方便

init = tf.global\_variables\_initializer()

sess = tf.InteractiveSession()

init.run()

result = f.eval()

print(result)

因而TensorFlow的代码分为两部分，定义部分和执行部分

TensorFlow是一个图的操作，有自动缺省的默认图和你自己定义的图

[复制代码](javascript:void(0);)

#系统默认缺省的图

>>> x1 = tf.Variable(1)

>>> x1.graph is tf.get\_default\_graph()

True

#自定义的图

>>> graph = tf.Graph()

>>> with graph.as\_default():

x2 = tf.Variable(2)

>>> x2.graph is graph

True

>>> x2.graph is tf.get\_default\_graph()

False

[复制代码](javascript:void(0);)

节点的生命周期

第二种方法可以找出公共部分，避免x被计算2次。

运行结束后所有节点的值都被清空，如果没有单独保存，还需重新run一遍。

[复制代码](javascript:void(0);)

w = tf.constant(3)

x = w + 2

y = x + 5

z = x \* 3

with tf.Session() as sess:

print(y.eval()) # 10

print(z.eval()) # 15

with tf.Session() as sess:

y\_val, z\_val = sess.run([y, z])

print(y\_val) # 10

print(z\_val) # 15

[复制代码](javascript:void(0);)

 Linear Regression with TensorFlow（线性回归上的应用）

y = wx+b = wx'　　//这里x'是相较于x多了一维全是1的向量

这里引用California housing的数据

TensorFlow上向量是列向量，需要reshape(-1,1)即转置成列向量

使用normal equation方法求解

[复制代码](javascript:void(0);)

import numpy as np

from sklearn.datasets import fetch\_california\_housing

housing = fetch\_california\_housing()

#获得数据维度，矩阵的行列长度

m, n = housing.data.shape

#np.c\_是连接的含义，加了一个全为1的维度

housing\_data\_plus\_bias = np.c\_[np.ones((m, 1)), housing.data]

#数据量并不大，可以直接用常量节点装载进来，但是之后海量数据无法使用（会用minbatch的方式导入数据）

X = tf.constant(housing\_data\_plus\_bias, dtype=tf.float32, name="X")

#转置成列向量

y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")

XT = tf.transpose(X)

#使用normal equation的方法求解theta，之前线性模型中有提及

theta = tf.matmul(tf.matmul(tf.matrix\_inverse(tf.matmul(XT, X)), XT), y)

#求出权重

with tf.Session() as sess:

theta\_value = theta.eval()

[复制代码](javascript:void(0);)

如果是原本的方法，可能更直接些。但由于使用底层的库不同，它们计算出来的值不完全相同。

[复制代码](javascript:void(0);)

#使用numpy

X = housing\_data\_plus\_bias

y = housing.target.reshape(-1, 1)

theta\_numpy = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(y)

#使用sklearn

from sklearn.linear\_model import LinearRegression

lin\_reg = LinearRegression()

lin\_reg.fit(housing.data, housing.target.reshape(-1, 1))

[复制代码](javascript:void(0);)

这里不禁感到疑惑，为什么TensorFlow感觉变复杂了呢？其实，这不过因为这里数据规模较小，进行大规模的计算时，TensorFlow的自动优化所发挥的效果，是十分厉害的。

使用gradient descent（梯度下降）方法求解

[复制代码](javascript:void(0);)

#使用gradient时需要scale一下

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

scaled\_housing\_data = scaler.fit\_transform(housing.data)

scaled\_housing\_data\_plus\_bias = np.c\_[np.ones((m, 1)), scaled\_housing\_data]

#迭代1000次

n\_epochs = 1000

learning\_rate = 0.01

#由于使用gradient，写入x的值需要scale一下

X = tf.constant(scaled\_housing\_data\_plus\_bias, dtype=tf.float32, name="X")

y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")

#使用gradient需要有一个初值

theta = tf.Variable(tf.random\_uniform([n + 1, 1], -1.0, 1.0), name="theta")

#当前预测的y，x是m\*（n+1），theta是（n+1）\*1，刚好是y的维度

y\_pred = tf.matmul(X, theta, name="predictions")

#整体误差

error = y\_pred - y

#TensorFlow求解均值功能强大，可以指定维数，也可以像下面方法求整体的

mse = tf.reduce\_mean(tf.square(error), name="mse")

#暂时自己写出训练过程，实际可以采用TensorFlow自带的功能更强大的自动求解autodiff方法

gradients = 2/m \* tf.matmul(tf.transpose(X), error)

training\_op = tf.assign(theta, theta - learning\_rate \* gradients)

#初始化并开始求解

init = tf.global\_variables\_initializer()

with tf.Session() as sess:

sess.run(init)

for epoch in range(n\_epochs):

#每运行100次打印一下当前平均误差

if epoch % 100 == 0:

print("Epoch", epoch, "MSE =", mse.eval())

sess.run(training\_op)

best\_theta = theta.eval()

[复制代码](javascript:void(0);)

上述代码中的autodiff如下，可以自动求出gradient

gradients = tf.gradients(mse, [theta])[0]

 使用Optimizer

上述的整个梯度下降和迭代方法，都封装了在如下方法中

 optimizer = tf.train.GradientDescentOptimizer(learning\_rate=learning\_rate)

 training\_op = optimizer.minimize(mse)

 这样的optimizer还有很多

例如带冲量的optimizer = tf.train.MomentumOptimizer(learning\_rate=learning\_rate,*momentum=0.9)*

 Feeding data to training algorithm

 当数据量达到几G，几十G时，使用constant直接导入数据显然是不现实的，因而我们用placeholder做一个占位符

（一般行都是none，即数据量是任意的）

 真正运行，run的时候再feed数据。可以不断使用新的数据。

[复制代码](javascript:void(0);)

>>> A = tf.placeholder(tf.float32, shape=(None, 3))

>>> B = A + 5

>>> with tf.Session() as sess:

... B\_val\_1 = B.eval(feed\_dict={A: [[1, 2, 3]]})

... B\_val\_2 = B.eval(feed\_dict={A: [[4, 5, 6], [7, 8, 9]]})

...

>>> print(B\_val\_1)

[[ 6. 7. 8.]]

>>> print(B\_val\_2)

[[ 9. 10. 11.]

[ 12. 13. 14.]]

[复制代码](javascript:void(0);)

这样，就可以通过定义min\_batch来分批次随机抽取指定数量的数据，即便是几T的数据也可以抽取。

[复制代码](javascript:void(0);)

batch\_size = 100

n\_batches = int(np.ceil(m / batch\_size))

#有放回的随机抽取数据

def fetch\_batch(epoch, batch\_index, batch\_size):

#定义一个随机种子

np.random.seed(epoch \* n\_batches + batch\_index) # not shown in the book

indices = np.random.randint(m, size=batch\_size) # not shown

X\_batch = scaled\_housing\_data\_plus\_bias[indices] # not shown

y\_batch = housing.target.reshape(-1, 1)[indices] # not shown

return X\_batch, y\_batch

#开始运行

with tf.Session() as sess:

sess.run(init)

#每次都抽取新的数据做训练

for epoch in range(n\_epochs):

for batch\_index in range(n\_batches):

X\_batch, y\_batch = fetch\_batch(epoch, batch\_index, batch\_size)

sess.run(training\_op, feed\_dict={X: X\_batch, y: y\_batch})

#最终结果

best\_theta = theta.eval()

[复制代码](javascript:void(0);)

Saving and Restoring models（保存模型）

有时候，运行几天的模型可能因故暂时无法继续跑下去，因而需要暂时保持已训练好的部分模型到硬盘上。

[复制代码](javascript:void(0);)

init = tf.global\_variables\_initializer()

saver = tf.train.Saver()

#保存模型

with tf.Session() as sess:

sess.run(init)

for epoch in range(n\_epochs):

if epoch % 100 == 0:

#print("Epoch", epoch, "MSE =", mse.eval())

save\_path = saver.save(sess, "/tmp/my\_model.ckpt")

sess.run(training\_op)

best\_theta = theta.eval()

save\_path = saver.save(sess, "/tmp/my\_model\_final.ckpt")

[复制代码](javascript:void(0);)

#恢复模型

with tf.Session() as sess:

saver.restore(sess, "/tmp/my\_model\_final.ckpt")

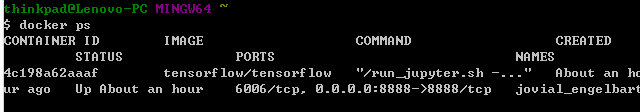
best\_theta\_restored = theta.eval()

关于TensorBoard

众所周知，神经网络和机器学习大多是黑盒模型，让人有点忐忑。TensorBoard所起的功能就是将这个黑盒稍微变白一些~

启用tensorboard

输入docker ps查看当前容器id



进入容器

http://images2017.cnblogs.com/blog/1214565/201711/1214565-20171113101934124-718126814.png

使用tensorboard --log-dir=tf\_logs命令打开已经存入的tf\_logs文件，其生成代码如下所示

[复制代码](javascript:void(0);)

from datetime import datetime

now = datetime.utcnow().strftime("%Y%m%d%H%M%S")

root\_logdir = "tf\_logs"

logdir = "{}/run-{}/".format(root\_logdir, now)

...

mse\_summary = tf.summary.scalar('MSE', mse)

file\_writer = tf.summary.FileWriter(logdir, tf.get\_default\_graph())

...

if batch\_index % 10 == 0:

summary\_str = mse\_summary.eval(feed\_dict={X: X\_batch, y: y\_batch})

step = epoch \* n\_batches + batch\_index

file\_writer.add\_summary(summary\_str, step)

**（四）人工神经网络ANN**

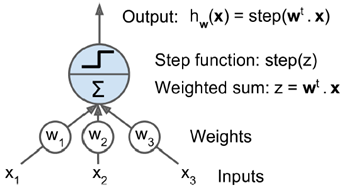
神经网络是从生物领域自然的鬼斧神工中学习智慧的一种应用。人工神经网络（ANN）的发展经历的了几次高潮低谷，如今，随着数据爆发、硬件计算能力暴增、深度学习算法的优化，我们迎来了又一次的ANN雄起时代，以深度学习为首的人工神经网络，又一次走入人们的视野。

感知机模型perceptron

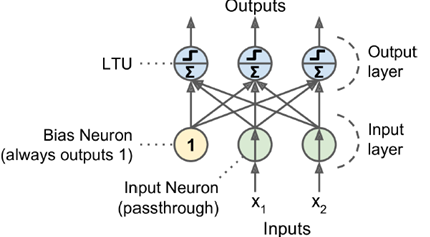
不再处理离散情况，而是连续的数值，学习时权值在变化，从而记忆存储学到的知识

神经元输入：类似于线性回归z =w1x1+w2x2 +⋯ +wnxn= wT・x（linear threshold unit (LTU)）

神经元输出：激活函数，类似于二值分类，模拟了生物学中神经元只有激发和抑制两种状态。



增加偏值，输出层哪个节点权重大，输出哪一个。



采用Hebb准则，下一个权重调整方法参考当前权重和训练效果

http://images2017.cnblogs.com/blog/1214565/201711/1214565-20171120082550805-2020381011.png

[复制代码](javascript:void(0);)

#一个感知机的例子  
import numpy as np

from sklearn.datasets import load\_iris

from sklearn.linear\_model import Perceptron

iris = load\_iris()

X = iris.data[:, (2, 3)] # petal length, petal width

y = (iris.target == 0).astype(np.int) # Iris Setosa?

per\_clf = Perceptron(random\_state=42)

per\_clf.fit(X, y)

y\_pred = per\_clf.predict([[2, 0.5]]

[复制代码](javascript:void(0);)

之后有人提出，perceptron无法处理异或问题，但是，使用多层感知机（MLP）可以处理这个问题

[复制代码](javascript:void(0);)

def heaviside(z):

return (z >= 0).astype(z.dtype)

def sigmoid(z):

return 1/(1+np.exp(-z))

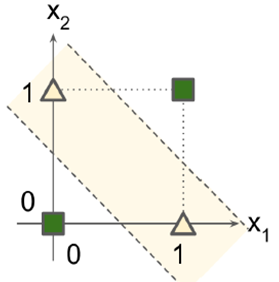
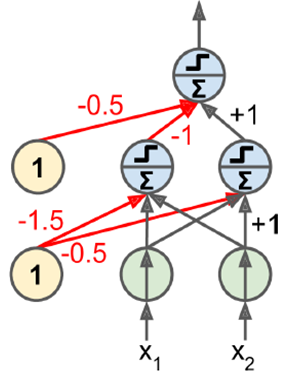
#做了多层activation，手工配置权重

def mlp\_xor(x1, x2, activation=heaviside):

return activation(-activation(x1 + x2 - 1.5) + activation(x1 + x2 - 0.5) - 0.5)

