TensorFlow人工智能引擎入门教程之五 AlphaGo 的策略网络（CNN）简单的实

2016-05-15 基陆伯 阅 8

alphago网上 很多 文章写了怎么实现，一个策略网络 一个估值网络 一个蒙卡利索树，其中蒙卡利搜索树用于最后下棋的最优解运算，而策略网络棋子该下的位置，而估值网络用于 下了棋子，运算胜出的概率。

http://renzhichu1987.blogchina.com/2917056.html 这是网上alphaGo的介绍

我们知道 训练下棋， 就是通过好多棋盘 的 棋局 作为训练的数据，来教电脑怎么下棋

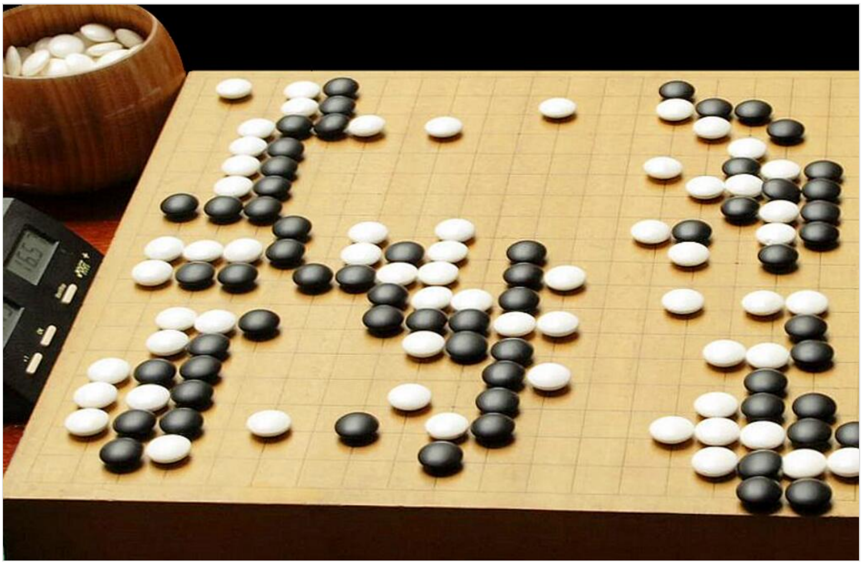
我们先看看 围棋 的棋盘 19 x 19的 格子 ，上面位置地方 不是黑棋 就是 白棋 ，而如果我们把他当做一个一个像素点，那么黑子为1 白子为0 的 二值化的灰度图片

看看输出，输出是下子的位置，如果我们把Y当做一个19x19矩阵的展开的一维向量，那么实际上每一个点的位置 [1 0 ......0]

都可以代表一个向量Y 比如 比如 [[0,0],[1,0]] 可以flat展开成[0,0,1,0] 的一维向量

那么每一次下棋的位置 就是y的1的点，这样 每一次棋盘有多个可能的y 也就是很多的x y 对

把X 19x19图像 作为输入 Y 下棋的19x19=361长度的一维向量[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0.....1,0,0,0,0,0,0,0] 作为Y输出来做训练。



下面我们修改网络实现我们的 alphago CNN 卷积神经网络

首先看一个AlphaGo官方开源https://github.com/Rochester-NRT/AlphaGo 但是他不是使用tensorflow而是使用 http://keras.io/ 这里我们把它使用

