第三次实验报告

郑子浩 2017202117

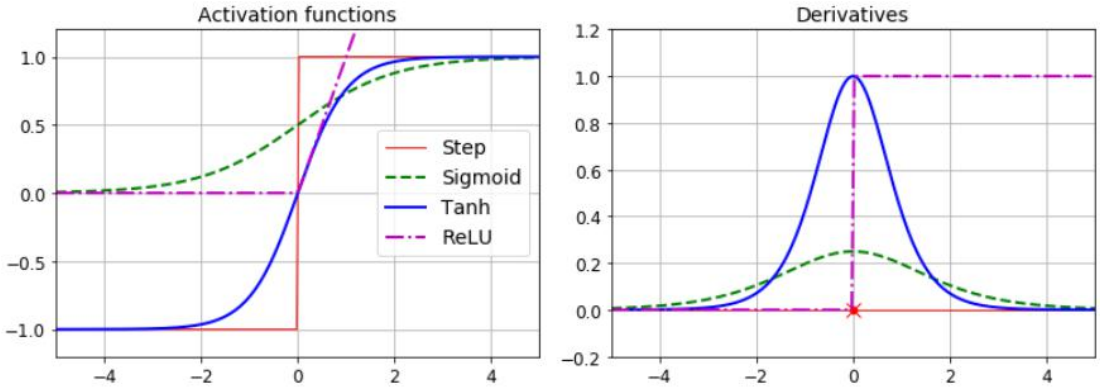
目录

第三次实验报告.....1
一、实验内容：.....2
1、Artificial Neural Networks.....2
1.2、Using plain TensorFlow.....3
1.3、Using dense() instead of neuron_layer().....5
2、Deep learning.....5
2.1、Batch Normalization.....6
2.2、Faster Optimizers：Adam.....6
2.3、Avoiding Overfitting Through Regularization.....7
3、 Policy Gradient.....8
4、DQN.....9
4、DQN 实现：.....12
二、 分析.....15
三、问题及解决方法.....17
1、tensorflow 下载过慢.....17

一、实验内容：

1、Artificial Neural Networks

目前流行的激活函数及其衍生物如下：



1.1、Using the Estimator API

与 TensorFlow 一起训练 MLP 最简单的方法是使用高级 API TF.Learn，这与 sklearn 的 API 非常相似。 DNNClassifier 可以很容易训练具有任意数量隐层的深度神经网络，而 softmax 输 出层输出估计的类概率.下面的代码训练两个隐藏层的 DNN（一个具有 300 个神经 元，另一个具有 100 个神经元）和一个具有 10 个神经元的 SOFTMax 输出层进行分类：

```
import tensorflow as tf
import pandas as pd
from sklearn.model_selection import train_test_split
import tensorflow as tf
import numpy as np

dataset = pd.read_csv('C:/Users/ASUS/Desktop/500.csv')
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)

feature_cols = [tf.feature_column.numeric_column("X", shape=[4* 1])]
dnn_clf = tf.estimator.DNNClassifier(hidden_units=[300,100], n_classes=10,
                                     feature_columns=feature_cols)

input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"X": X_train}, y=y_train, num_epochs=40, batch_size=50, shuffle=True)
dnn_clf.train(input_fn=input_fn)
```

模型评估如下：

```
eval_results:
 {'accuracy': 0.94758064, 'average_loss': 0.12100767, 'loss': 15.0049515, 'global_step': 794}
```

在 open AI 中，采用上述模型进行预测，其预测的核心函数为：

```
test_input_fn = tf.estimator.inputs.numpy_input_fn(
x={"X": np.array(act_pred)}, y=np.array([0]), shuffle=False)#

y_pred_iter = dnn_clf.predict(input_fn=test_input_fn)
y_pred = list(y_pred_iter)
act = y_pred[0]['class_ids'][0]
```

运行结果如下：

```
eval_results:
 {'accuracy': 0.94758064, 'average_loss': 0.12100767, 'loss': 15.0049515, 'global_step': 794}
```

```
stop after 200 steps
 the total reward is 200.0
Execution Time:  93.66106724739075
```

1.2、Using plain TensorFlow

首先我们需要导入 tensorflow 库。 然后我们必须指定输入和输出的数量，并设置 每个层中隐藏的神经元数量：

```
import pandas as pd
from sklearn.model_selection import train_test_split
import tensorflow as tf
import numpy as np

n_inputs = 4*1  # MNIST
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10
```

接下来，使用占位符节点来表示训练数据和目标。X 的形状仅有部分被定义。我们知道它将是一个 2D 张量（即一个矩阵），沿着第一个维度的实例和第二个维度的特征，我们知道特征的数量将是 4×1（每像素一个特征）但是我们不知道每个训练批次将包含多少个实例。所以 X 的形状是(None,n_inputs)。同样，我们知道 y 将是一个 1D 张量，每个实例有一个入口，但是我们再次不知道在这一点上训练批次的大小，所以形状(None)。

```
dataset = pd.read_csv('C:/Users/ASUS/Desktop/500.csv')
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int32, shape=(None), name="y")
```

