人工智能与量化投资--一个简单的股票预测深度学习模型

Q科技 2020-01-13 11:19:37

使用TensorFlow来处理数据并构建深度学习模型是一件非常激动人心的事情，本文将构建一个使用TensorFlow来预测股价走势的范例。本文将对如果构建一个基础的预测模型进行介绍，文中的Python代码没有针对效率进行优化，而是更偏重可理解性。

请注意，实际的股票价格预测是一项非常具有挑战性和复杂性的任务，需要付出巨大的努力，尤其是在更高的频率的策略下，比如你可以使用分钟甚至秒作为预测周期。

数据

我们可以使用Python抓取一些数据存入csv文件。该数据集包含41266分钟的数据，范围从2017年4月到8月，500种股票以及标准普尔500指数的价格。

```
# Import data
```

```
data = pd.read_csv('data_stocks.csv')
```

```
# Drop date variable
```

```
data = data.drop(['DATE'], 1)
```

```
# Dimensions of dataset
```

```
n = data.shape[0]
```

```
p = data.shape[1]
```

```
# Make data a numpy array
```

```
data = data.values
```

对于数据，我们预先做了处理，所以不存在无效数据。建议大家对数据进行清洗和处理后，再进行存储，这样你就不需要每次使用数据时都去处理一遍，但是和模型相关的一些数据处理，我们可以放在模型相关的代码里去执行，这方便我们去使用不同的模型。

数据集分为训练和测试数据。训练数据包含总数据集的80%。数据没有打乱，而是按顺序切片。训练数据范围从4月到7月底，测试数据则从7月底到8月底结束。

```
# Training and test data
```

```
train_start = 0
```

```
train_end = int(np.floor(0.8*n))
```

```
test_start = train_end
```

```
test_end = n
```

```
data_train = data[np.arange(train_start, train_end), :]
```

```
data_test = data[np.arange(test_start, test_end), :]
```

时间序列交叉验证有很多不同的方法，例如滚动预测，或者时间序列重新采样。

数据归一化

大多数神经网络模型都需要做归一化处理。为什么？因为神经元的最常见激活函数，例如tanh或sigmoid，分别在[-1,1]或[0,1]间隔上定义。如今，ReLU激活是常用的激活，其在可能的激活值的轴上是无界的。但是，无论如何，我们都会使用归一化。使用sklearn的MinMaxScaler可以在Python中轻松完成归一化。

```
# Scale data
```

```
from sklearn.preprocessing import MinMaxScaler
```

```
scaler = MinMaxScaler()
```

```
data_train = scaler.fit_transform(data_train)
```

```
data_test = scaler.transform(data_test)
```

```
# Build X and y
```

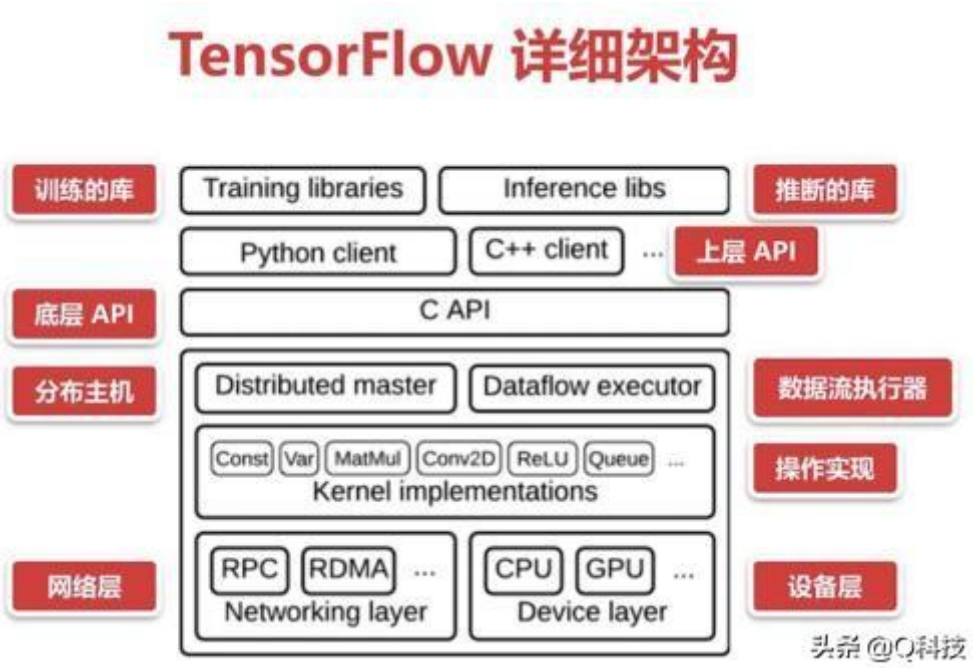
```
X_train = data_train[:, 1:]
```

```
y_train = data_train[:, 0]
```

```
X_test = data_test[:, 1:]
```

```
y_test = data_test[:, 0]
```