这是实际使用的网络

```
def createPolicyNetwork(self):
    # network weights
    W_conv1 = self.weight_variable([5,5,48,192])
    b_conv1 = self.bias_variable([192])
    W_conv2 = self.weight_variable([3,3,192,192])
    b_conv2 = self.bias_variable([192])
    W_conv3 = self.weight_variable([3,3,192,192])
    b_conv3 = self.bias_variable([192])
    W_conv4 = self.weight_variable([3,3,192,192])
    b_conv4 = self.bias_variable([192])
    W_conv5 = self.weight_variable([3,3,192,192])
    b_conv5 = self.bias_variable([192])
    W_conv6 = self.weight_variable([3,3,192,192])
    b_conv6 = self.bias_variable([192])
    W_conv7 = self.weight_variable([3,3,192,192])
    b_conv7 = self.bias_variable([192])
    W_conv8 = self.weight_variable([3,3,192,192])
    b_conv8 = self.bias_variable([192])
    W_conv9 = self.weight_variable([3,3,192,192])
    b_conv9 = self.bias_variable([192])
    W_conv10 = self.weight_variable([3,3,192,192])
    b_conv10 = self.bias_variable([192])
    W_conv11 = self.weight_variable([3,3,192,192])
    b_conv11 = self.bias_variable([192])
    W_conv12 = self.weight_variable([3,3,192,192])
    b_conv12 = self.bias_variable([192])
    W_conv13 = self.weight_variable([1,1,192,1])
    b_conv13 = self.bias_variable([1])

    # input layer
    self.state = tf.placeholder("float",[None,19,19,48])

    # conv layers
    h_conv1 = tf.nn.relu(self.conv2d(self.state,W_conv1,2) + b_conv1)
    h_conv2 = tf.nn.relu(self.conv2d(h_conv1,W_conv2,1) + b_conv2)
    h_conv3 = tf.nn.relu(self.conv2d(h_conv2,W_conv3,1) + b_conv3)
    h_conv4 = tf.nn.relu(self.conv2d(h_conv3,W_conv4,1) + b_conv4)
    h_conv5 = tf.nn.relu(self.conv2d(h_conv4,W_conv5,1) + b_conv5)
    h_conv6 = tf.nn.relu(self.conv2d(h_conv5,W_conv6,1) + b_conv6)
    h_conv7 = tf.nn.relu(self.conv2d(h_conv6,W_conv7,1) + b_conv7)
    h_conv8 = tf.nn.relu(self.conv2d(h_conv7,W_conv8,1) + b_conv8)
    h_conv9 = tf.nn.relu(self.conv2d(h_conv8,W_conv9,1) + b_conv9)
    h_conv10 = tf.nn.relu(self.conv2d(h_conv9,W_conv10,1) + b_conv10)
    h_conv11 = tf.nn.relu(self.conv2d(h_conv10,W_conv11,1) + b_conv11)
    h_conv12 = tf.nn.relu(self.conv2d(h_conv11,W_conv12,1) + b_conv12)
    self.probability = tf.nn.softmax(self.conv2d(h_conv12,W_conv13,1) + b_conv13)

    self.action = tf.placeholder("float",[None,19,19])
    self.cost = tf.reduce_mean(-tf.reduce_sum(self.action*tf.log(self.probability),
        reduction_indices=1))
    self.train = tf.train.GradientDescentOptimizer(learning_rate).minimize(self.cost)
```