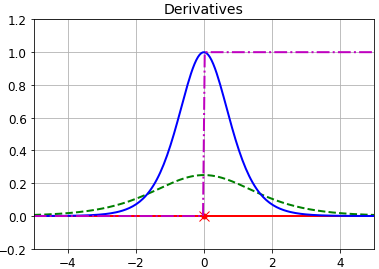
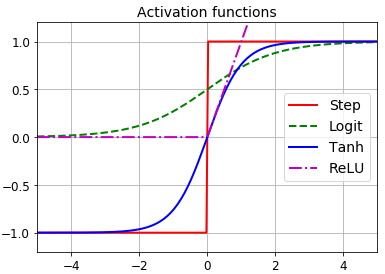
[复制代码](javascript:void(0);)

如图所示，两层MLP，包含输入层，隐层，输出层。所谓的深度神经网络，就是隐层数量多一些。



激活函数

以下是几个激活函数的例子，其微分如右图所示



step是最早提出的一种激活函数，但是它在除0外所有点的微分都是0，没有办法计算梯度

logit和双曲正切函数tanh梯度消失，数据量很大时，梯度无限趋近于0，

relu在层次很深时梯度也不为0，无限传导下去。

 如何自动化学习计算权重——backpropagation

首先正向做一个计算，根据当前输出做一个error计算，作为指导信号反向调整前一层输出权重使其落入一个合理区间，反复这样调整到第一层，每轮调整都有一个学习率，调整结束后，网络越来越合理。

 step函数换成逻辑回归函数σ(z) = 1 / (1 + exp(–z))，无论x落在哪个区域，最后都有一个非0的梯度可以使用，落在（0,1）区间。

双曲正切函数The hyperbolic tangent function tanh (z) = 2σ(2z) – 1，在（-1,1）的区间。

The ReLU function ReLU (z) = max (0, z)，层次很深时不会越传递越小。

多分类时，使用softmax（logistics激活函数）最为常见。

使用MLP多分类输出层为softmax，隐层倾向于使用ReLU，因为向前传递时不会有数值越来越小得不到训练的情况产生。

以mnist数据集为例

[复制代码](javascript:void(0);)

import tensorflow as tf

# construction phase

n\_inputs = 28\*28 # MNIST

# 隐藏层节点数目

n\_hidden1 = 300

n\_hidden2 = 100

n\_outputs = 10

X = tf.placeholder(tf.float32, shape=(None, n\_inputs), name="X")

y = tf.placeholder(tf.int64, shape=(None), name="y")

def neuron\_layer(X, n\_neurons, name, activation=None):

with tf.name\_scope(name):

n\_inputs = int(X.get\_shape()[1])

# 标准差初始设定，研究证明设为以下结果训练更快

stddev = 2 / np.sqrt(n\_inputs)

# 使用截断的正态分布，过滤掉极端的数据，做了一个初始权重矩阵，是input和neurons的全连接矩阵

init = tf.truncated\_normal((n\_inputs, n\_neurons), stddev=stddev)

W = tf.Variable(init, name="weights")

# biases项初始化为0

b = tf.Variable(tf.zeros([n\_neurons]), name="biases")

# 该层输出

z = tf.matmul(X, W) + b

# 根据activation选择激活函数

if activation=="relu":

return tf.nn.relu(z)

else:

return z

with tf.name\_scope("dnn"):

# 算上输入层一共4层的dnn结构

hidden1 = neuron\_layer(X, n\_hidden1, "hidden1", activation="relu")

hidden2 = neuron\_layer(hidden1, n\_hidden2, "hidden2", activation="relu")

# 直接输出最后结果值

logits = neuron\_layer(hidden2, n\_outputs, "outputs")

# 使用TensorFlow自带函数实现，最新修改成dense函数

from tensorflow.contrib.layers import fully\_connected

with tf.name\_scope("dnn"):

hidden1 = fully\_connected(X, n\_hidden1, scope="hidden1")

hidden2 = fully\_connected(hidden1, n\_hidden2, scope="hidden2")

logits = fully\_connected(hidden2, n\_outputs, scope="outputs", activation\_fn=None)

# 使用logits（网络输出）计算交叉熵，取均值为误差

with tf.name\_scope("loss"):

xentropy = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=y, logits=logits)

loss = tf.reduce\_mean(xentropy, name="loss")

learning\_rate = 0.01

with tf.name\_scope("train"):

optimizer = tf.train.GradientDescentOptimizer(learning\_rate)

training\_op = optimizer.minimize(loss)

with tf.name\_scope("eval"):

correct = tf.nn.in\_top\_k(logits, y, 1)

accuracy = tf.reduce\_mean(tf.cast(correct, tf.float32))

init = tf.global\_variables\_initializer()

saver = tf.train.Saver()

# Execution Phase

from tensorflow.examples.tutorials.mnist import input\_data

mnist = input\_data.read\_data\_sets("/tmp/data/")

# 外层大循环跑400次，每个循环中小循环数据量50

n\_epochs = 400

batch\_size = 50

with tf.Session() as sess:

init.run()

for epoch in range(n\_epochs):

for iteration in range(mnist.train.num\_examples // batch\_size):

X\_batch, y\_batch = mnist.train.next\_batch(batch\_size)

sess.run(training\_op, feed\_dict={X: X\_batch, y: y\_batch})

acc\_train = accuracy.eval(feed\_dict={X: X\_batch, y: y\_batch})

acc\_test = accuracy.eval(feed\_dict={X: mnist.test.images,y: mnist.test.labels})

print(epoch, "Train accuracy:", acc\_train, "Test accuracy:", acc\_test)

# 下次再跑模型时不用再次训练了

save\_path = saver.save(sess, "./my\_model\_final.ckpt")

# 下次调用

with tf.Session() as sess:

saver.restore(sess, "./my\_model\_final.ckpt") # or better, use save\_path

X\_new\_scaled = mnist.test.images[:20]

Z = logits.eval(feed\_dict={X: X\_new\_scaled})

y\_pred = np.argmax(Z, axis=1)

[复制代码](javascript:void(0);)

超参数设置

隐层数量：一般来说单个隐层即可，对于复杂问题，由于深层模型可以实现浅层的指数级别的效果，且每层节点数不多，加至overfit就不要再加了。

每层神经元数量：以漏斗形逐层递减，输入层最多，逐渐features更少代表性更强。

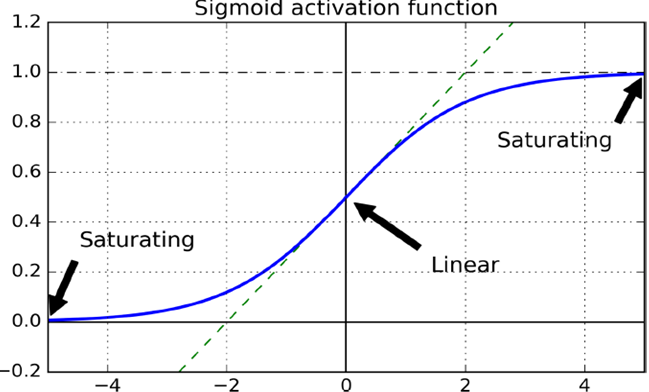
激活函数选择（activation function）：隐层多选择ReLU，输出层多选择softmax

**（五）DNN训练中的问题与方法**

由于深度神经网络（DNN）层数很多，每次训练都是逐层由后至前传递。传递项<1，梯度可能变得非常小趋于0，以此来训练网络几乎不会有什么变化，即vanishing gradients problem；或者>1梯度非常大，以此修正网络会不断震荡，无法形成一个收敛网络。因而DNN的训练中可以形成很多tricks。。

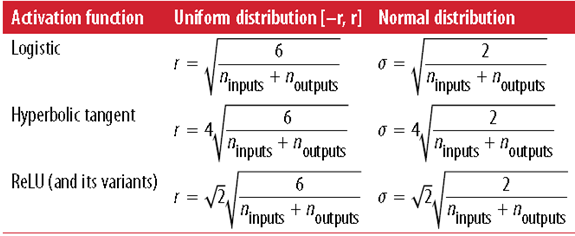
1、初始化权重

起初采用正态分布随机化初始权重，会使得原本单位的variance逐渐变得非常大。例如下图的sigmoid函数，靠近0点的梯度近似线性很敏感，但到了，即很强烈的输入产生木讷的输出。



采用Xavier initialization，根据fan-in（输入神经元个数）和fan-out（输出神经元个数）设置权重。

并设计针对不同激活函数的初始化策略，如下图（左边是均态分布，右边正态分布较为常用）

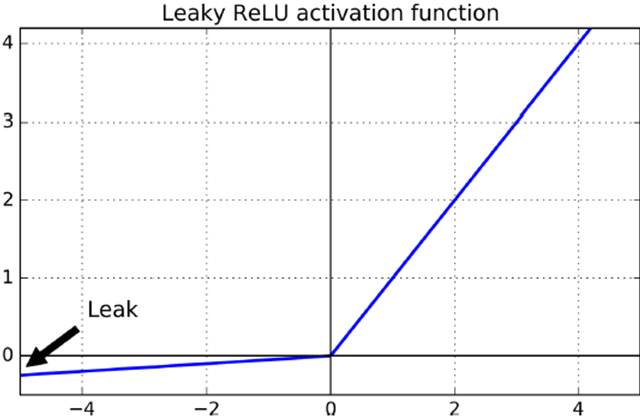


2、激活函数

一般使用ReLU，但是不能有小于0的输入（dying ReLUs）

a.Leaky RELU

改进方法Leaky ReLU=max(αx,x)，小于0时保留一点微小特征。



具体应用

[复制代码](javascript:void(0);)

from tensorflow.examples.tutorials.mnist import input\_data

mnist = input\_data.read\_data\_sets("/tmp/data/")

reset\_graph()

n\_inputs = 28 \* 28 # MNIST

n\_hidden1 = 300

n\_hidden2 = 100

n\_outputs = 10

X = tf.placeholder(tf.float32, shape=(None, n\_inputs), name="X")

y = tf.placeholder(tf.int64, shape=(None), name="y")

with tf.name\_scope("dnn"):

hidden1 = tf.layers.dense(X, n\_hidden1, activation=leaky\_relu, name="hidden1")

hidden2 = tf.layers.dense(hidden1, n\_hidden2, activation=leaky\_relu, name="hidden2")

logits = tf.layers.dense(hidden2, n\_outputs, name="outputs")

with tf.name\_scope("loss"):

xentropy = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=y, logits=logits)

loss = tf.reduce\_mean(xentropy, name="loss")

learning\_rate = 0.01

with tf.name\_scope("train"):

optimizer = tf.train.GradientDescentOptimizer(learning\_rate)

training\_op = optimizer.minimize(loss)

with tf.name\_scope("eval"):

correct = tf.nn.in\_top\_k(logits, y, 1)

accuracy = tf.reduce\_mean(tf.cast(correct, tf.float32))

init = tf.global\_variables\_initializer()

saver = tf.train.Saver()

n\_epochs = 40

batch\_size = 50

with tf.Session() as sess:

init.run()

for epoch in range(n\_epochs):

for iteration in range(mnist.train.num\_examples // batch\_size):

X\_batch, y\_batch = mnist.train.next\_batch(batch\_size)

sess.run(training\_op, feed\_dict={X: X\_batch, y: y\_batch})

if epoch % 5 == 0:

acc\_train = accuracy.eval(feed\_dict={X: X\_batch, y: y\_batch})

acc\_test = accuracy.eval(feed\_dict={X: mnist.validation.images, y: mnist.validation.labels})

print(epoch, "Batch accuracy:", acc\_train, "Validation accuracy:", acc\_test)

