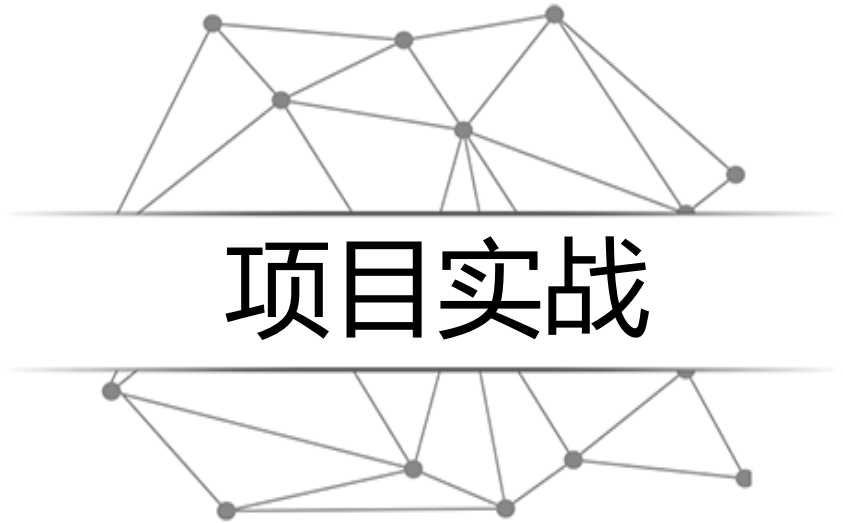

项目实战

tensorflow基本使用

理解TensorFlow:

- 使用图(graph)来表示计算任务；
- 在被称之为会话(Session)的上下文(context)中执行图；
- 使用tensor（张量）表示数据；
- 通过变量(Variable)维护状态；
- 使用feed和fetch可以为任意的操作(arbitrary operation)赋值或者从其中获取数据。

tensorflow基本使用

- TensorFlow是一个编程系统，使用图来表示计算任务。图中的节点被称作op（Operation），op可以获得0个或多个tensor，产生0个或多个tensor。每个tensor是一个类型化的多维数组。例如：可以将一组图像集表示成一个四维的浮点数组，四个维度分别是[batch，height，weight，channels]。
- 图（graph）描述了计算的过程。为了进行计算，图必须在会话中启动，会话负责将图中的op分发到CPU或GPU上进行计算，然后将产生的tensor返回。在Python中，tensor就是numpy.ndarray对象。

tensorflow基本使用

- TensorFlow程序通常被组织成两个阶段：构建阶段和执行阶段。
  - 构建阶段:op的执行顺序被描述成一个图；
  - 执行阶段:使用会话执行图中的op。
  - 例如：通常在构建阶段创建一个图来表示神经网络，在执行阶段反复执行图中的op训练神经网络。

tensorflow基本使用

实例1：

```
>>> import tensorflow as tf    #导入tensorflow库
>>> mat1 = tf.constant([[3., 3.]])  #创建一个1*2的矩阵
>>> mat2 = tf.constant([[2.],[2.]]) #创建一个2*1的矩阵
>>> product = tf.matmul(mat1, mat2) #创建op执行两个矩阵的乘法
>>> sess = tf.Session()         #启动默认图
>>> res = sess.run(product)     #在默认图中执行op操作
>>> print(res)                  #输出乘积结果
[[ 12.]]
>>> sess.close()                #关闭session
```

tensorflow基本使用

交互式会话（InteractiveSession）：

为了方便使用Ipython之类的Python交互环境，可以使用交互式会话（InteractiveSession）来代替Session，使用类似Tensor.run()和Operation.eval()来代替Session.run()，避免使用一个变量来持有会话。

tensorflow基本使用

实例2：

```
>>> import tensorflow as tf    #导入tensorflow库
>>> sess = tf.InteractiveSession()   #创建交互式会话
>>> a = tf.Variable([1.0, 2.0]) #创建变量数组
>>> b = tf.constant([3.0, 4.0]) #创建常量数组
>>> sess.run(tf.global_variables_initializer()  #变量初始化
>>> res = tf.add(a, b)            #创建加法操作
>>> print(res.eval())             #执行操作并输出结果
[4. 5.]
```