，我们这里使用自己自定义的网络、

首先贴上上一章使用的CNN模型 部分 我们只需要修改那其中的一步分就可以了

首先是X 是 19x19 的图像 ，Y值 不是0-9 的10个 而是19x19的361个

这里我们没有特别注重网络，因为网络效果 很多精度需要测试，如果要精度好，可以使用vgg16

googlenet大概有92的准确率 而resnet大概96.5的准确率，所以 后面直接修改那些网络层即可

这里看到 我们修改了输入 以及输出 以及 shape 后面调整卷积 我们卷积 层数先不调整 ，网络方面现在

```
# Network Parameters
n_input =361 # MNIST data input (img shape: 28*28)
n_classes = 361 # MNIST total classes (0-9 digits)
dropout = 0.618 # Dropout, probability to keep units

# tf Graph input
x = tf.placeholder(tf.float32, [None, n_input])
y = tf.placeholder(tf.float32, [None, n_classes])
keep_prob = tf.placeholder(tf.float32) # dropout (keep probability)

# Create custom model
def conv2d(name, l_input, w, b):
    return tf.nn.relu(tf.nn.bias_add(tf.nn.conv2d(l_input, w, strides=[1, 1, 1, 1], padding='SAME'),b), name=

def max_pool(name, l_input, k):
    return tf.nn.max_pool(l_input, ksize=[1, k, k, 1], strides=[1, k, k, 1], padding='SAME', name=name)

def norm(name, l_input, lsize=4):
    return tf.nn.lrn(l_input, lsize, bias=1.0, alpha=0.001 / 9.0, beta=0.75, name=name)

def customnet(_X, _weights, _biases, _dropout):
    # Reshape input picture
    _X = tf.reshape(_X, shape=[-1, 19, 19, 1])

    # Convolution Layer
```

现在我们计算每一步的shape

分别是

19x19 ===>21x21 ===>10x10===>12x12===>6x6===>8x8===>4x4

OK 我们得到了全连接层的输入 如果是用alexnet CNN来训练 alphago那么 他的输入应该是4X4X256 注意光标修改的地方

```

# Store layers weight & bias
weights = {
    'wc1': tf.Variable(tf.random_normal([3, 3, 1, 64])),
    'wc2': tf.Variable(tf.random_normal([3, 3, 64, 128])),
    'wc3': tf.Variable(tf.random_normal([3, 3, 128, 256])),
    'wd1': tf.Variable(tf.random_normal([4*4*256, 1024])),
    'wd2': tf.Variable(tf.random_normal([1024, 1024])),
    'out': tf.Variable(tf.random_normal([1024, 10]))
}
biases = {
    'bc1': tf.Variable(tf.random_normal([64])),
    'bc2': tf.Variable(tf.random_normal([128])),
    'bc3': tf.Variable(tf.random_normal([256])),
    'bd1': tf.Variable(tf.random_normal([1024])),
    'bd2': tf.Variable(tf.random_normal([1024])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}

```

ok

我们有测试 数据来训练 ，我只讲一下 怎么使用CNN 或者我们自己自定义的CNN 来训练alphaGo的策略网络

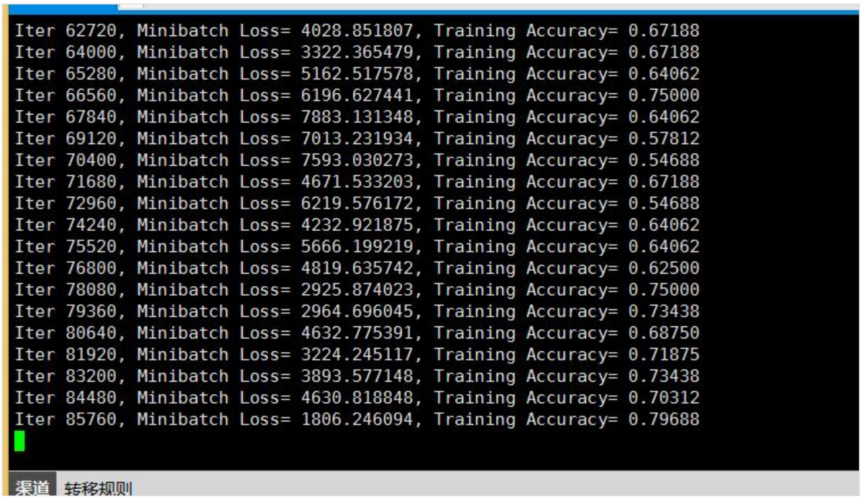
其实我一直想的是用那些googleNet有名的CNN网络模型 来训练CNN策略网络是不是会更加智能