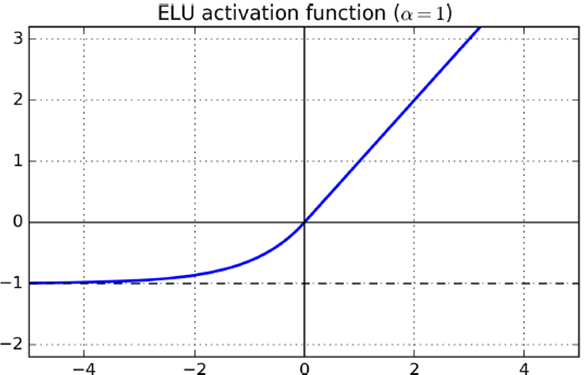
save\_path = saver.save(sess, "./my\_model\_final.ckpt")

[复制代码](javascript:void(0);)

 b. ELU改进

另一种改进ELU，在神经元小于0时采用指数变化

https://images2018.cnblogs.com/blog/1214565/201711/1214565-20171127082548534-1559842929.png



#just specify the activation function when building each layer

X = tf.placeholder(tf.float32, shape=(None, n\_inputs), name="X")

hidden1 = tf.layers.dense(X, n\_hidden1, activation=tf.nn.elu, name="hidden1")

c. SELU

最新提出的是SELU（仅给出关键代码）

with tf.name\_scope("dnn"):

hidden1 = tf.layers.dense(X, n\_hidden1, activation=selu, name="hidden1")

hidden2 = tf.layers.dense(hidden1, n\_hidden2, activation=selu, name="hidden2")

logits = tf.layers.dense(hidden2, n\_outputs, name="outputs")

[复制代码](javascript:void(0);)

# train 过程

means = mnist.train.images.mean(axis=0, keepdims=True)

stds = mnist.train.images.std(axis=0, keepdims=True) + 1e-10

with tf.Session() as sess:

init.run()

for epoch in range(n\_epochs):

for iteration in range(mnist.train.num\_examples // batch\_size):

X\_batch, y\_batch = mnist.train.next\_batch(batch\_size)

X\_batch\_scaled = (X\_batch - means) / stds

sess.run(training\_op, feed\_dict={X: X\_batch\_scaled, y: y\_batch})

if epoch % 5 == 0:

acc\_train = accuracy.eval(feed\_dict={X: X\_batch\_scaled, y: y\_batch})

X\_val\_scaled = (mnist.validation.images - means) / stds

acc\_test = accuracy.eval(feed\_dict={X: X\_val\_scaled, y: mnist.validation.labels})

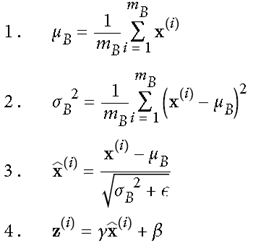
print(epoch, "Batch accuracy:", acc\_train, "Validation accuracy:", acc\_test)

save\_path = saver.save(sess, "./my\_model\_final\_selu.ckpt")

[复制代码](javascript:void(0);)

3、Batch Normalization

在2015年，有研究者提出，既然使用mini-batch进行操作，对每一批数据也可采用，在调用激活函数之前，先做一下normalization，使得输出数据有一个较好的形状，初始时，超参数scaling（γ）和shifting（β）进行适度缩放平移后传递给activation函数。步骤如下：



现今batch normalization已经被TensorFlow实现成一个单独的层，直接调用

测试时，由于没有mini-batch，故训练时直接使用训练时的mean和standard deviation（），实现代码如下

[复制代码](javascript:void(0);)

import tensorflow as tf

n\_inputs = 28 \* 28

n\_hidden1 = 300

n\_hidden2 = 100

n\_outputs = 10

batch\_norm\_momentum = 0.9

X = tf.placeholder(tf.float32, shape=(None, n\_inputs), name="X")

y = tf.placeholder(tf.int64, shape=(None), name="y")

training = tf.placeholder\_with\_default(False, shape=(), name='training')

with tf.name\_scope("dnn"):

he\_init = tf.contrib.layers.variance\_scaling\_initializer()

#相当于单独一层

my\_batch\_norm\_layer = partial(

tf.layers.batch\_normalization,

training=training,

momentum=batch\_norm\_momentum)

my\_dense\_layer = partial(

tf.layers.dense,

kernel\_initializer=he\_init)

hidden1 = my\_dense\_layer(X, n\_hidden1, name="hidden1")

bn1 = tf.nn.elu(my\_batch\_norm\_layer(hidden1))# 激活函数使用ELU

hidden2 = my\_dense\_layer(bn1, n\_hidden2, name="hidden2")

bn2 = tf.nn.elu(my\_batch\_norm\_layer(hidden2))

logits\_before\_bn = my\_dense\_layer(bn2, n\_outputs, name="outputs")

logits = my\_batch\_norm\_layer(logits\_before\_bn)# 输出层也做一个batch normalization

with tf.name\_scope("loss"):

xentropy = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=y, logits=logits)

loss = tf.reduce\_mean(xentropy, name="loss")

with tf.name\_scope("train"):

optimizer = tf.train.GradientDescentOptimizer(learning\_rate)

training\_op = optimizer.minimize(loss)

with tf.name\_scope("eval"):

correct = tf.nn.in\_top\_k(logits, y, 1)

accuracy = tf.reduce\_mean(tf.cast(correct, tf.float32))

init = tf.global\_variables\_initializer()

saver = tf.train.Saver()

n\_epochs = 20

batch\_size = 200

#需要显示调用训练时得出的方差均值，需要额外调用这些算子

extra\_update\_ops = tf.get\_collection(tf.GraphKeys.UPDATE\_OPS)

#在training和testing时不一样

with tf.Session() as sess:

init.run()

for epoch in range(n\_epochs):

for iteration in range(mnist.train.num\_examples // batch\_size):

X\_batch, y\_batch = mnist.train.next\_batch(batch\_size)

sess.run([training\_op, extra\_update\_ops],

feed\_dict={training: True, X: X\_batch, y: y\_batch})

accuracy\_val = accuracy.eval(feed\_dict={X: mnist.test.images,

y: mnist.test.labels})

print(epoch, "Test accuracy:", accuracy\_val)

save\_path = saver.save(sess, "./my\_model\_final.ckpt")

[复制代码](javascript:void(0);)

4、Gradient Clipp

处理gradient之后往后传，一定程度上解决梯度爆炸问题。（但由于有了batch normalization，此方法用的不多）

[复制代码](javascript:void(0);)

threshold = 1.0

optimizer = tf.train.GradientDescentOptimizer(learning\_rate)

grads\_and\_vars = optimizer.compute\_gradients(loss)

capped\_gvs = [(tf.clip\_by\_value(grad, -threshold, threshold), var)

for grad, var in grads\_and\_vars]

training\_op = optimizer.apply\_gradients(capped\_gvs)

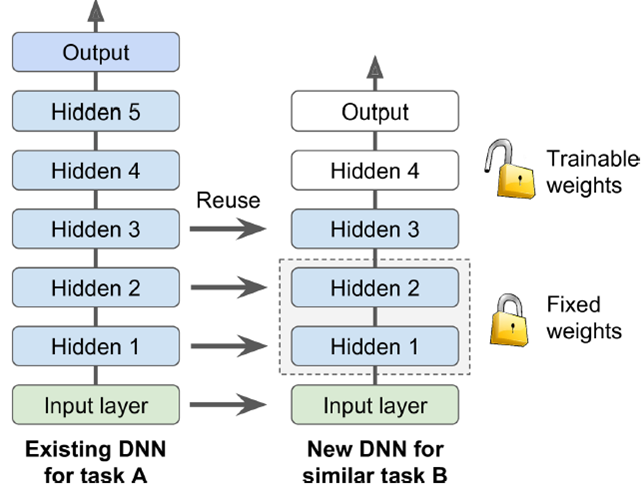
[复制代码](javascript:void(0);)

5、重用之前训练过的层（Reusing Pretrained Layers）

对之前训练的模型稍加修改，节省时间，在深度模型训练（由于有很多层）中经常使用。

一般相似问题，分类数等和问题紧密相关的output层与最后一个直接与output相关的隐层不可以直接用，仍需自己训练。

如下图所示，在已训练出一个复杂net后，迁移到相对简单的net时，hidden1和2固定不动，hidden3稍作变化，hidden4和output自己训练。。这在没有自己GPU情况下是非常节省时间的做法。



[复制代码](javascript:void(0);)

# 只选取需要的操作

X = tf.get\_default\_graph().get\_tensor\_by\_name("X:0")

y = tf.get\_default\_graph().get\_tensor\_by\_name("y:0")

accuracy = tf.get\_default\_graph().get\_tensor\_by\_name("eval/accuracy:0")

training\_op = tf.get\_default\_graph().get\_operation\_by\_name("GradientDescent")

# 如果你是原模型的作者，可以赋给模型一个清楚的名字保存下来

for op in (X, y, accuracy, training\_op):

tf.add\_to\_collection("my\_important\_ops", op)

# 如果你要使用这个模型

X, y, accuracy, training\_op = tf.get\_collection("my\_important\_ops")

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

# 训练时

with tf.Session() as sess:

saver.restore(sess, "./my\_model\_final.ckpt")

for epoch in range(n\_epochs):

for iteration in range(mnist.train.num\_examples // batch\_size):

X\_batch, y\_batch = mnist.train.next\_batch(batch\_size)

sess.run(training\_op, feed\_dict={X: X\_batch, y: y\_batch})

accuracy\_val = accuracy.eval(feed\_dict={X: mnist.test.images,

y: mnist.test.labels})

print(epoch, "Test accuracy:", accuracy\_val)

save\_path = saver.save(sess, "./my\_new\_model\_final.ckpt")

[复制代码](javascript:void(0);)

a. Freezing the Lower Layers

训练时固定底层参数，达到Freezing the Lower Layers的目的

[复制代码](javascript:void(0);)

# 以MINIST为例

n\_inputs = 28 \* 28 # MNIST

n\_hidden1 = 300 # reused

n\_hidden2 = 50 # reused

n\_hidden3 = 50 # reused

n\_hidden4 = 20 # new!

n\_outputs = 10 # new!

X = tf.placeholder(tf.float32, shape=(None, n\_inputs), name="X")

y = tf.placeholder(tf.int64, shape=(None), name="y")

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

with tf.name\_scope("dnn"):

hidden1 = tf.layers.dense(X, n\_hidden1, activation=tf.nn.relu,

name="hidden1") # reused frozen

hidden2 = tf.layers.dense(hidden1, n\_hidden2, activation=tf.nn.relu,

name="hidden2") # reused frozen

hidden2\_stop = tf.stop\_gradient(hidden2)

hidden3 = tf.layers.dense(hidden2\_stop, n\_hidden3, activation=tf.nn.relu,

name="hidden3") # reused, not frozen

hidden4 = tf.layers.dense(hidden3, n\_hidden4, activation=tf.nn.relu,

name="hidden4") # new!

logits = tf.layers.dense(hidden4, n\_outputs, name="outputs") # new!

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

with tf.name\_scope("loss"):

xentropy = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=y, logits=logits)

loss = tf.reduce\_mean(xentropy, name="loss")

with tf.name\_scope("eval"):

correct = tf.nn.in\_top\_k(logits, y, 1)

accuracy = tf.reduce\_mean(tf.cast(correct, tf.float32), name="accuracy")

with tf.name\_scope("train"):

optimizer = tf.train.GradientDescentOptimizer(learning\_rate)

training\_op = optimizer.minimize(loss)

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

reuse\_vars = tf.get\_collection(tf.GraphKeys.GLOBAL\_VARIABLES,

scope="hidden[123]") # regular expression

reuse\_vars\_dict = dict([(var.op.name, var) for var in reuse\_vars])

restore\_saver = tf.train.Saver(reuse\_vars\_dict) # to restore layers 1-3

init = tf.global\_variables\_initializer()

saver = tf.train.Saver()

with tf.Session() as sess:

init.run()

restore\_saver.restore(sess, "./my\_model\_final.ckpt")

for epoch in range(n\_epochs):

for iteration in range(mnist.train.num\_examples // batch\_size):

X\_batch, y\_batch = mnist.train.next\_batch(batch\_size)

sess.run(training\_op, feed\_dict={X: X\_batch, y: y\_batch})

accuracy\_val = accuracy.eval(feed\_dict={X: mnist.test.images,

y: mnist.test.labels})

print(epoch, "Test accuracy:", accuracy\_val)

save\_path = saver.save(sess, "./my\_new\_model\_final.ckpt")

[复制代码](javascript:void(0);)

b. Catching the Frozen Layers

训练时直接从lock层之后的层开始训练，Catching the Frozen Layers

[复制代码](javascript:void(0);)

# 以MINIST为例  
  
n\_inputs = 28 \* 28 # MNIST

n\_hidden1 = 300 # reused

n\_hidden2 = 50 # reused

n\_hidden3 = 50 # reused

n\_hidden4 = 20 # new!

n\_outputs = 10 # new!