现在让我们创建一个实际的神经网络。占位符 X 将作为输入层;在执行阶段，它将一次更换一个训练批次（注意训练批中的所有实例将由神经网络同时处理）。现在需要创建两个隐藏层和输出层。两个隐藏的层几乎相同：它们只是它们所连接的输入和它们包含的神经元的数量不同。输出层也非常相似，但它使用 softmax 激活函数而不是 ReLU 激活函数。所以让我们创建一个 neuron_layer()函数，我们将一次创建一个图层。它将需要参数来指定输入，神经元数量，激活函数和图层的名称：

```
def neuron_layer(X, n_neurons, name, activation=None):
    with tf.name_scope(name):
        n_inputs = int(X.get_shape()[1])
        stddev = 2 / np.sqrt(n_inputs)
        init = tf.truncated_normal((n_inputs, n_neurons), stddev=stddev)
        W = tf.Variable(init, name="kernel")
        b = tf.Variable(tf.zeros([n_neurons]), name="bias")
        Z = tf.matmul(X, W) + b
        if activation is not None:
            return activation(Z)
        else:
            return Z
```

好了，现在有一个很好的函数来创建一个神经元层。让我们用它来创建深层神经网络！第一个隐藏层以 X 为输入。第二个将第一个隐藏层的输出作为其输入。最后，输出层将第二个隐藏层的输出作为其输入。

```
with tf.name_scope("dnn"):
    hidden1 = neuron_layer(X, n_hidden1, name="hidden1",
                           activation=tf.nn.relu)
    hidden2 = neuron_layer(hidden1, n_hidden2, name="hidden2",
                           activation=tf.nn.relu)
    logits = neuron_layer(hidden2, n_outputs, name="outputs")

with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y,
                                                              logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")
```

损失函数和评估函数：

```
learning_rate = 0.01

with tf.name_scope("train"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)

with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

执行阶段：

```
init = tf.global_variables_initializer()
saver = tf.train.Saver()

n_epochs = 40
batch_size = 50

def shuffle_batch(X, y, batch_size):
    rnd_idx = np.random.permutation(len(X))
    n_batches = len(X) // batch_size
    for batch_idx in np.array_split(rnd_idx, n_batches):
        X_batch, y_batch = X[batch_idx], y[batch_idx]
        yield X_batch, y_batch

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for X_batch, y_batch in shuffle_batch(X_train, y_train, batch_size):
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        acc_batch = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
        acc_val = accuracy.eval(feed_dict={X: X_test, y: y_test})
        print(epoch, "Batch accuracy:", acc_batch, "Val accuracy:", acc_val)

    save_path = saver.save(sess, "./my_model_final.ckpt")
```

结果如下：

```
39 Batch accuracy: 0.9423077 Val accuracy: 0.9314516
stop after 200 steps
 the total reward is 200.0
Execution Time:  21.915095806121826
```

1.3、Using dense() instead of neuron_layer()

现在最好使用 tf.layers.dense（），因为 contrib 模块中的任何内容可能会更改或删除。dense（）函数与 fully_connected（）函数几乎相同，除了一些细微的差别：
几个参数被重命名：scope 变为名称，activation_fn 变为激活（同样_fn 后缀从其他参数（如 normalizer_fn）中删除），weights_initializer 成为 kernel_initializer 等。默认激活现在是无，而不是 tf.nn.relu。

```
with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1",
                              activation=tf.nn.relu,reuse=True)
    hidden2 = tf.layers.dense(hidden1, n_hidden2, name="hidden2",
                              activation=tf.nn.relu,reuse=True)
    logits = tf.layers.dense(hidden2, n_outputs, name="outputs",reuse=True)
    y_proba = tf.nn.softmax(logits)

with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")

learning_rate = 0.01

with tf.name_scope("train"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)

with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

init = tf.global_variables_initializer()
saver = tf.train.Saver()

