基于tf实现稀疏自编码和在推荐中的应用

原创 Thinkgamer 搜索与推荐Wiki 2020-05-29

收录于话题
#推荐相关笔记

32个

稀疏自编码

自编码器（Auto-Encoder）顾名思义，即可以利用自身的高阶特征编码自己。自编码器也是一种神经网络，他的输入和输出是一致的，他借助稀疏编码的思想，目标是使用稀疏的一些高阶特征重新组合来重构自己。

因此他的特征十分明显：

- 期望输入与输出一致
- 希望使用高阶特征来重构自己，而不只是复制像素点

自编码器的输入节点和输出节点的数量是一致的，但如果只是单纯的逐个复制输入节点则没有意义，像前面提到的，自编码器通常希望使用少量稀疏的高维特征来重构输入，所以加入几种限制：

- （1）中间隐含层节点的数量。如果中间隐含节点数量小于输入/输出的数量，则为一个降维的过程。此时不可能出现复制所有节点的情况，只能学习数据中最重要的特征，将不太相关的特征去除。**如果再加一个L1正则，则可以根据惩罚系数控制隐含节点的稀疏程度，惩罚系数越大，学到特征越稀疏**

- （2）给数据加入噪声，变成了Denoising AutoEncoder（去噪自编码器），将从噪声中学习出数据的特征。此时只有学习数据频繁出现的模式和结构，将噪声去除，才可以复原数据。

去噪自编码器中最常使用的噪声有：

- 加性高斯噪声（Additive Gaussian Noise，AGN）
- 用Masking Noise，即有随机遮挡的噪声
- Variational AutoEncoder（VAE）

Xavier initialization

特点：根据某一层网络的输入、输出节点数量自动调整最合适的分布。Xavier让权重满足0均值，同时方差为$\frac{2}{n_{in}+n_{out}}$，分布可以是均匀分布或者高斯分布。

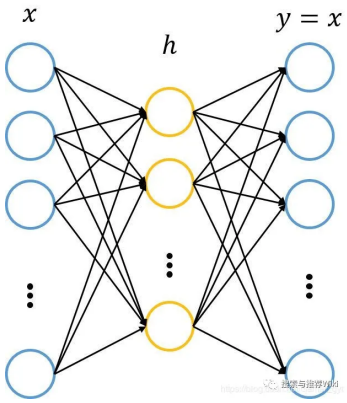
tf.random_uniform 创造一个 $(-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}})$范围内的均匀分布

```
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, dty
```

稀疏自编码可以被解释为

- 如果当神经元的输出接近于1的时候我们认为它被激活，而输出接近于0的时候认为它被抑制，那么使得神经元大部分的时间都是被抑制的限制则被称作稀疏性限制。这里我们假设的神经元的激活函数是sigmoid函数。如果你使用tanh作为激活函数的话，当神经元输出为-1的时候，我们认为神经元是被抑制的。

只有一个隐藏层的稀疏自编码结构如下图



这时候隐藏层则是原始特征的另一种表达形式。

在推荐中的应用

一个大型推荐系统，物品的数量级为千万，用户的数量级为亿。使用稀疏编码进行数据降维后，用户或者物品均可用一组低维基向量表征，便于存储计算，可供在线层实时调用。

在推荐实践中，我们主要使用稀疏编码的方法，输入用户点击/收藏/购买数据，训练出物品及用户的特征向量，具体构造自编码网络的方法如下：

输入层：每首物品的输入向量为 $(u_1, u_2, \ldots, u_n)$，其中u_i表示用户i是否点击/收藏/购买这个物品。输入矩阵为$(m+1)*n$维（包含一个截距项），m为用户数量，n为物品数量。

输出层：指定为和输出层一致（无截距项）。

隐藏层：强制指定神经元的数量为$k+1$个，此时隐藏层其实就是物品的低维特征向量，矩阵为（ $k+1$ ） $*n, k+1$为特征维数（包含一个截距项1，之所以保留，是为了可以重构出输出层），n为物品数量。

隐藏层到输出层的连接。一般的神经网络中，往往会忽略隐藏层到输出层的连接权重$W^{(1,1)}, W^{(1,2)}, b^1, b^2$的意义，只是将其作为一个输出预测的分类器；但在自编码网络中，连接层是有实际意义的。这些权重作用是将物品特征向量映射到用户是否听过/喜欢该物品，其实可就是用户的低维特征，所以该稀疏网络同样可以学习到用户的特征矩阵$m*(k+1)$。值得注意的是，当网络结构为3层时，其目标函数与svd基本一致，算法上是相通的。

tf实现

代码阅读不便，可以参考blog：https://thinkgamer.blog.csdn.net/article/details/106413077

1、引入需要的package

```
import numpy as np
import sklearn.preprocessing as prep
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
```

2、定义个Xavier初始化器

```
# 定义一个 Xavier初始化器，让权重不大不小，正好合适
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))
```

```
    weight = tf.random_uniform( (fan_in, fan_out), minval=low, maxval=high, dty
    return weight
```