X = tf.placeholder(tf.float32, shape=(None, n\_inputs), name="X")

y = tf.placeholder(tf.int64, shape=(None), name="y")

with tf.name\_scope("dnn"):

hidden1 = tf.layers.dense(X, n\_hidden1, activation=tf.nn.relu,

name="hidden1") # reused frozen

hidden2 = tf.layers.dense(hidden1, n\_hidden2, activation=tf.nn.relu,

name="hidden2") # reused frozen & cached

hidden2\_stop = tf.stop\_gradient(hidden2)

hidden3 = tf.layers.dense(hidden2\_stop, n\_hidden3, activation=tf.nn.relu,

name="hidden3") # reused, not frozen

hidden4 = tf.layers.dense(hidden3, n\_hidden4, activation=tf.nn.relu,

name="hidden4") # new!

logits = tf.layers.dense(hidden4, n\_outputs, name="outputs") # new!

with tf.name\_scope("loss"):

xentropy = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=y, logits=logits)

loss = tf.reduce\_mean(xentropy, name="loss")

with tf.name\_scope("eval"):

correct = tf.nn.in\_top\_k(logits, y, 1)

accuracy = tf.reduce\_mean(tf.cast(correct, tf.float32), name="accuracy")

with tf.name\_scope("train"):

optimizer = tf.train.GradientDescentOptimizer(learning\_rate)

training\_op = optimizer.minimize(loss)

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

reuse\_vars = tf.get\_collection(tf.GraphKeys.GLOBAL\_VARIABLES,

scope="hidden[123]") # regular expression

reuse\_vars\_dict = dict([(var.op.name, var) for var in reuse\_vars])

restore\_saver = tf.train.Saver(reuse\_vars\_dict) # to restore layers 1-3

init = tf.global\_variables\_initializer()

saver = tf.train.Saver()

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

import numpy as np

n\_batches = mnist.train.num\_examples // batch\_size

with tf.Session() as sess:

init.run()

restore\_saver.restore(sess, "./my\_model\_final.ckpt")

h2\_cache = sess.run(hidden2, feed\_dict={X: mnist.train.images})

h2\_cache\_test = sess.run(hidden2, feed\_dict={X: mnist.test.images}) # not shown in the book

for epoch in range(n\_epochs):

shuffled\_idx = np.random.permutation(mnist.train.num\_examples)

hidden2\_batches = np.array\_split(h2\_cache[shuffled\_idx], n\_batches)

y\_batches = np.array\_split(mnist.train.labels[shuffled\_idx], n\_batches)

for hidden2\_batch, y\_batch in zip(hidden2\_batches, y\_batches):

sess.run(training\_op, feed\_dict={hidden2:hidden2\_batch, y:y\_batch})

accuracy\_val = accuracy.eval(feed\_dict={hidden2: h2\_cache\_test, # not shown

y: mnist.test.labels}) # not shown

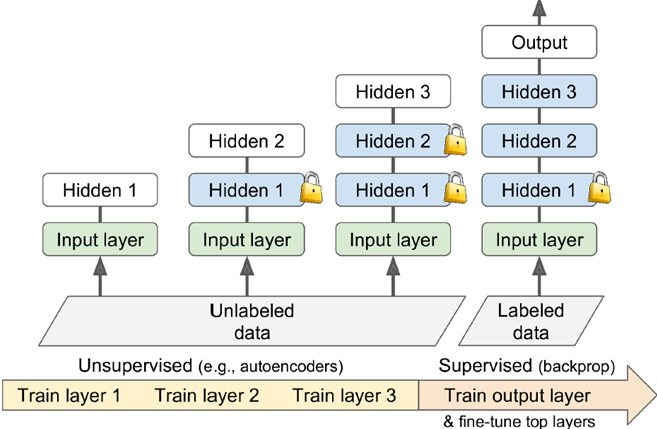
print(epoch, "Test accuracy:", accuracy\_val) # not shown

save\_path = saver.save(sess, "./my\_new\_model\_final.ckpt")

[复制代码](javascript:void(0);)

6、Unsupervised Pretraining

该方法的提出，让人们对深度学习网络的训练有了一个新的认识，可以利用不那么昂贵的未标注数据，训练数据时没有标注的数据先做一个Pretraining训练出一个差不多的网络，再使用带label的数据做正式的训练进行反向传递，增进深度模型可用性



也可以在相似模型中做pretraining

7、Faster Optimizers

在传统的SGD上提出改进

有Momentum optimization（最早提出，利用惯性冲量），Nesterov Accelerated Gradient，AdaGrad（adaptive gradient每层下降不一样），RMSProp，Adam optimization（结合adagrad和momentum，用的最多，是缺省的optimizer）

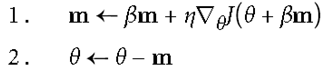
a. momentum optimization

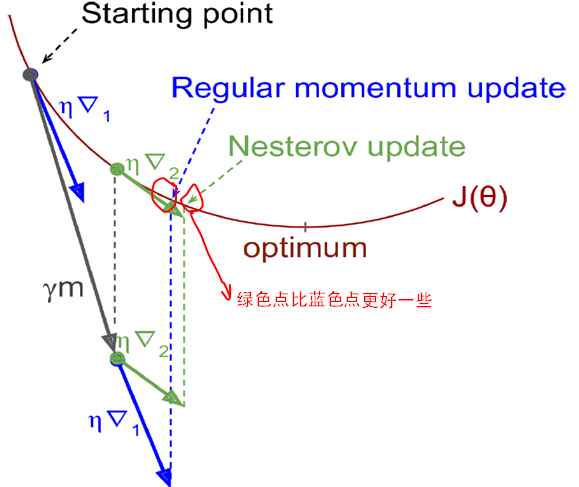
记住之前算出的gradient方向，作为惯性加到当前梯度上。相当于下山时，SGD是静止的之判断当前最陡的是哪里，而momentum相当于在跑的过程中不断修正方向，显然更加有效。

https://images2018.cnblogs.com/blog/1214565/201711/1214565-20171127091752284-1789699169.png

b. Nesterov Accelerated Gradient

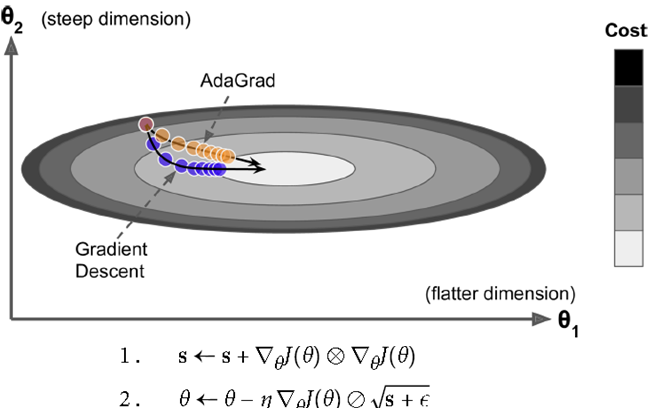
只计算当前这点的梯度，超前一步，再往前跑一点计算会更准一些。





c. AdaGrad

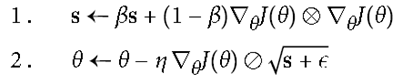
各个维度计算梯度作为分母，加到当前梯度上，不同维度梯度下降不同。如下图所示，横轴比纵轴平缓很多，传统gradient仅仅单纯沿法线方向移动，而AdaGrad平缓的θ1走的慢点，陡的θ2走的快点，效果较好。



但也有一定缺陷，s不断积累，分母越来越大，可能导致最后走不动。

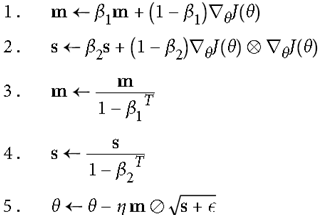
d. RMSProp（Adadelta）

只加一部分，加一个衰减系数只选取相关的最近几步相关系数



e. Adam Optimization

目前用的最多效果最好的方法，结合AdaGrad和Momentum的优点



[复制代码](javascript:void(0);)

# TensorFlow中调用方法

optimizer = tf.train.MomentumOptimizer(learning\_rate=learning\_rate,momentum=0.9)

optimizer = tf.train.MomentumOptimizer(learning\_rate=learning\_rate,momentum=0.9, use\_nesterov=True)

optimizer = tf.train.RMSPropOptimizer(learning\_rate=learning\_rate,momentum=0.9, decay=0.9, epsilon=1e-10)

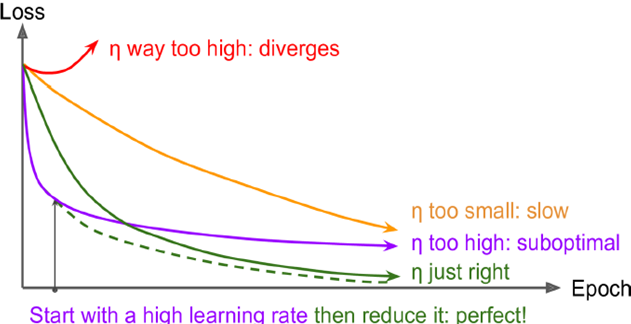
# 可以看出AdamOptimizer最省心了

optimizer = tf.train.AdamOptimizer(learning\_rate=learning\_rate)

[复制代码](javascript:void(0);)

8、learning rate scheduling

learning rate的设置也很重要，如下图所示，太大不会收敛到全局最优，太小收敛效果最差。最理想情况是都一定情况缩小learning rate，先大后小



a. Exponential Scheduling

指数级下降学习率

[复制代码](javascript:void(0);)

initial\_learning\_rate = 0.1

decay\_steps = 10000

decay\_rate = 1/10

global\_step = tf.Variable(0, trainable=False)

learning\_rate = tf.train.exponential\_decay(initial\_learning\_rate, global\_step, decay\_steps, decay\_rate)

optimizer = tf.train.MomentumOptimizer(learning\_rate, momentum=0.9)

training\_op = optimizer.minimize(loss, global\_step=global\_step)

[复制代码](javascript:void(0);)

9、Avoiding Overfitting Through Regularization

解决深度模型过拟合问题

a. Early Stopping

训练集上错误率开始上升时停止

b. l1和l2正则化

[复制代码](javascript:void(0);)

# construct the neural network

base\_loss = tf.reduce\_mean(xentropy, name="avg\_xentropy")

reg\_losses = tf.reduce\_sum(tf.abs(weights1)) + tf.reduce\_sum(tf.abs(weights2))

loss = tf.add(base\_loss, scale \* reg\_losses, name="loss")

with arg\_scope( [fully\_connected], weights\_regularizer=tf.contrib.layers.l1\_regularizer(scale=0.01)):

hidden1 = fully\_connected(X, n\_hidden1, scope="hidden1")

hidden2 = fully\_connected(hidden1, n\_hidden2, scope="hidden2")

logits = fully\_connected(hidden2, n\_outputs, activation\_fn=None,scope="out")

reg\_losses = tf.get\_collection(tf.GraphKeys.REGULARIZATION\_LOSSES)

loss = tf.add\_n([base\_loss] + reg\_losses, name="loss")

[复制代码](javascript:void(0);)

c. dropout

 一种新的正则化方法，随机生成一个概率，大于某个阈值就扔掉，随机扔掉一些神经元节点，结果表明dropout很能解决过拟合问题。可强迫现有神经元不会集中太多特征，降低网络复杂度，鲁棒性增强。

 加入dropout后，training和test的准确率会很接近，一定程度解决overfit问题

[复制代码](javascript:void(0);)

training = tf.placeholder\_with\_default(False, shape=(), name='training')

dropout\_rate = 0.5 # == 1 - keep\_prob

X\_drop = tf.layers.dropout(X, dropout\_rate, training=training)

with tf.name\_scope("dnn"):

hidden1 = tf.layers.dense(X\_drop, n\_hidden1, activation=tf.nn.relu,

name="hidden1")

hidden1\_drop = tf.layers.dropout(hidden1, dropout\_rate, training=training)

hidden2 = tf.layers.dense(hidden1\_drop, n\_hidden2, activation=tf.nn.relu,

name="hidden2")

hidden2\_drop = tf.layers.dropout(hidden2, dropout\_rate, training=training)

logits = tf.layers.dense(hidden2\_drop, n\_outputs, name="outputs")