n_epochs = 20
n_batches = 50
```

结果如下：

```
19 Batch accuracy: 0.53846157 Validation accuracy: 0.6854839
stop after 91 steps
 the total reward is 91.0
Execution Time:  14.499105453491211
```

2、Deep learning

上文中训练了我们的初步的深度神经网络。 但它非常浅，只有两个隐藏层。当面对非常复杂的问题时，将会面对如下的问题：

首先，你将面临棘手的梯度消失问题（或相关的梯度爆炸问题），这会影响深度神经网络，并使较低层难以训练。

其次，对于如此庞大的网络，训练将非常缓慢。

第三，具有数百万参数的模型将会有严重的过拟合训练集的风险。

接下来将对上述问题逐一作出尝试解决。

2.1、Batch Normalization

反向传播算法的工作原理是从输出层到输入层，传播误差的梯度。 一旦该算法已经计算了网络中每个参数的损失函数的梯度，它就使用这些梯度来用梯度下降步骤来更新每个参数。

不幸的是，梯度往往变得越来越小，随着算法进展到较低层。 结果，梯度下降更新使得低层连接权重实际上保持不变，并且训练永远不会收敛到良好的解决方案。 这被称为梯度消失问题。 在某些情况下，可能会发生相反的情况：梯度可能变得越来越大，许多层得到了非常大的权重更新，算法发散。这是梯度爆炸的问题，在循环神经网络中最为常见。 更一般地说，深度神经网络受梯度不稳定之苦; 不同的层次可能以非常不同的速度学习。

在 2015 年的一篇论文中，Sergey Ioffe 和 Christian Szegedy 提出了一种称为批量标准化（Batch Normalization，BN）的技术来解决梯度消失/爆炸问题，每层输入的分布在训练期间改变的问题，更普遍的问题是当前一层的参数改变，每层输入的分布会在训练过程中发生变化（他们称之为内部协变量偏移问题）。

该技术包括在每层的激活函数之前在模型中添加操作，简单地对输入进行 zero-centering 和规范化，然后每层使用两个新参数（一个用于尺度变换，另一个用于偏移）对结果进行尺度变换和偏移。 换句话说，这个操作可以让模型学习到每层输入值的最佳尺度,均值。为了对输入进行归零和归一化，算法需要估计输入的均值和标准差。 它通过评估当前小批量输入的均值和标准差（因此命名为“批量标准化”）来实现。

TensorFlow 提供了一个 batch_normalization()函数，它简单地对输入进行居中和标准化，但是您必须自己计算平均值和标准差(基于训练期间的小批量数据或测试过程中的完整数据集） 作为这个函数的参数，并且还必须处理缩放和偏移量参数的创建（并将它们传递给此函数）。还要注意，为了在每个隐藏层激活函数之前运行批量标准化，我们手动应用 RELU 激活函数，在批量规范层之后。

最后结果如下：

```
stop after 8 steps
 the total reward is 8.0
Execution Time:  24.62008810043335
```

```
19 Batch accuracy: 0.9465726 Validation accuracy: 0.9354839
```

2.2、Faster Optimizers：Adam

训练一个非常大的深度神经网络可能会非常缓慢。 到目前为止，我们已经看到了四种加速训练的方法（并且达到更好的解决方案）：对连接权重应用良好的初始化策略，使用良好的激活函数，使用批量规范化以及重用预训练网络的部分。 另一个巨大的速度提升来自使用比普通渐变下降优化器更快的优化器。 目前最流行的优化器有：动量优化，Nesterov 加速梯度，AdaGrad，RMSProp，最后是 Adam 优化。

而实际上几乎总是应该使用 Adam_optimization，所以如果不关心它是如何工作的，只需使用 AdamOptimizer 替换您的 GradientDescentOptimizer。 只需要这么小的改动，训练通常会快几倍。 但是，Adam 优化确实有三个可以调整的超参数（加上学习率）。

只需做出如下的修改即可

```
with tf.name_scope('train'):
    optimizer = tf.train.AdamOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)
```

结果如下：

```
19 Batch accuracy: 1.0 Validation accuracy: 0.96370965

stop after 9 steps
 the total reward is 9.0
Execution Time:  1.5688748359680176
```

2.3、Avoiding Overfitting Through Regularization

深度神经网络通常具有数以万计的参数，有时甚至是数百万。 有了这么多的参数，网络拥有难以置信的自由度，可以适应各种复杂的数据集。 但是这个很大的灵活性也意味着它很容易过拟合训练集。通过神经网络正则化技术，实现 tensorflow 避免过拟合，l1 和 l2 正则化，drop out，最大范数正则化和数据增强。

可以使用 l1 和 l2 正则化约束一个神经网络的连接权重（但通常不是它的偏置）。

使用 TensorFlow 做到这一点的一种方法是简单地将适当的正则化项添加到您的损失函数中。 例如，假设您只有一个权重为 weights1 的隐藏层和一个权重为 weight2 的输出层，那么可以像这样应用 l1 正则化：

我们可以将正则化函数传递给 tf.layers.dense()函数，该函数将使用它来创建计算正则化损失的操作，并将这些操作添加到正则化损失集合中。 接下来，我们将使用 Python partial()函数来避免一遍又一遍地重复相同的参数。 请注意，我们设置了内核正则化参数（正则化函数有 l1_regularizer()，l2_regularizer()，l1_l2_regularizer()）：

```
scale = 0.001

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name = 'X')
y = tf.placeholder(tf.int64, shape=None, name = 'y')
training = tf.placeholder_with_default(False, shape=(), name = 'training')#给Batch norm加一个placeholder

my_dense_layer = partial(
    tf.layers.dense, activation=tf.nn.relu,
    kernel_regularizer=tf.contrib.layers.l1_regularizer(scale))

with tf.name_scope("dnn"):
    hidden1 = my_dense_layer(X, n_hidden1, name="hidden1",reuse=tf.AUTO_REUSE)
    hidden2 = my_dense_layer(hidden1, n_hidden2, name="hidden2",reuse=tf.AUTO_REUSE)
    logits = my_dense_layer(hidden2, n_outputs, activation=None,
                            name="outputs",reuse=tf.AUTO_REUSE)
```

该代码创建了一个具有两个隐藏层和一个输出层的神经网络，并且还在图中创建节点以计算与每个层的权重相对应的 l1 正则化损失。TensorFlow 会自动将这些节点添加到包含所有正则化损失的特殊集合中。接下来，我们必须将正则化损失加到基本损失上：

