

1. 了解linear regression, logistic regression 和 softmax

1.1 了解Linear Regression

对于一个被研究的物体，它有多个属性(x_1, x_2, \dots, x_n)和一个值 y 。线性回归假设 y 与(x_1, x_2, \dots, x_n)有线性关系，

$$y = \theta + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$$

也就是我们可以把 y 表示成关于(x_1, x_2, \dots, x_n)的函数 其中($\theta, \theta_1, \theta_2, \dots$)都是常数。线性回归就是要找到最合适的($\theta, \theta_1, \theta_2, \dots$)使得这个函数离真实分布的偏差尽量小，尽量“贴切”真实的分布，让**损失**最小化。

于是这就引出两个问题：

1. 怎么定义这个与真实分布的偏差？
2. 如何去靠近这个真实分布？

这两个问题的答案就是

1. 选择合适的**损失函数**，此处使用距离方差。
2. 选择合适的**优化策略**，有最小二乘法和梯度下降。

1.1.1 距离方差

$$C = \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

距离方差的定义是：
差的平方，再把这些平方求和。 [为什么损失函数要用平方距离](#)

即对每一组数据求真实值与预测值的

1.1.2 优化策略：最小二乘法

本来的表达式 $y = WX + b$ ，为便于研究，接下来我们试图把转换为 $y = WX$ (或 $y = XW$ ，总之要去掉 b 这个“小尾巴”)的形式。

1. 我们把 W 与 b 吸收为一个新的向量 $\hat{w} = (w; b)$

$$\mathbf{X} = \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1d} & 1 \\ x_{21} & x_{22} & \dots & x_{2d} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{m1} & x_{m2} & \dots & x_{md} & 1 \end{pmatrix} = \begin{pmatrix} \mathbf{x}_1^T & 1 \\ \mathbf{x}_2^T & 1 \\ \vdots & \vdots \\ \mathbf{x}_m^T & 1 \end{pmatrix},$$

2. 把X转为

3. 这样就可以表达 $y = \mathbf{XW}$ 了

$$\hat{\mathbf{w}}^* = \arg \min_{\hat{\mathbf{w}}} (\mathbf{y} - \mathbf{X}\hat{\mathbf{w}})^T (\mathbf{y} - \mathbf{X}\hat{\mathbf{w}}).$$

那么最合适的W可以表达为

令 $E_{\hat{\mathbf{w}}} = (\mathbf{y} - \mathbf{X}\hat{\mathbf{w}})^T (\mathbf{y} - \mathbf{X}\hat{\mathbf{w}})$, 对 $\hat{\mathbf{w}}$ 求导得到

$$\frac{\partial E_{\hat{\mathbf{w}}}}{\partial \hat{\mathbf{w}}} = 2 \mathbf{X}^T (\mathbf{X}\hat{\mathbf{w}} - \mathbf{y}).$$

令上式得0可得

$$\hat{\mathbf{w}}^* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y},$$

如果 $(\mathbf{X}^T \mathbf{X})$ 满秩, 则有逆, 就可以求出 \mathbf{W}^* 。

但它的缺点在于, 现实任务中 $(\mathbf{X}^T \mathbf{X})$ 往往不是满秩, 就需要**正则化(regularization)**来使其有逆, 也就是为这个矩阵的加一个偏差。但即使是很小的偏差, 有时候都会导致预测结果的较大变化。所以, 梯度下降的策略更稳定可行。

1.1.3 优化策略：梯度下降

这篇文章比较好地解释了梯度下降的机制。

- [机器学习笔记-线性回归与梯度下降](#)。梯度下降有随机梯度下降与批处理梯度下降, 这篇文章以linear regression为例, 比较了两种梯度下降算法。
- [机器学习-随机梯度下降 \(Stochastic gradient descent\)](#) 和 [批量梯度下降 \(Batch gradient descent\)](#)

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (y^i - h_{\theta}(x^i))^2$$

1. 首先, 损失函数是距离差的平方, 如上文所说。

2. 接下来要对每个变量求偏导 对于一个样本的误差(随机梯度下降用到它):

$$\begin{aligned}\frac{\partial}{\partial \theta_j} J(\theta) &= \frac{\partial}{\partial \theta_j} \frac{1}{2} (h_{\theta}(x) - y)^2 \\ &= 2 \cdot \frac{1}{2} (h_{\theta}(x) - y) \cdot \frac{\partial}{\partial \theta_j} (h_{\theta}(x) - y) \\ &= (h_{\theta}(x) - y) \cdot \frac{\partial}{\partial \theta_j} \left(\sum_{i=0}^n \theta_i x_i - y \right) \\ &= (h_{\theta}(x) - y) x_j\end{aligned}$$

$$\frac{\partial J(\theta)}{\partial \theta_j} = -\frac{1}{m} \sum_{i=1}^m (y^i - h_{\theta}(x^i)) x_j^i$$

对于所有m个样本的误差(批量梯度下降用到它):

3. 更新 θ_j 时, 用它的偏导数乘以步长, 就是对应 θ_j 的变化程度了。设偏导数在该点的值为 ∂ , 则 $\theta_j = \theta_j - \alpha \partial$, 其中 ∂ 用上一步的式子求得。于是 θ_j 可以更新为: 对于一个样本点(随机梯度下降用到它):

$$\theta_j := \theta_j - \alpha (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

对于多个样本点(批量梯度下降用到它):

$$\theta_j := \theta_j - \alpha \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

4. 批量梯度下降的计算量过大, 每次都要计算所有m组数据。我们引入**随机梯度下降**: 每次只计算一组样本, 每一次迭代不一定要用所有样本, 可以只取其中m个。公式如下:

```
Loop {  
    for i=1 to m, {  
         $\theta_j := \theta_j + \alpha (y^{(i)} - h_{\theta}(x^{(i)})) x_j^{(i)}$     (for every j)  
    }  
}
```

5. 我们让他迭代一定次数, 比如1500次, 只要次数足够多, 迭代停止时应该是达到最优的。另外可以设一个变量, 监控theta, theta变化小于某一个很小的数的时候停止迭代。

1.1.4 两种优化策略的比较

参考：[机器学习知识体系 - 线性回归](#)

梯度下降算法	正规方程
需要选择学习速率参数	不需要学习速率参数
需要很多次迭代	不需要迭代
$O(kn^2)$	$O(n^3)$
n如果很大依旧还能工作	n如果很大，速度会非常慢

1.1.5 编程实现

```
import tensorflow as tf
import numpy as np

# 随机生成一批训练数据
#  $y = 2 * x + 10$ 
# np.random.randn标准正态分布
train_X = np.linspace(-1, 1, 100)
train_Y = 5 * train_X + np.random.randn(*train_X.shape) * 0.33 + 10 # *的作用是解包

# 构建计算模型
W = tf.Variable(tf.random_normal([1]), name='weight')
b = tf.Variable(tf.random_normal([1]), name='bias')

X = tf.placeholder(tf.float32, shape=[None]) # 只有一维，第一维长度随意
Y = tf.placeholder(tf.float32) # 默认一维

# 构建计算模型
hypothesis = X * W + b
cost = tf.reduce_mean(tf.square(hypothesis - Y))
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01)
train = optimizer.minimize(cost) # 反向传播

sess = tf.Session()
sess.run(tf.global_variables_initializer())

# 迭代2000次
for step in range(2001):
    cost_val, W_val, b_val, _ = sess.run([cost, W, b, train], feed_dict={X: train_X, Y: train_Y})
    if step % 40 == 0:
        print(step, cost_val, W_val, b_val)
```

