Deep learning series-1: auto-encoder, MLP, CNNs on MNIST dataset

2017-04-20 |

这里准备开一个深度学习系列, 将会从最简单的autoencoder, MLP, 到CNN, RNN, LSTM, GAN等, 每种将会伴随一些经典的实现, 可能用的框架有Tensorflow和Pytorch. 本文代码已放在github上.

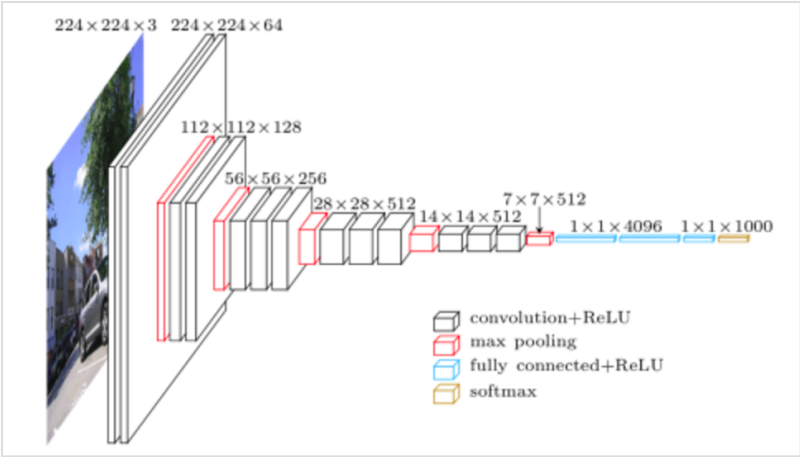


FIG. 1: MACROARCHITECTURE OF VGG16

Softmax

简单的说Softmax层就是一个全连接层加一个Softmax运算, 注意有一个Softplus, 相当于ReLu的一个激活函数, 两者是不同的.

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

print(mnist.train.images.shape, mnist.train.labels.shape)
print(mnist.test.images.shape, mnist.test.labels.shape)
print(mnist.validation.images.shape, mnist.validation.labels.shape)

import tensorflow as tf
sess = tf.InteractiveSession()
x = tf.placeholder(tf.float32, [None, 784])

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

y = tf.nn.softmax(tf.matmul(x, W) + b)

y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))

train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)

tf.global_variables_initializer().run()

for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    train_step.run({x: batch_xs, y_: batch_ys})

correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))

```

```
30  accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
31
32  print(accuracy.eval({x: mnist.test.images, y_: mnist.test.labels}))
```

92%的准确率

自编码

自编码器属于无监督模型, 不需要标注的数据, 用自身的高阶特征来编码自己.

这里涉及到稀疏编码, 所谓稀疏编码, 就是把高阶特征分解为低阶特征来稀疏表示.

可以把神经网络的前向过程看做从低阶特征向高阶特征过渡.

那么稀疏编码是反过来的过程, 从高阶特征向低阶特征转移. Sparce Encoding, 这里的稀疏可以是双向的.

在计算机视觉中, 我们需要从像素这样的低阶特征中学习高阶特征, 所谓编码就是从高阶到低阶过渡, 从而压缩信息, 自编码器的关键在编码, 用自身的高阶特征来编码自己. 特点如下

1. 希望输入输出一致
2. 希望使用高阶特征来重构自己.

DBN第一次使得训练深层网络成为可能.

从原始的自编码引入一些限制, 可以有一个特性.

1. 中间节点数目小于输入节点, 相当于降维.

2. 在输入数据中加入噪声, 就成了去噪自编码器. 常用的噪声是加性高斯噪声(Additive Gaussian Noise, AGN), 记忆Masking Noise, 即有随机遮挡的噪声, 这种情况下, 图像中的一部分像素被置0, 自编码器需要从其它像素的结构来推断被遮挡的像素, 因此依然需要学习高阶特征.

如果中间只有一层的话, 那么就相当于PCA, DBN中间有很多隐含层, 每一个隐含层都是一个RMB, 训练时, 对每两层之间进行无监督学习, 相当于多层的自编码器, 提取特征初始化网络权重, 最后在进行有监督的反向传播训练, 解决了梯度弥散问题. 这里相当于进行预训练.

主要的作用

1. 特征提取
2. 预训练

一般用的都是去噪自编码器. 更进一步的有变分自编码器.其特征如下

1. 对中间节点的分布有强假设
2. 有额外的损失项
3. 使用SGVB(Stochastic Gradient Variational Bayes)
4. 与GAN一起作为强大的生成模型

Mnist Autoencoder

参数初始化

这里用到一个比较常用的网络参数初始化方法, xavier initialization. 利用网络输出输出节点个数, 使得网络权重的均值等于0, 同时方差等于$\frac{2}{n_{in}+n_{out}}, 生成方法可以利用高斯分布或者均匀分布, 如果是均匀分布, 则可以利用(-\sqrt{\frac{6}{n_{in}+n_{out}}},\sqrt{\frac{6}{n_{in}+n_{out}}}), 方差利用公式\frac{(max-min)^2}{12}$