下面贴出修改后的代码

```
# Import AlphaGo Data  import input_data mnist = input_data.read_data_sets("/tmp/data/", one_hot=True) import tensorflow as tf # Parameters learning_ra
g_iters = 200000 batch_size = 64 display_step = 20 # Network Parameters n_input =361 # alphaGo data input (img shape: 19*19) n_classes = 361 # AlphaGo
19=361 digits) dropout = 0.618 # Dropout, probability to keep units 这里是随机概率当掉一些节点来训练，随你填，我一般用黄金分割点 # tf Graph input x = 1
loat32, [None, n_input]) y = tf.placeholder(tf.float32, [None, n_classes]) keep_prob = tf.placeholder(tf.float32) # dropout (keep probability) # Create
f conv2d(name, l_input, w, b):     return tf.nn.relu(tf.nn.bias_add(tf.nn.conv2d(l_input, w, strides=[1, 1, 1, 1], padding='SAME'),b), name=name) def r
put, k):     return tf.nn.max_pool(l_input, ksize=[1, k, k, 1], strides=[1, k, k, 1], padding='SAME', name=name) def norm(name, l_input, lsize=4):
n(l_input, lsize, bias=1.0, alpha=0.001 / 9.0, beta=0.75, name=name) def alphago(_X, _weights, _biases, _dropout):     # Reshape input picture     _X =
ape=[-1, 19, 19, 1])     # Convolution Layer     conv1 = conv2d('conv1', _X, _weights['wc1'], _biases['bc1'])     # Max Pooling (down-sampling)     poo
l1', conv1, k=2)     # Apply Normalization     norm1 = norm('norm1', pool1, lsize=4)     # Apply Dropout     norm1 = tf.nn.dropout(norm1, _dropout)
    tf.image_summary("conv1", conv1)     # Convolution Layer     conv2 = conv2d('conv2', norm1, _weights['wc2'], _biases['bc2'])     # Max Pooling (dow
ol2 = max_pool('pool2', conv2, k=2)     # Apply Normalization     norm2 = norm('norm2', pool2, lsize=4)     # Apply Dropout     norm2 = tf.nn.dropout(r
    # Convolution Layer     conv3 = conv2d('conv3', norm2, _weights['wc3'], _biases['bc3'])     # Max Pooling (down-sampling)     pool3 = max_pool('poo
    # Apply Normalization     norm3 = norm('norm3', pool3, lsize=4)     # Apply Dropout     norm3 = tf.nn.dropout(norm3, _dropout)     # Fully conn
nsel = tf.reshape(norm3, [-1, _weights['wd1'].get_shape().as_list()[0]]) # Reshape conv3 output to fit dense layer input     dense1 = tf.nn.relu(tf.mat
ts['wd1']) + _biases['bd1'], name='fc1') # Relu activation     dense2 = tf.nn.relu(tf.matmul(dense1, _weights['wd2']) + _biases['bd2'], name='fc2') # I
```

```
    # Output, class prediction     out = tf.matmul(dense2, _weights['out']) + _biases['out']     return out # Store layers weight & bias weights = {
le(tf.random_normal([3, 3, 1, 64])),      'wc2': tf.Variable(tf.random_normal([3, 3, 64, 128])),      'wc3': tf.Variable(tf.random_normal([3, 3, 128, 25
f.Variable(tf.random_normal([4*4*256, 1024])),      'wd2': tf.Variable(tf.random_normal([1024, 1024])),      'out': tf.Variable(tf.random_normal([1024,
    'bc1': tf.Variable(tf.random_normal([64])),      'bc2': tf.Variable(tf.random_normal([128])),      'bc3': tf.Variable(tf.random_normal([256])),      '
e(tf.random_normal([1024])),      'bd2': tf.Variable(tf.random_normal([1024])),      'out': tf.Variable(tf.random_normal([n_classes])) } # Construct mode
t(x, weights, biases, keep_prob) # Define loss and optimizer cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(pred, y)) optimizer = tf.tra
earning_rate=learning_rate).minimize(cost) # Evaluate model correct_pred = tf.equal(tf.argmax(pred,1), tf.argmax(y,1)) accuracy = tf.reduce_mean(tf.ca
f.float32)) # Initializing the variables init = tf.initialize_all_variables() #  tf.scalar_summary("loss", cost) tf.scalar_summary("accuracy", accurac
aries to a single operator merged_summary_op = tf.merge_all_summaries() # Launch the graph with tf.Session() as sess:     sess.run(init)     summary_w
mmaryWriter('/tmp/logs', graph_def=sess.graph_def)     step = 1     # Keep training until reach max iterations     while step * batch_size < training_
h_xs, batch_ys = mnist.train.next_batch(batch_size)         # Fit training using batch data         sess.run(optimizer, feed_dict={x: batch_xs, y: bat
ropout})         if step % display_step == 0:             # Calculate batch accuracy             acc = sess.run(accuracy, feed_dict={x: batch_xs, y: b
b: 1.})             # Calculate batch loss             loss = sess.run(cost, feed_dict={x: batch_xs, y: batch_ys, keep_prob: 1.})             print "I
tch_size) + ", Minibatch Loss= " + "{:.6f}".format(loss) + ", Training Accuracy= " + "{:.5f}".format(acc)             summary_str = sess.run(merged_su
t={x: batch_xs, y: batch_ys, keep_prob: 1.})             summary_writer.add_summary(summary_str, step)         step += 1     print "Optimization Finis
te accuracy for 256 mnist test images     print "Testing Accuracy:", sess.run(accuracy, feed_dict={x: mnist.test.images[:256], y: mnist.test.labels[:2
b: 1.})
```