1.1.6 运行结果

如图，当运行到2000次时，两个参数已经约等于5和10，误差仅有0.119

1040	0.119697	[4.98574018]	[10.01683235]
1080	0.119696	[4.9865613]	[10.01683235]
1120	0.119695	[4.98718643]	[10.01683235]
1160	0.119694	[4.98766184]	[10.01683235]
1200	0.119694	[4.98802376]	[10.01683235]
1240	0.119693	[4.98829937]	[10.01683235]
1280	0.119693	[4.98850918]	[10.01683235]
1320	0.119693	[4.98866844]	[10.01683235]
1360	0.119693	[4.98879099]	[10.01683235]
1400	0.119693	[4.98888206]	[10.01683235]
1440	0.119693	[4.98895216]	[10.01683235]
1480	0.119693	[4.98900747]	[10.01683235]
1520	0.119693	[4.98904562]	[10.01683235]
1560	0.119693	[4.98907852]	[10.01683235]
1600	0.119693	[4.9890976]	[10.01683235]
1640	0.119693	[4.98911667]	[10.01683235]
1680	0.119693	[4.98913574]	[10.01683235]
1720	0.119693	[4.98914289]	[10.01683235]
1760	0.119693	[4.98914289]	[10.01683235]
1800	0.119693	[4.98914289]	[10.01683235]
1840	0.119693	[4.98914289]	[10.01683235]
1880	0.119693	[4.98914289]	[10.01683235]
1920	0.119693	[4.98914289]	[10.01683235]
1960	0.119693	[4.98914289]	[10.01683235]
2000	0.119693	[4.98914289]	[10.01683235]

1.2 了解Logistic Regression

这篇文章解释的比较好 [Logistic Regression（逻辑回归）原理及公式推导](#)

我们知道，线性回归的公式如下：

$$z = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 \dots + \theta_n x_n = \theta^T x$$

而对于Logistic

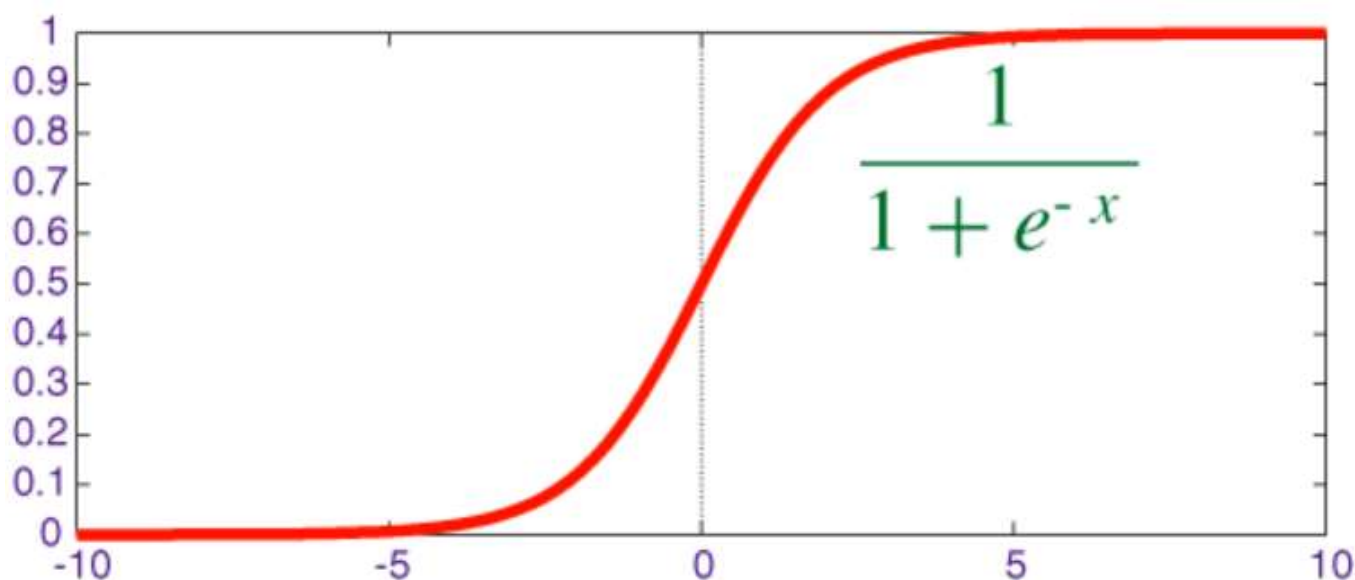
Regression来说，其思想也是基于线性回归（Logistic Regression属于广义线性回归模型）。其公式如下：

$$h_{\theta}(x) = \frac{1}{1 + e^{-z}} = \frac{1}{1 + e^{-\theta^T x}} \quad y = \frac{1}{1 + e^{-x}}$$

其中，

被称作

sigmoid函数，我们可以看到，Logistic Regression算法是将线性函数的结果映射到了sigmoid函数中。图形如下：



我们把函数的值视为物体属于某一类别的概率，当 $h_{\theta}(x) > 0.5$ ，我们把它分到A类，当 $h_{\theta}(x) < 0.5$ ，我们把它分到B类。

1.2.1 极大似然估计求损失函数

(该推论在博客中也有详细解释) 首先我们知道概率的表示为

$$P(y|x; \theta) = (h_{\theta}(x))^y * (1 - h_{\theta}(x))^{1-y}$$

那么m个样本的概率分布为

$$L(\theta) = \prod_{i=1}^m (h_{\theta}(x^{(i)}))^y * (1 - h_{\theta}(x^{(i)}))^{1-y}$$

取对数似然函数：

$$l(\theta) = \log(L(\theta)) = \sum_{i=1}^m \log((h_{\theta}(x^{(i)}))^y) + \log((1 - h_{\theta}(x^{(i)}))^{1-y})$$

$$l(\theta) = \log(L(\theta)) = \sum_{i=1}^m y^{(i)} \log(h_{\theta}(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))$$

最大似然估计

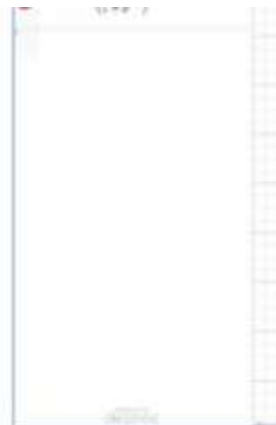
就是要求得使 $l(\theta)$ 取最大值时的 θ ，这里可以使用梯度上升法求解。我们稍微变换一下：

$$J(\theta) = -\frac{1}{m} l(\theta)$$

因为乘了一个负的系数 $-1/m$ ，然后就可以使用**梯度下降算法**进行参数求解了。于

是损失函数与梯度下降的函数都如下所示

Logistic Regression



$$H(X) = \text{sigmoid}(XW) = \frac{1}{1 + e^{-XW}}$$

$$\text{cost}(W) = -\frac{1}{m} \sum \underbrace{y \log(H(x))}_{\text{Case for } y=1} + \underbrace{(1-y) \log(1-H(x))}_{\text{Case for } y=0}$$

Case for y=0



$$W := W - \alpha \frac{\partial}{\partial W} \text{cost}(W)$$

1.2.2 编程实现

```
import tensorflow as tf
import numpy as np

# 生成随机数据
# x为(-10, -10), (-10, -9) 到 (4, 4)
# 分界为 3 * x1 + 4 * x2 + 5 = 0
x_data = []
y_data = []
for i1 in range(15):
    for i2 in range(15):
        x1 = i1 - 10
        x2 = i2 - 10
        x_data.append([x1, x2])
        y = int(3 * x1 + 4 * x2 + 5 > 0)
        y_data.append([y])

# 构建变量与占位符
X = tf.placeholder(tf.float32, shape=[None, 2])
Y = tf.placeholder(tf.float32, shape=[None, 1])

W = tf.Variable(tf.random_normal([2, 1]), name='weight')
b = tf.Variable(tf.random_normal([1]), name='bias')

# 构建模型
hypothesis = tf.sigmoid(tf.matmul(X, W) + b)
cost = -tf.reduce_mean(Y * tf.log(hypothesis) + (1 - Y) * tf.log(1 - hypothesis))
train = tf.train.GradientDescentOptimizer(learning_rate=0.001).minimize(cost)
predicted = tf.cast(hypothesis > 0.5, dtype=tf.float32)
accuracy = tf.reduce_mean(tf.cast(tf.equal(predicted, Y), dtype=tf.float32))

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
```

```
#迭代训练
for step in range(10001):
    cost_val, _ = sess.run([cost, train], feed_dict={X: x_data, Y: y_data})
    if step % 200 == 0:
        print(step, cost_val)

# 预测参数与准确率
h, c, a = sess.run([hypothesis, predicted, accuracy],
                    feed_dict={X: x_data, Y: y_data})
print("\nHypothesis: ", h, "\nCorrect (Y): ", c, "\nAccuracy: ", a)
```