```
1 import numpy as np
```

```
import sklearn.preprocessing as prep
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
def xavier_init(fan_in, fan_out, constant = 1):
    low = -constant * np.sqrt(6.0 /(fan_in + fan_out))
    high = constant * np.sqrt(6.0 / (fan_in + fan_out))
    return tf.random_uniform((fan_in, fan_out), minval = low, maxval = high, dtype = tf.float32)
```

```
class AdditiveGaussianNoiseAutoencoder(object):
    def __init__(self, n_input, n_hidden, transfer_function = tf.nn.softplus, optimizer = tf.train.AdamOptimizer(),
                 scale = 0.1):
        self.n_input = n_input
        self.n_hidden = n_hidden
        self.transfer = transfer_function
        self.scale = tf.placeholder(tf.float32)
        self.training_scale = scale
        network_weights = self._initialize_weights()
        self.weights = network_weights

        # model
        self.x = tf.placeholder(tf.float32, [None, self.n_input])
        self.hidden = self.transfer(tf.add(tf.matmul(self.x + scale * tf.random_normal((n_input,)),
                self.weights['w1']),
                self.weights['b1']))
        self.reconstruction = tf.add(tf.matmul(self.hidden, self.weights['w2']), self.weights['b2'])

        # cost
        self.cost = 0.5 * tf.reduce_sum(tf.pow(tf.subtract(self.reconstruction, self.x), 2.0))
        self.optimizer = optimizer.minimize(self.cost)

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

```

```
27      def _initialize_weights(self):
28          all_weights = dict()
29          all_weights['w1'] = tf.Variable(xavier_init(self.n_input, self.n_hidden))
30          all_weights['b1'] = tf.Variable(tf.zeros([self.n_hidden], dtype = tf.float32))
31          all_weights['w2'] = tf.Variable(tf.zeros([self.n_hidden, self.n_input], dtype = tf.float32))
32          all_weights['b2'] = tf.Variable(tf.zeros([self.n_input], dtype = tf.float32))
33          return all_weights
34
35      def partial_fit(self, X):
36          cost, opt = self.sess.run((self.cost, self.optimizer), feed_dict = {self.x: X,
37                                                                               self.scale: self.training_scale
38                                                                               })
39          return cost
40
41      def calc_total_cost(self, X):
42          return self.sess.run(self.cost, feed_dict = {self.x: X,
43                                                       self.scale: self.training_scale
44                                                       })
45
46      def transform(self, X):
47          return self.sess.run(self.hidden, feed_dict = {self.x: X,
48                                                         self.scale: self.training_scale
49                                                         })
50
51      def generate(self, hidden = None):
52          if hidden is None:
53              hidden = np.random.normal(size = self.weights["b1"])
54          return self.sess.run(self.reconstruction, feed_dict = {self.hidden: hidden})
55
56      def reconstruct(self, X):
57          return self.sess.run(self.reconstruction, feed_dict = {self.x: X,
58                                                                 self.scale: self.training_scale
59                                                                 })
60
61      def getWeights(self):
```

```
        return self.sess.run(self.weights['w1'])

    def getBiases(self):
        return self.sess.run(self.weights['b1'])
```

```
mnist = input_data.read_data_sets('MNIST_data', one_hot = True)
```

```
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
```

```
def standard_scale(X_train, X_test):
    preprocessor = prep.StandardScaler().fit(X_train)
    X_train = preprocessor.transform(X_train)
    X_test = preprocessor.transform(X_test)
    return X_train, X_test

def get_random_block_from_data(data, batch_size):
    start_index = np.random.randint(0, len(data) - batch_size)
    return data[start_index:(start_index + batch_size)]
```

```
X_train, X_test = standard_scale(mnist.train.images, mnist.test.images)
```

```
n_samples = int(mnist.train.num_examples)
training_epochs = 20
batch_size = 128
display_step = 1
```

