Tensorflow

2019年4月27日 21:33

```python
In [1]: import os
        import tensorflow as tf
        import numpy as np
        import math
        import timeit
        import matplotlib.pyplot as plt

        %matplotlib inline
```

```python
In [2]: def load_cifar10(num_training=49000, num_validation=1000, num_test=10000):
            """
            Fetch the CIFAR-10 dataset from the web and perform preprocessing to prepare
            it for the two-layer neural net classifier. These are the same steps as
            we used for the SVM, but condensed to a single function.
            """
            # Load the raw CIFAR-10 dataset and use appropriate data types and shapes
            cifar10 = tf.keras.datasets.cifar10.load_data()
            (X_train, y_train), (X_test, y_test) = cifar10
            X_train = np.asarray(X_train, dtype=np.float32)
            y_train = np.asarray(y_train, dtype=np.int32).flatten()
            X_test = np.asarray(X_test, dtype=np.float32)
            y_test = np.asarray(y_test, dtype=np.int32).flatten()

            # Subsample the data
            mask = range(num_training, num_training + num_validation)
            X_val = X_train[mask]
            y_val = y_train[mask]
            mask = range(num_training)
            X_train = X_train[mask]
            y_train = y_train[mask]
            mask = range(num_test)
            X_test = X_test[mask]
            y_test = y_test[mask]

            # Normalize the data: subtract the mean pixel and divide by std
            mean_pixel = X_train.mean(axis=(0, 1, 2), keepdims=True)
            std_pixel = X_train.std(axis=(0, 1, 2), keepdims=True)
            X_train = (X_train - mean_pixel) / std_pixel
            X_val = (X_val - mean_pixel) / std_pixel
            X_test = (X_test - mean_pixel) / std_pixel

            return X_train, y_train, X_val, y_val, X_test, y_test


        # Invoke the above function to get our data.
        NHW = (0, 1, 2)
        X_train, y_train, X_val, y_val, X_test, y_test = load_cifar10()
        print('Train data shape: ', X_train.shape)
        print('Train labels shape: ', y_train.shape, y_train.dtype)
        print('Validation data shape: ', X_val.shape)
        print('Validation labels shape: ', y_val.shape)
        print('Test data shape: ', X_test.shape)
        print('Test labels shape: ', y_test.shape)
```

```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,) int32
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

不得不说......有点牛逼你，keras里面有内置cifar-10数据集

cifar10 = tf.keras.datasets.cifar10.load_data()　　该条语句直接加载数据。

array和asarray都可以将结构数据转化为ndarray，但是主要区别就是当数据源是ndarray时，array仍然会copy出一个副本，占用新的内存，但asarray不会。

```python
import numpy as np

#example 2:
arr1=np.ones((3,3))
arr2=np.array(arr1)
arr3=np.asarray(arr1)
arr1[1]=2
print 'arr1:\n',arr1
print 'arr2:\n',arr2
print 'arr3:\n',arr3
```

```
arr1:
[[ 1.  1.  1.]
 [ 2.  2.  2.]
 [ 1.  1.  1.]]
arr2:
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]
arr3:
[[ 1.  1.  1.]
 [ 2.  2.  2.]
 [ 1.  1.  1.]]
```

a是个矩阵或者数组，a.flatten()就是把a降到一维，默认是按横的方向降。

这里cifar-10加载的数据应该是N*32*32*3的，这里将它们全向量化。

下面的处理是cifar-10的训练集的前49000个数据作为训练集，后1000个数据作为验证集，cifar-10的测试集里前10000个作为测试集。
然后将所有数据做z-score标准化处理，即减去平均值再除以标准差。

```
In [6]: class Dataset(object):
            def __init__(self, X, y, batch_size, shuffle=False):
                """
                Construct a Dataset object to iterate over data X and labels y