3、构建融合高斯噪音的稀疏自编码

```
class AdditiveGaussianNoiseAutoEncoder(object):
    # 构造函数
    def __init__(self, n_input, n_hidden, transfer_function = tf.nn.softplus,
        # 输入变量数
        self.n_input = n_input
        # 隐含层节点数
        self.n_hidden = n_hidden
        # 隐含层激活函数
        self.transfer_func = transfer_function
        # 优化器，默认使用Adma
        self.optimizer = optiminzer
        # 高斯噪声系数，默认使用0.1
        self.scale = tf.placeholder(tf.float32)
        self.training_scale = scale
        # 初始化神经网络参数
        network_weights = self._initialize_weights()
        # 获取神经网络参数
        self.weights = network_weights
        # 初始化输入的数据， 数据的维度为 n_input 列，行数未知
        self.x = tf.placeholder(tf.float32, [None, self.n_input])
        # 计算隐藏层值，输入数据为融入噪声的数据，然后与w1权重相乘，再加上偏置
        self.hidden = self.transfer_func(
            tf.add(
                tf.matmul(
                    self.x + scale * tf.random_normal((self.n_input,)), s
                ),
                self.weights["b1"]
            )
        )
        # 计算预测结果值，将隐藏层的输出结果与w2相乘，再加上偏置
        self.reconstruction = tf.add(
            tf.matmul(
                self.hidden, self.weights["w2"]
            ),
```

```
            self.weights["b2"]
        )
        # 计算平方损失函数（Squared Error） subtract：计算差值
        self.cost = 0.5 * tf.reduce_mean(
            tf.pow(
                tf.subtract(
                    self.reconstruction, self.x
                ),
                2.0
            )
        )
        # 定义优化方法，这里默认使用的是Adma
        self.optimizer = optiminzer.minimize(self.cost)
        init = tf.global_variables_initializer()
        self.sess = tf.Session()
        self.sess.run(init)

    # 权值初始化函数
    def _initialize_weights(self):
        all_weights = dict()
        all_weights["w1"] = tf.Variable( xavier_init(self.n_input, self.n_hic
        all_weights["b1"] = tf.Variable( tf.zeros([self.n_hidden], dtype = t
        all_weights["w2"] = tf.Variable( tf.zeros([self.n_hidden, self.n_inpu
        all_weights["b2"] = tf.Variable( tf.zeros([self.n_input], dtype = tf
        return all_weights

    # 计算损失函数和优化器
    def partial_fit(self, X):
        cost, opt = self.sess.run(
            (self.cost, self.optimizer),
            feed_dict= {self.x: X, self.scale: self.training_scale}
        )
        return cost

    # 计算损失函数
    def calc_total_cost(self, X):
        return self.sess.run(
            self.cost, feed_dict= {self.x: X, self.scale: self.training_scal
        )

```

```
    # 输出自编码器隐含层的输出结果，用来提取高阶特征，是三层（输入层，隐含层，输出层）
    def transform(self, X):
        return self.sess.run(
            self.hidden, feed_dict= {self.x: X, self.scale: self.training_sc
        )

    # 将隐含层的输出作为结果，复原原数据，是整体拆分的后半部分
    def generate(self, hidden = None):
        if hidden is None:
            hidden = np.random.normal(size= self.weights["b1"])
        return self.sess.run(
            self.reconstruction, feed_dict= {self.hidden: hidden}
        )

    # 构建整个流程，包括：transform和generate
    def reconstruct(self, X):
        return self.sess.run(
            self.reconstruction, feed_dict={self.x: X, self.scale: self.trai
        )

    # 获取权重w1
    def getWeights(self):
        return self.sess.run(self.weights['w1'])

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

4、定义模型训练类

```
class ModelTrain:
    def __init__(self, training_epochs = 20, batch_size = 128, display_step =
        self.mnist = self.load_data()
        # 格式化训练集和测试集
        self.x_train, self.x_test = self.standard_scale(self.mnist.train.image
        # 总的训练样本数
        self.n_samples = int(self.mnist.train.num_examples)
        # 训练次数
        self.training_epochs = training_epochs
```

```
        # 每次训练的批大小
        self.batch_size = batch_size
        # 设置每多少轮显示一次loss值
        self.display_step = display_step

    # 加载数据集
    def load_data(self):
        return input_data.read_data_sets("../MNIST_data", one_hot=True)

    # 数据标准化处理函数
    def standard_scale(self, x_train, x_test):
        # StandardScaler: z = (x - u) / s (u 均值， s 标准差)
        preprocessor = prep.StandardScaler().fit(x_train)
        x_train = preprocessor.transform(x_train)
        x_test = preprocessor.transform(x_test)
        return x_train, x_test

    # 最大限度不重复的获取数据
    def get_random_block_from_data(self, data, batch_size):
        start_index = np.random.randint(0, len(data) - batch_size)
        return data[start_index:(start_index + batch_size)]
```

5、主函数调用进行模型训练

```
if __name__ == "__main__":
    autoencoder = AdditiveGaussianNoiseAutoEncoder(
        n_input=784,
        n_hidden= 200,
        transfer_function=tf.nn.softplus,
        optiminzer= tf.train.AdamOptimizer(learning_rate=0.01),
        scale= 0.01
    )
    modeltrain = ModelTrain(training_epochs= 20, batch_size= 128, display_step
    for epoch in range(modeltrain.training_epochs):
        avg_cost = 0
        # 一共计算多少次数据集
        total_bacth = int(modeltrain.n_samples / modeltrain.batch_size)
        for i in range(total_bacth):
            batch_x = modeltrain.get_random_block_from_data(modeltrain.x_trai
```

```
            cost = autoencoder.partial_fit(batch_x)
            avg_cost += cost / modeltrain.n_samples * modeltrain.batch_size
        if epoch % modeltrain.display_step == 0:
            print("Epoch:", '%04d' % (epoch + 1), "cost=", "{:.9f}".format(a

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

ok，不知道你对稀疏自编码是否有一定的了解，欢迎加微信交流！

原创不易，点个“在看”鼓励鼓励吧

收录于话题 #推荐相关笔记·32个

上一篇
机器学习在微博信息流推荐中的应用实践

下一篇
基于TF-IDF算法的短标题关键词提取

阅读原文