```
autoencoder = AdditiveGaussianNoiseAutoencoder(n_input = 784,
```

```
                                    n_hidden = 200,
                                    transfer_function = tf.nn.softplus,
                                    optimizer = tf.train.AdamOptimizer(learning_rate = 0.001),
                                    scale = 0.01)
```

```
import matplotlib.pyplot as plt
import matplotlib
%matplotlib inline
```

```
def plot_10_by_10_images(images):
    """ Plot 100 MNIST images in a 10 by 10 table. Note that we crop
    the images so that they appear reasonably close together.  The
    image is post-processed to give the appearance of being continued."""
    images = np.array([np.reshape(image, (28,28)) for image in images])
    fig = plt.figure()
    #images = [image[3:25, 3:25] for image in images]
    #image = np.concatenate(images, axis=1)
    for x in range(10):
        for y in range(10):
            ax = fig.add_subplot(10, 10, 10*y+(x+1))
            ax.matshow(images[10*y+x], cmap = matplotlib.cm.binary)
            plt.xticks(np.array([]))
            plt.yticks(np.array([]))
    plt.show()
```

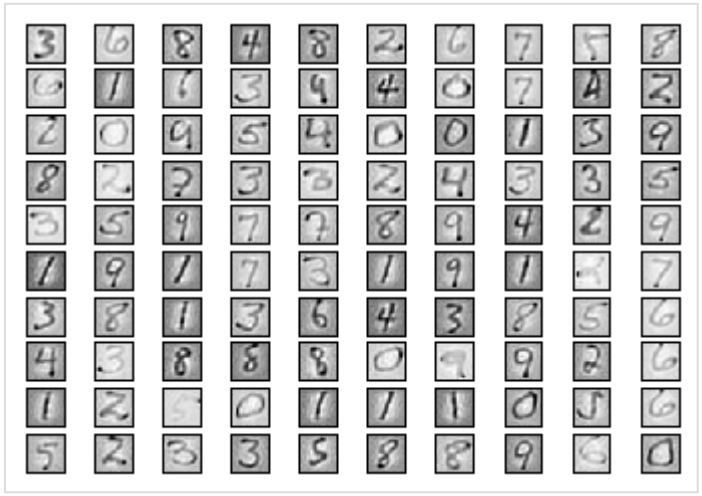
```
for epoch in range(training_epochs):
    avg_cost = 0.
    total_batch = int(n_samples / batch_size)
    # Loop over all batches
    for i in range(total_batch):
        batch_xs = get_random_block_from_data(X_train, batch_size)

```

```
        # Fit training using batch data
        cost = autoencoder.partial_fit(batch_xs)
        # Compute average loss
        avg_cost += cost / n_samples * batch_size

    # Display logs per epoch step
    if epoch % display_step == 0 or epoch == 0:
        batch_xs = batch_xs[:100]
        reconstruct_img = autoencoder.reconstruct(batch_xs)
        plot_10_by_10_images(reconstruct_img)
        print("Epoch:", '%04d' % (epoch + 1), "cost=", "{:.9f}".format(avg_cost))

print("Total cost: " + str(autoencoder.calc_total_cost(X_test)))
```



('Epoch:', '0020', 'cost=', '7491.963491477')
Total cost: 675218.0

Autoencoder作为深度学习在无监督学习中的应用是非常成功的, 可以用来为模型初始化参数, 但是现在用的比较少了, 可以用来提取高阶特征.

MLP

为了防止深层网络的过拟合, 一般有三种方法(DAR)

1. Dropout
2. Adam, AdaGrad(改进的随机梯度算法)
3. ReLu

其中Dropout相当于随机丢弃某一层的输出数据, 相当于创造出很多的随机样本, 通过增大样本量, 减少特征数目来防止过拟合, 也是一种bagging方法.

MLP可以解决XOR问题, 因为XOR问题是线性不可分的, 引入隐含层之后可以学习非线性的曲线来进行划分.

借助隐含层, 我们可以抽象出高阶特征, 进一步利用高阶特征来进行推断, 而如果没有隐含层, 我们只能直接从像素推断. 如Softmax如果只有一层的话和在之前加入一个隐含层后再进行一次Softmax的准确率的差别是非常高的. 前者是92%, 后者则有98%.