注意：必须注意归一数据的哪个部分以及何时归一。常见的错误是在应用训练和测试分割之前缩放整个数据集。为什么这是一个错误？因为缩放会调用统计数据，例如变量的最小值/最大值。在现实生活中进行时间序列预测时，您在预测时没有来自未来观测的信息。因此，必须对训练数据进行缩放统计的计算，然后必须将其应用于测试数据。否则，您在预测时使用未来信息，这通常会使预测指标偏向正向。模型对于任何你透露的未来信息都绝对不会错过。



TensorFlow简介

TensorFlow是一个伟大的软件，目前是领先的深度学习和神经网络计算框架。它基于C ++低层后端，但通常通过Python控制（R还有一个简洁的TensorFlow库，由RStudio维护）。 TensorFlow对基础计算任务的图形表示进行操作。该方法允许用户将数学运算指定为数据，变量和运算符图中的元素。由于神经网络实际上是数据和数学运算的图形，因此TensorFlow非常适合神经网络和深度学习。如下是一段示例代码：

```
# Import TensorFlow
```

```
import tensorflow as tf
```

```
# Define a and b as placeholders
```

```
a = tf.placeholder(dtype=tf.int8)
```

```
b = tf.placeholder(dtype=tf.int8)
```

```
# Define the addition
```

```
c = tf.add(a, b)
```

```
# Initialize the graph
```

```
graph = tf.Session()
```

```
# Run the graph
```

```
graph.run(c, feed_dict={a: 5, b: 4})
```

导入TensorFlow库后，使用tf.placeholder（）定义了两个占位符。然后，通过tf.add（）定义数学加法。计算结果为c = 9。设置占位符后，可以使用a和b的任何整数值执行图形。当然，神经网络中所需的图和计算要复杂得多。

占位符

为了适应我们的模型，我们需要两个占位符：X包含网络的输入（在时间T = t时所有标准普尔500指数成份股的股票价格）和Y网络的输出（在时间T = t +时标准普尔500指数的指数值） 1）。

占位符的形状对应于[None，n_stocks]，[None]表示输入是2维矩阵，输出是1维向量。了解神经网络需要哪些输入和输出维度以便正确设计它至关重要。

```
# Placeholder
```

```
X = tf.placeholder(dtype=tf.float32, shape=[None, n_stocks])
```

```
Y = tf.placeholder(dtype=tf.float32, shape=[None])
```

