Tensorflow_神经网络

<Excerpt in index | 首页摘要>
使用Tensorflow搭建神经网络，熟悉一些基本概念与操作。
1、梯度下降 2、方向传播 3、Adam 4、SGD 5、多分类
Github：Tensorflow
Resource:深度学习理论与实战（基于TensorFlow实现）

<The rest of contents | 余下全文>

线性模型和梯度下降

1、一元线性模型

一元线性回归

一元线性模型非常简单，假设我们有变量 x_i 和目标 y_i，每个 i 对应于一个数据点，希望建立一个模型

$$\hat{y}_i = wx_i + b \tag{1}$$

$\hat{y}_i$ 是我们预测的结果，希望通过 $\hat{y}_i$ 来拟合目标 y_i，通俗来讲就是找到这个函数拟合 y_i 使得误差最小，即最小化

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

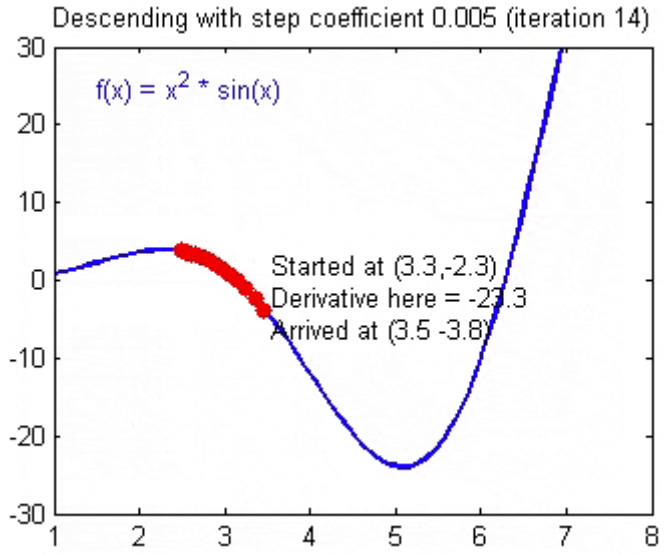
那么如何最小化这个误差呢?

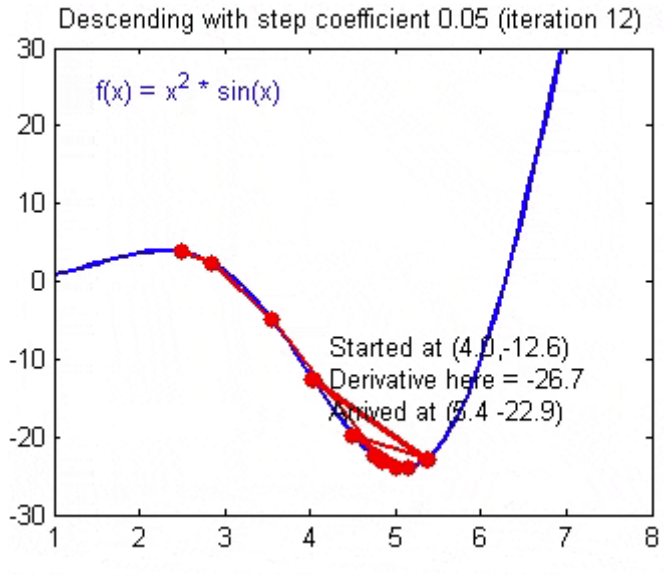
这里需要用到**梯度下降**，这是我们接触到的第一个优化算法，非常简单，但是却非常强大，在深度学习中被大量使用，所以让我们从简单的例子出发了解梯度下降法的原理

2、梯度下降

我们使用梯度下降法来优化算法，**就是沿着梯度的反方向，我们不断改变 w 和 b 的值，最终找到一组最好的 w 和 b 使得误差最小。**

在更新中，**学习率非常重要**，不同的学习率都会导致不同的结果，学习率太小会导致下降非常缓慢，学习率太大又会导致跳动非常明显，可以看看下面的例子：





最后得出的更新公式为：

$$w := w - \eta \frac{\partial f(w,\, b)}{\partial w}$$

$$b := b - \eta \frac{\partial f(w,\, b)}{\partial b}$$

3、代码实例

定义一个多项式回归函数

```
# 定义一个多变量函数
w_target = np.array([0.5, 3, 2.4]) # 定义参数
b_target = np.array([0.9]) # 定义参数

f_des = 'y = {:.2f} + {:.2f} * x + {:.2f} * x^2 + {:.2f} * x^3'.format(
    b_target[0], w_target[0], w_target[1], w_target[2]) # 打印出函数的式子
```

画出图像

```
%matplotlib inline
#%matplotlib inline 可以在Ipython编译器里直接使用，功能是可以内嵌绘图，并且可以省略掉plt.show()这一步。
import matplotlib.pyplot as plt
# 画出这个函数的曲线
x_sample = np.arange(-3, 3.1, 0.1)
y_sample = b_target[0] + w_target[0] * x_sample + w_target[1] * x_sample
 ** 2 + w_target[2] * x_sample ** 3

plt.plot(x_sample, y_sample, label='real curve')
plt.legend()
```

构造线性模型

```
x_train = np.stack([x_sample ** i for i in range(1, 4)], axis=1)
x_train = tf.constant(x_train, dtype=tf.float32, name='x_train')
y_train = tf.constant(y_sample, dtype=tf.float32, name='y_train')
w = tf.Variable(initial_value=tf.random_normal(shape=(3, 1)), dtype=tf.fl
oat32, name='weights')
b = tf.Variable(initial_value=0, dtype=tf.float32, name='bias')

def multi_linear(x):
    return tf.squeeze(tf.matmul(x, w) + b)
y_ = multi_linear(x_train)
```

定义loss并用梯度下降法优化

```
#定义loss
loss = tf.reduce_mean(tf.square(y_train - y_))
loss_numpy = sess.run(loss)

# 利用`tf.gradients()`自动求解导数
w_grad, b_grad = tf.gradients(loss, [w, b])


# 利用梯度下降更新参数
lr = 1e-3
w_update = w.assign_sub(lr * w_grad)
b_update = b.assign_sub(lr * b_grad)
```