```
from tensorflow.examples.tutorials.mnist import input_data
import tensorflow as tf
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
sess = tf.InteractiveSession()

in_units = 784
h1_units = 300
W1 = tf.Variable(tf.truncated_normal([in_units, h1_units], stddev=0.1))
b1 = tf.Variable(tf.zeros([h1_units]))
W2 = tf.Variable(tf.zeros([h1_units, 10]))
b2 = tf.Variable(tf.zeros([10]))

x = tf.placeholder(tf.float32, [None, in_units])
keep_prob = tf.placeholder(tf.float32)

hidden1 = tf.nn.relu(tf.matmul(x, W1) + b1)
hidden1_drop = tf.nn.dropout(hidden1, keep_prob)
y = tf.nn.softmax(tf.matmul(hidden1_drop, W2) + b2)
```

```
19
20  # Define loss and optimizer
21  y_ = tf.placeholder(tf.float32, [None, 10])
22  cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
23  train_step = tf.train.AdagradOptimizer(0.3).minimize(cross_entropy)
24
25  # Train
26  tf.global_variables_initializer().run()
27  for i in range(3000):
28    batch_xs, batch_ys = mnist.train.next_batch(100)
29    train_step.run({x: batch_xs, y_: batch_ys, keep_prob: 0.75})
30
31  # Test trained model
32  correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
33  accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
34  print(accuracy.eval({x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))
```

```
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
0.9785
```

CNNs

借助隐含层, 我们可以抽象出高阶特征, 进一步利用高阶特征来进行推断, 而如果没有隐含层, 我们只能直接从像素推断. 如Softmax如果只有一层的话和在之前加入一个隐含层后再进行一次Softmax的准确率的差别是非常高的.

CNN最初提出是用来解决图像问题. 在CNNs之前, 我们可以利用SIFT, HoG等算法来提取出图像的高阶特征, 然后利用SVM等一些算法对图像进行分类.

CNN中的参数数目只与卷积核的大小有关, 每一个卷积核只能提取一种特性的特征.

1. 局部连接
2. 共享权重
3. 池化中的降采样

其中1,2使得模型复杂度下降, 减轻过拟合, 池化进一步降低了模型的参数数目, 提高对变形的容忍性, 增大泛化能力.

```
from tensorflow.examples.tutorials.mnist import input_data
import tensorflow as tf
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
sess = tf.InteractiveSession()


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')

x = tf.placeholder(tf.float32, [None, 784])
y_ = tf.placeholder(tf.float32, [None, 10])
x_image = tf.reshape(x, [-1,28,28,1])

W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
```

```
28  h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
29  h_pool1 = max_pool_2x2(h_conv1)
30
31  W_conv2 = weight_variable([5, 5, 32, 64])
32  b_conv2 = bias_variable([64])
33  h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
34  h_pool2 = max_pool_2x2(h_conv2)
35
36  W_fc1 = weight_variable([7 * 7 * 64, 1024])
37  b_fc1 = bias_variable([1024])
38  h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
39  h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
40
41  keep_prob = tf.placeholder(tf.float32)
42  h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
43
44  W_fc2 = weight_variable([1024, 10])
45  b_fc2 = bias_variable([10])
46  y_conv=tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)
47
48  cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y_conv), reduction_indices=[1]))
49  train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
50
51  correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
52  accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
53  tf.global_variables_initializer().run()
54  for i in range(2000):
55    batch = mnist.train.next_batch(50)
56    if i%100 == 0:
57      train_accuracy = accuracy.eval(feed_dict={
58          x:batch[0], y_: batch[1], keep_prob: 1.0})
59      print("step %d, training accuracy %g"%(i, train_accuracy))
60    train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})
61
62  print("test accuracy %g"%accuracy.eval(feed_dict={
```

```
63          x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))
```

```
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
step 0, training accuracy 0.12
step 100, training accuracy 0.84
step 200, training accuracy 0.96
step 300, training accuracy 0.86
step 400, training accuracy 0.98
step 500, training accuracy 0.98
step 600, training accuracy 0.98
step 700, training accuracy 0.96
step 800, training accuracy 0.9
step 900, training accuracy 1
step 1000, training accuracy 0.98
step 1100, training accuracy 0.98
step 1200, training accuracy 0.92
step 1300, training accuracy 0.96
step 1400, training accuracy 1
step 1500, training accuracy 0.98
step 1600, training accuracy 0.96
step 1700, training accuracy 0.94
step 1800, training accuracy 0.96
step 1900, training accuracy 0.94
test accuracy 0.9748
```

这里只训练了一会儿, 理论上训练的久点是可以达到98%-99%的准确率的, 相比上面的softmax效果更好.

deep-learning

ANNS

Deep learning series-2: typical CNN net