1.2.3 运行结果

[illegible]

1.2.4 关于计算过程出现nan问题

```
0 nan
200 nan
400 nan
600 nan
```

最初是打算在(-100, -100) 到 (100, 100)上生成一万组x的数据, 结果出现了

查阅了 [Tensorflow 实现稠密输入数据的逻辑回归二分类](#)，文中提到

注意如果设置的batch_size 比较大 而learning rate也比较大 可能会出现nan， 可以通过减小batch_size 或者调小learning rate来避免


```

0 1.65092
200 0.0860151
400 0.0654475
600 0.0554888
800 0.049398
1000 0.0451773
1200 nan
1400 nan
1600 nan
1800 nan

```

原来是一批处理的数据量太大了，改成15 * 15组数据后，仍然有

```

[[ 1.]
 [ 1.]
 [ 1.]
 [ 1.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 1.]
 [ 1.]
 [ 1.]
 [ 1.]
 [ 1.]
 [ 1.]
 [ 1.]
 [ 1.]
 [ 1.]
 [ 1.]]
Accuracy: 0.982222

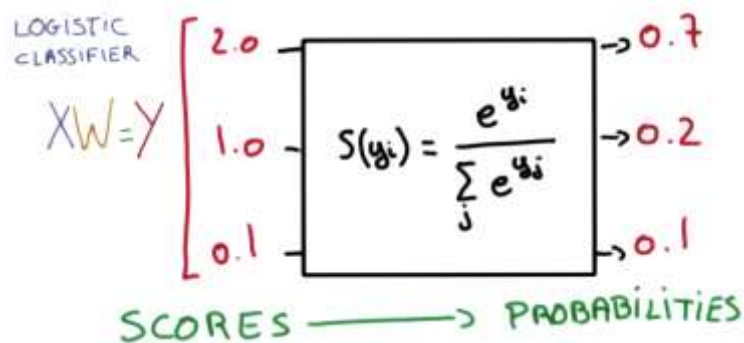
```

根据上文，我推测是学习率0.1太大了，于是改成了0.01，当数据量适中，且学习率足够小，代码可以顺利运行

当数据量适中，且学习率

1.3 了解softmax回归

softmax适用于多分类问题，比如有3个分类，就为它们分别计算一个值，并把它们通过



映射区间为(0, 1)且和为1的值上，可以视为某种"概率"，但不是真正的概率。它的意义在于相互比较，得到最可能的类别。其损失函数是交叉熵(cross entropy loss)，表示了信息的混乱程度。运算公式为 `cost = tf.reduce_mean(-tf.reduce_sum(Y * tf.log(hypothesis), axis=1))`

1.3.1 编程实现

因为MINST中的数字识别问题就是一个多分类问题，我们以MINST作为数据源进行编程。

```

import tensorflow as tf
import numpy as np
import tensorflow.examples.tutorials.mnist.input_data as input_data

```

```

# 读取MNIST数据集
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
#trX, trY, teX, teY = mnist.train.images, mnist.train.labels, mnist.test.images, mnist.test.labels

# 一副图像有784个像素
X = tf.placeholder(tf.float32, [None, 784])
Y = tf.placeholder(tf.float32, [None, 10]) # 正确分类

W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))

# 构建计算图
y = tf.nn.softmax(tf.matmul(X, W) + b) # 预测分类
cross_entropy = -tf.reduce_sum(Y * tf.log(y))
optimizer = tf.train.GradientDescentOptimizer(0.01).minimize(cross_entropy)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    # 每次迭代读取128张图, 迭代1000轮
    for i in range(1000):
        batch_x, batch_y = mnist.train.next_batch(128)
        sess.run(optimizer, feed_dict={X: batch_x, Y: batch_y})

    # 计算准确率
    correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(Y, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
    print(sess.run(accuracy, feed_dict={X: mnist.test.images, Y: mnist.test.labels}))

```

1.3.2 执行结果

```
(py36) C:\Users\jj\Desktop\科研实训\week3 CNN>python softmax_regression_mnist.py
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
2018-04-09 03:54:55.634458: W c:\1\tensorflow_1501918863922\work\tensorflow-1.2.1\t
rd.cc:45] The TensorFlow library wasn't compiled to use SSE instructions, but these
d speed up CPU computations.
2018-04-09 03:54:55.640442: W c:\1\tensorflow_1501918863922\work\tensorflow-1.2.1\t
rd.cc:45] The TensorFlow library wasn't compiled to use SSE2 instructions, but thes
ld speed up CPU computations.
2018-04-09 03:54:55.645252: W c:\1\tensorflow_1501918863922\work\tensorflow-1.2.1\t
rd.cc:45] The TensorFlow library wasn't compiled to use SSE3 instructions, but thes
ld speed up CPU computations.
2018-04-09 03:54:55.649639: W c:\1\tensorflow_1501918863922\work\tensorflow-1.2.1\t
rd.cc:45] The TensorFlow library wasn't compiled to use SSE4.1 instructions, but th
ould speed up CPU computations.
2018-04-09 03:54:55.654324: W c:\1\tensorflow_1501918863922\work\tensorflow-1.2.1\t
rd.cc:45] The TensorFlow library wasn't compiled to use SSE4.2 instructions, but th
ould speed up CPU computations.
2018-04-09 03:54:55.658942: W c:\1\tensorflow_1501918863922\work\tensorflow-1.2.1\t
rd.cc:45] The TensorFlow library wasn't compiled to use AVX instructions, but thes
d speed up CPU computations.
2018-04-09 03:54:55.664231: W c:\1\tensorflow_1501918863922\work\tensorflow-1.2.1\t
rd.cc:45] The TensorFlow library wasn't compiled to use AVX2 instructions, but thes
ld speed up CPU computations.
2018-04-09 03:54:55.668363: W c:\1\tensorflow_1501918863922\work\tensorflow-1.2.1\t
rd.cc:45] The TensorFlow library wasn't compiled to use FMA instructions, but thes
d speed up CPU computations.
0.9099
```

2.用简单CNN解决MNIST数字识别问题

2.1 理解CNN

卷积神经网络CNN原理以及TensorFlow实现 形象地描述了CNN的大致流程

另外这篇文章也很好: <https://blog.csdn.net/u013082989/article/details/53673602>

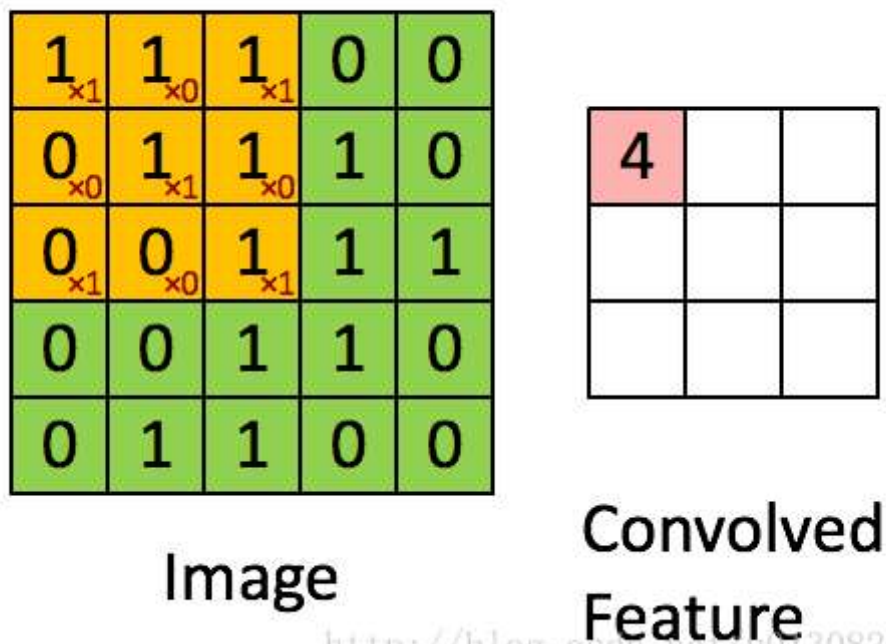
2.1.1 理解卷积层

在接收到图片数据方阵时,我们用卷积层对其“特征”进行提取,生成一个新的数据方阵。接收的图片数据方阵一般是两种形式,有一个或三个Channel:

