2019 寒假训练项目 – 深度学习

深度神经网络的搭建和训练——以 MNIST 为例

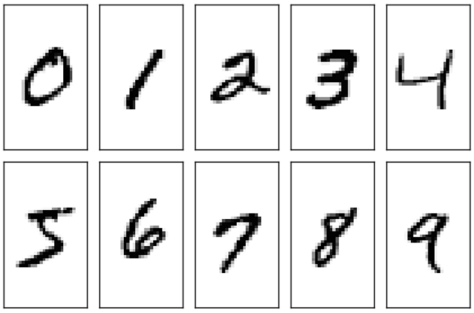
目录

一、MNIST 数据集及程序介绍

1、MNIST 简介

MNIST 是一个著名的数据集，以至于 TensorFlow 库中自带了有关 MNIST 数据集的读取和处理功能的封装函数，而本次训练项目提供的程序也将直接使用现成的读取函数。

MNIST 的特点主要是其典型性（手写数字识别问题）和简洁性（55000+10000 个 28×28 单通道图像，整个数据集不到 60MB），很适合初学者练习训练小规模的神经网络（一般计算机直接用 CPU 就可以带得动）。整个数据集中全是如下图所示的手写数字图片



2、读取接口

TensorFlow 库中封装了操作 MNIST 的相关函数，首先使用

```
from tensorflow.examples.tutorials.mnist import input_data
```

来引入这个类。在程序开始，需要使用读取的函数将数据存到一个具体的类中（实例化）

```
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

其中 `MNIST_data` 是存储数据集文件（4 个压缩包）的路径名，如果路径不存在/文件不存在会自动创建并下载文件，因此任意指定即可。`one_hot` 参数指定了标签是否要使用 one-hot 编码，在分类问题中 one-hot 编码比直接编号要来得更好，因此一般定为 True。

在程序运行过程中，需要不断使用 next_batch 方法从数据集中提取指定数量（BATCH_SIZE）的数据（图片+相应的标签），可以使用

```
data_batch = mnist.train.next_batch(BATCH_SIZE)
```

来提取，对于测试集，则用

```
data_batch = mnist.test.next_batch(BATCH_SIZE)
```

这行代码可以从数据集中按顺序提取数据，也就是每执行一次，指针都会后移，而且到达数据集数据量的上限后会自动循环至开头。例如对于一个 500 个数据的数据集，连续执行 3 次 next_batch(200)，每次得到的数据编号是 0~199、200~399、400~499 加上 0~99

另外，next_batch 函数还有额外的参数。其声明如下

```
def next_batch(self, batch_size, fake_data=False, shuffle=False):
```

其中 fake_data 是无效参数，shuffle 参数决定了最终返回的数据是否需要随机打乱（网上可以找到这个函数的完整代码）

注：对于任意一个数据集，如果没有现成的操作函数，而是需要自行编写的话，操作的接口也建议模仿类似的形式来写

next_batch 函数的返回值（就是上文中的 `data_batch`）是一个元组，包含两个元素，[0]是图像的列表，[1]是标签的列表。其中图像列表的维度是[BATCH_SIZE, 784]，它是将 BATCH_SIZE 张图片放入一个列表，每张图片（28×28）都 reshape 成一个向量（784 维）。

如果使用 one-hot 编码，则标签列表的维度是[BATCH_SIZE, 10]。

以上内容就是目前所需的有关数据集本身的操作。

二、全连接网络的搭建

1、单个全连接层

全连接网络主要包含若干个全连接层，其中还可能加入了其他辅助运算，例如数据归一化（本次不使用）和 Dropout（本次需要使用）。

每个全连接层的输入是一个向量$\boldsymbol{X}$，假设是n维的，它的输出也是一个向量$\boldsymbol{Y}$，假设是m维的（就是这层具有的神经元数量）。全连接层的核心运算是矩阵乘法加偏置，即

$$\boldsymbol{Y} = \boldsymbol{X}W + b$$

其中W是权重矩阵，是$n \times m$维（$[n, m]$）的，b是偏置项，是m维向量，与$\boldsymbol{Y}$相同。在输出时，也可选择令某个激活函数作用在运算结果上，即

$$\boldsymbol{Z} = \sigma(\boldsymbol{X}W + B)$$

激活函数常用 Sigmoid 或 ReLU，但因 ReLU 形式简单运算快，一般用得比较多。

搭建单个全连接层的函数如下