更新100次，查看结果

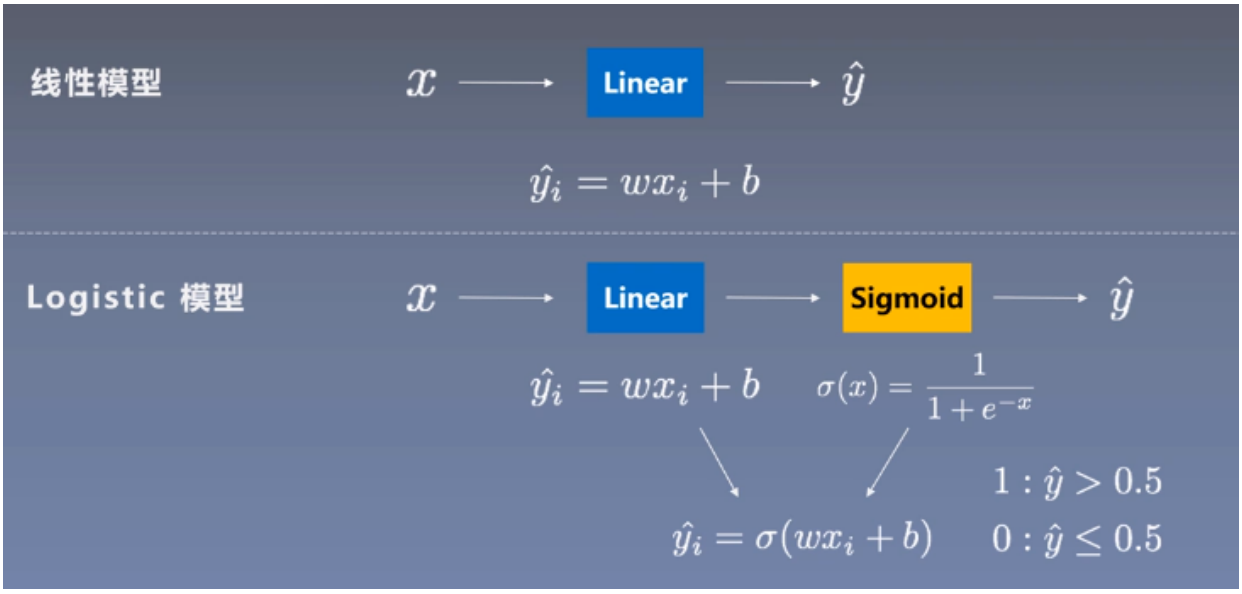
```
%matplotlib notebook
```

```
    fig = plt.figure()
    ax = fig.add_subplot(111)
    plt.ion()
    fig.show()
    fig.canvas.draw()
    sess.run(tf.global_variables_initializer())
    for e in range(100):
        sess.run([w_update, b_update])
        x_train_value = x_train.eval(session=sess)
        y_train_value = y_train.eval(session=sess)
        y_pred_value = y_.eval(session=sess)
        loss_numpy = loss.eval(session=sess)

        ax.clear()
        ax.plot(x_train_value[:,0], y_pred_value, label='fitting curve', colo
    r='r')
        ax.plot(x_train_value[:,0], y_train_value, label='real curve',
    color='b')
        ax.legend()
        fig.canvas.draw()
        plt.pause(0.1)
        if (e + 1) % 20 == 0:
            print('epoch: {}, loss: {}'.format(e + 1, loss_numpy))
```

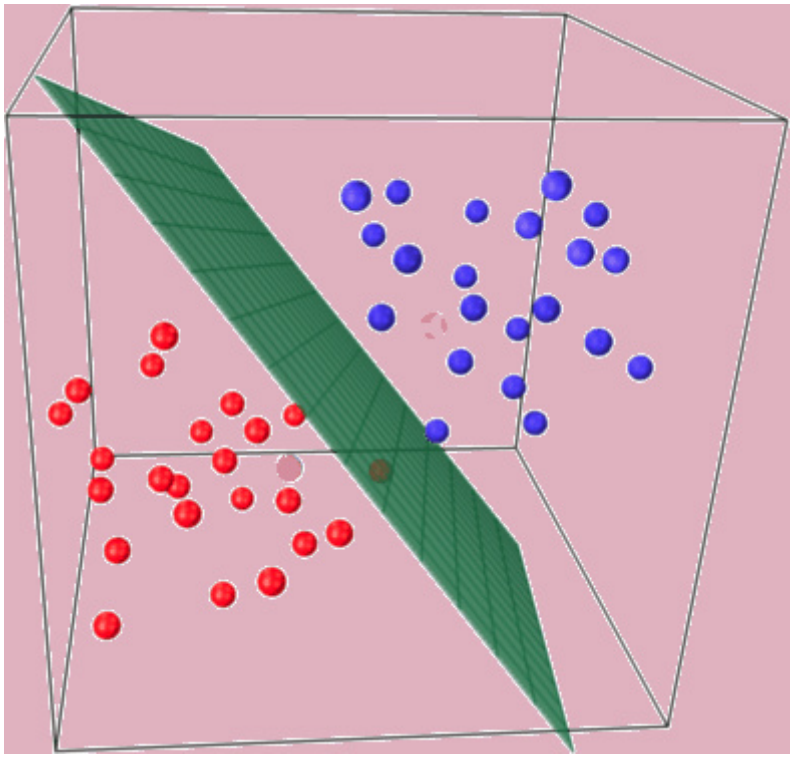
Logistic回归

1、Logistic概念

Logistic 回归是一种广义的回归模型,经常用在分类问题上，同时又以二分类更为常用。**其与线性模型的主要区别如下:**



任何一个值经过了 Sigmoid 函数的作用，都会变成 0 ~ 1 之间的一个值，这个值可以形象地理解为一个概率，**比如对于二分类问题，这个值越小就表示属于第一类，这个值越大就表示属于第二类。**