1. 标准的数码相机有红、绿、蓝三个通道 (Channels), 每一种颜色的像素值在0-255之间, 构成三个堆叠的二维矩阵, 大小为 $\text{width} * \text{height} * 3$
2. 灰度图像则只有一个通道, 可以用一个二维矩阵来表示。大小为 $\text{width} * \text{height}$

2.1.1.1 卷积操作

1. 下图是channel为1时，3x3的卷积核(又称过滤器，filter)在5x5的图像上做的一次卷积操作，就是矩阵做



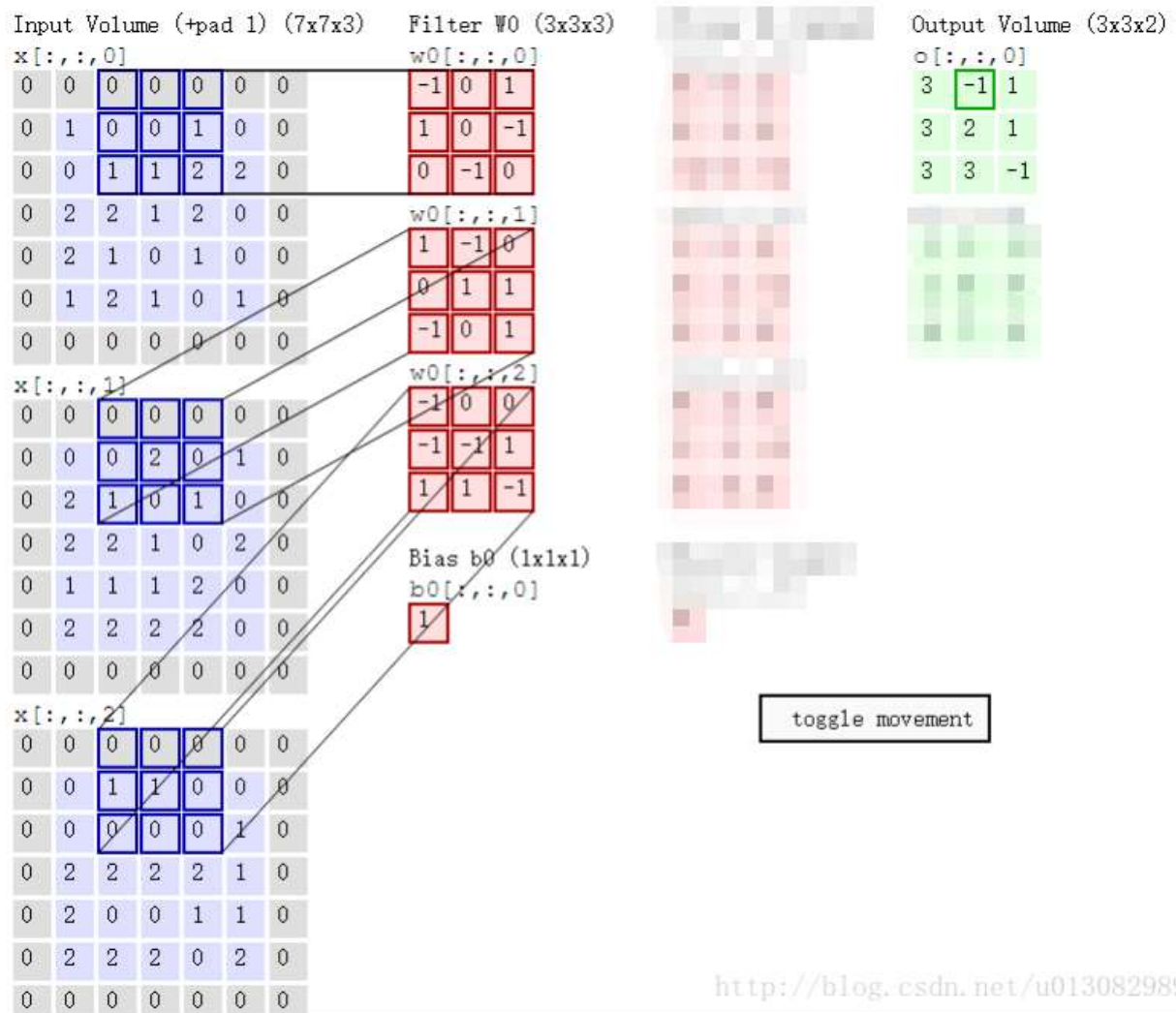
点乘之后的和。

$$W_i x_{small} + b_i$$

第 i 个隐含单元的输入就是：

，其中 x_{small} 就时与过滤器filter过滤到的图片局部区域。上图中，filter为[[1,0,1], [0,1,1], [0,0,1]]，b为0，上图的点乘结果为4，再加上0的偏置，得到结果为4。

2. 当channel为3时，则对应的filter过滤器也是三维的。下图是只有一个过滤器，channel为3的情况。



则卷积操作是

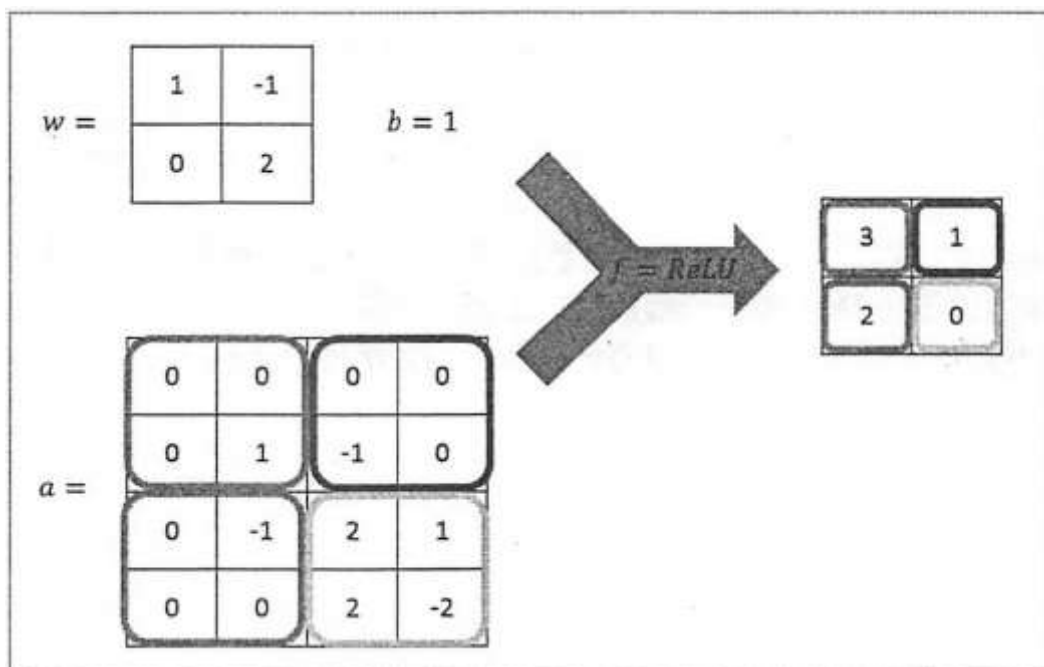
- 让每一个channel上的该图片区域 与 filter中对应的矩阵作点乘
- 把三个点乘的结果加起来
- 加上偏置：

上图的具体操作是：与 $w0[:, :, 0]$ 卷积： $0 \times (-1) + 0 \times 0 + 0 \times 1 + 0 \times 1 + 0 \times 0 + 1 \times (-1) + 1 \times 0 + 1 \times (-1) + 2 \times 0 = -2$ 与 $w0[:, :, 1]$ 卷积： $2 \times 1 + 1 \times (-1) + 1 \times 1 = 2$ 与 $w0[:, :, 2]$ 卷积： $1 \times (-1) + 1 \times (-1) = -2$ 最终结果： $-2 + 2 + (-2) + 1 = -1$ (1为偏置)