```
def dense(inputs, weight_shape, name_scope, relu=False):
    with tf.name_scope(name_scope):
        weight = tf.get_variable(name=name_scope + '_weight', shape=weight_shape,
initializer=tf.contrib.keras.initializers.he_normal())
        bias = tf.Variable(tf.constant(value=0.1, shape=[weight_shape[-1]],
dtype=tf.float32), name=name_scope + '_bias')
        if relu:
            return tf.nn.relu(tf.matmul(inputs, weight) + bias)
        else:
            return tf.matmul(inputs, weight) + bias
```

这里首先需要注意的是变量的作用域问题。虽然变量 weight 和 bias 看似是函数中的临时变量，函数运行完毕后就不再存在，但它们其实是在 TensorFlow 中定义了两个具有特定类别（任意设定的 name_scope）和名称（name）的变量，程序运行期间会一直存在于 TensorFlow 中。

2、有关 Variable 和 Tensor 的问题

对于权重变量 weight，get_variable 函数能够调用某个特定名称的变量（指定 name），如果该名称的变量不存在则创建，如果要新建变量需要注意名称不能重复。shape 则指定了该变量（实际表示矩阵）的尺寸，initializer 指定了变量的初始化方式。初始化方式应根据使用的激活函数来选择，其中 he_normal()在使用 ReLU 激活函数时效果较好。

对于偏置项变量 bias，使用 Variable 同样能够新建一个指定名称的变量。因为偏置项较简单，因此直接初始化为常数即可，但不要取得太大。

函数的返回值是一个 tensor，分为有和没有激活函数两种。

注：可以这样理解，在程序中的任何地方，定义的 TensorFlow 中的变量（variable）和张量（tensor），都可以看成是（具有特定名称的，如果定义了 name 的话）全局变量。函数返回的 tensor 是依赖于这两个变量的，因此训练过程中如果涉及到这个 tensor，变量也会相应地被更新。有关 Tensor 和 Variable 的更多解释，可参考以下资料

https://stackoverflow.com/questions/37849322/how-to-understand-the-term-tensor-in-tensorflow

https://stackoverflow.com/questions/40866675/implementation-difference-between-tensorflow-variable-and-tensorflow-tensor

https://stackoverflow.com/questions/38556078/in-tensorflow-what-is-the-difference-between-a-variable-and-a-tensor
https://www.tensorflow.org/programmers_guide/variables
https://www.tensorflow.org/api_docs/python/tf/Variable
因为这是一个比较重要的概念，因此需要尽量理解清楚

3、全连接网络的搭建

以上函数就完成了一个带 ReLU 激活函数的全连接层的搭建，要构成一个完整的全连接网络，只需要组合几个全连接层，以及部分辅助层即可。网络的定义如下

```
def __init__(self):
    # Layers
    self.dense_layer_1 = dense(X_image, [784, 1024], 'dense1', relu=True)
    self.drop_1 = tf.nn.dropout(self.dense_layer_1, keep_prob=keep_prob)
    self.dense_layer_2 = dense(self.drop_1, [1024, 512], 'dense2', relu=True)
    self.drop_2 = tf.nn.dropout(self.dense_layer_2, keep_prob=keep_prob)
    self.output = dense(self.drop_2, [512, 10], 'output', relu=False)

    # Loss
    self.loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=Y_label,
logits=self.output))

    # Training op
    self.train_step = tf.train.AdamOptimizer(LEARNING_RATE).minimize(self.loss)

    # Prediction accuracy
    self.correct_predictions = tf.equal(tf.argmax(self.output, 1), tf.argmax(Y_label, 1))
    self.eval_accuracy = tf.reduce_mean(tf.cast(self.correct_predictions, tf.float32))
```

其中，第一部分定义了网络的结构，它包含了 3 个全连接层，含有的神经元数量分别是 1024、512、10，最后一个是输出层。另外，输入层的神经元数量是 784 个。其中两个隐层（前面两层）的输出使用了 ReLU 激活函数，层之后加了 Dropout 层（TensorFlow 自带了定义 Dropout 层的函数），而输出层没有激活函数。

输出层是不需要激活函数的，因为输出的是 one-hot 编码，理想情况下，应当是取输出层中值最大的神经元下标作为分类结果（编号，就是用 argmax 函数）。额外加入 Softmax 层还可以将其转换为预测的概率，但因损失函数使用了带 Softmax 的交叉熵，因此不需要加。

对于分类问题，常用交叉熵作为损失函数。本次程序中使用带 Softmax 的交叉熵 `tf.nn.softmax_cross_entropy_with_logits`，也就是将输出层先进行 Softmax 运算，转换为全是 0~1 之间的、相加为 1 的数（预测把握），与标签（one-hot）做交叉熵。

训练的过程 `train_step`，就是使用优化器类（这里使用 `AdamOptimizer`），以最小化损失函数为优化目标进行训练。

对于每一批数据，预测正确与否取决于输出层中值最大的神经元编号是否和标签中 1 的编号相同，就是输出结果和标签取 `argmax` 后结果是否相同。使用 `tf.equal` 运算可以计算有多少个是相同的。将其除以一批中数据的总数就得到了正确率 `eval_accuracy`。

三、卷积神经网络的搭建

1、单个卷积层的搭建

定义一个卷积层的函数如下