2、损失函数

Logistic损失函数如下，其中 **y 表示真实的 label，只能取 {0, 1} 这两个值**，因为输入表示经过 Logistic 回归预测之后的结果，是一个 0 ~ 1 之间的小数。如果 y 是 0，表示该数据属于第一类，我们希望 loss越小越好，反之我们希望loss越大越好，所以把损失函数设为如下：

线性模型的 loss 函数非常简单 $\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y)^2$

Logistic 模型的 loss 函数如下 $-\frac{1}{n}\sum_{i=1}^{n}(y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i))$

y_i 是真实的 label，只能取 0 和 1 两个值

$y_i = 0$ $-\frac{1}{n}\sum_{i=1}^{n}\log(1-\hat{y}_i)$

$y_i = 1$ $-\frac{1}{n}\sum_{i=1}^{n}\log \hat{y}_i$

3、代码实例

构建模型

```
w = tf.get_variable(initializer=tf.random_normal_initializer(seed=2017),
 shape=(2, 1), name='weights')
b = tf.get_variable(initializer=tf.zeros_initializer(), shape=(1),
name='bias')

def logistic_regression(x):
    # 使用 tf.sigmoid 将结果映射到 [0, 1] 区间
    return tf.sigmoid(tf.matmul(x, w) + b)
```

LOSS损失函数

```
def binary_loss(y_pred, y):
    logit = tf.reduce_mean(y * tf.log(y_pred) + (1 - y) * tf.log(1 - y_pr
ed))
    return -logit
```

优化器

```
# 首先从tf.train中定义一个优化方法
optimizer = tf.train.GradientDescentOptimizer(learning_rate=1, name='opti
mizer')

# 利用这个优化方法去优化一个损失函数，得到的这个`op`就是我们想要的
train_op = optimizer.minimize(loss)
```

训练模型

```
sess.run(tf.global_variables_initializer())

# 这一行用于时间统计，不重要
start = time.time()
for e in range(1000):
    sess.run(train_op)

    if (e + 1) % 200 == 0:
        # 计算正确率
        y_true_label = y_data.eval(session=sess)
        y_pred_numpy = y_pred.eval(session=sess)
        y_pred_label = np.greater_equal(y_pred_numpy, 0.5).astype(np.floa
t32)
        accuracy = np.mean(y_pred_label == y_true_label)
        loss_numpy = loss.eval(session=sess)
        print('Epoch %d, Loss: %.4f, Acc: %.4f' % (e + 1, loss_numpy, acc
uracy))

print()
```

```
print('manual_GD cost time: %.4f' % (time.time() - start))
```

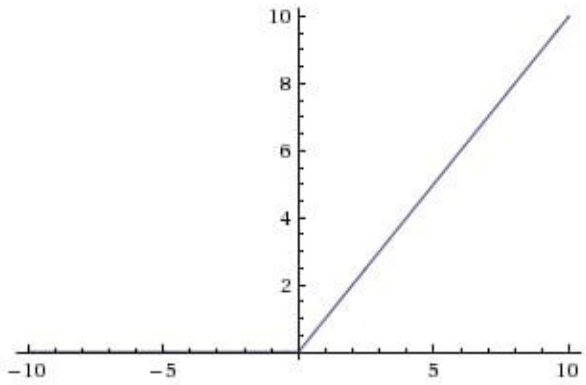
多层神经网络

1、激活函数

ReLU 激活函数

神经网络使用的激活函数都是**非线性的**，每个激活函数都输入一个值，然后做一种特定的数学运算得到一个结果。

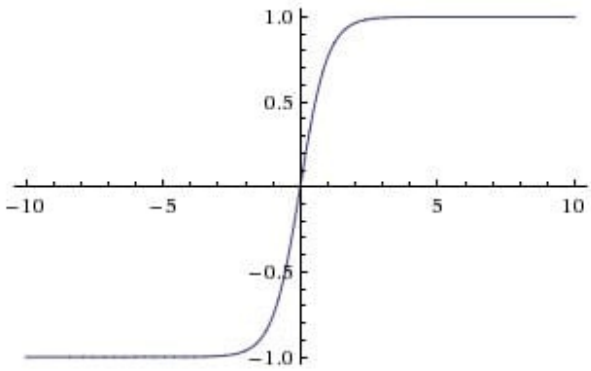
$$ReLU(x) = max(0, x)$$



我们下面重点讲一讲 ReLU 激活函数，因为现在神经网络中 90% 的情况都是使用这个激活函数。使用这个激活函数能够加快梯度下降法的收敛速度，同时对比与其他的激活函数，这个激活函数计算更加简单.

tanh 激活函数

$$tanh(x) = 2\sigma(2x) - 1$$



2、代码实例

构建一个两层的网络

```
# 首先构建第一个隐藏层
with tf.variable_scope('layer1'):

    # 构建参数weight
    w1 = tf.get_variable(initializer=tf.random_normal_initializer(stddev=
0.01), shape=(2, 4), name='weights1')

    # 构建参数bias
    b1 = tf.get_variable(initializer=tf.zeros_initializer(), shape=(4), n
ame='bias1')

# 同样地，我们再构建第二个隐藏层
with tf.variable_scope('layer2'):
    w2 = tf.get_variable(initializer=tf.random_normal_initializer(stddev=
0.01), shape=(4, 1), name='weights2')
    b2 = tf.get_variable(initializer=tf.zeros_initializer(), shape=(1), n
ame='bias2')

# 通过上面的参数构建一个两层的神经网络
def two_network(nn_input):
    with tf.variable_scope('two_network'):
        # 第一个隐藏层
        net = tf.matmul(nn_input, w1) + b1
        # tanh 激活层
        net = tf.tanh(net)
        # 第一个隐藏层
        net = tf.matmul(net, w2) + b2

        # 经过 sigmoid 得到输出
        return tf.sigmoid(net)
```

Loss函数和优化器

```
# 构建神经网络的训练过程
loss_two = tf.losses.log_loss(predictions=net, labels=y, scope='loss_tw
o')
lr = 1
optimizer = tf.train.GradientDescentOptimizer(learning_rate=lr)
train_op = optimizer.minimize(loss=loss_two, var_list=[w1, w2, b1, b2])
```