```
with tf.name_scope("loss"):                                     # not shown in the book
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(  # not shown
        labels=y, logits=logits)                                # not shown
    base_loss = tf.reduce_mean(xentropy, name="avg_xentropy")   # not shown
    reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
    loss = tf.add_n([base_loss] + reg_losses, name="loss")
```

其他如常。

最后结果如下：

```
19 Batch accuracy: 0.9576613 Validation accuracy: 0.9677419

stop after 10 steps
 the total reward is 10.0
Execution Time:  1.0142643451690674
```

3、Policy Gradient

让我们创建一个神经网络策略。就像之前我们编码的策略一样，这个神经网络将把观察作为输入，输出要执行的动作。更确切地说，它将估计每个动作的概率，然后我们将根据估计的概率随机地选择一个动作。在 CartPole 环境中，只有两种可能的动作（左或右），所以我们只需要一个输出神经元。它将输出动作 0（左）的概率 p，动作 1（右）的概率显然将是 1 - p。

通读代码：

1、在导入之后，我们定义了神经网络体系结构。输入的数量是观测空间的大小（在 CartPole 的情况下是 4 个），我们只有 4 个隐藏单元，并且不需要更多，并且我们只有 1 个输出概率（向左的概率）。

2、接下来我们构建了神经网络。在这个例子中，它是一个 vanilla 多层感知器，只有一个输出。注意，输出层使用 Logistic（Sigmoid）激活函数，以便输出从 0 到 1 的概率。如果有两个以上的可能动作，每个动作都会有一个输出神经元，相应的你将使用 Softmax 激活函数。

3、最后，我们调用 multinomial()函数来选择一个随机动作。该函数独立地采样一个（或多个）整数，给定每个整数的对数概率。例如，如果通过设置 num_samples=5，令数组为[np.log(0.5), np.log(0.2), np.log(0.3)]来调用它，那么它将输出五个整数，每个整数都有 50% 的概率是 0，20% 为 1，30% 为 2。在我们的情况下，我们只需要一个整数来表示要采取的行动。由于输出张量（output）仅包含向左的概率，所以我们必须首先将 1 - output 连接它，以得到包含左和右动作的概率的张量。请注意，如果有两个以上的可能动作，神经网络将不得不输出每个动作的概率，这时你就不需要连接步骤了。

好了，现在我们有一个可以观察和输出动作的神经网络了。

```
import tensorflow as tf
from tensorflow.contrib.layers import fully_connected
# 1. 声明神经网络结构
n_inputs = 4  # == env.observation_space.shape[0]
n_hidden = 4  # 这只是个简单的测试，不需要过多的隐藏层
n_outputs = 1 # 只输出向左加速的概率
initializer = tf.contrib.layers.variance_scaling_initializer()
# 2. 建立神经网络
X = tf.placeholder(tf.float32, shape=[None, n_inputs]) hidden = fully_connected(X, n_hidden, activation_fn=tf.nn.elu,weights_initiali
logits = fully_connected(hidden, n_outputs, activation_fn=None,weights_initializer=initializer)
outputs = tf.nn.sigmoid(logits)
# 3. 在概率基础上随机选择动作
p_left_and_right = tf.concat(axis=1, values=[outputs, 1 - outputs])
action = tf.multinomial(tf.log(p_left_and_right), num_samples=1)
init = tf.global_variables_initializer()
```

如果我们知道每一步的最佳动作，我们可以像通常一样训练神经网络，通过最小化估计概率和目标概率之间的交叉熵。这只是通常的监督学习。然而，在强化学习中，智能体获得的指导的唯一途径是通过奖励，奖励通常是稀疏的和延迟的。例如，如果智能体在 100 个步骤内设法平衡杆，它怎么知道它采取的 100 个行动中的哪一个是好的，哪些是坏的？它所知道的是，在最后一次行动之后，杆子坠落了，但最后一次行动肯定不是完全负责的。这被称为信用分配问题：当智能体得到奖励时，很难知道哪些行为应该被信任（或责备）。想想一只狗在行为良好后几小时就会得到奖励，它会明白它得到了什么回报吗？

为了解决这个问题，一个通常的策略是基于这个动作后得分的总和来评估这个个动作，通常在每个步骤中应用衰减率 r。例如，如果一个智能体决定连续三次向右，在第一步之后得到 +10 奖励，第二步后得到 0，最后在第三步之后得到 -50，然后假设我们使用衰减率 r=0.8，那么第一个动作将得到 10 +r×0 + r2×(-50)=-22 的分述。如果衰减率接近 0，那么与即时奖励相比，未来的奖励不会有多大意义。相反，如果衰减率接近 1，那么对未来的奖励几

乎等于即时回报。典型的衰减率通常为是 0.95 或 0.99。如果衰减率为 0.95，那么未来 13 步的奖励大约是即时奖励的一半（0.9513×0.5），而当衰减率为 0.99，未来 69 步的奖励是即时奖励的一半。在 CartPole 环境下，行为具有相当短期的影响，因此选择 0.95 的折扣率是合理的。

正如前面所讨论的，PG 算法通过遵循更高回报的梯度来优化策略参数。一种流行的 PG 算法，称为增强算法，在 1929 由 Ronald Williams 提出。这是一个常见的变体：

1、首先，让神经网络策略玩几次游戏，并在每一步计算梯度，这使得智能体更可能选择行为，但不应用这些梯度。

2、运行几次后，计算每个动作的得分（使用前面段落中描述的方法）。

3、如果一个动作的分数是正的，这意味着动作是好的，可应用较早计算的梯度，以便将来有更大的的概率选择这个动作。但是，如果分数是负的，这意味着动作是坏的，要应用负梯度来使得这个动作在将来采取的可能性更低。我们的方法就是简单地将每个梯度向量乘以相应的动作得分。

4、最后，计算所有得到的梯度向量的平均值，并使用它来执行梯度下降步骤。

让我们使用 TensorFlow 实现这个算法。我们将训练我们早先建立的神经网络策略，让它学会平衡车上的平衡杆。让我们从完成之前编码的构造阶段开始，添加目标概率、代价函数和训练操作。因为我们的意愿是选择的动作是最好的动作，如果选择的动作是动作 0（左），则目标概率必须为 1，如果选择动作 1（右）则目标概率为 0：