```
def convolution(inputs, kernel_shape, name_scope):
    with tf.name_scope(name_scope):
```

```
        kernel = tf.get_variable(name_scope + '_kernel', shape=kernel_shape,
initializer=tf.contrib.keras.initializers.he_normal())
        bias = tf.Variable(tf.constant(value=0.1, shape=[kernel_shape[-1]],
dtype=tf.float32), name=name_scope + '_bias')
        return tf.nn.relu(tf.nn.conv2d(inputs, kernel, strides=[1, 1, 1, 1], padding='SAME')
+ bias)
```

其中 kernel 是卷积核，bias 是偏置项。**卷积核和卷积结果的维度**是需要重点理解的，在此仅给出简要解释，详见相关资料。

2、卷积涉及的维度问题

卷积核和卷积结果都是 4 维的。卷积结果（可看作图片）各维度的含义分别是$[Batch\ size,\ Height, Width, Channel]$。其中$Batch\ size$表示一批数据中有几张图片，$Height$和$Width$表示图片的尺寸，$Channel$表示图片的通道。输入图像一般有 1 个（灰度图，如 MNIST）或 3 个（彩色图，如 CIFAR），而经卷积操作后可能会有更多的通道。其中，定义 placeholder 或 reshape 时，$Batch\ size$可以取-1 或者 None，表示这个维度的数量不确定。

卷积核各维度的含义分别是

$$[Kernel\ height, Kernel\ width, Input\ channel, Output\ channel]$$

其中$Kernel\ height$和$Kernel\ width$表示卷积核的尺寸（一般长宽相等），$Input\ channel$表示输入图像有多少个通道，$Output\ channel$表示本卷积层用多少个不同的卷积核进行卷积，卷积后将每个卷积核的结果叠到一起，就等于输出结果的通道数。

因此，卷积后，结果的$Channel$等于该层卷积核的$Output\ channel$，卷积核尺寸与输出图像尺寸无关。如果多个卷积层输入输出相连，则下一层的$Input\ channel$等于上一层的$Output\ channel$。

另外，strides 表示每卷积一次，卷积核每个维度上移动的距离，也是一个 4 维向量，一般用[1,1,1,1]或[1,2,2,1]，它决定了输出图像的尺寸。使用[1,1,1,1]时输入输出图像尺寸相同。padding 表示卷积核位于图像边界处、有一部分在图像外时如何处理，'SAME'表示在外面补零，'VALID'表示保证卷积核始终完全在图像内，此时输出图像的尺寸会比输入图像小。

3、卷积神经网络的搭建

卷积神经网络的定义代码如下

```
def __init__(self):
    # Layers
    self.input_image = tf.reshape(X_image, [-1, 28, 28, 1])
    self.convolution_layer_1_1 = convolution(self.input_image, [3, 3, 1, 64], 'conv1_1')
    self.convolution_layer_1_2 = convolution(self.convolution_layer_1_1, [3, 3, 64, 64],
'conv1_2')
    self.max_pool_1 = tf.nn.max_pool(self.convolution_layer_1_2, ksize=[1, 2, 2, 1],
strides=[1, 2, 2, 1], padding='SAME')

    self.convolution_layer_2_1 = convolution(self.max_pool_1, [3, 3, 64, 128], 'conv2_1')
    self.convolution_layer_2_2 = convolution(self.convolution_layer_2_1, [3, 3, 128, 128],
'conv2_2')
    self.max_pool_2 = tf.nn.max_pool(self.convolution_layer_2_2, ksize=[1, 2, 2, 1],
strides=[1, 2, 2, 1], padding='SAME')

    self.flatten = tf.reshape(self.max_pool_2, [-1, 7*7*128])

    self.dense_layer_1 = dense(self.flatten, [7*7*128, 1024], 'dense1', relu=True)
    self.drop_1 = tf.nn.dropout(self.dense_layer_1, keep_prob=keep_prob)
    self.output = dense(self.drop_1, [1024, 10], 'output', relu=False)
```

```
    # Loss
    self.loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=Y_label,
logits=self.output))