我没有数据，这里我就不能给大家演示截图了，但是思想方式是一样的 ，上面所有参数都是对的 。



```
Iter 62720, Minibatch Loss= 4028.851807, Training Accuracy= 0.67188
Iter 64000, Minibatch Loss= 3322.365479, Training Accuracy= 0.67188
Iter 65280, Minibatch Loss= 5162.517578, Training Accuracy= 0.64062
Iter 66560, Minibatch Loss= 6196.627441, Training Accuracy= 0.75000
Iter 67840, Minibatch Loss= 7883.131348, Training Accuracy= 0.64062
Iter 69120, Minibatch Loss= 7013.231934, Training Accuracy= 0.57812
Iter 70400, Minibatch Loss= 7593.030273, Training Accuracy= 0.54688
Iter 71680, Minibatch Loss= 4671.533203, Training Accuracy= 0.67188
Iter 72960, Minibatch Loss= 6219.576172, Training Accuracy= 0.54688
Iter 74240, Minibatch Loss= 4232.921875, Training Accuracy= 0.64062
Iter 75520, Minibatch Loss= 5666.199219, Training Accuracy= 0.64062
Iter 76800, Minibatch Loss= 4819.635742, Training Accuracy= 0.62500
Iter 78080, Minibatch Loss= 2925.874023, Training Accuracy= 0.75000
Iter 79360, Minibatch Loss= 2964.696045, Training Accuracy= 0.73438
Iter 80640, Minibatch Loss= 4632.775391, Training Accuracy= 0.68750
Iter 81920, Minibatch Loss= 3224.245117, Training Accuracy= 0.71875
Iter 83200, Minibatch Loss= 3893.577148, Training Accuracy= 0.73438
Iter 84480, Minibatch Loss= 4630.818848, Training Accuracy= 0.70312
Iter 85760, Minibatch Loss= 1806.246094, Training Accuracy= 0.79688
```


上一篇：TensorFlow人工智能引擎入门教程之四 TensorBoard面板可视化管理

下一篇：原 TensorFlow人工智能引擎入门教程之六 训练的模型Model 保存 文件 并使用

Deep learning：四十一(Dropout简单理解... 百首舞曲DJ欣赏

当AlphaGo火了以后，我们来聊聊深度学习... 《性格决定命运》

发表评论：

请 登录 或者 注册 后再进行评论 社交帐号登录：