```
y = 1. - tf.to_float(action)
```

现在我们有一个目标概率，我们可以定义损失函数（交叉熵）并计算梯度：

```
learning_rate = 0.01
cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(                    labels=y, logits=logits)
optimizer = tf.train.AdamOptimizer(learning_rate)
grads_and_vars = optimizer.compute_gradients(cross_entropy)
```

注意，我们正在调用优化器的 compute_gradients()方法，而不是 minimize()方法。这是因为我们想要在使用它们之前调整梯度。compute_gradients()方法返回梯度向量/变量对的列表（每个可训练变量一对）。让我们把所有的梯度放在一个列表中，以便方便地获得它们的值：

```
gradients = [grad for grad, variable in grads_and_vars]
```

好，现在是棘手的部分。在执行阶段，算法将运行策略，并在每个步骤中评估这些梯度张量并存储它们的值。在多次运行之后，它如先前所解释的调整这些梯度（即，通过动作分数乘以它们并使它们归一化），并计算调整后的梯度的平均值。接下来，需要将结果梯度反馈到优化器，以便它可以执行优化步骤。这意味着对于每一个梯度向量我们需要一个占位符。此外，我们必须创建操作去应用更新的梯度。为此，我们将调用优化器的 apply_gradients()函数，该函数接受梯度向量/变量对的列表。我们不给它原始的梯度向量，而是给它一个包含更新梯度的列表（即，通过占位符递送的梯度）：

```
gradient_placeholders = []
grads_and_vars_feed = []
for grad, variable in grads_and_vars:
    gradient_placeholder = tf.placeholder(tf.float32, shape=grad.get_shape())
    gradient_placeholders.append(gradient_placeholder)
    grads_and_vars_feed.append((gradient_placeholder, variable))
training_op = optimizer.apply_gradients(grads_and_vars_feed)
```

4、DQN

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

编写主程序：

按照至上而下的编程方式，我们先写主函数用来执行这个实验，然后再具体编写 DQN 算法实现。

先 import 所需的库：

```
import gym
import tensorflow as tf
import numpy as np
import random
from collections import deque
```

编写主函数如下：

```
# Hyper Parameters
ENV_NAME = 'CartPole-v0'
EPISODE = 10000 # Episode limitation
STEP = 300 # Step limitation in an episode

def main():
  # initialize OpenAI Gym env and dqn agent
  env = gym.make(ENV_NAME)
  agent = DQN(env)

  for episode in xrange(EPISODE):
    # initialize task
    state = env.reset()
    # Train
    for step in xrange(STEP):
      action = agent.egreedy_action(state) # e-greedy action for train
      next_state,reward,done,_ = env.step(action)
      # Define reward for agent
      reward_agent = -1 if done else 0.1
      agent.perceive(state,action,reward,next_state,done)
      state = next_state
      if done:
        break

if __name__ == '__main__':
  main()
```

我们将编写一个 DQN 的类，DQN 的一切都将封装在里面。在主函数中，我们只需调用

```
agent.egreedy_action(state) # 获取包含随机的动作
agent.perceive(state,action,reward,next_state,done) # 感知信息
```

本质上就是一个输出动作，一个输入状态。当然我们这里输入的是整个 transition。

然后环境自己执行动作，输出新的状态：

```
next_state,reward,done,_ = env.step(action)
```

然后整个过程就反复循环，一个 episode 结束，就再来一个。

这就是训练的过程。

但只有训练显然不够，我们还需要测试。因此，在 main()的最后，我们再加上几行的测试代码：

```
# Test every 100 episodes
    if episode % 100 == 0:
      total_reward = 0
      for i in xrange(TEST):
        state = env.reset()
        for j in xrange(STEP):
          env.render()
          action = agent.action(state) # direct action for test
          state,reward,done,_ = env.step(action)
          total_reward += reward
          if done:
            break
      ave_reward = total_reward/TEST
      print 'episode: ',episode,'Evaluation Average Reward:',ave_reward
      if ave_reward >= 200:
        break
```

测试中唯一的不同就是我们使用

```
action = agent.action(state)
```

来获取动作，也就是完全没有随机性，只根据神经网络来输出，没有探索，同时这里也就不再 perceive 输入信息来训练。

最后结果如下：