    # Training op
    self.train_step = tf.train.AdamOptimizer(LEARNING_RATE).minimize(self.loss)

    # Prediction accuracy
    self.correct_predictions = tf.equal(tf.argmax(self.output, 1), tf.argmax(Y_label, 1))
    self.eval_accuracy = tf.reduce_mean(tf.cast(self.correct_predictions, tf.float32))
```

因为数据集中直接得到的图像数据是 784 维的向量，因此需要 reshape 变成矩阵（二维图像）的形式。max_pool 表示最大池化，池化的定义、ksize 的说明见相关资料，strides、padding 的含义同卷积层。本次程序中模仿了 VGG 模型，也就是每个卷积层中连做两次卷积，中间没有池化（可以用连续的、小尺寸卷积核的操作代替一次大尺寸卷积核的卷积，可以减少参数数目），每两个卷积层之间有池化层。

在卷积之后，因为结果是二维图像，而全连接层需要输入向量，因此需要进行 reshape。之后就是全连接层、损失函数、训练步骤的定义，同全连接层。

四、全连接网络和卷积神经网络的训练

训练的核心类是 `tf.Session()`，一些操作（例如训练、获取某个 tensor 的值）都需要借助它的 `run` 方法。因为神经网络依赖 placeholder 给出输入数据，而 placeholder 的值是事先不知道的，因此需要借助 `feed_dict` 参数以字典的形式给出值。

初始化所有参数需要使用

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

每一次训练，需要调用 `train_step`，使用

```
sess.run(self.train_step, feed_dict={X_image: input_images, Y_label: input_labels,
keep_prob: keep_prob_value})
```

其中字典的键的名称就是 placeholder 的名称，`input_images` 和 `input_labels` 是需要给出的图像和标签，`keep_prob_value` 是指定的 Dropout 层的保留概率，训练时应小于 1，测试时应等于 1。这个操作是没有实际返回值的。

在用一批数据评价当前模型时，使用

```
accuracy, loss = sess.run([self.eval_accuracy, self.loss], feed_dict={X_image:
input_images, Y_label: input_labels, keep_prob: 1.0})
```

来计算 `eval_accuracy` 和 `loss` 这两个 tensor 的值，从而得到模型在这一批数据上的准确率和损失。

而对于一批新的数据（有可能没有标签），如果要知道模型的预测结果，以及每个结果的把握，可以使用

```
prediction = sess.run(tf.nn.softmax(self.output), feed_dict={X_image: input_images,
keep_prob: 1.0})
```

来计算，注意到这里使用了 Softmax 层。

模型训练的流程是（两种网络完全相同）

（1）初始化参数，随机初始化或者读取现有的参数（在上一次的基础上继续训练）

（2）每次迭代，读取一批训练数据及其标签，用 `train_step` 来训练

（3）每隔一定的迭代次数，可以对模型进行评价，使用 `[self.eval_accuracy, self.loss]`

（4）模型训练完成，在测试集上使用 `self.eval_accuracy` 计算其准确率。如果测试集太大，可以每次测试一小部分，最后取平均数。

本次提供的实例程序中，两种神经网络的分类准确率均可达 98%，其中全连接网络计算量小，训练快，而卷积神经网络收敛快，需要的迭代次数少，但每次迭代计算量大。

五、K-近邻算法

严格地说，KNN 并不算是深度神经网络，K-近邻算法只是机器学习中的一种分类算法。但因其思路比较清晰简洁，同时在分类问题中也有很大的作用，因此也在此列出，仅供参考。

KNN 的主要思路是，首先对所有数据提取特征（降低数据维度），之后对于每个待分类数据，求它到每个训练数据的距离（可以取任一种范数），取最近的 K 个进行投票，最终选出最接近的类别。详细解释见相关资料。

因为 MNIST 数据集的数据比较简单，只有 784 维，因此实际可以直接计算而不先提取特征。KNN 分类的代码如下

```
def KNN(test_data_vector, train_data, train_label, k):
    size = train_data.shape[0]
    difference = np.tile(test_data_vector, (size, 1)) - train_data
    Euclid_distance = (difference ** 2).sum(axis=1)
    sorted_index = Euclid_distance.argsort()

    classes = {}
    max_count = 0
    best_class = 0
    for i in range(k):
        current_class = train_label[sorted_index[i]]
        current_count = classes.get(current_class, 0) + 1
        classes[current_class] = current_count
        if current_count > max_count:
            max_count = current_count
            best_class = current_class
    return best_class
```

对于每个待分类数据 `test_data_vector`，求它到每个训练数据的欧几里得距离 `Euclid_distance`（二范数），并进行从小到大排序（`argsort`），在循环中选前 k 个（距离最小的）训练数据的标签，找出其中出现最多的（投票）作为最终的分类结果。因为这里只是使用了标签，因此 one-hot 编码不是必要的。

KNN 的缺点是运算量很大，本次提供的实例程序中，因为没有进行特征提取，而且使用了全部的 55000 个训练数据，因此计算很慢。测试时只是从测试集中随机选了少量数据进行测试，精度可达 99%。

此次提供的程序只是演示了用 K-近邻算法进行分类，甚至没有用到 TensorFlow。而实际上真正的 KNN 在前面是需要加上特征提取的步骤的，感兴趣的同学可以作更多了解。