2.1.1.2 经过ReLU神经元激活

一次卷积操作得到的值还要经过ReLU神经元进行激活，于是 $g(-1) = 0$ 就是卷积层对矩阵这个位置上处理的最终结果。

一次完整的传播过程如下图所示。我们取第一块区域为例，卷积操作得到 1×2 (点乘结果) + 1 (偏置) = 3，然后经过ReLU函数，得到最终结果为3。



2.1.1.3 其它

- 滑动的步长-stride 上面(2.1.1.2)那张图片从左到右，每次滑动的时候移动了两格，stride为2。但是其实它一次滑动多格，这就是步长。
- 卷积的边界处理-padding 原本的图片是3 * 3的，而过滤器是 2 * 2，这样卷积操作出来的图片就缩小为2 * 2了，比原来的图片要小了。

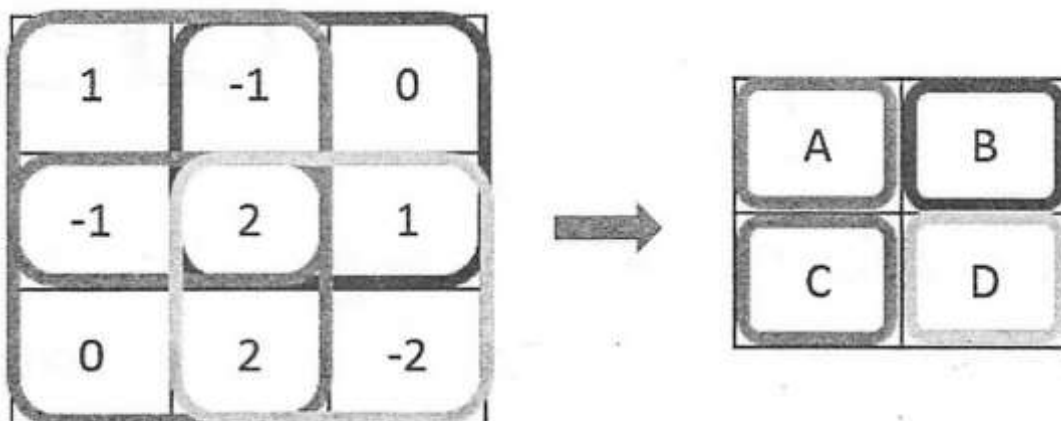


图 6-10 卷积层前向传播过程示意图

所以要考虑这个边界的问题。卷积的边界处理有两种方式：一、丢掉边界，也就是就按右边那个缩小的矩阵来。二、使用全0填充，如下图所示

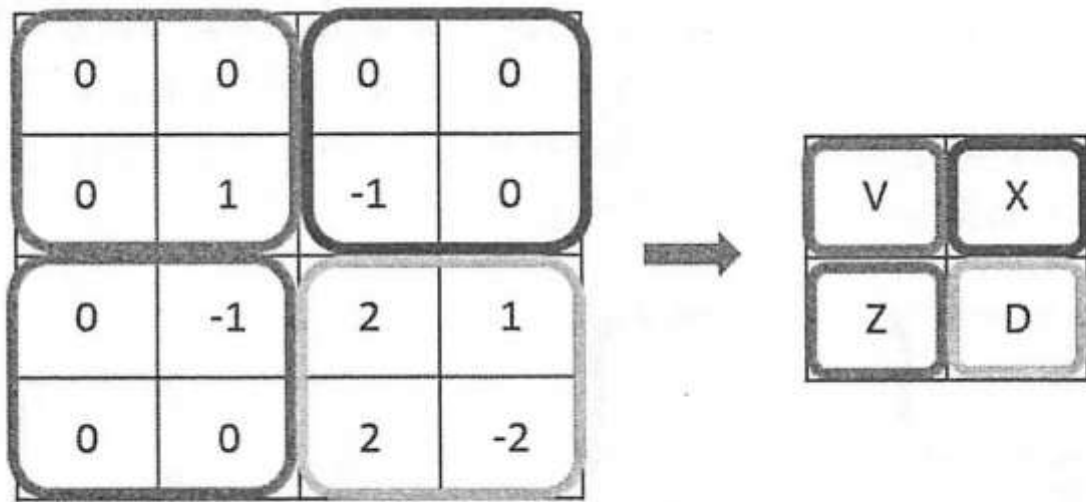


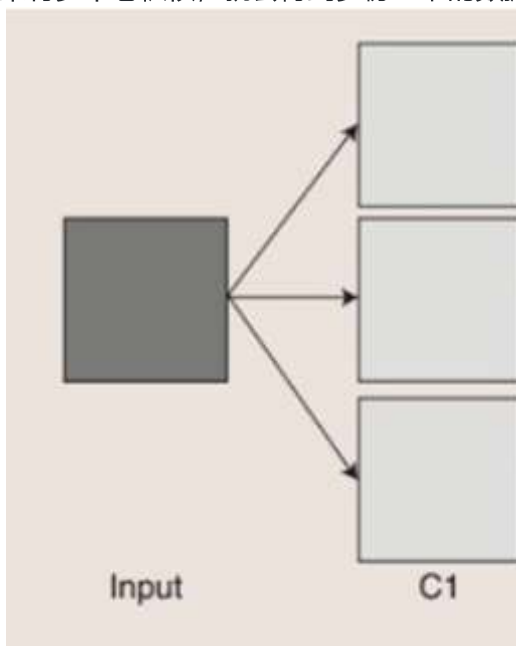
图 6-12 过滤器移动步长为 2 且使用全 0 填充时卷积层前向传播过程示意图

一般

来说，卷积操作的意义不在于压缩图片，所以我们倾向于用全0填充

2.1.1.4 多个卷积核的情形

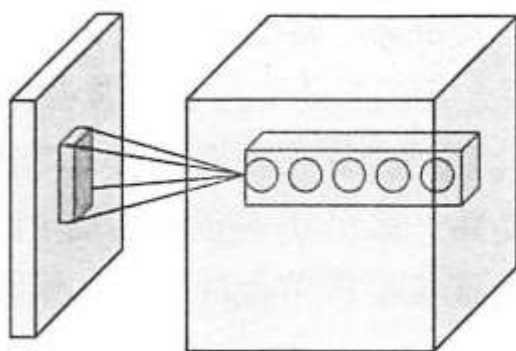
从上文中的图"2.1.1.1 一个卷积核且channel为3的情形"我们可以知道，不管channel是几，**一个卷积核**都会对图片处理得到一份**二维的数据**。那么如果有多个卷积核，就会得到多份二维的数据。如下图，就是一副图片经