```
episode: 42   reward: 114.0
Execution Time:  0.061577558517456055
```

4、DQN 实现：

```
class DQN():
  # DQN Agent
  def __init__(self, env): #初始化

  def create_Q_network(self): #创建Q网络

  def create_training_method(self): #创建训练方法

  def perceive(self,state,action,reward,next_state,done): #感知存储信息

  def train_Q_network(self): #训练网络

  def egreedy_action(self,state): #输出带随机的动作

  def action(self,state): #输出动作
```

主要只需要以上几个函数。上面已经注释得很清楚，这里不再加以解释。

我们知道，我们的 DQN 一个很重要的功能就是要能存储数据，然后在训练的时候 minibatch 出来。所以，我们需要构造一个存储机制。这里使用 deque 来实现。

```
self.replay_buffer = deque()
```

初始化：

这里要注意一点就是 egreedy 的 epsilon 是不断变小的，也就是随机性不断变小。怎么理解呢？就是一开始需要更多的探索，所以动作偏随机，慢慢的我们需要动作能够有效，因此减少随机。

```
def __init__(self, env):
  # init experience replay
  self.replay_buffer = deque()
  # init some parameters
  self.time_step = 0
  self.epsilon = INITIAL_EPSILON
  self.state_dim = env.observation_space.shape[0]
  self.action_dim = env.action_space.n

  self.create_Q_network()
  self.create_training_method()

  # Init session
  self.session = tf.InteractiveSession()
  self.session.run(tf.initialize_all_variables())
```

创建 Q 网络：

我们这里创建最基本的 MLP，中间层设置为 20：

```
def create_Q_network(self):
  # network weights
  W1 = self.weight_variable([self.state_dim,20])
  b1 = self.bias_variable([20])
  W2 = self.weight_variable([20,self.action_dim])
  b2 = self.bias_variable([self.action_dim])
  # input layer
  self.state_input = tf.placeholder("float",[None,self.state_dim])
  # hidden layers
  h_layer = tf.nn.relu(tf.matmul(self.state_input,W1) + b1)
  # Q Value layer
  self.Q_value = tf.matmul(h_layer,W2) + b2

def weight_variable(self,shape):
  initial = tf.truncated_normal(shape)
  return tf.Variable(initial)

def bias_variable(self,shape):
  initial = tf.constant(0.01, shape = shape)
  return tf.Variable(initial)
```

只有一个隐层，然后使用 relu 非线性单元。相信对 MLP 有了解的知友看上面的代码很 easy！要注意的是我们 state 输入的格式，因为使用 minibatch，所以格式是[None,state_dim]

编写 perceive 函数：

```
def perceive(self,state,action,reward,next_state,done):
  one_hot_action = np.zeros(self.action_dim)
  one_hot_action[action] = 1
   self.replay_buffer.append((state,one_hot_action,reward,next_state,done))
  if len(self.replay_buffer) > REPLAY_SIZE:
    self.replay_buffer.popleft()

  if len(self.replay_buffer) > BATCH_SIZE:
    self.train_Q_network()
```

这里需要注意的一点就是动作格式的转换。我们在神经网络中使用的是 one hot key 的形式，而在 OpenAI Gym 中则使用单值。什么意思呢？比如我们输出动作是 1，那么对应的 one hot 形式就是[0,1]，如果输出动作是 0，那么 one hot 形式就是[1,0]。这样做的目的是为了之后更好的进行计算。

在 perceive 中一个最主要的事情就是存储。然后根据情况进行 train。这里我们要求只要存储的数据大于 Batch 的大小就开始训练。

编写 action 输出函数：

```
def egreedy_action(self,state):
    Q_value = self.Q_value.eval(feed_dict = {
      self.state_input:[state]
      })[0]
    if random.random() <= self.epsilon:
      return random.randint(0,self.action_dim - 1)
    else:
      return np.argmax(Q_value)

    self.epsilon -= (INITIAL_EPSILON - FINAL_EPSILON)/10000

def action(self,state):
    return np.argmax(self.Q_value.eval(feed_dict = {
      self.state_input:[state]
      })[0])
```

区别之前已经说过，一个是根据情况输出随机动作，一个是根据神经网络输出。由于神经网络输出的是每一个动作的 Q 值，因此我们选择最大的那个 Q 值对应的动作输出。

编写 training method 函数：

```
def create_training_method(self):
  self.action_input = tf.placeholder("float",[None,self.action_dim]) # one hot presentation
  self.y_input = tf.placeholder("float",[None])
  Q_action = tf.reduce_sum(tf.mul(self.Q_value,self.action_input),reduction_indices = 1)
  self.cost = tf.reduce_mean(tf.square(self.y_input - Q_action))
  self.optimizer = tf.train.AdamOptimizer(0.0001).minimize(self.cost)
```

这里的 y_input 就是 target Q 值。我们这里采用 Adam 优化器，其实随便选择一个必然 SGD，RMSProp 都是可以的。可能比较不好理解的就是 Q 值的计算。这里大家记住动作输入是 one hot key 的形式，因此将 Q_value 和 action_input 向量相乘得到的就是这个动作对应的 Q_value。然后用 reduce_sum 将数据维度压成一维。

编写 training 函数：