tensorflow基本使用

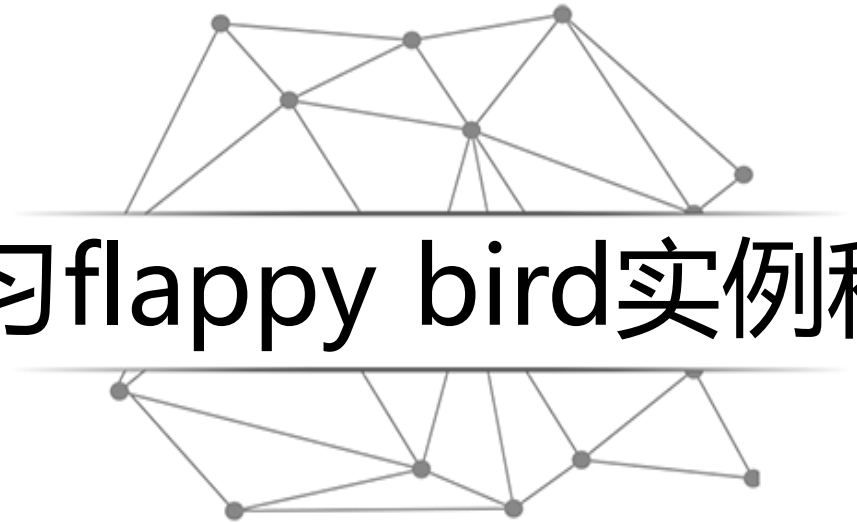
Feed操作：

前面的例子中，数据均以变量或常量的形式进行存储。Tensorflow还提供了Feed机制，该机制可以临时替代图中任意操作中的tensor。最常见的用例是使用tf.placeholder()创建占位符，相当于是作为图中的输入，然后使用Feed机制向图中占位符提供数据进行计算，具体使用方法见接下来的样例。

tensorflow基本使用

实例3：

```
>>> import tensorflow as tf    #导入tensorflow库
>>> sess = tf.InteractiveSession()   #创建交互式会话
>>> input1 = tf.placeholder(tf.float32) #创建占位符
>>> input2 = tf.placeholder(tf.float32) #创建占位符
>>> res = tf.mul(input1, input2)          #创建乘法操作
>>> res.eval(feed_dict={input1:[7.], input2:[2.]}) #求值
array([ 14.], dtype=float32)
```



自主学习flappy bird实例程序编写

实例程序编写

1. 建立工程，导入相关工具包：

```
>>> import tensorflow as tf    #导入tensorflow库
>>> import cv2                 #导入opencv库
>>> import sys                 #导入sys模块
>>> sys.path.append("game/")   #添加game目录到系统环境变量
>>> import wrapped_flappy_bird as game #加载游戏
>>> import random              #加载随机模块
>>> import numpy as np         #加载numpy模块
>>> from collections import deque #导入双端队列
```

实例程序编写

2. 设置超参数：

```
>>> GAME = 'bird'          #设置游戏名称
>>> ACTIONS = 2            #设置游戏动作数目（点击不点击屏幕）
>>> GAMMA = 0.99           #设置增强学习更新公式中的累计折扣因子
>>> OBSERVE = 10000.       #观察期 1万次迭代（随机指定动作获得D）
>>> EXPLORE = 2000000.     #探索期
>>> FINAL_EPSILON = 0.0001     #设置ε的最终最小值
>>> INITIAL_EPSILON = 0.1      #设置ε贪心策略中的ε初始值
>>> REPLAY_MEMORY = 50000      #设置Replay Memory的容量
>>> BATCH = 32             #设置每次网络参数更新时用的样本数目
>>> FRAME_PER_ACTION = 1       #设置几帧图像进行一次动作
```