保存模型与恢复

加载模型也叫做模型的恢复，包括两个阶段

- 首先，恢复模型的结构
- 再恢复模型的参数值

```
```

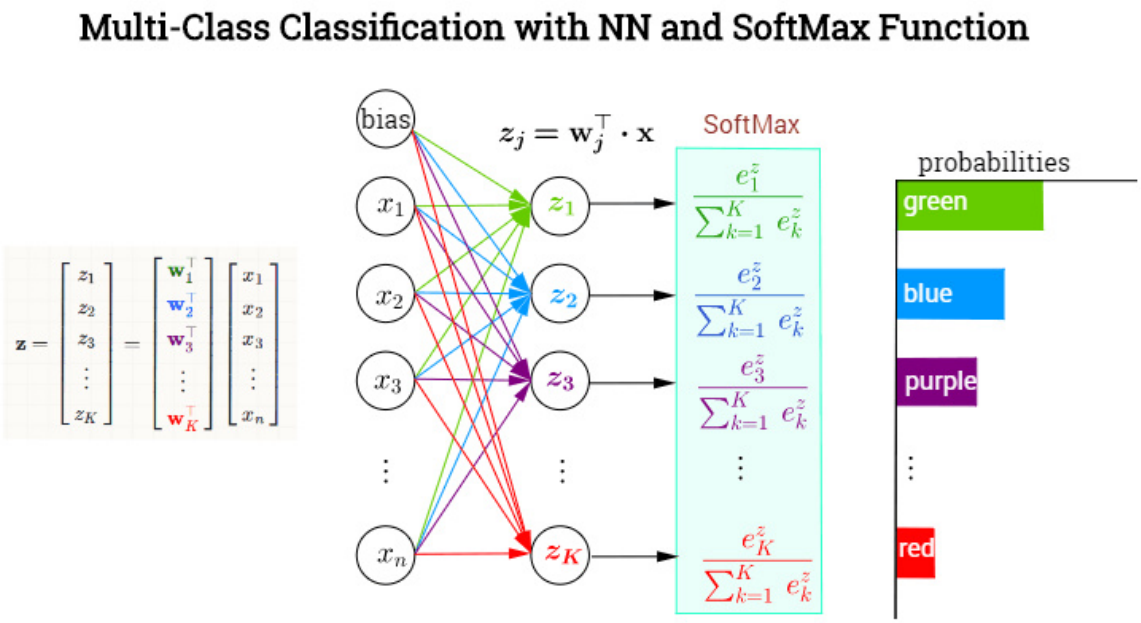
```
saver = tf.train.Saver()
saver.save(sess=sess, save_path='First_Save/model.ckpt', global_step=(e +
 1))

# 恢复模型结构
saver = tf.train.import_meta_graph('First_Save/model.ckpt-10000.meta')

# 恢复模型参数
saver.restore(sess, 'First_Save/model.ckpt-10000')
```

深层神经网络

1、Softmax

因为对于二分类问题，如果不属于第一类，那么必定属于第二类，所以只需要用一个值来表示其属于其中一类概率，但是**对于多分类问题**，这样并不行，需要知道其属于每一类的概率，这个时候就需要 **softmax 函数了。**

softmax运算过程如下:



softmax公式如下:

对于网络的输出 $z_1, z_2, \cdots z_k$，我们首先对他们每个都取指数变成 $e^{z_1}, e^{z_2}, \cdots, e^{z_k}$，那么每一项都除以他们的求和，也就是

$$z_i \to \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \tag{1}$$

如果对经过 softmax 函数的**所有项求和就等于 1**，所以他们每一项都分别表示属于其中某一类的概率。

2、交叉熵

交叉熵衡量两个分布相似性的一种度量方式。**公式如下：**

$$cross\_entropy(p,q) = E_p[-\log q] = -\frac{1}{m}\sum_x p(x)\log q(x) \tag{2}$$

交叉熵作为loss的优势:**在接近上边界的时候，其仍然可以保持在高梯度状态，因此模型的收敛速度不会受到影响**

3、代码实例

定义网络框架

```
def hidden_layer(layer_input, output_depth,scope='hidden_layer',reuse=None):
    input_depth = layer_input.get_shape()[-1]
    with tf.variable_scope(scope,reuse=reuse):
        #注意这里的初始化方法是truncated_normal
        w = tf.get_variable(initializer=tf.truncated_normal_initializer(stddev=0.1),shape=(input_depth,output_depth),name='weights')
        #注意这里用0.1对偏置值进行初始化
        b = tf.get_variable(initializer=tf.constant_initializer(0.1),shape=(output_depth),name='bias')
        net = tf.matmul(layer_input, w) + b
        return net

def DNN(x, output_depths, scope='DNN',reuse=None):
    net = x
    for i,output_depth in enumerate(output_depths):
        net = hidden_layer(net, output_depth,scope='layer%d' % i,reuse=reuse)
        #激活函数
        net = tf.nn.relu(net)

    # 数字分为0, 1, ..., 9 所以这是10分类问题
    # 对应于 one_hot 的标签, 所以这里输出一个 10维 的向量
    net = hidden_layer(net, 10 , scope='classification',reuse=reuse)
    return net
```

定义交叉熵并优化

```
#这是一个分类问题，因此我们采用交叉熵来计算损失函数
loss = tf.losses.softmax_cross_entropy(logits=dnn , onehot_labels=label_ph)

#定义正确率，注意理解它为什么是这样定义的
```

```
acc = tf.reduce_mean(tf.cast(tf.equal(tf.argmax(dnn, axis=-1),tf.argmax(l
abel_ph, axis=-1)),dtype=tf.float32))


lr = 0.01
optimizer = tf.train.GradientDescentOptimizer(learning_rate=lr)
train_op = optimizer.minimize(loss)
```

4、Tensorboard & tf.summary

首先介绍一下**tf.summary**, 它能够收集训练过程中的各种tensor的信息并把它保存起来以供Tensorboard读取并展示. 按照下面的方法来使用它:

构造summary

- 如果你想收集表示一个标量或者一个数的tensor的信息, 比如上面的loss

  ```
  loss_sum = tf.summary.scalar('loss', loss)
  ```

  上面的语句就会告诉Tensorflow, 在运行过程中, 我要让Tensorboard显示误差的变化了

- 如果你想收集一个tensor的分布情况, 这个tensor可以是任意形状, 比如我们定义了一个(784, 400)的权重w

  ```
  w_hist = tf.summary.histogram('w_hist', w)
  ```

- 如果你想收集一个4维的1-通道(灰度图), 3-通道(RGB), 4-通道(RGBA)的tensor的变化, 比如我们输出了一个(1, 8, 8, 1)的灰度图image

  ```
  image_sum = tf.summary.image('image', image)
  ```

- 如果你想收集一个3维(batch, frame, channel), 2维(batch, frame)的变化, 比如我们输出了一个(10, 50, 3)的tensor:audio

  ```
  audio_sum = tf.summary.audio('audio', audio)
  ```

(可选)融合summary

- 我们可以把前面定义的所有summary都融合成一个summary

  ```
  merged = tf.summary.merge_all()
  ```

- 也可以只是融合某些summary

  ```
  merged = tf.summary.merge([loss_sum, image_sum])
  ```

输出summary

summary是需要导出到外部文件的

- 首先定义一个文件读写器

```
summary_writer = tf.summary.FileWriter('summaries', sess.graph)
```

- 然后在训练的过程中，在你希望的时候运行一次merged或者是你之前自己定义的某个通过summary定义的op

```
summaries = sess.run(merged, feed_dict={...})
```

- 然后将这个summaries写入到summari_writer内

```
summary_writer.add_summary(summaries, step)
```

注意step表示你当前训练的步数，当然你也可以设定为其他你想要用的数值

- 最后关闭文件读写器

```
summary_writer.close()
```

记录损失函数准确率

```
for e in range(20000):
    images, labels = train_set.next_batch(batch_size)
    sess.run(train_op, feed_dict={input_ph: images, label_ph: labels})
    if e % 1000 == 999:
        test_imgs, test_labels = test_set.next_batch(batch_size)
        # 获取`train`数据的`summaries`以及`loss`, `acc`信息
        sum_train, loss_train, acc_train = sess.run([merged, loss, acc],
 feed_dict={input_ph: images, label_ph: labels})
        # 将`train`的`summaries`写入到`train_writer`中
        train_writer.add_summary(sum_train, e)
        # 获取`test`数据的`summaries`以及`loss`, `acc`信息
        sum_test, loss_test, acc_test = sess.run([merged, loss, acc], fee
d_dict={input_ph: test_imgs, label_ph: test_labels})
        # 将`test`的`summaries`写入到`test_writer`中
        test_writer.add_summary(sum_test, e)
        print('STEP {}: train_loss: {:.6f} train_acc: {:.6f} test_loss:
 {:.6f} test_acc: {:.6f}'.format(e + 1, loss_train, acc_train, loss_test,
 acc_test))

# 关闭读写器
train_writer.close()
test_writer.close()
```

打开Tensorboard

在之前对计算图可视化的时候, 我们用tensorboard –logdir=.命令打开过Tensorboard显示当前目录下`. 但Tensorboard支持打开多个目录下的.events`文件, 方便我们对比不同模型或者训练和测试之间的差别

在test_summary目录中输入以下命令:

```
$ tensorboard –logdir=train:train/,test:test/
```

反向传播

链式法则是反向传播算法的核心

Sigmoid函数举例

下面我们通过Sigmoid函数来演示反向传播过程在一个复杂的函数上是如何进行的。

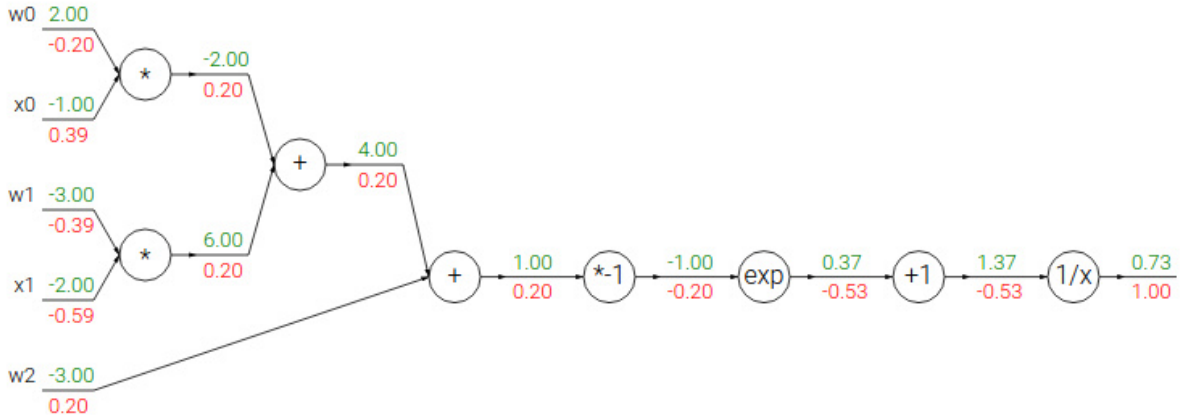
$$f(w,x)=\frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

我们需要求解出 $\frac{\partial f}{\partial w_0}, \frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}$

首先我们将这个函数抽象成一个计算图来表示，即

$$f(x)=\frac{1}{x}$$
$$f_c(x)=1+x$$
$$f_e(x)=e^x$$
$$f_w(x)=-(w_0x_0+w_1x_1+w_2)$$

按照上面的链式法则能够画出下面的计算图:



按照链式法则反向计算每个参数的梯度并进行更新:

同样上面绿色的数字表示数值，下面红色的数字表示梯度，我们从后往前计算一下各个参数的梯度。首先最后面的梯度是1,，然后经过$\frac{1}{x}$这个函数，这个函数的梯度是$-\frac{1}{x^2}$，所以往前传播的梯度是$1 \times -\frac{1}{1.37^2} = -0.53$，然后是$+1$这个操作，梯度不变，接着是$e^x$这个运算，它的梯度就是$-0.53 \times e^{-1} = -0.2$，这样不断往后传播就能够求得每个参数的梯度。

优化算法

1、SGD随机梯度下降

随机梯度下降法是我们前面一直在使用的优化算法，**就是每次都选取下降最快的方向。**

对于损失函数 $L(\theta)$，我们使用其梯度 $\nabla L(\theta)$ 来更新参数 θ，学习率为 η，那么更新公式为

$$\theta^i = \theta^{i-1} - \eta \nabla L(\theta^{i-1})$$

最终我们希望求得**最小的损失函数。**

虽然都有了数学的证明，但是为什么我们训练的时候损失函数并不是一直减少，而是一个不断上下跳动，慢慢减小的过程呢？一是因为 $\eta L'(\theta_i)$ 并不一定是一个特别小的量，所以不能完美的适用于 taylor 公式，第二点就是我们一般使用的叫做随机梯度下降，也就是每次取一定量的数据集来计算梯度，而不是全部的数据集，因为全部的数据集太过于庞大，而且计算非常慢，随机取数据点不但计算快，还有利于我们跳出局部极小点和鞍点。

2、动量法

梯度下降法的问题：

考虑一个二维输入，$[x_1, x_2]$，输出的损失函数 $L: R^2 \to R$，下面是这个函数的等高线

可以想象成一个很扁的漏斗，这样在竖直方向上，梯度就非常大，在水平方向上，梯度就相对较小，所以我们在设置学习率的时候就不能设置太大，为了防止竖直方向上参数更新太过了，这样一个较小的学习率又导致了水平方向上参数在更新的时候太过于缓慢，所以就导致最终收敛起来非常慢。

动量法：

动量法相当于每次在进行参数更新的时候，都会**将之前的速度考虑进来。**

$$v_i = \gamma v_{i-1} + \eta \nabla L(\theta)$$

$$\theta_i = \theta_{i-1} - v_i$$

其中 v_i 是当前速度，γ 是动量参数，是一个小于 1的正数，η 是学习率

每个参数在各方向上的移动幅度不仅取决于当前的梯度，还取决于过去各个梯度在各个方向上是否一致，如果一个梯度一直沿着当前方向进行更新，那么每次更新的幅度就越来越大，如果一个梯度在一个方向上不断变化，那么其更新幅度就会被衰减，这样我们就可以使用一个较大的学习率，使得收敛更快，同时梯度比较大的方向就会因为动量的关系每次更新的幅度减少。

比如我们的梯度每次都等于 g，而且方向都相同，那么动量法在该方向上使**参数加速移动**，有下面的公式：

$$v_0 = 0$$

$$v_1 = \gamma v_0 + \eta g = \eta g$$

$$v_2 = \gamma v_1 + \eta g = (1+\gamma)\eta g$$

$$v_3 = \gamma v_2 + \eta g = (1+\gamma+\gamma^2)\eta g$$

$$\cdots$$

$$v_{+\infty} = (1+\gamma+\gamma^2+\gamma^3+\cdots)\eta g = \frac{1}{1-\gamma}\eta g$$

上式的意思是如果我们将y定为0.9，那么更新幅度的峰值就是原本梯度乘学习率的 10 倍。

动量法中的动量项会沿着梯度指向方向相同的方向不断增大，对于梯度方向改变的方向逐渐减小，得到了**更快的收敛速度以及更小的震荡。**

3、Adagrad 算法

Adagrad 的想法非常简答，在每次使用一个 batch size 的数据进行参数更新的时候，我们需要计算所有参数的梯度，那么其想法就是对于每个参数，初始化一个变量 s 为 0，然后每次将该参数的梯度平方求和累加到这个变量 s 上，然后在更新这个参数的时候，学习率就变为

$$\frac{\eta}{\sqrt{s+\epsilon}}$$

由上式可知不同的参数**由于梯度不同**，他们对应的 s 大小也就不同，所以上面的公式得到的**学习率也就不同**，这也就实现了**自适应的学习率。**

Adagrad 的核心想法就是，如果一个参数的梯度一直都非常大，那么其对应的学习率就变小一点，防止震荡，而一个参数的梯度一直都非常小，那么这个参数的学习率就变大一点，使得其能够更快地更新。

Adagrad 也有一些问题，因为 s 不断累加梯度的平方，所以会越来越大，导致学习率在后期会变得较小，导致收敛乏力的情况，可能无法收敛到表较好的结果。

4、RMSProp 算法

RMSProp 仍然会使用梯度的平方量，不同于 Adagrad，其会使用一个指数加权移动平均来计算这个 s，也就是

$$s_i = \alpha s_{i-1} + (1-\alpha)\, g^2 \tag{10}$$

这里 g 表示当前求出的参数梯度，然后最终更新和 Adagrad 是一样的，学习率变成了

$$\frac{\eta}{\sqrt{s+\epsilon}} \tag{11}$$

这里 α 是一个移动平均的系数，也是因为这个系数，导致了 RMSProp 和 Adagrad 不同的地方，这个系数使得 RMSProp 更新到后期累加的梯度平方较小，从而保证 s 不会太大，也就使得模型后期依然能够找到比较优的结果