```
def train_Q_network(self):
    self.time_step += 1
    # Step 1: obtain random minibatch from replay memory
    minibatch = random.sample(self.replay_buffer,BATCH_SIZE)
    state_batch = [data[0] for data in minibatch]
    action_batch = [data[1] for data in minibatch]
    reward_batch = [data[2] for data in minibatch]
    next_state_batch = [data[3] for data in minibatch]

    # Step 2: calculate y
    y_batch = []
    Q_value_batch = self.Q_value.eval(feed_dict={self.state_input:next_state_batch})
    for i in range(0,BATCH_SIZE):
      done = minibatch[i][4]
      if done:
        y_batch.append(reward_batch[i])
      else :
        y_batch.append(reward_batch[i] + GAMMA * np.max(Q_value_batch[i]))

    self.optimizer.run(feed_dict={
      self.y_input:y_batch,
      self.action_input:action_batch,
      self.state_input:state_batch
      })
```

首先就是进行 minibatch 的工作，然后根据 batch 计算 y_batch。最后就是用 optimizer 进行优化。

最后结果如下：

```
score: 200.0
Execution Time:  3.5551085472106934
```

二、分析

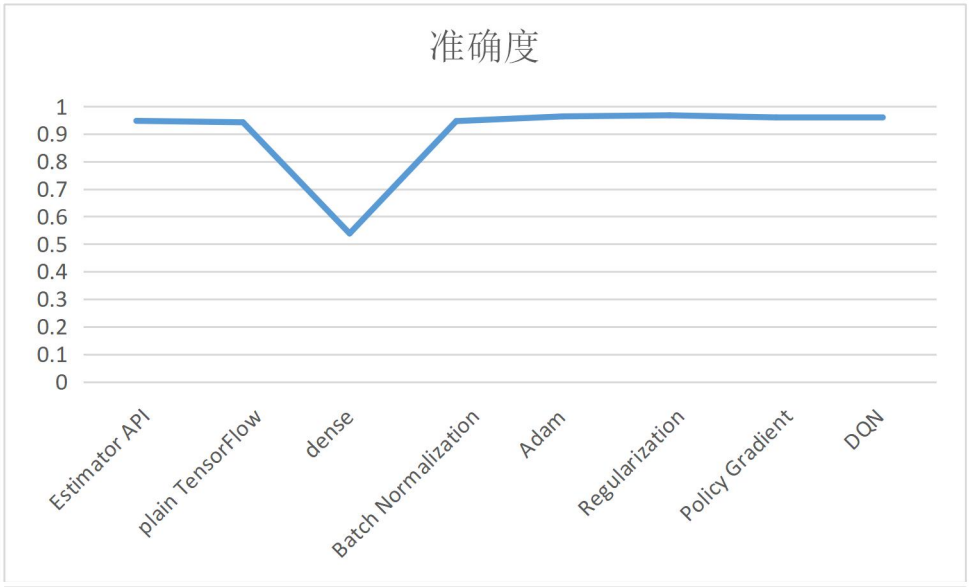
实验结果汇总：

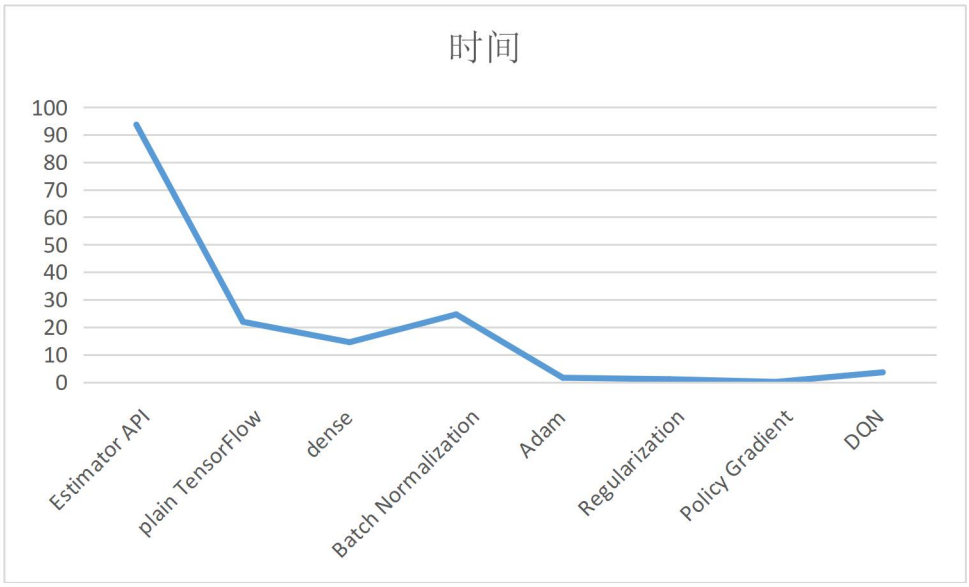
方法	准确度	时间	分数
Estimator API	0.9475	93.66	200