[复制代码](javascript:void(0);)

d. Max-Norm Regularization

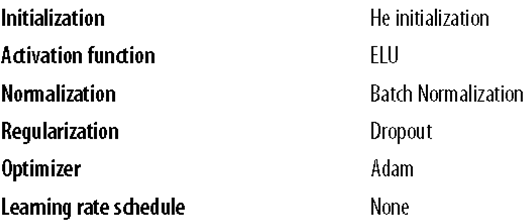
 可以把超出threshold的权重截取掉，一定程度上让网络更加稳定

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

e. Date Augmentation

 深度学习网络是一个数据饥渴模型，需要很多的数据。扩大数据集，例如图片左右镜像翻转，随机截取，倾斜随机角度，变换敏感度，改变色调等方法，扩大数据量，减少overfit可能性

10、default DNN configuration



**（六）彻底搞懂CNN**

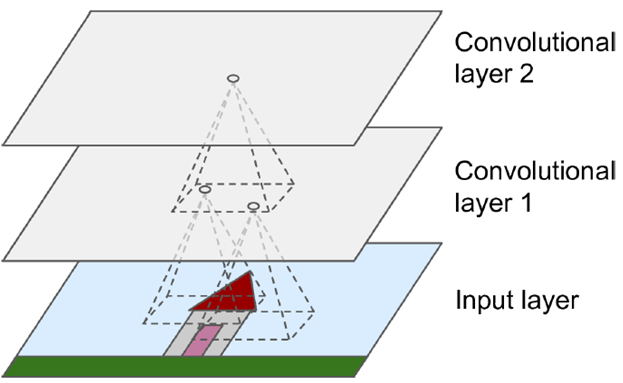
上世纪科学家们发现了几个视觉神经特点，视神经具有局部感受野，一整张图的识别由多个局部识别点构成；不同神经元对不同形状有识别能力，且视神经具有叠加能力，高层复杂的图案可以由低层简单线条组成。之后人们发现经过conclusional的操作，可以很好反映视神经处理计算的过程，典型的是1998年LeCun发明的LeNet-5，可以极大地提升识别效果。

本文主要就convolutional layer、pooling layer和整体CNN结构展开

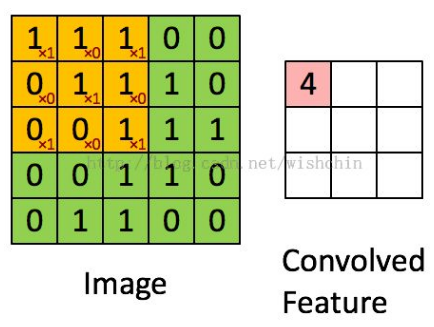
一、Convolutional Layer卷积层

1、原理和参数

可以模拟局部感受野的性质，同上一层不是全连接，而是一小块区域连接，这一小块就是局部感受野（receptive field）。并且通过构造特定的卷积神经元，可以模拟不同神经元对不同形状刺激不同反应的性质。如下图所示，一个神经元处理一层会形成一个feature map，多层叠加，层数逐渐加深。

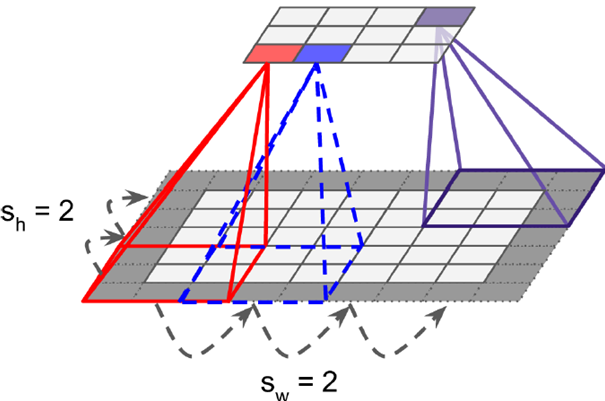


感受野（kernel或filter）的尺寸可以看做fh\*fw，由于感受野本身具有尺寸，feature map会不断缩小，为了处理方便，使得每层大小不变，于是我们每层加值为0的边（zero padding），保证经过处理以后的feature map同前一层尺寸一样。多层之间的卷积运算操作，相当于和原来像素对应位置做乘法。如下左图所示，加了边后可以保证上下层大小一致，右图表示每层之间convolve的操作（如果不加zero padding）。

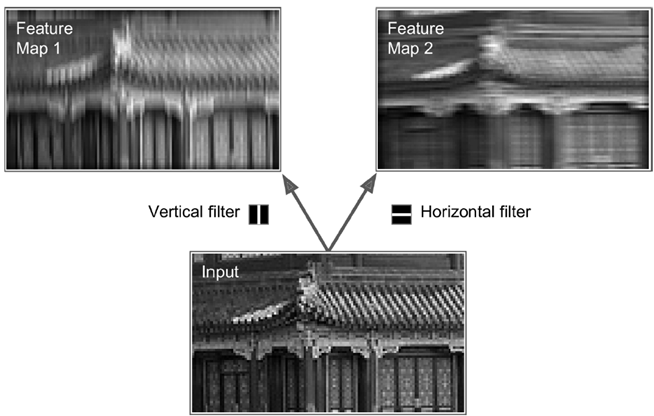


  但上图所示只是简单例子，一般扫描的是三维图像（RGB），就不是一个矩阵，而是一个立方体，我们用一个三维块去扫描它，原理同上图相同。

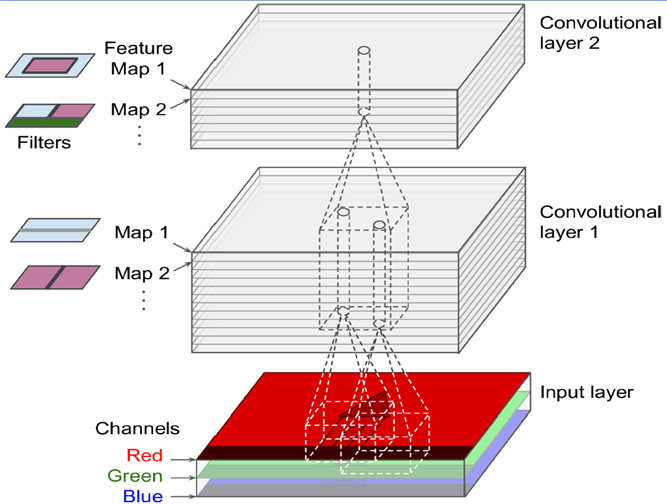
有时扫描时不是顺序去扫，而是跳跃着扫描，每次移动2-3个像素值（stride），但并非完全分离不会造成信息丢失，这样形成的feature map相较于原始图片缩小，实现信息聚集的效果。



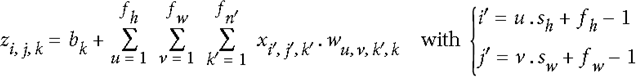
就像如下灰度图（2d）中所示，左边只提取竖线（vertical filter），右边只提取横线（horizontal filter）可看出横梁部分变亮，大量不同的这样的filter（比如可以识别边角、折线的filter）的叠加，可形成多张feature maps



下图是一个3d的RGB效果，每个kernel（filter）可以扫描出一张feature map，多个filter可以叠加出很厚的feature maps，前一层filter做卷积可以形成后一层的一个像素点



如下图，可以代表i行j列k深度的一个输出像素值，k’代表第k个filter，w代表filter中的值，x代表输入，b是偏值。



2、TensorFlow实现

以下是使用TensorFlow实现的代码，主要使用conv2d这个函数

[复制代码](javascript:void(0);)

import numpy as np

from sklearn.datasets import load\_sample\_images

# Load sample images

dataset = np.array(load\_sample\_images().images, dtype=np.float32)

# 一共4维，channel表示通道数，RGB是3

batch\_size, height, width, channels = dataset.shape

# Create 2 filters

# 一般感受野大小7\*7,5\*5,3\*3，设置2个kernel，输出2层feature map

filters\_test = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)

# 第一个（0）filter的设定，7\*7矩阵中，3是中间

filters\_test[:, 3, :, 0] = 1 # vertical line

# 第二个（1）filter的设定

filters\_test[3, :, :, 1] = 1 # horizontal line

# a graph with input X plus a convolutional layer applying the 2 filters

X = tf.placeholder(tf.float32, shape=(None, height, width, channels))

# 虽然输入是一个四维图像，但是由于batch\_size和channel都已经固定，所以使用conv2d

# strides设定，第一个和第四个都是1表示不可以跳过batch\_size和channel

# 那两个2表示横纵向都缩减2，相当于整张图片缩减为原来四分之一，做了75%的缩减

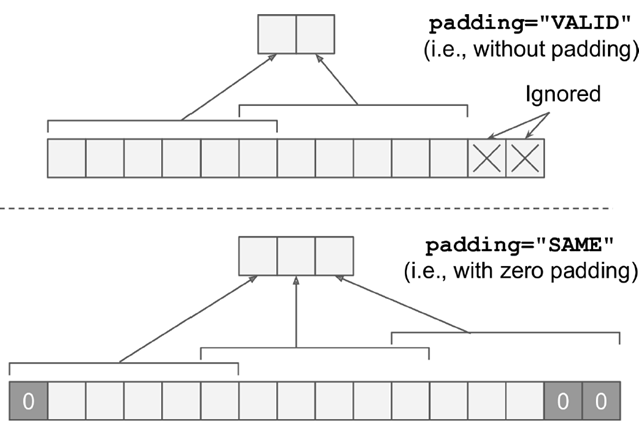
convolution = tf.nn.conv2d(X, filters, strides=[1,2,2,1], padding="SAME")

with tf.Session() as sess:

output = sess.run(convolution, feed\_dict={X: dataset})

[复制代码](javascript:void(0);)

下面是padding的值SAME和VALID的区别（filter的宽度为6，stride为5），SAME确保所有图像信息都被convolve添加zero padding，而VALID只添加包含在内的像素点



3、所耗内存计算

相比于传统的全连接层，卷积层只是部分连接，节省了很多内存。

比如：一个具有5\*5大小filter的卷积层，输出200张150\*100大小的feature maps，stride取1（即不跳跃），padding为SAME。输入是150\*100大小的RGB图像（channel=3），总共的参数个数是200\*（5\*5\*3+1）=15200，其中+1是bias；如果输出采用32-bits float表示（np.float32），那么每张图片会占据200\*150\*100\*32=9600000bits（11.4MB），如果一个training batch包含100张图片（mini-batch=100），那么这一层卷积层就会占据1GB的RAM。

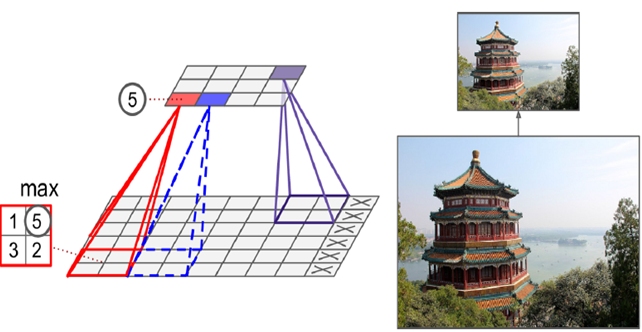
可以看出，训练卷积神经网络是非常消耗内存的，但是使用时，只用到最后一层的输出即可。

二、Pooling Layer池化层

1、原理和参数

当图片大小很大时内存消耗巨大，而Pooling Layer所起的作用是浓缩效果，缓解内存压力。

即选取一定大小区域，将该区域用一个代表元素表示。具体的Pooling有两种，取平均值（mean）和取最大值（max）。如下图所示是一个取最大值的pooling layer，kernel大小为2\*2，stride大小取决于kernel大小，这里是2，即刚好使得所有kernel都不重叠的值，因而实现高效的信息压缩，将原始图像横纵压缩一半，如右图所示，特征基本都完全保留了下来。



pooling这个操作不影响channel数，在feature map上也一般不做操作（即z轴一般不变），只改变横纵大小。

 2、TensorFlow实现

[复制代码](javascript:void(0);)

# Create a graph with input X plus a max pooling layer

X = tf.placeholder(tf.float32, shape=(None, height, width, channels))