过**三个卷积核**得到了三组二维的数据。

但有时文章会把这三个卷积

核认为是一个整体，称为一个**具有深度的卷积核**。所以上文一共用到了三个卷积核，我们也可以称为用到了一



个**深度为3的卷积核**。

2.1.1.5 总结

有了上文，我们再次回顾整个卷积层处理的过程 首先将图片分割成如下图的重叠的独立小块；下图中，这张照

Input Image



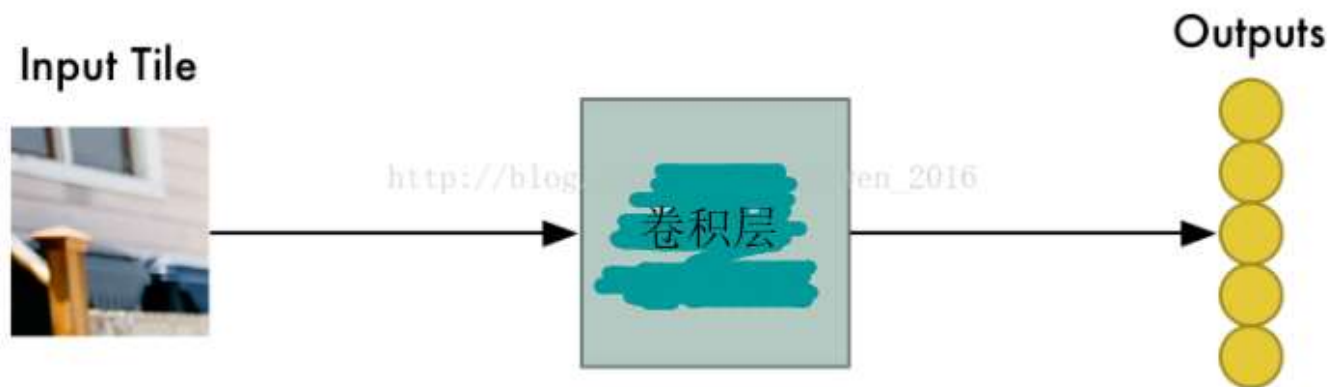
片被分割成了77张大小相同的小图片。



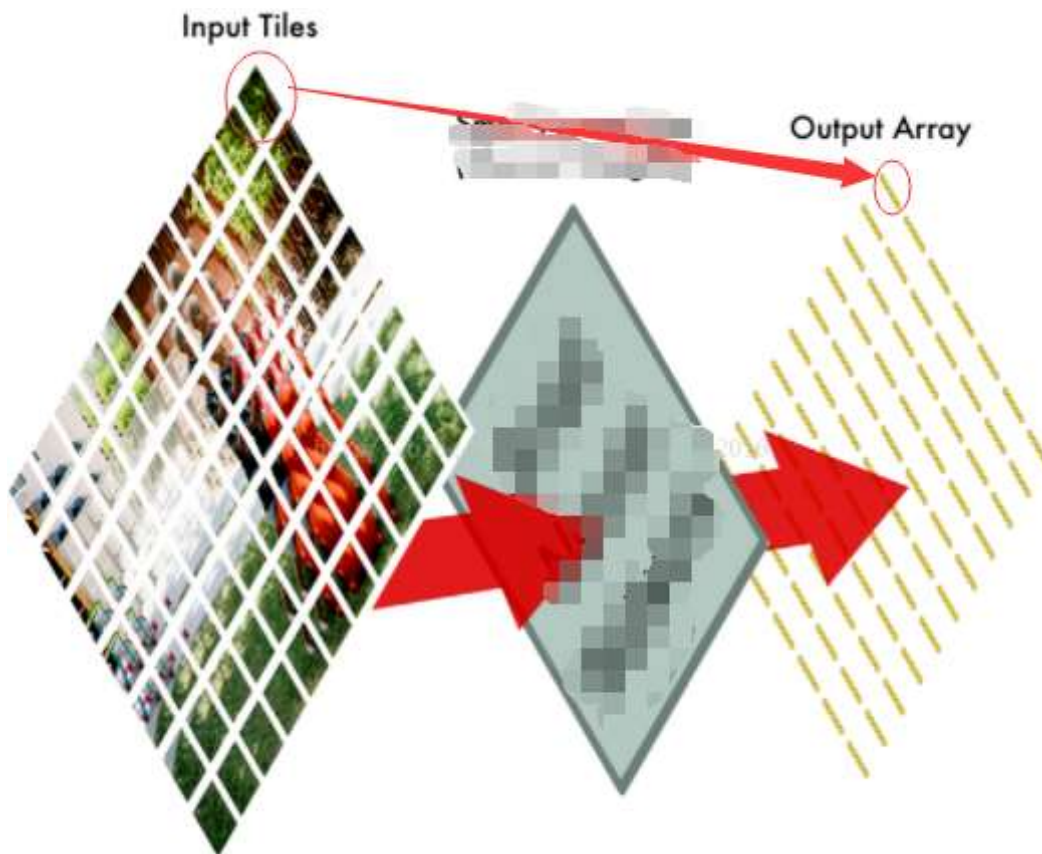
然后，对每一

块图片，我们用一个卷积层来处理。下图中的卷积层的深度为5，有就是有5个卷积核。每个卷积核与图片进行卷积操作并经过ReLU激活后，得到一个值。

Processing a single tile



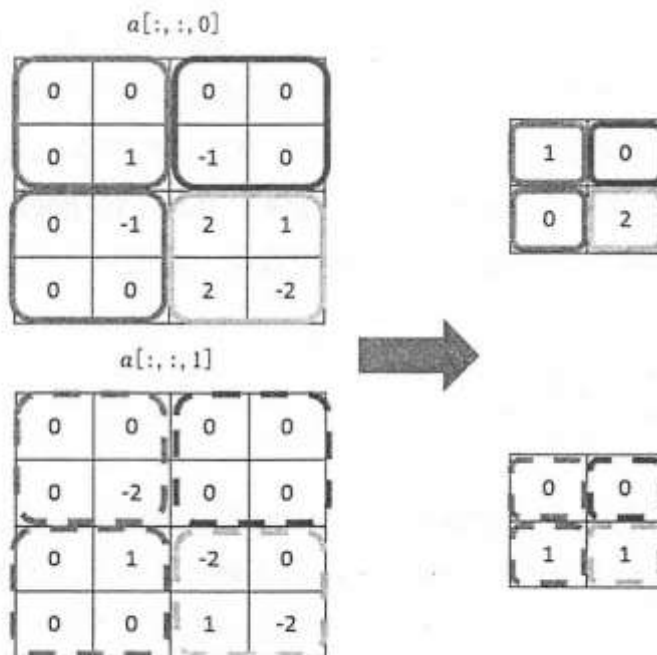
将所有的独立小块**分别**进行**卷积**和**激活**操作后，再将每一个输出的特征数组按照第一步时77个独立小块的相对位置做排布，就得到一个新数组。