5、Adam 算法

Adam 是一个结合了动量法和 RMSProp 的优化算法，其结合了两者的优点。

Adam 算法会使用一个动量变量 v 和一个 RMSProp 中的梯度元素平方的移动指数加权平均 s，首先将他们全部初始化为 0，然后在每次迭代中，计算他们的移动加权平均进行更新:

$$v = \beta_1 v + (1-\beta_1) g$$
$$s = \beta_2 s + (1-\beta_2) g^2$$

在 adam 算法里，为了减轻 v 和 s 被初始化为 0 的初期对计算指数加权移动平均的影响，每次 v 和 s 都做下面的修正:

$$\hat{v} = \frac{v}{1-\beta_1^t}$$
$$\hat{s} = \frac{s}{1-\beta_2^t}$$

最后使用修正之后的 $\hat{v}$ 和 $\hat{s}$ 进行学习率的重新计算

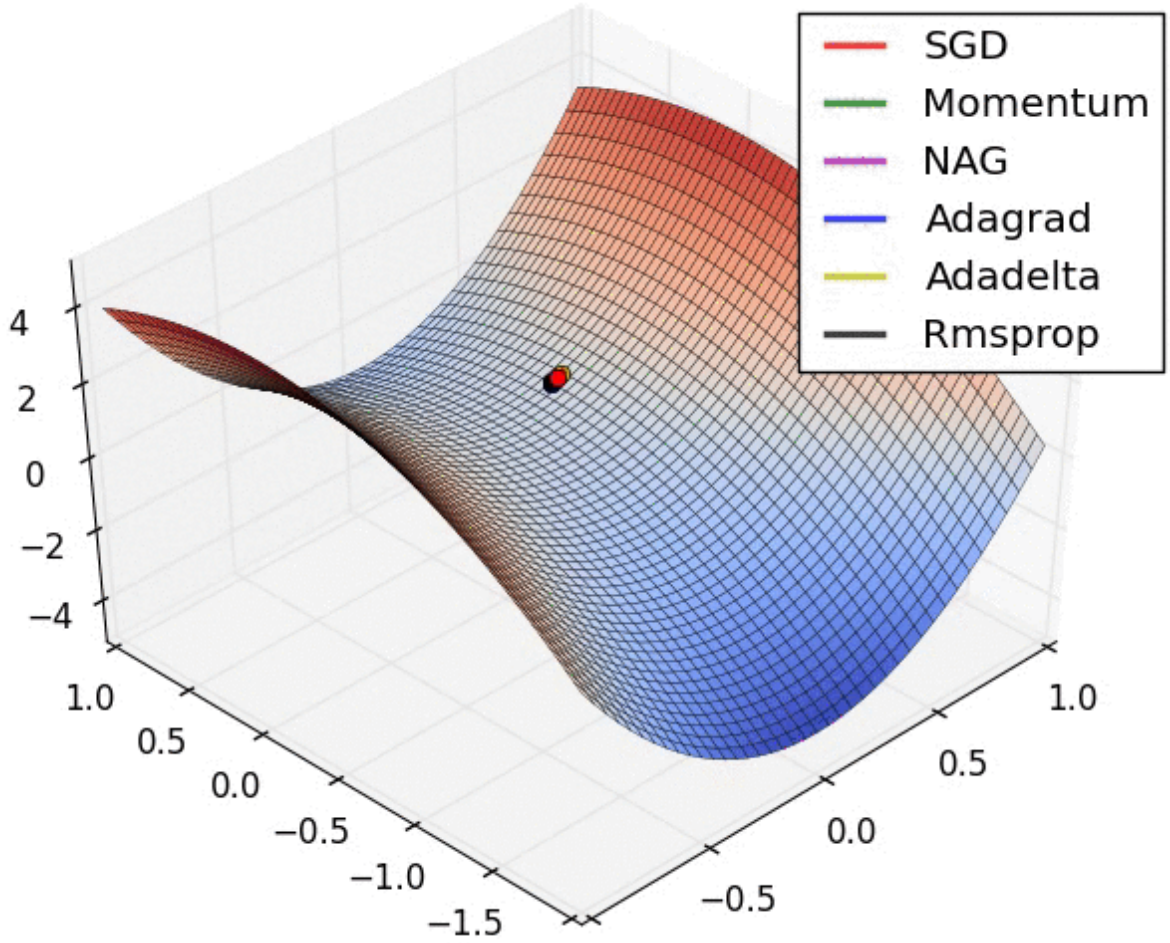
$$g' = \frac{\eta \hat{v}}{\sqrt{\hat{s} + \epsilon}}$$

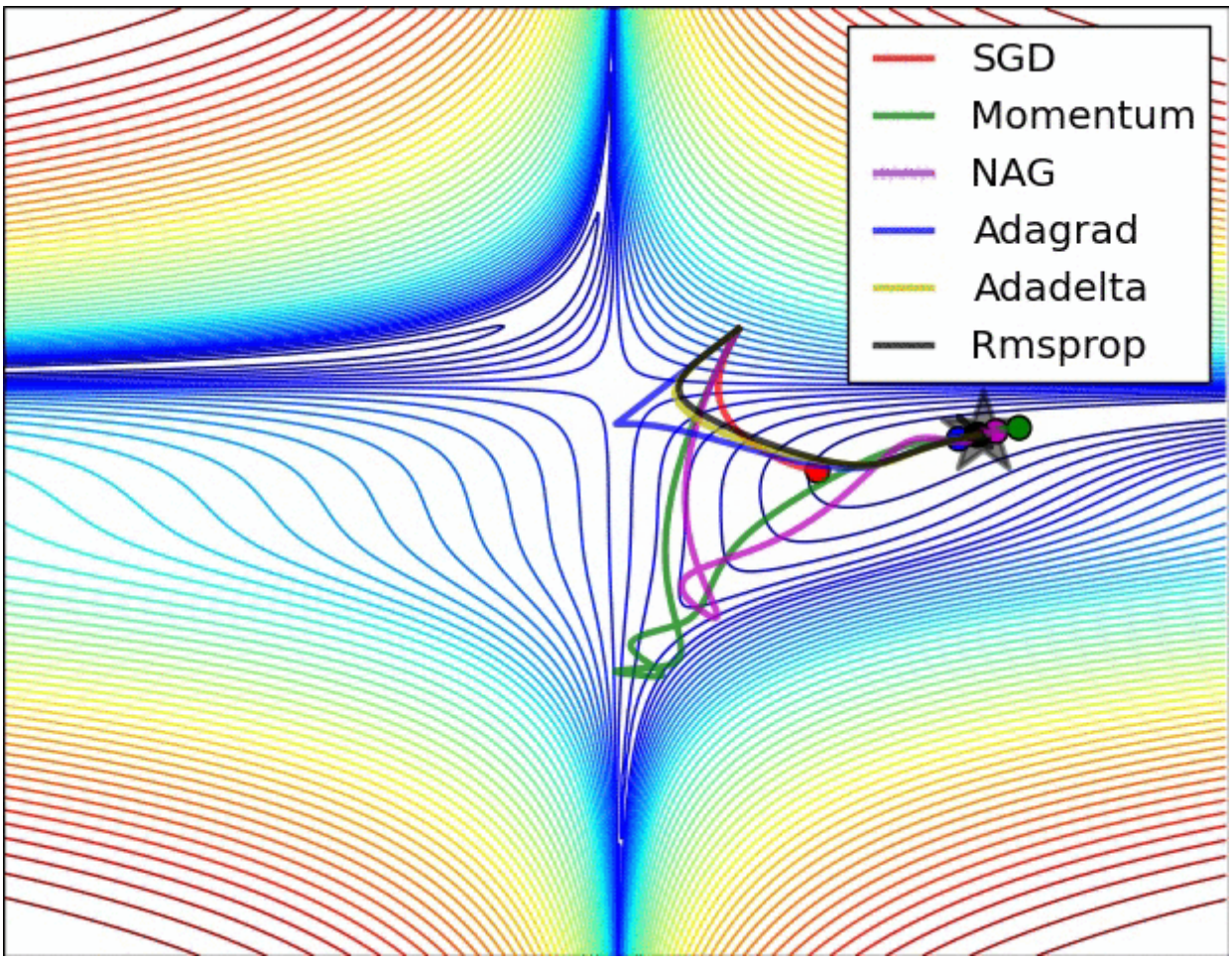
这里 η 是学习率，$epsilon$ 仍然是为了数值稳定性而添加的常数，最后参数更新有