# 选用取最大值的max\_pool方法

# 如果是取平均值，这里是mean\_pool  
# ksize就是kernel大小，feature map和channel都是1,横向纵向是2

max\_pool = tf.nn.max\_pool(X, ksize=[1,2,2,1], strides=[1,2,2,1],padding="VALID")

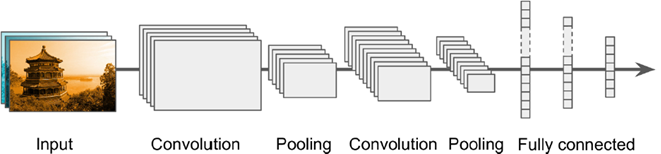
with tf.Session() as sess:

output = sess.run(max\_pool, feed\_dict={X: dataset})

[复制代码](javascript:void(0);)

三、整体CNN框架

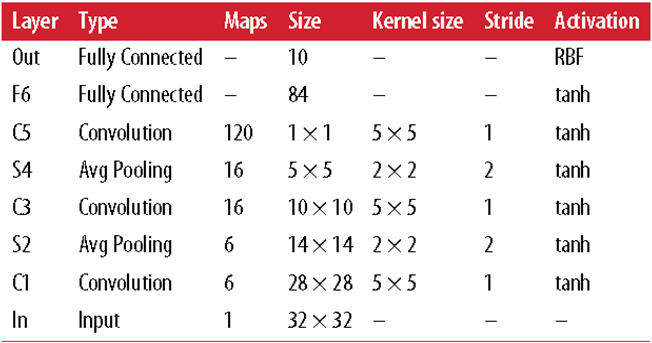
 典型CNN architecture



有名的CNN架构：

LeNet（MISIT上）-1998：输入32\*32（在28\*28图像上加了zero padding）。第一层kernel用了6个神经元，kernel大小5\*5，stride取1，输出就是28\*28；第二层做了average pooling，2\*2的kernel，stride是2，输出就变为原来的一半，不改变feature map数目；第三层放了16个神经元，其他同理；第五层用了120个神经元，5\*5的kernel对5\*5的输入做卷积，没法再滑动，输出为1\*1；F6用120个1\*1的输出全连接84个神经元，Out全连接10个神经元，对应手写体识别输出的10个数字。

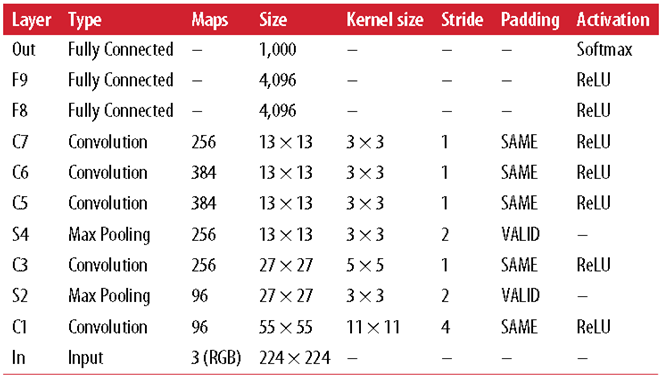
激活函数前面都用的tanh，是传统CNN中常用的，输出层用了RBF比较特殊，是一个计算距离的方式去判断和目标输出间距离做lost。。



AlexNet-2012：最早应用于竞赛中，近10%的提高了准确度

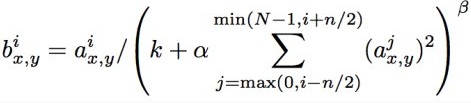
输入224\*224的彩色图像，C1是个很大的11\*11的filter，stride=4。。最后连做3层convolution。。最后输出1000个类的分类结果。

激活函数使用ReLU，这在现今很流行，输出层用的softmax



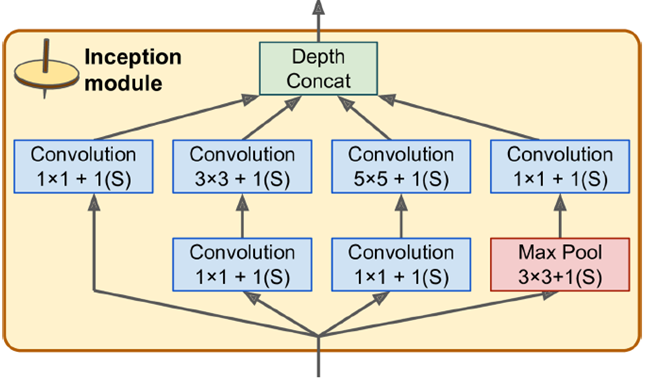
AlexNet使用了一个小技巧是Local Response Normalization（LRN局部响应归一化）

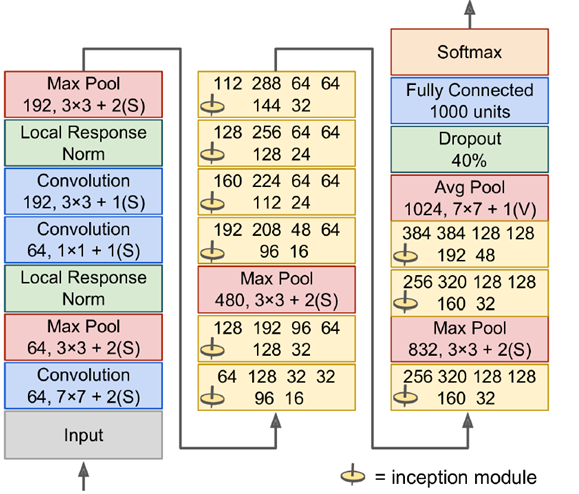
这种操作可以在传统输出上加一个bias，考虑到近邻的一些输出影响。即一个输出旁边有很牛掰的输出的话，它的输出就会怂了，收到抑制，可以看到，含β的整个项都在分母上。但后来发现，这个技术对分类器的提升也不是很明显，有的就没有用。



GoogleLeNet-2014：

大量应用Inception module，一个输入进来，直接分四步进行处理，这四步处理完后深度直接进行叠加。在不同的尺度上对图片进行操作。大量运用1\*1的convolution，可以灵活控制输出维度，可以降低参数数量。

如右图所示，输入是192，使用了9层inception module，如果直接用3\*3,5\*5参数，可以算一下，之后inception参数数目是非常大的，深度上可以调节，可以指定任意数目的feature map，通过增加深度把维度减下来。inception模块6个参数刚好对应这6个convolution，上面4个参数对应上面4个convolution，加入max pool不会改变feature map数目（如480=128+192+96+64）。

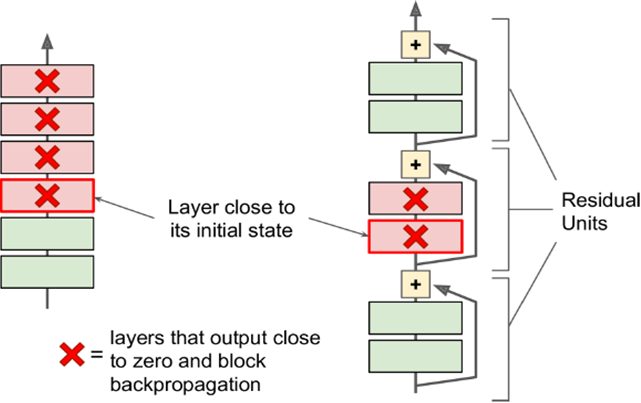
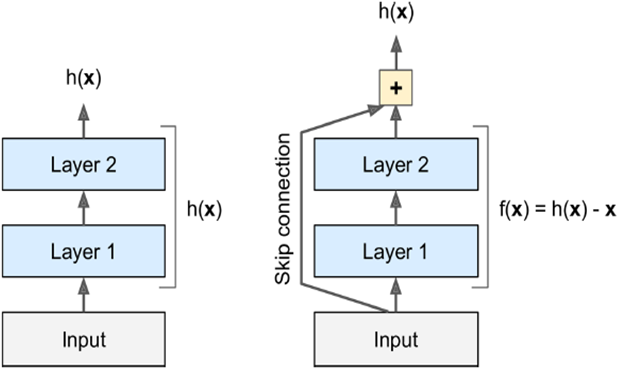


将正确率升高到95-96%，超过人类分辨率，因为image net中但是狗的种类就有很多，人类无法完全一一分辨出。

ReSNet残差网络-2015：

不再直接学习一个目标函数，输入直接跳过中间层直接连到输出上，要学习的是残差f（x），输入跳过中间层直接加到输出上。

好处是：深度模型路径依赖的限制，即gradient向前传导时要经过所有层，如果中间有层死掉了，前面的层就无法得到训练。残差网络不断跳跃，即使中间有的层已经死掉，信息仍旧能够有效流动，使得训练信号有效往回传导。

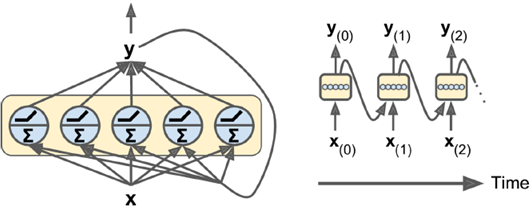
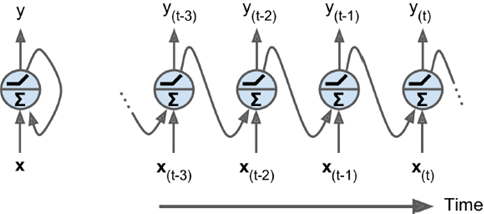


**（六）RNN学习**

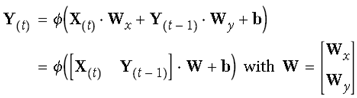
 之前提到的CNN模型主要用到人类的视觉中枢，但其有一劣势，无论是人类的视觉神经还是听觉神经，所接受到的都是一个连续的序列，使用CNN相当于割裂了前后的联系。从而诞生了专门为处理序列的Recurrent Neural Network（RNN），每一个神经元除了当前信息的输入外，还有之前产生的记忆信息，保留序列依赖型。

一、RNN基本原理

如下图所示有两种表示方法，每张图片左边是RNN的神经元（称为memory cell），右边是按时间轴展开后的情况。每次输入两个信息输出两个信息，每轮处理hidden state。把同样神经元在时间上展开处理，比CNN更加节省参数，是一个相当高效的表示方法。



可参考如下公式表示，最后简化后的形式同一般神经元相同，输入信息乘权重加偏值：

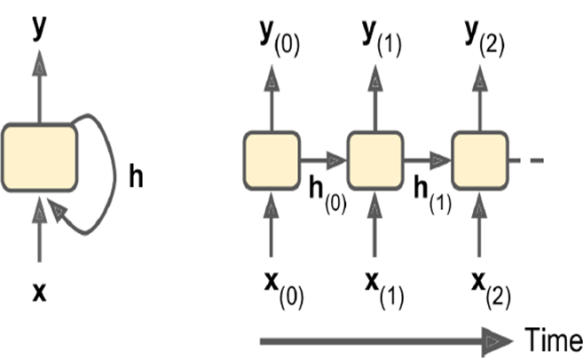


由于t状态的t由t-1时候决定，因而具有记忆功能，也叫作memory cell（或cell）

隐状态可由如下表示：

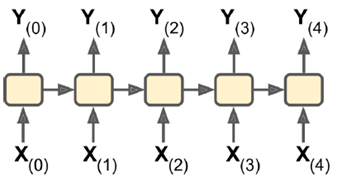
http://images2017.cnblogs.com/blog/1214565/201712/1214565-20171211082434006-1885352676.png

隐状态是当前t时刻的状态，也由t-1时刻决定，简单情况下，hidden state等同于output（y），但大多较为复杂的cell中，它们并不相同。如下图所示：



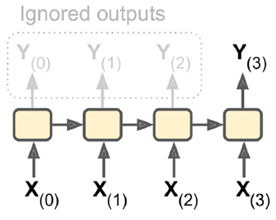
二、RNN种类：

1. sequence-to-sequence：输入输出都是一个序列。例如股票预测中的RNN，输入是前N天价格，输出明天的股市价格。

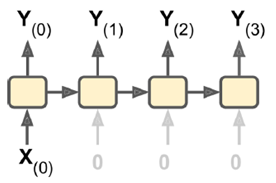


2. sequence-to-vector：输入是一个序列，输出单一向量。

　　例如，输入一个电影评价序列，输出一个分数表示情感趋势（喜欢还是讨厌）。

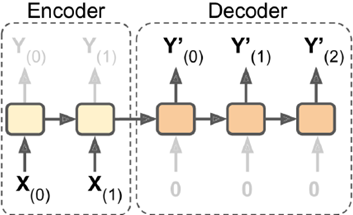


3. vector-to-sequence：输入单一向量，输出一个序列。



4.Encoder-Decoder：输入sequence-to-vector，称作encoder，输出vector-to-sequence，称作decoder。

这是一个delay模型，经过一段延迟，即把所有输入都读取后，在decoder中获取输入并输出一个序列。这个模型在机器翻译中使用较广泛，源语言输在入放入encoder，浓缩在状态信息中，生成目标语言时，可以生成一个不长度的目标语言序列。