None参数表示此时我们还不知道每批中流经神经网络图的观测数量，因此我们保持灵活性。稍后我们将定义变量batch_size，它控制每个训练批次的观察次数。

变量

除了占位符，变量是TensorFlow的另一个基石。权重和偏差表示为变量，以便在训练期间适应。在模型训练之前，需要初始化变量。稍后我们将更详细地讨论这个问题。

该模型由四个隐藏层组成。第一层包含1024个神经元，略大于输入大小的两倍。后续的隐藏层总是前一层的一半大小，这意味着512,256和最后128个神经元。每个后续层的神经元数量的减少压缩了网络在先前层中识别的信息。

```
# Model architecture parameters
```

```
n_stocks = 500
```

```
n_neurons_1 = 1024
```

```
n_neurons_2 = 512
```

```
n_neurons_3 = 256
```

```
n_neurons_4 = 128
```

```
n_target = 1
```

```
# Layer 1: Variables for hidden weights and biases
```

```
W_hidden_1 = tf.Variable(weight_initializer([n_stocks, n_neurons_1]))
```

```
bias_hidden_1 = tf.Variable(bias_initializer([n_neurons_1]))
```

```
# Layer 2: Variables for hidden weights and biases
```

```
W_hidden_2 = tf.Variable(weight_initializer([n_neurons_1, n_neurons_2]))
```

```
bias_hidden_2 = tf.Variable(bias_initializer([n_neurons_2]))
```

```
# Layer 3: Variables for hidden weights and biases
```

```
W_hidden_3 = tf.Variable(weight_initializer([n_neurons_2, n_neurons_3]))
```

```
bias_hidden_3 = tf.Variable(bias_initializer([n_neurons_3]))
```

```
# Layer 4: Variables for hidden weights and biases
```

```
W_hidden_4 = tf.Variable(weight_initializer([n_neurons_3, n_neurons_4]))
```

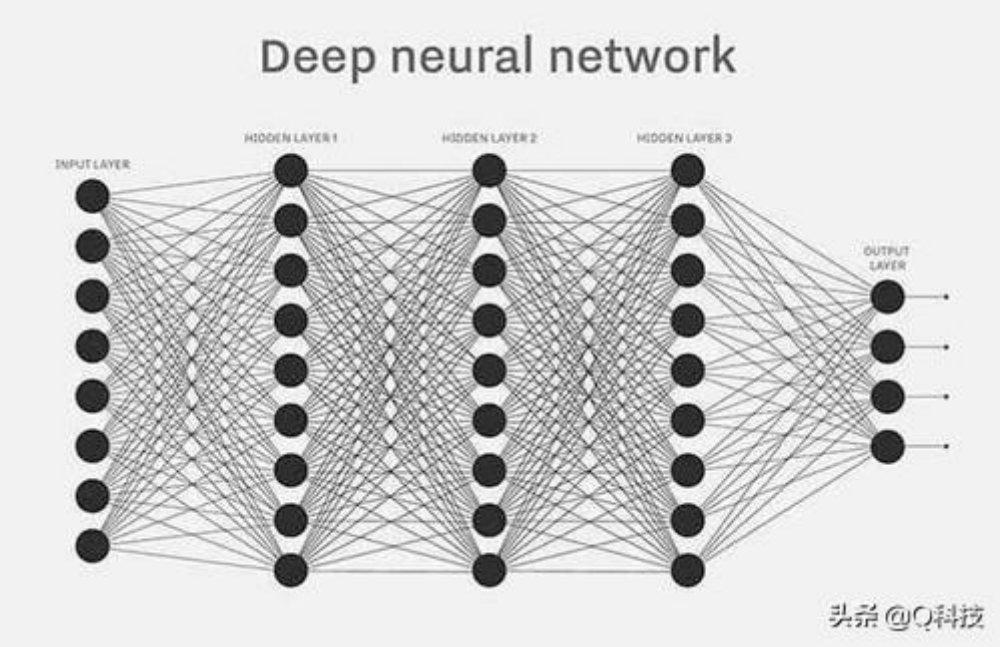
```
bias_hidden_4 = tf.Variable(bias_initializer([n_neurons_4]))
```

```
# Output layer: Variables for output weights and biases
```

```
W_out = tf.Variable(weight_initializer([n_neurons_4, n_target]))
```

```
bias_out = tf.Variable(bias_initializer([n_target]))
```