实例程序编写

3. 创建深度神经网络：

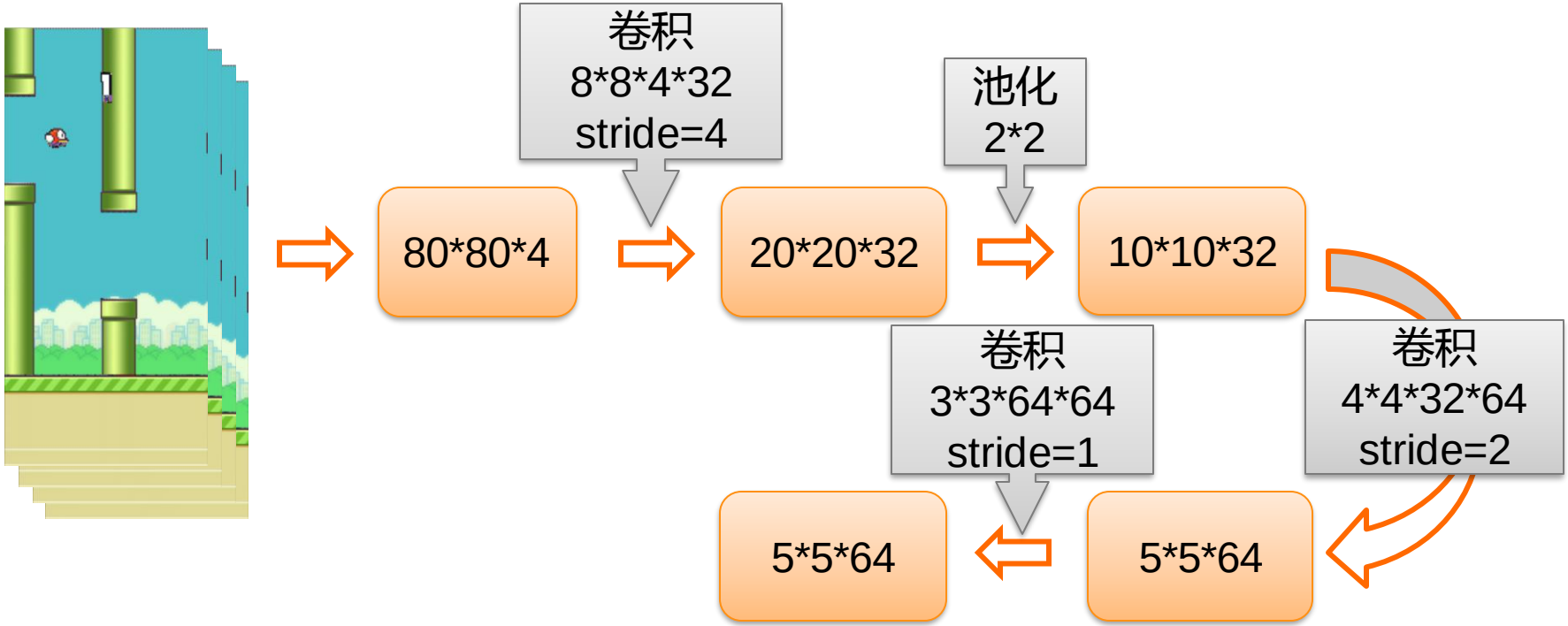
```
>>> def weight_variable(shape):
        initial = tf.truncated_normal(shape, stddev = 0.01)
    return tf.Variable(initial)
#首先定义一个函数，该函数用于生成形状为shape的张量（高纬数组）
#张量中的初始化数值服从正太分布，且方差为0.01
>>> def bias_variable(shape):
        initial = tf.constant(0.01, shape = shape)
    return tf.Variable(initial)
#定义另外一个函数，用于生成偏置项，初始值为0.01
```

实例程序编写

3. 创建深度神经网络：

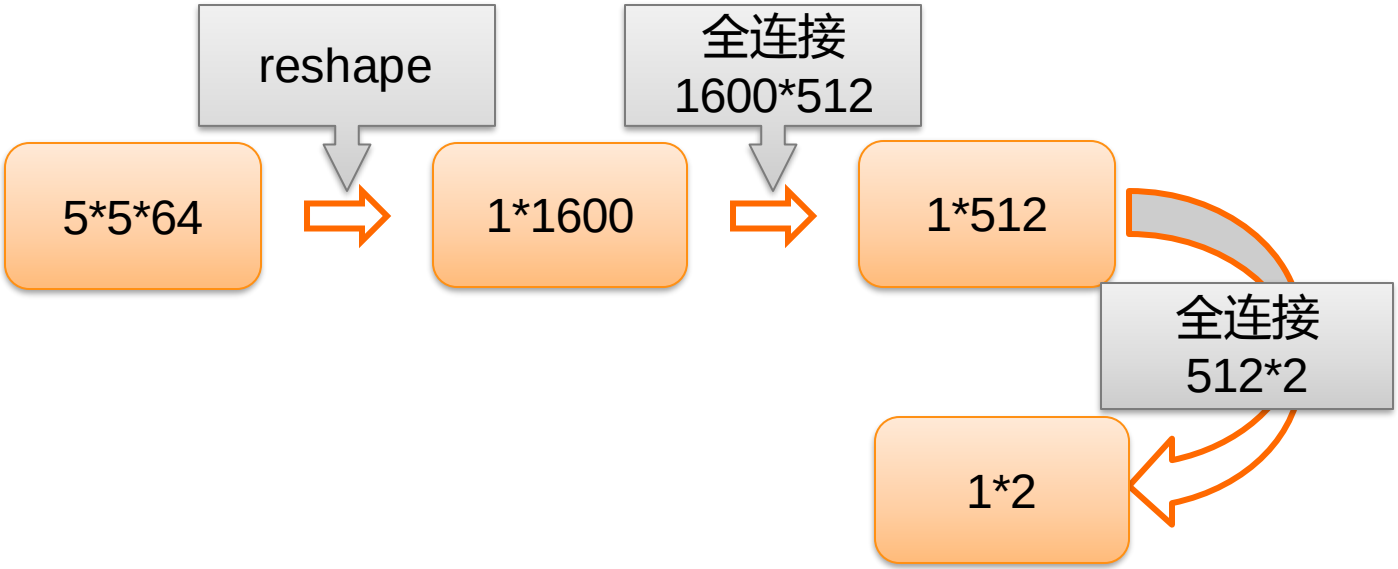
```
>>> def conv2d(x, W, stride):
    return tf.nn.conv2d(x, W, strides=[1,stride,stride,1],
padding = "SAME")
#定义卷积操作，实现卷积核W在数据x上卷积操作
#strides为卷积核的移动步长，padding为卷积的一种模式，参数为same
表示滑动范围超过边界时，
>>> def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1,2,2,1], strides=[1,2,
2,1], padding = "SAME")
#定义池化函数，此程序中通过调用max_pool执行最大池化操作，大小为
2*2， stride步长为2.
```