 三、RNN实例

1.手动实现

以下是一个手动实现RNN的实例

[复制代码](javascript:void(0);)

n\_inputs = 3

# hidden state

n\_neurons = 5

X0 = tf.placeholder(tf.float32, [None, n\_inputs])

X1 = tf.placeholder(tf.float32, [None, n\_inputs])

# 由于Wx要和X相乘，故低维是n\_inputs

Wx = tf.Variable(tf.random\_normal(shape=[n\_inputs, n\_neurons],dtype=tf.float32))

# 低维，高维都是n\_neurons，为了使得输出也是hidden state的深度

# 这样下一次才可以继续运算

Wy = tf.Variable(tf.random\_normal(shape=[n\_neurons,n\_neurons],dtype=tf.float32))

b = tf.Variable(tf.zeros([1, n\_neurons], dtype=tf.float32))

# Y0初始化为0，初始时没有记忆

Y0 = tf.tanh(tf.matmul(X0, Wx) + b)

# 把上一轮输出Y0也作为输入

Y1 = tf.tanh(tf.matmul(Y0, Wy) + tf.matmul(X1, Wx) + b)

init = tf.global\_variables\_initializer()

import numpy as np

X0\_batch = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 0, 1]]) # t = 0

X1\_batch = np.array([[9, 8, 7], [0, 0, 0], [6, 5, 4], [3, 2, 1]]) # t = 1

with tf.Session() as sess:

　　init.run()

　　Y0\_val, Y1\_val = sess.run([Y0, Y1], feed\_dict={X0: X0\_batch, X1: X1\_batch})

# Y0，Y1都是4\*5大小，4是mini-batch数目，5是输出神经元个数

[复制代码](javascript:void(0);)

TensorFlow函数集成后实现

2.static unrolling through time

static\_rnn()是使用链式cells实现一个按时间轴展开的RNN

[复制代码](javascript:void(0);)

# 这种和上面那种手动实现的效果相同

n\_inputs = 3

n\_neurons = 5

X0 = tf.placeholder(tf.float32, [None, n\_inputs])

X1 = tf.placeholder(tf.float32, [None, n\_inputs])

basic\_cell = tf.contrib.rnn.BasicRNNCell(num\_units=n\_neurons)

output\_seqs, states = tf.contrib.rnn.static\_rnn(basic\_cell, [X0, X1], dtype=tf.float32)

Y0, Y1 = output\_seqs

# run部分

init = tf.global\_variables\_initializer()

X0\_batch = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 0, 1]])

X1\_batch = np.array([[9, 8, 7], [0, 0, 0], [6, 5, 4], [3, 2, 1]])

with tf.Session() as sess:

init.run()

Y0\_val, Y1\_val = sess.run([Y0, Y1], feed\_dict={X0: X0\_batch, X1: X1\_batch})

[复制代码](javascript:void(0);)

packing sequence

[复制代码](javascript:void(0);)

n\_steps = 2

n\_inputs = 3

n\_neurons = 5

# 输入是一个三维tensor，none是mini-batch大小不限，n\_steps是序列长度

X = tf.placeholder(tf.float32, [None, n\_steps, n\_inputs])

# 把一个高维度n的tensor展开成一个n-1维，降维，这里是3位降到2维列表

# unstack之前要做一个1,2维转置，相当于构造了n\_steps个数的列表

X\_seqs = tf.unstack(tf.transpose(X, perm=[1, 0, 2]))

basic\_cell = tf.contrib.rnn.BasicRNNCell(num\_units=n\_neurons)

# states是最新状态

output\_seqs, states = tf.contrib.rnn.static\_rnn( basic\_cell, X\_seqs, dtype=tf.float32)

# 再做一个转置，和输入对应

outputs = tf.transpose(tf.stack(output\_seqs), perm=[1, 0, 2])

# 输入大小4\*2\*3

X\_batch = np.array([

# t = 0　　　　 t = 1

[[0, 1, 2], [9, 8, 7]], # instance 0

[[3, 4, 5], [0, 0, 0]], # instance 1

[[6, 7, 8], [6, 5, 4]], # instance 2

[[9, 0, 1], [3, 2, 1]], # instance 3

])

with tf.Session() as sess:

init.run()

outputs\_val = outputs.eval(feed\_dict={X: X\_batch})

# output\_val是一个4\*2\*5，仅输出维度神经元个数改变

[复制代码](javascript:void(0);)

3. dynamic RNN

本身支持高维tensor输入，内嵌一个循环运行足够多次数的cell，不需要unstack步骤。

这个内嵌循环while\_loop()在前向传播中将每次迭代的tensor值存储下来，以便于反向传播过程中使用其计算梯度值。

X = tf.placeholder(tf.float32, [None, n\_steps, n\_inputs])

# 动态RNN内部封装一个循环

# 根据输入，动态决定自己需要展开几次

basic\_cell = tf.contrib.rnn.BasicRNNCell(num\_units=n\_neurons)

outputs, states = tf.nn.dynamic\_rnn(basic\_cell, X, dtype=tf.float32)

dynamicRNN可以动态规定输入大小（就像句子输入）

[复制代码](javascript:void(0);)

n\_steps = 2

n\_inputs = 3

n\_neurons = 5

X = tf.placeholder(tf.float32, [None, n\_steps, n\_inputs])

basic\_cell = tf.contrib.rnn.BasicRNNCell(num\_units=n\_neurons)

seq\_length = tf.placeholder(tf.int32, [None])

outputs, states = tf.nn.dynamic\_rnn(basic\_cell, X, dtype=tf.float32, sequence\_length=seq\_length)

init = tf.global\_variables\_initializer()

# X\_batch的大小4\*2\*3

X\_batch = np.array([

# step 0 step 1

[[0, 1, 2], [9, 8, 7]], # instance 1

[[3, 4, 5], [0, 0, 0]], # instance 2 (padded with zero vectors)

[[6, 7, 8], [6, 5, 4]], # instance 3

[[9, 0, 1], [3, 2, 1]], # instance 4

])

# 这里设置sequence大小，一共4个batch，第二维上只取第一个

seq\_length\_batch = np.array([2, 1, 2, 2])

with tf.Session() as sess:

init.run()

outputs\_val, states\_val = sess.run(

[outputs, states], feed\_dict={X: X\_batch, seq\_length: seq\_length\_batch})

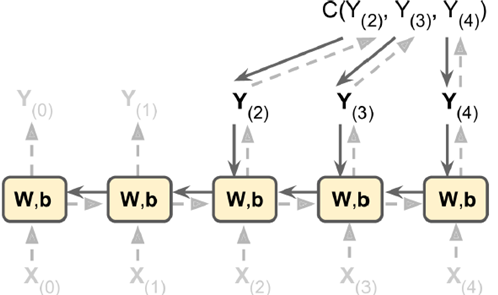
[复制代码](javascript:void(0);)

 如果事先不知道输出序列的长度，就需要定义一个end-of-sequence token（eos token），无论是课上还是网上相关信息都很少，这里就不展开了。。

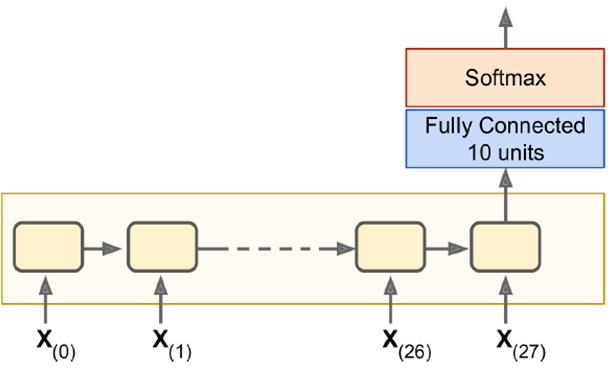
四、RNN训练

1. 拟合分类

RNN比较难以训练，但是如下图的节点中，cost function就包含y2,y3,y4三个输出，往回回溯。



以下MINIST中使用150个RNN神经元，最后加一个全连接层，得到10个神经元的输出（分别对应0-9），最后看对应在



[复制代码](javascript:void(0);)

from tensorflow.contrib.layers import fully\_connected

n\_steps = 28

n\_inputs = 28

n\_neurons = 150

n\_outputs = 10

learning\_rate = 0.001

X = tf.placeholder(tf.float32, [None, n\_steps, n\_inputs])

# 一维输出

y = tf.placeholder(tf.int32, [None])

# 使用最简单的basicRNNcell

basic\_cell = tf.contrib.rnn.BasicRNNCell(num\_units=n\_neurons)

#使用dynamic\_rnn

outputs, states = tf.nn.dynamic\_rnn(basic\_cell, X, dtype=tf.float32)

# 原始输出

logits = fully\_connected(states, n\_outputs, activation\_fn=None)

# 计算和真实的交叉熵

xentropy = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=y, logits=logits)

loss = tf.reduce\_mean(xentropy)

# 使用AdamOptimizer

optimizer = tf.train.AdamOptimizer(learning\_rate=learning\_rate)

training\_op = optimizer.minimize(loss)

# 计算准确率，只有等于y才是对的，其他都错

correct = tf.nn.in\_top\_k(logits, y, 1)

accuracy = tf.reduce\_mean(tf.cast(correct, tf.float32))

init = tf.global\_variables\_initializer()

from tensorflow.examples.tutorials.mnist import input\_data

mnist = input\_data.read\_data\_sets("/tmp/data/")

# 转换到合理的输入shape

X\_test = mnist.test.images.reshape((-1, n\_steps, n\_inputs))

y\_test = mnist.test.labels

# run100遍，每次处理150个输入

n\_epochs = 100

batch\_size = 150

# 开始循环

with tf.Session() as sess:

init.run()

for epoch in range(n\_epochs):

for iteration in range(mnist.train.num\_examples // batch\_size):

# 读入数据并reshape

X\_batch, y\_batch = mnist.train.next\_batch(batch\_size)

X\_batch = X\_batch.reshape((-1, n\_steps, n\_inputs))

# X大写，y小写

sess.run(training\_op, feed\_dict={X: X\_batch, y: y\_batch})

acc\_train = accuracy.eval(feed\_dict={X: X\_batch, y: y\_batch})

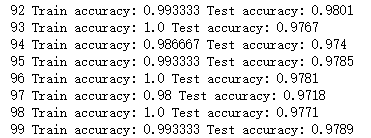
acc\_test = accuracy.eval(feed\_dict={X: X\_test, y: y\_test})

# 每次打印一下当前信息

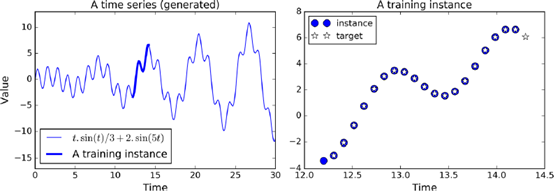
print(epoch, "Train accuracy:", acc\_train, "Test accuracy:", acc\_test)

[复制代码](javascript:void(0);)

以下，只用了150个参数，做了单层。就可以达到非常高的效果，可以看出rnn效果非常不错



序列预测，前20个状态作为输入，则第2个到21个作为输出，作为训练集



[复制代码](javascript:void(0);)

# 输入x0-x19

n\_steps = 20

# 只预测一个值

n\_inputs = 1

# rnn有100个

n\_neurons = 100

n\_outputs = 1

# none表示min\_batch大小这里任意

X = tf.placeholder(tf.float32, [None, n\_steps, n\_inputs])