了解输入，隐藏和输出层之间所需的变量维度非常重要。作为多层感知器（MLP，此处使用的网络类型）的经验法则，前一层的第二维是当前层中权重矩阵的第一维。这可能听起来很复杂，但基本上只是每个层将其输出作为输入传递给下一层。偏差维度等于当前图层权重矩阵的第二维，它对应于该图层中的神经元数量。



设计网络架构

在定义了所需的权重和偏差变量之后，需要指定网络拓扑，网络的体系结构。因此，需要将占位符（数据）和变量（权重和偏差）组合成顺序矩阵乘法系统。

此外，网络的隐藏层由激活功能转换。激活函数是网络体系结构的重要元素，因为它们会给系统带来非线性。有许多可能的激活功能，其中最常见的是ReLU，它也将用于该模型。

```
# Hidden layer
```

```
hidden_1 = tf.nn.relu(tf.add(tf.matmul(X, W_hidden_1), bias_hidden_1))
```

```
hidden_2 = tf.nn.relu(tf.add(tf.matmul(hidden_1, W_hidden_2), bias_hidden_2))
```

```
hidden_3 = tf.nn.relu(tf.add(tf.matmul(hidden_2, W_hidden_3), bias_hidden_3))
```

```
hidden_4 = tf.nn.relu(tf.add(tf.matmul(hidden_3, W_hidden_4), bias_hidden_4))
```

```
# Output layer (must be transposed)
```

```
out = tf.transpose(tf.add(tf.matmul(hidden_4, W_out), bias_out))
```

该模型由三个主要构建块组成。输入层，隐藏层和输出层。该体系结构称为前馈网络。前馈表示该批数据仅从左向右流动。其他网络架构，例如递归神经网络，也允许数据在网络中“向后”流动。

损失函数

网络的成本函数用于生成网络预测与实际观察到的训练目标之间的偏差度量。对于回归问题，通常使用均方误差（MSE）函数。 MSE计算预测和目标之间的平均平方偏差。基本上，可以实现任何可微函数以便计算预测和目标之间的偏差度量。

```
# Cost function
```

```
mse = tf.reduce_mean(tf.squared_difference(out, Y))
```

然而，MSE表现出某些特性，这些特性对于要解决的一般优化问题是有利的。

优化

优化器负责在训练期间用于调整网络权重和偏差变量的必要计算。这些计算调用所谓的梯度的计算，其指示在训练期间必须改变权重和偏差的方向，以便最小化网络的成本函数。稳定和快速优化

是神经网络的一个主要领域。

```
# Optimizer
```

```
opt = tf.train.AdamOptimizer().minimize(mse)
```

这里使用了Adam Optimizer，它是深度学习开发中当前的默认优化器之一。 Adam代表“自适应力矩估计”，可以视为两个其他流行优化器AdaGrad和RMSProp之间的组合。

初始化器

初始化器用于在训练之前初始化网络变量。由于神经网络是使用数值优化技术训练的，因此优化问题的出发点是找到底层问题的良好解决方案的关键因素之一。 TensorFlow中有不同的初始化程序，每个初始化程序都有不同的初始化方法。在这里，我使用tf.variance_scaling_initializer（），这是默认的初始化策略之一。

```
# Initializers
```

```
sigma = 1
```

```
weight_initializer = tf.variance_scaling_initializer(mode="fan_avg",
distribution="uniform", scale=sigma)
```

```
bias_initializer = tf.zeros_initializer()
```

注意，使用TensorFlow，可以为图中的不同变量定义多个初始化函数。但是，在大多数情况下，统一初始化就足够了。

神经网络

在定义了网络的占位符，变量，初始值设定项，成本函数和优化程序之后，需要对模型进行训练。通常，这是通过minibatch培训完成的。在批训练期间，从训练数据中抽取n = batch_size的随机数据样本并将其馈送到网络中。训练数据集分为n / batch_size批次，这些批次按顺序送入网络。此时占位符X和Y开始起作用。它们存储输入和目标数据，并将它们作为输入和目标呈现给网络。