深度神经网络-框架回顾



本实验中使用的深度神经网络结构就是多个卷积操作和池化操作的累加。

深度神经网络-框架回顾



输出分别对应于两个动作，即不做操作和跳跃的状态动作值函数。

实例程序编写

3. 创建深度神经网络，定义网络结构：

```
def createNetwork():
    #定义深度神经网络的参数和偏置
    W_conv1 = weight_variable([8, 8, 4, 32])
    b_conv1 = bias_variable([32])

    W_conv2 = weight_variable([4, 4, 32, 64])
    b_conv2 = bias_variable([64])

    W_conv3 = weight_variable([3, 3, 64, 64])
    b_conv3 = bias_variable([64])

    W_fc1 = weight_variable([1600, 512])
    b_fc1 = bias_variable([512])

    W_fc2 = weight_variable([512, ACTIONS])
    b_fc2 = bias_variable([ACTIONS])
```

第一个卷积层

第二个卷积层

第三个卷积层

第一个全连接层

第二个全连接层

实例程序编写

. 创建深度神经网络：

```
# 输入层
s = tf.placeholder("float", [None, 80, 80, 4])

# 隐藏层
h_conv1 = tf.nn.relu(conv2d(s, W_conv1, 4) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)

h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2, 2) + b_conv2)

h_conv3 = tf.nn.relu(conv2d(h_conv2, W_conv3, 1) + b_conv3)

h_conv3_flat = tf.reshape(h_conv3, [-1, 1600])

h_fc1 = tf.nn.relu(tf.matmul(h_conv3_flat, W_fc1) + b_fc1)

# 输出层
readout = tf.matmul(h_fc1, W_fc2) + b_fc2

return s, readout, h_fc1
```

输入层，placeholder用于占位，可用作网络的输入

对各层进行连接

实例程序编写

4. 训练深度神经网络-1：

```
def trainNetwork(s, readout, h_fc1, sess):
    # 定义损失函数
    a = tf.placeholder("float", [None, ACTIONS])
    y = tf.placeholder("float", [None])
    readout_action = tf.reduce_sum(tf.multiply(readout, a),
                                   reduction_indices=1)
    cost = tf.reduce_mean(tf.square(y - readout_action))
    train_step = tf.train.AdamOptimizer(1e-6).minimize(cost)

    # 开启游戏模拟器，会打开一个模拟器的窗口，实时显示游戏的信息
    game_state = game.GameState()

    # 创建双端队列用于存放replay memory
    D = deque()
```

定义神经网络训练函数；定义损失函数。

4. 训练深度神经网络-2：

```
# 获取游戏的初始状态，设置动作为不执行跳跃，并将初始状态修改成80*80*4大小
do_nothing = np.zeros(ACTIONS)
do_nothing[0] = 1
x_t, r_0, terminal = game_state.frame_step(do_nothing)
x_t = cv2.cvtColor(cv2.resize(x_t, (80, 80)), cv2.COLOR_BGR2GRAY)
ret, x_t = cv2.threshold(x_t,1,255,cv2.THRESH_BINARY)
s_t = np.stack((x_t, x_t, x_t, x_t), axis=2)

# 用于加载或保存网络参数
saver = tf.train.Saver()
sess.run(tf.initialize_all_variables())
checkpoint = tf.train.get_checkpoint_state("saved_networks")
if checkpoint and checkpoint.model_checkpoint_path:
    saver.restore(sess, checkpoint.model_checkpoint_path)
    print("Successfully loaded:", checkpoint.model_checkpoint_path)
else:
    print("Could not find old network weights")
```