plain TensorFlow	0.9423	21.92	200
dense	0.5385	14.50	91
Batch Normalization	0.9466	24.62	18
Adam	0.9637	1.56	10
Regularization	0.9677	1.04	10
Policy Gradient	0.96	0.06	114
DQN	0.96	3.55	200

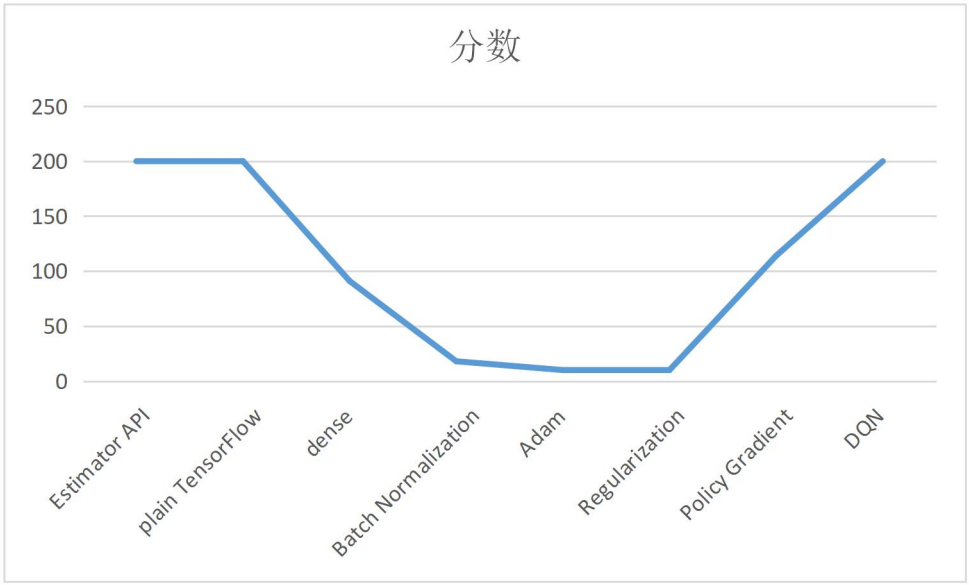
首先揭示分数与时间的含义：

是指平均情况下，每一场（当 done 等于 1 时结束一场），所花费的时间和所获得的分数。

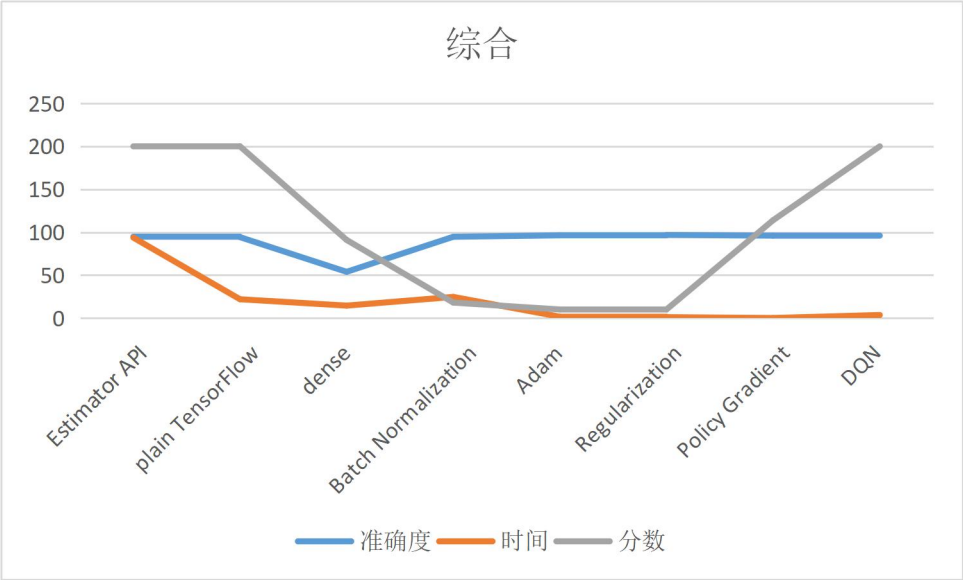
可视化如下：


准确度
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Estimator API
plain TensorFlow
dense
Batch Normalization
Adam
Regularization
Policy Gradient
DQN


时间
100
90
80
70
60
50
40
30
20
10
0
Estimator API
plain TensorFlow
dense
Batch Normalization
Adam
Regularization
Policy Gradient
DQN


分数
250
200
150
100
50
0
Estimator API
plain TensorFlow
dense
Batch Normalization
Adam
Regularization
Policy Gradient
DQN

总体来看：



原则上应该是分数越高越好；时间越短越好。

综合来看 DQN 方法最好。

而从折线图上看，我们发现 dense 分数低的原因主要是准确度太低，欠拟合；

而 Batch Normalization、Adam、Regularization 虽然准确度很高但是分数很低，这是因为准确度过高导致过拟合。

三、问题及解决方法

1、tensorflow 下载过慢

一般不建议采用 pip install tensorflow-gpu1.15.0 --upgrade tensorflow-gpu 方式，这种方式需要翻墙而且下载速度超级慢。可以使用国内镜像，pip install -i https://pypi.tuna.tsinghua.edu.cn/simple/ --upgrade tensorflow-gpu1.15.0