X的采样数据批次流经网络，直到到达输出层。在那里，TensorFlow将模型预测与当前批次中实际观察到的目标Y进行比较。之后，TensorFlow进行优化步骤并更新与所选学习方案相对应的网络参

数。更新了权重和偏差后，对下一批进行采样，并重复该过程。该过程将继续，直到所有批次都已呈现给网络。所有批次的全扫描被称为一个epoch。

一旦达到最大epoch数或者用户定义的另一个停止条件，网络的训练就停止。

```
# Make Session
```

```
net = tf.Session()
```

```
# Run initializer
```

```
net.run(tf.global_variables_initializer())
```

```
# Setup interactive plot
```

```
plt.ion()
```

```
fig = plt.figure()
```

```
ax1 = fig.add_subplot(111)
```

```
line1, = ax1.plot(y_test)
```

```
line2, = ax1.plot(y_test*0.5)
```

```
plt.show()
```

```
# Number of epochs and batch size
```

```
epochs = 10
```

```
batch_size = 256
```

```
for e in range(epochs):
```

```
# Shuffle training data
```

```
shuffle_indices = np.random.permutation(np.arange(len(y_train)))
```

```
X_train = X_train[shuffle_indices]
```

```
y_train = y_train[shuffle_indices]
```

```
# Minibatch training
```

```
for i in range(0, len(y_train) // batch_size):
```

```
start = i * batch_size
```

```
batch_x = X_train[start:start + batch_size]
```

```
batch_y = y_train[start:start + batch_size]
```

```
# Run optimizer with batch
```

```
net.run(opt, feed_dict={X: batch_x, Y: batch_y})
```

```
# Show progress
```

```
if np.mod(i, 5) == 0:
```

```
# Prediction
```

```
pred = net.run(out, feed_dict={X: X_test})
```

```
line2.set_ydata(pred)
```

```
plt.title('Epoch ' + str(e) + ', Batch ' + str(i))
```

```
file_name = 'img/epoch_' + str(e) + '_batch_' + str(i) + '.jpg'
```

```
plt.savefig(file_name)
```

```
plt.pause(0.01)
```

```
# Print final MSE after Training
```

```
mse_final = net.run(mse, feed_dict={X: X_test, Y: y_test})
```

```
print(mse_final)
```

在训练期间，我们评估测试集上的网络预测 - 每个第5批中未学习但预留的数据 - 并将其可视化。此外，图像将导出，然后组合成训练过程的视频动画。该模型快速了解测试数据中时间序列的形状和位置，并能够在一段时间之后产生预测。

可以看出，网络迅速适应时间序列的基本形状，并继续学习更精细的数据模式。这也对应于adam优化，该方案在模型训练期间降低学习速率以便不超过优化最小值。在10个epoch之后，我们非常接近测试数据！最终测试MSE等于0.00078（它非常低，因为目标是缩放的）。测试集上预测的平均绝对百分比误差等于5.31％，这是相当不错的。请注意，这仅适用于测试数据，实际场景中没有实际的样本指标。

请注意，有很多方法可以进一步改善这一结果：层和神经元的设计，选择不同的初始化和激活方案，引入神经元的丢失层，提前停止等等。此外，不同类型的深度学习模型，例如递归神经网络，可以在此任务上实现更好的性能。但是，这不是这篇文章的范围。

结论和展望

TensorFlow的发布是深度学习研究中的一个里程碑事件。它的灵活性和性能使研究人员能够开发各种复杂的神经网络架构以及其他ML算法。但是，与更高级别的API（如Keras或MxNet）相比，灵活性需要更长的模型周期时间，当然目前的2.0版本已经整合了Keras。我确信TensorFlow将在神经网络和研究和实际应用中的深度学习开发方面成为商业标准。