这就是一个完整的卷积层所做的事情了

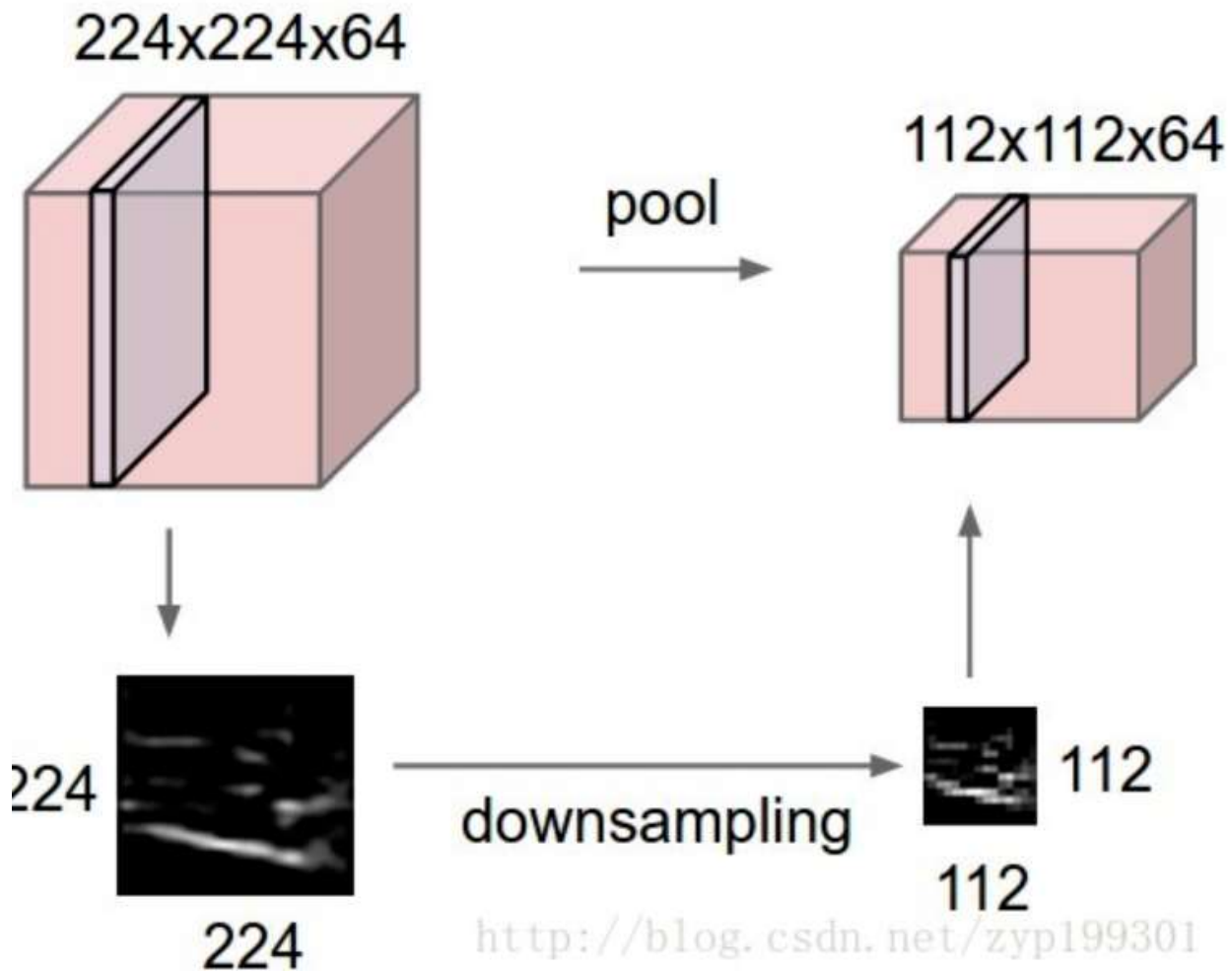
2.1.2 理解池化层

池化就是对卷积层处理后得到的信息进行压缩，有最大池化(max pooling)和平均池化(average pooling)。前者就是在一个范围内找最大值，后者就是在范围内算平均值。实践发现最大池化的表现更好，现在一般都采用 max pooling。下图是 $3 \times 3 \times 2$ 的矩阵经过最大池化的过程示意图。可以得知在上一层的卷积层的过滤器的深度为2。



度为2。 $3 \times 3 \times 2$ 节点矩阵经过全 0 填充且步长为 2 的最大池化层前向传播过程示意图。下图是 224×224

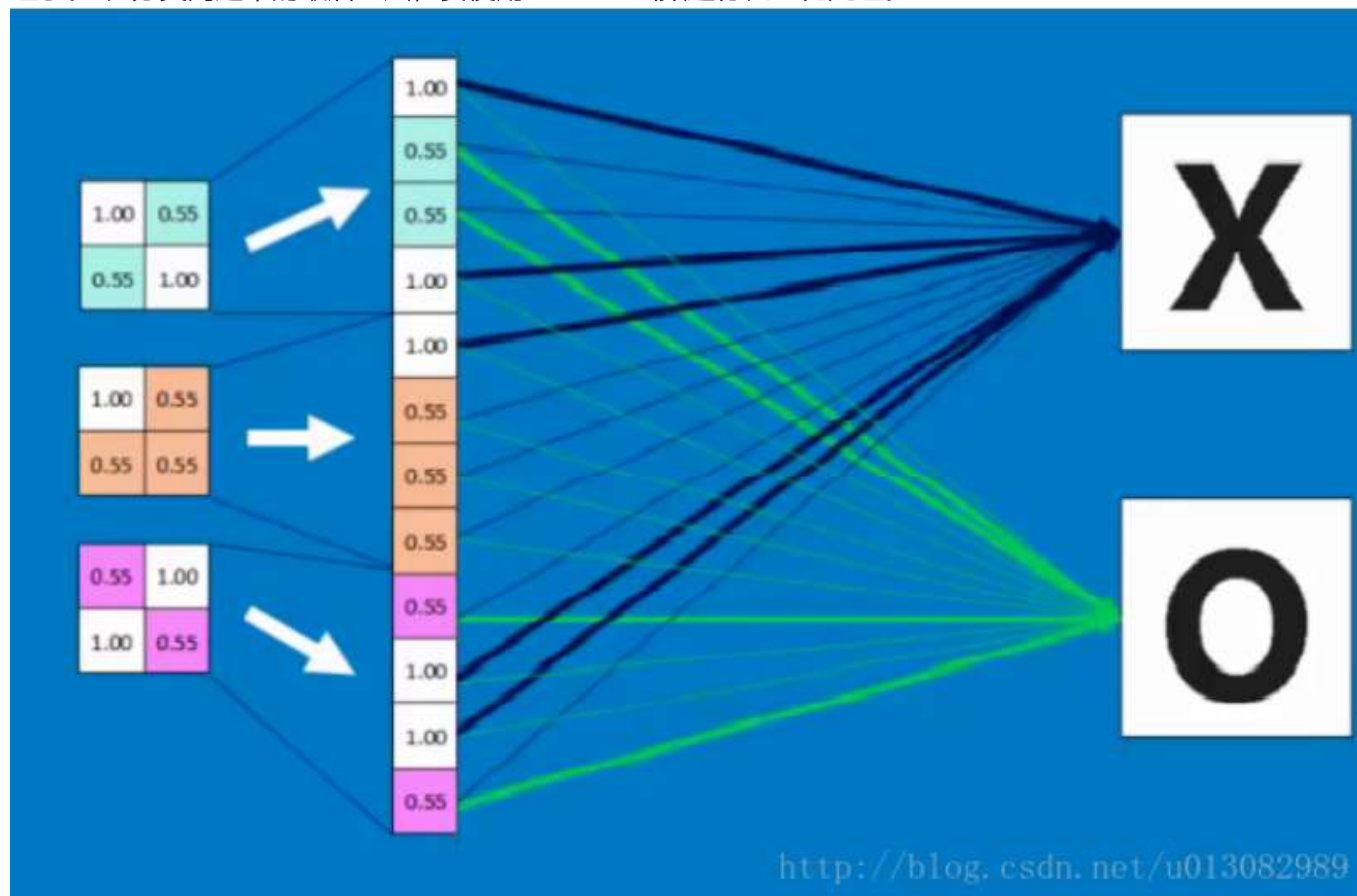
* 64的矩阵经过最大池化的过程示意图。可以得知在上一层的卷积层的过滤器的深度为64。



2.1.3 理解全连接层

将多次卷积和池化后的图像展开进行全连接，如下图所示。之所以要放在卷积和池化之后，是因为全连接层的每个神经元都与所有输入相关联，**如果不对图片进行“压缩”，其中的神经元的连接数量是巨大的**。通过对图片多次卷积和池化后，最后留下的矩阵大小已大大缩小，且保留了原图片的特征，于是就可以使用全连接层处

理了。在分类问题中的最后一层，要使用softmax函数进行归一化处理。



2.2 用CNN编程解决MNIST数字识别问题

2.2.1 代码

```
import tensorflow as tf
import tensorflow.examples.tutorials.mnist.input_data as input_data

mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)  # 下载并加载mnist数据
x = tf.placeholder(tf.float32, [None, 784])  # 输入的数据占位符
y_actual = tf.placeholder(tf.float32, shape=[None, 10])  # 输入的标签占位符

# 定义一个函数，用于初始化所有的权值 w
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

# 定义一个函数，用于初始化所有的偏置项 b
def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)

# 定义一个函数，用于构建卷积层
def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

# 定义一个函数，用于构建池化层
def max_pool(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
```

```

#构建网络
x_image = tf.reshape(x, [-1,28,28,1])      # 转换输入数据shape,以便于用于网络中
W_conv1 = weight_variable([5, 5, 1, 32])   # 深度为32的随机权重矩阵
b_conv1 = bias_variable([32]) # 初始偏差为全1的偏差向量
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)    #第一个卷积层
h_pool1 = max_pool(h_conv1)                  #第一个池化层

W_conv2 = weight_variable([5, 5, 32, 64])   # 通道为32, 深度为64的随机权重矩阵
b_conv2 = bias_variable([64])
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)    #第二个卷积层
h_pool2 = max_pool(h_conv2)                  #第二个池化层

h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])           #reshape成向量

"""
    有的代码还会在此添加一个全连接层和dropout层
"""

W_fc2 = weight_variable([7*7*64, 10]) #softmax层
b_fc2 = bias_variable([10])
y_predict=tf.nn.softmax(tf.matmul(h_pool2_flat, W_fc2) + b_fc2)

cross_entropy = -tf.reduce_sum(y_actual*tf.log(y_predict))    #交叉熵
train_step = tf.train.GradientDescentOptimizer(1e-4).minimize(cross_entropy)    #梯度下降法
correct_prediction = tf.equal(tf.argmax(y_predict,1), tf.argmax(y_actual,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))    #精确度计算
sess=tf.InteractiveSession()
sess.run(tf.global_variables_initializer())
for i in range(20000):
    batch = mnist.train.next_batch(50)
    if i%100 == 0:          #训练100次, 验证一次
        train_acc = accuracy.eval(feed_dict={x:batch[0], y_actual: batch[1]})
        print('step',i,'training accuracy',train_acc)
        train_step.run(feed_dict={x: batch[0], y_actual: batch[1]})

test_acc=accuracy.eval(feed_dict={x: mnist.test.images, y_actual: mnist.test.labels})
print("test accuracy",test_acc)