$$\theta_i = \theta_{i-1} - g'$$

6、对比

下面是常用的几种优化算法的对比图：





优化算法—实现

1、Adadelta 法

Adadelta 跟 RMSProp 一样，先使用移动平均来计算 s

$$s = \rho s + (1-\rho)g^2$$

这里 ρ 和 RMSProp 中的 α 都是移动平均系数，g 是参数的梯度，然后我们会计算需要更新的参数的变化量

$$g' = \frac{\sqrt{\Delta\theta + \epsilon}}{\sqrt{s + \epsilon}} g$$

Δθ 初始为 0 张量，每一步做如下的指数加权移动平均更新

$$\Delta\theta = \rho\Delta\theta + (1-\rho)g'^2$$

最后参数更新如下

$$\theta = \theta - g'$$

代码示范

```
# 构建`loss`和`acc`
loss = tf.losses.softmax_cross_entropy(logits=dnn, onehot_labels=label_p
h)

acc = tf.reduce_mean(tf.cast(tf.equal(tf.argmax(dnn, axis=-1), tf.argmax
(label_ph, axis=-1)), dtype=tf.float32))
train_op = tf.train.AdadeltaOptimizer(learning_rate=1.0, rho=0.9).minimiz
e(loss)
```

2、Adagrad 算法

原理上面已经解释了，这里直接贴代码：

代码示范

```
# 构建`loss`和`acc`
loss = tf.losses.softmax_cross_entropy(logits=dnn, onehot_labels=label_p
h)

acc = tf.reduce_mean(tf.cast(tf.equal(tf.argmax(dnn, axis=-1), tf.argmax
(label_ph, axis=-1)), dtype=tf.float32))
train_op = tf.train.AdagradOptimizer(0.01).minimize(loss)
```

3、Adam 算法

原理如上，这里直接贴代码：

代码示范

```
# 构建`loss`和`acc`
loss = tf.losses.softmax_cross_entropy(logits=dnn, onehot_labels=label_p
h)

acc = tf.reduce_mean(tf.cast(tf.equal(tf.argmax(dnn, axis=-1), tf.argmax
(label_ph, axis=-1)), dtype=tf.float32))
train_op = tf.train.AdamOptimizer(1e-3).minimize(loss)
```

4、动量法

原理如上，这里直接贴代码：

```
# 构建`loss`和`acc`
loss = tf.losses.softmax_cross_entropy(logits=dnn, onehot_labels=label_p
h)

acc = tf.reduce_mean(tf.cast(tf.equal(tf.argmax(dnn, axis=-1), tf.argmax
(label_ph, axis=-1)), dtype=tf.float32))
train_op = tf.train.MomentumOptimizer(0.01, 0.9).minimize(loss)
```

5、RMSProp 算法

原理如上，这里直接贴代码：

```
# 构建`loss`和`acc`
loss = tf.losses.softmax_cross_entropy(logits=dnn, onehot_labels=label_p
h)

acc = tf.reduce_mean(tf.cast(tf.equal(tf.argmax(dnn, axis=-1), tf.argmax
(label_ph, axis=-1)), dtype=tf.float32))
train_op = tf.train.RMSPropOptimizer(0.01, 0.9).minimize(loss)
```

6、SGD随机梯度下降法

定义损失函数

```
# 构建`loss`和`acc`
loss = tf.losses.softmax_cross_entropy(logits=dnn, onehot_labels=label_p
h)

acc = tf.reduce_mean(tf.cast(tf.equal(tf.argmax(dnn, axis=-1), tf.argmax
(label_ph, axis=-1)), dtype=tf.float32))
```

在这里, 我们利用到 `tensorflow` 中的函数 **`tf.add_to_collection`** , **`tf.get_collection`**

- `tf.add_to_collection(name, var)` 会将 `var` 加入到名为 `name` 的 `collection` 的收集列表里面
- `tf.get_collection(name)` 则会将名为 `name` 的收集列表中所有的变量取出

那么我们在定义变量的过程中就可以把他们加到名为 `params` 的列表中, 然后通过 `get_collection` 一次性取出

```
params = tf.get_collection('params')
gradients_for_params = tf.gradients(loss, params)  #计算梯度
```

使用神经网络预测房价

具体参考GitHub

反馈与建议

- 微博：@柏林designer
- 邮箱：wwj123@zju.edu.cn