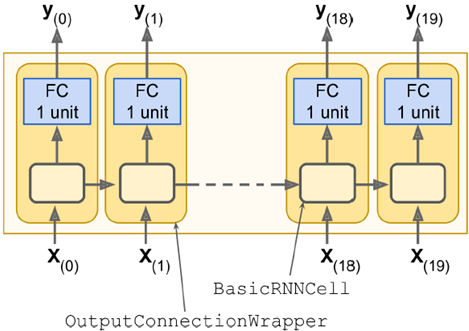
y = tf.placeholder(tf.float32, [None, n\_steps, n\_outputs])

cell = tf.contrib.rnn.BasicRNNCell(num\_units=n\_neurons, activation=tf.nn.relu)

outputs, states = tf.nn.dynamic\_rnn(cell, X, dtype=tf.float32)

[复制代码](javascript:void(0);)

如上代码中，每次输出的vector都是100维的，加入一个output rejections后，使得每次只输出1个值



output rejection实现代码如下

[复制代码](javascript:void(0);)

# 设置输出为上面设定的n\_outputs大小

cell = tf.contrib.rnn.OutputProjectionWrapper(

tf.contrib.rnn.BasicRNNCell(num\_units=n\_neurons, activation=tf.nn.relu), output\_size=n\_outputs)

learning\_rate = 0.001

loss = tf.reduce\_mean(tf.square(outputs - y))

optimizer = tf.train.AdamOptimizer(learning\_rate=learning\_rate)

training\_op = optimizer.minimize(loss)

init = tf.global\_variables\_initializer()

# 开始训练

n\_iterations = 10000

batch\_size = 50

with tf.Session() as sess:

init.run()

for iteration in range(n\_iterations):

X\_batch, y\_batch = [...] # fetch the next training batch

sess.run(training\_op, feed\_dict={X: X\_batch, y: y\_batch})

if iteration % 100 == 0:

mse = loss.eval(feed\_dict={X: X\_batch, y: y\_batch})

print(iteration, "\tMSE:", mse)

[复制代码](javascript:void(0);)

 2.预测

当一个RNN训练好后，它就可以生成很多新的东西。RNN的强大的生成能力非常有魅力，用很多曲子去训练它，它就可以生成新的曲子，用很多文章训练它，他就可以生成新的文章。如果可以训练出功能非常强的RNN模型，就有可能代替人的工作。

[复制代码](javascript:void(0);)

with tf.Session() as sess:

# 导入训练好的模型

saver.restore(sess, "./my\_time\_series\_model")

# 生成新的曲线

sequence = [0.] \* n\_steps

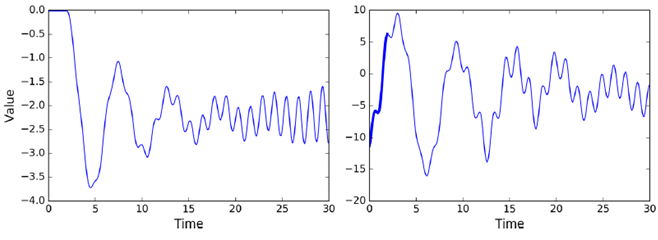
for iteration in range(300):

X\_batch = np.array(sequence[-n\_steps:]).reshape(1, n\_steps, 1)

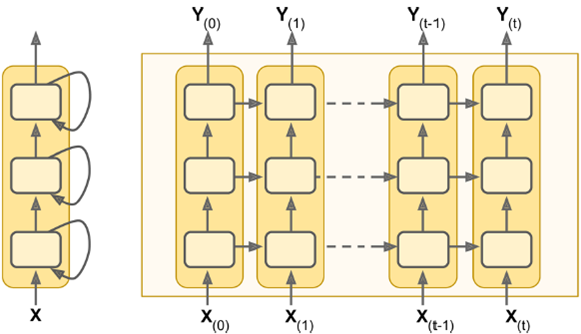
y\_pred = sess.run(outputs, feed\_dict={X: X\_batch})

sequence.append(y\_pred[0, -1, 0])

[复制代码](javascript:void(0);)



RNN也可以不断叠加，形成很深的网络，如下图所示，每一层输出都反馈到当前位置的输入，时间轴展开后，如右边所示。



[复制代码](javascript:void(0);)

n\_inputs = 2

n\_steps = 5

X = tf.placeholder(tf.float32, [None, n\_steps, n\_inputs])

n\_neurons = 100

n\_layers = 3 # 做了3层rnn

# 模型不是越复杂越好，越复杂所需数据量越大，否则会有过拟合的风险

# 可以加dropout来控制

layers = [tf.contrib.rnn.BasicRNNCell(num\_units=n\_neurons)

for layer in range(n\_layers)]

multi\_layer\_cell = tf.contrib.rnn.MultiRNNCell(layers)

outputs, states = tf.nn.dynamic\_rnn(multi\_layer\_cell, X, dtype=tf.float32)

init = tf.global\_variables\_initializer()

X\_batch = np.random.rand(2, n\_steps, n\_inputs)

with tf.Session() as sess:

init.run()

outputs\_val, states\_val = sess.run([outputs, states], feed\_dict={X: X\_batch})

[复制代码](javascript:void(0);)

五、困难及优化

反向训练时，对于RNN来说，要横向往前推，一直往前推到序列开始的地方。当序列非常长时，梯度消失，梯度爆炸都与路径长度太长有关，前面的权重都基本固定不变，没有训练效果。

为了解决这个困难，有了很多更复杂RNN模型的提出

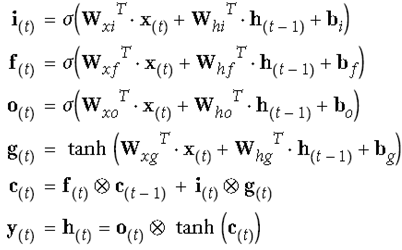
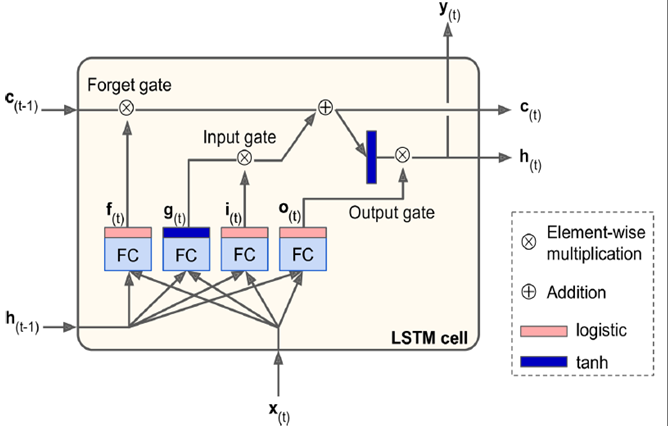
 1.LSTM（Long Short Term Memory）

97年提出，直到深度学习提出，使用LSTM做出具体实事后，才火起来。或许是因为现在有大数据的环境，以及训练能力很强的硬件这些客观条件得具备，才能真正发挥LSTM的威力。

它把训练信息分为长期记忆（c）和短期记忆（h），上面的长期记忆信息，可以穿到很远，即使序列长到1000，也可以向前传导。

它分了很多个门（gate），输出信息趋近于0，门关闭，趋近于1门打开。i是输入门控制新输入加多少到长期记忆中，f是forget控制是否受长期记忆的影响，哪些长期记忆被忘掉，o是输出门控制哪些长期记忆可以输出并作为短期记忆ht传递下去，通过这3个门控制信息的流动。

可以保证长期记忆变换的缓慢，相对稳定，可以对距离比较远的序列影响，ht和ht-1可以看到距离也比较远，短期记忆ht-1变化明显。



[复制代码](javascript:void(0);)

# TensorFlow中LSTM具体实现

n\_steps = 28

n\_inputs = 28

n\_neurons = 150

n\_outputs = 10

n\_layers = 3

learning\_rate = 0.001

X = tf.placeholder(tf.float32, [None, n\_steps, n\_inputs])

y = tf.placeholder(tf.int32, [None])

lstm\_cells = [tf.contrib.rnn.BasicLSTMCell(num\_units=n\_neurons)

for layer in range(n\_layers)]

multi\_cell = tf.contrib.rnn.MultiRNNCell(lstm\_cells)

outputs, states = tf.nn.dynamic\_rnn(multi\_cell, X, dtype=tf.float32)

top\_layer\_h\_state = states[-1][1]

logits = tf.layers.dense(top\_layer\_h\_state, n\_outputs, name="softmax")

xentropy = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=y, logits=logits)

loss = tf.reduce\_mean(xentropy, name="loss")

optimizer = tf.train.AdamOptimizer(learning\_rate=learning\_rate)

training\_op = optimizer.minimize(loss)

correct = tf.nn.in\_top\_k(logits, y, 1)

accuracy = tf.reduce\_mean(tf.cast(correct, tf.float32))

init = tf.global\_variables\_initializer()

[复制代码](javascript:void(0);)

LSTM还有一点改进Peephole Connection

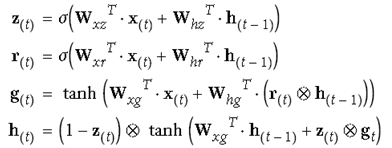
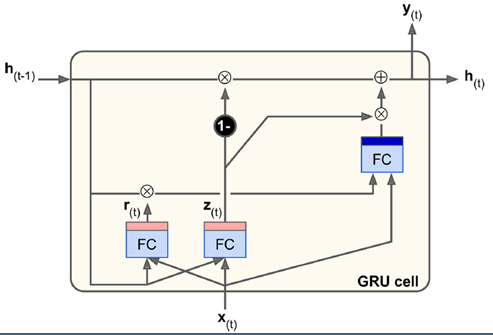
 lstm\_cell = tf.contrib.rnn.LSTMCell(num\_units=n\_neurons, use\_peepholes=True)

 2. GRU（Gated recurrent unit）

 GRU是对LSTM简化后的版本，去掉了长短期记忆的区分（都是h），减少了几个门，2014年提出，从参数上来说较LSTM简单些。

统一用update gate控制原来的i门和f门。z趋近于0就用ht-1来更新，趋近于1就取当前输入。

比LSTM还少一个矩阵乘法，实际表现不比LSTM差，也成为现在很多研究者越来越看重的方法。



调用时，直接调用GRU cell即可

 gru\_cell = tf.contrib.rnn.GRUCell(num\_units=n\_neurons)

 六、RNN在NLP（natural language processing）中的应用

RNN的输入原本是one-hot的表示，但这样会使得输入极其稀疏，不好训练。于是将高维空间映射到低维（如100维）空间，用这个低维嵌入的输入做训练，非常有效。

Word Embeddings

相同含义的词在低维空间中距离近，含义差的多的离得远。

[复制代码](javascript:void(0);)

# 把50000维数据映射到150维数据空间上

vocabulary\_size = 50000

embedding\_size = 150

# 做一个全连接

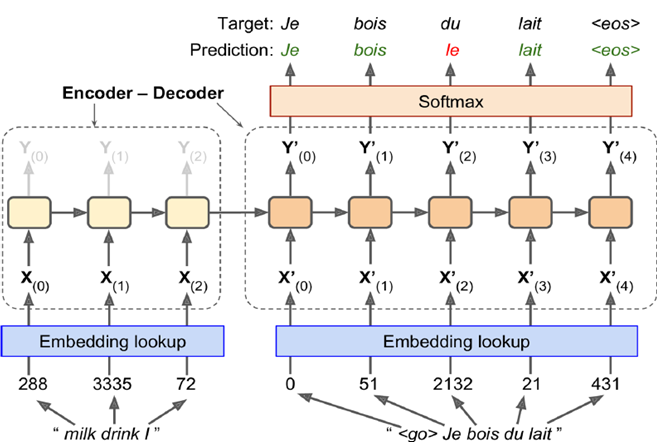
embeddings = tf.Variable(tf.random\_uniform([vocabulary\_size, embedding\_size], -1.0, 1.0))

train\_inputs = tf.placeholder(tf.int32, shape=[None]) # from ids

embed = tf.nn.embedding\_lookup(embeddings, train\_inputs)# to embd

[复制代码](javascript:void(0);)

例如下图所示，把要翻译的英文句子做输入，用训练后的状态值做输入，和法语作为训练集的作为decoder输入。第一位放一个起始信号<go>，输出和输入刚好错一位，最后一位以一个结束标识<eos*>结束。这样做是为了后继应用时，翻译新句子没有training label，只有英文输入。把英文输入放进来后加一个<go>，得到第一个je输出，把第一个词放进来得到第二个输出bois。。最后<eos>翻译结束。*



最后实际应用时如下，输入<go>开始翻译。