这里需要使用Opencv对图像进行预处理

将像素值大于等于1的像素点处理成255，也就是黑白二值图

实例程序编写

4. 训练深度神经网络-2：

```
# 开始训练
epsilon = INITIAL_EPSILON
t = 0
while "flappy bird" != "angry bird":
    # 使用epsilon贪心策略选择一个动作
    readout_t = readout.eval(feed_dict={s : [s_t]})[0]
    a_t = np.zeros([ACTIONS])
    action_index = 0
    if t % FRAME_PER_ACTION == 0:
        # 执行一个随机动作
        if random.random() <= epsilon:
            print("----------Random Action----------")
            action_index = random.randrange(ACTIONS)
            a_t[random.randrange(ACTIONS)] = 1
        # 由神经网络计算的Q(s,a)值选择对应的动作
        else:
            action_index = np.argmax(readout_t)
            a_t[action_index] = 1
    else:
        a_t[0] = 1 # 不执行跳跃动作
```

实例程序编写

4. 训练深度神经网络-2：

```
# 随游戏的进行，不断降低epsilon，减少随机动作
if epsilon > FINAL_EPSILON and t > OBSERVE:
    epsilon -= (INITIAL_EPSILON - FINAL_EPSILON) / EXPLORE

# 执行选择的动作，并获得下一状态及回报
x_t1_colored, r_t, terminal = game_state.frame_step(a_t)
x_t1 = cv2.cvtColor(cv2.resize(x_t1_colored, (80, 80)),
                                        cv2.COLOR_BGR2GRAY)
ret, x_t1 = cv2.threshold(x_t1, 1, 255, cv2.THRESH_BINARY)
x_t1 = np.reshape(x_t1, (80, 80, 1))
s_t1 = np.append(x_t1, s_t[:, :, :3], axis=2)

# 将状态转移过程存储到D中，用于更新参数时采样
D.append((s_t, a_t, r_t, s_t1, terminal))
if len(D) > REPLAY_MEMORY:
    D.popleft()
```

4. 训练深度神经网络-2：

```
# 过了观察期，才会进行网络参数的更新
if t > OBSERVE:
    # 从D中随机采样，用于参数更新
    minibatch = random.sample(D, BATCH)

    # 分别将当前状态、采取的动作、获得的回报、下一状态分组存放
    s_j_batch = [d[0] for d in minibatch]
    a_batch = [d[1] for d in minibatch]
    r_batch = [d[2] for d in minibatch]
    s_j1_batch = [d[3] for d in minibatch]

    #计算Q(s,a)的新值
    y_batch = []
    readout_j1_batch = readout.eval(feed_dict = {s : s_j1_batch})
    for i in range(0, len(minibatch)):
        terminal = minibatch[i][4]
        # 如果游戏结束，则只有反馈值
        if terminal:
            y_batch.append(r_batch[i])
        else:
            y_batch.append(r_batch[i] +
                           GAMMA * np.max(readout_j1_batch[i]))

    # 使用梯度下降更新网络参数
    train_step.run(feed_dict = {
        y : y_batch,
        a : a_batch,
        s : s_j_batch}
    )
```

实例程序编写

4. 训练深度神经网络-2：

```
# 状态发生改变，用于下次循环
s_t = s_t1
t += 1

# 每进行10000次迭代，保留一下网络参数
if t % 10000 == 0:
    saver.save(sess, 'saved_networks/' + GAME + '-dqn', global_step=t)

# 打印游戏信息
state = ""
if t <= OBSERVE:
    state = "observe"
elif t > OBSERVE and t <= OBSERVE + EXPLORE:
    state = "explore"
else:
    state = "train"

print("TIMESTEP", t, "/ STATE", state, \
    "/ EPSILON", epsilon, "/ ACTION", action_index, "/ REWARD", r_t, \
    "/ Q_MAX %e" % np.max(readout_t))
```

实例程序编写

5. 开启整个训练过程：

```
def playGame():
    sess = tf.InteractiveSession()
    s, readout, h_fc1 = createNetwork()
    trainNetwork(s, readout, h_fc1, sess)

if __name__ == "__main__":
    playGame()
```

前面已经定义深度神经网络创建函数和网络训练函数，开启训练过程

结果展示