```

2.2.2 运行结果

```

step 0 training accuracy 0.04
step 100 training accuracy 0.7
step 200 training accuracy 0.78
step 300 training accuracy 0.9
step 400 training accuracy 0.9
step 500 training accuracy 0.94
step 600 training accuracy 0.9
step 700 training accuracy 0.92
step 800 training accuracy 0.86
step 900 training accuracy 0.92
step 1000 training accuracy 0.96
step 1100 training accuracy 0.98
step 1200 training accuracy 0.96
step 1300 training accuracy 0.94

```

```

step 3500 training accuracy 0.98
step 3600 training accuracy 0.94
step 3700 training accuracy 0.98

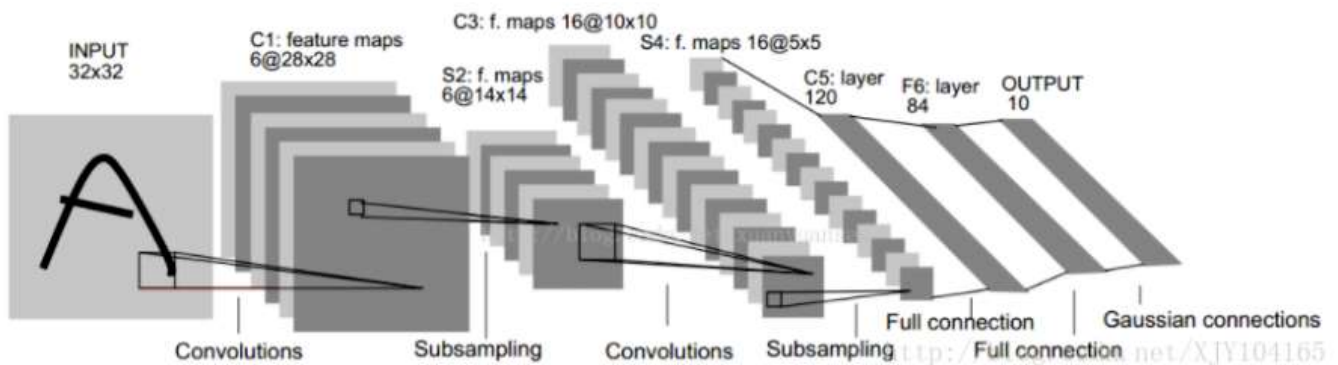
```


到学习后期已经基本稳定在95%左右

3 用LeNet-5解决MNIST数字识别问题

3.1 理解LeNet-5

LeNet-5模型一共有7层，下图展示了LeNet-5模型的架构：



其实就是由两次(卷积->池化)交替，后跟上三个全连接层。LeNet5的具体参数也在图中有所体现，比如C1接收 32×32 的图片，输出 28×28 的矩阵，且卷积核种类为6。

3.2 代码实现

这份代码参考了网上别人的代码。尽管认真思考过，我还是不确定这个是否算作LeNet-5。因为这份代码的构建格式与《tensorflow实战google深度学习框架》第六章所提到的LeNet-5格式基本相似。我觉得是符合LeNet-5的架构的，因为两次卷积->池化后确实有两个全连接层，代码应该只是把第六层和输出层合并了。

```
# 声明第六层全连接层的变量并实现前向传播过程。这一层的输入为一组长度为 512 的向量，
# 输出为一组长度为 10 的向量。这一层的输出通过 Softmax 之后就得到了最后的分类结果。
with tf.variable_scope('layer6-fc2'):
    fc2_weights = tf.get_variable(
        "weight", [FC_SIZE, NUM_LABELS],
        initializer=tf.truncated_normal_initializer(stddev=0.1))
    if regularizer != None:
        tf.add_to_collection('losses', regularizer(fc2_weights))
    fc2_biases = tf.get_variable(
        "bias", [NUM_LABELS],
        initializer=tf.constant_initializer(0.1))
    logit = tf.matmul(fc1, fc2_weights) + fc2_biases

# 返回第六层的输出。
return logit
```

代码：

```

import tensorflow as tf
import sys
from tensorflow.examples.tutorials.mnist import input_data

"""
训练3000次时已能接近1.000正确率
"""

x = tf.placeholder("float", shape=[None, 784])
y_ = tf.placeholder("float", shape=[None, 10])

def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)

def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

sess = tf.InteractiveSession()

# 第一层卷积核池化层
x_image = tf.reshape(x, [-1, 28, 28, 1])
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)

# 第二层卷积核池化层
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)

# Now image size is reduced to 7*7
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])
h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

# dropout层
keep_prob = tf.placeholder("float")
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

# full connect层与输出层
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])
y_conv=tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)

cross_entropy = -tf.reduce_sum(y_*tf.log(y_conv))

```

```
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
sess.run(tf.global_variables_initializer())

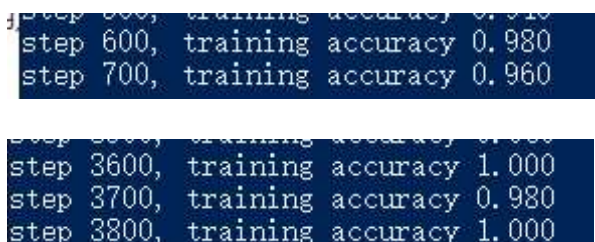
for i in range(20000):
    batch = mnist.train.next_batch(50)
    if i%100 == 0:
        train_accuracy = accuracy.eval(feed_dict={x:batch[0], y_: batch[1], keep_prob: 1.0})
        print("step %d, training accuracy %.3f"%(i, train_accuracy))
        train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

print("Training finished")

test_acc = accuracy.eval(feed_dict={x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0})
print("test accuracy %.3f" % test_acc)
```

3.3 运行结果

表现比较好，很容易就到95+%了。



The image shows two screenshots of a terminal window. The first screenshot displays the first three lines of training output: 'step 600, training accuracy 0.910', 'step 600, training accuracy 0.980', and 'step 700, training accuracy 0.960'. The second screenshot displays the next three lines: 'step 3600, training accuracy 1.000', 'step 3700, training accuracy 0.980', and 'step 3800, training accuracy 1.000'. The text is white on a dark blue background.

4. 疑惑与感想

1. 优化器的封装程度太高，比如梯度下降函数，只提供一个learning_rate参数就可以调用了。自己不确定函数的效率和具体执行策略。
2. 在试图构建LeNet-5时遇到了许多困难，在网上也一直找不到完全满意的样例
3. 在理解概念上花费了很多时间，自己的实操能力还有待提高。
4. 这周感觉比较赶，因为上周的代码需要修改，而我的代码的改动量比较大，主要逻辑基本相当于重写。所以在LeNet-5的实现上不太顺利。

5. 参考

- [卷积神经网络_ \(1\) 卷积层和池化层学习](#)
- [CNN详解 \(卷积层及下采样层\)](#)
- [\[TensorFlow\]入门学习笔记\(2\)-卷积神经网络mnist手写识别](#)
- [卷积神经网络CNN原理以及TensorFlow实现](#)
- [基于tensorflow的MNIST手写字识别 \(一\) --白话卷积神经网络模型](#)
- [深度学习Deep Learning \(01\) _CNN卷积神经网络](#)

实验代码

- https://www.tensorflow.org/tutorials/layers#intro_to_convolutional_neural_networks
- <https://www.cnblogs.com/denny402/p/5853538.html>(学习率要调到 $1e-4$)

LeNet-5

- <https://www.jianshu.com/p/ce609f9b5910>
- <https://blog.csdn.net/kaido0/article/details/53161684>