                Inputs:
                - X: Numpy array of data, of any shape
                - y: Numpy array of labels, of any shape but with y.shape[0] == X.shape[0]
                - batch_size: Integer giving number of elements per minibatch
                - shuffle: (optional) Boolean, whether to shuffle the data on each epoch
                """
                assert X.shape[0] == y.shape[0], 'Got different numbers of data and labels'
                self.X, self.y = X, y
                self.batch_size, self.shuffle = batch_size, shuffle

            def __iter__(self):
                N, B = self.X.shape[0], self.batch_size
                idxs = np.arange(N)
                if self.shuffle:
                    np.random.shuffle(idxs)
                return iter((self.X[i:i+B], self.y[i:i+B]) for i in range(0, N, B))


        train_dset = Dataset(X_train, y_train, batch_size=64, shuffle=True)
        val_dset = Dataset(X_val, y_val, batch_size=64, shuffle=False)
        test_dset = Dataset(X_test, y_test, batch_size=64)
```

assert：断言，如果不满足断言内容，则会返回后面 '' 内的内容
shuffle：打乱顺序，不过……这段代码好像有问题，貌似并没有起到打乱顺序的作用

```
In [10]: # We can iterate through a dataset like this:
         for t, (x, y) in enumerate(train_dset):
             print(t, x.shape, y.shape)
             if t > 5: break

         0 (64, 32, 32, 3) (64,)
         1 (64, 32, 32, 3) (64,)
         2 (64, 32, 32, 3) (64,)
         3 (64, 32, 32, 3) (64,)
         4 (64, 32, 32, 3) (64,)
         5 (64, 32, 32, 3) (64,)
         6 (64, 32, 32, 3) (64,)
```

以下展示了使用 enumerate() 方法的实例:

```
>>>seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>> list(enumerate(seasons, start=1))       # 下标从 1 开始
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

只是显示一下每批构造了一个64*32*32*3的数据集

```
In [11]: # Set up some global variables
         USE_GPU = True

         if USE_GPU:
             device = '/device:GPU:0'
         else:
             device = '/cpu:0'

         # Constant to control how often we print when training models
         print_every = 100

         print('Using device: ', device)

         # from tensorflow.python.client import device_lib
         # print(device_lib.list_local_devices())

         Using device:  /device:GPU:0
```

device：设置使用的设备的变量，这个作业就最后一个模型我用了GPU，之前都是选择的CPU，这里可以改成USE_GPU=False

```
In [12]: def flatten(x):
    """
    Input:
    - TensorFlow Tensor of shape (N, D1, ..., DM)

    Output:
    - TensorFlow Tensor of shape (N, D1 * ... * DM)
    """
    N = tf.shape(x)[0]
    return tf.reshape(x, (N, -1))
```

好像有点没必要，貌似自带flatten函数

```
In [7]: def test_flatten():
    # Clear the current TensorFlow graph.
    tf.reset_default_graph()

    # Stage I: Define the TensorFlow graph describing our computation.
    # In this case the computation is trivial: we just want to flatten
    # a Tensor using the flatten function defined above.

    # Our computation will have a single input, x. We don't know its
    # value yet, so we define a placeholder which will hold the value
    # when the graph is run. We then pass this placeholder Tensor to
    # the flatten function; this gives us a new Tensor which will hold
    # a flattened view of x when the graph is run. The tf.device
    # context manager tells TensorFlow whether to place these Tensors
    # on CPU or GPU.
    with tf.device(device):
        x = tf.placeholder(tf.float32)
        x_flat = flatten(x)

    # At this point we have just built the graph describing our computation,
    # but we haven't actually computed anything yet. If we print x and x_flat
    # we see that they don't hold any data; they are just TensorFlow Tensors
    # representing values that will be computed when the graph is run.
    print('x: ', type(x), x)
    print('x_flat: ', type(x_flat), x_flat)
    print()

    # We need to use a TensorFlow Session object to actually run the graph.
    with tf.Session() as sess:
        # Construct concrete values of the input data x using numpy
        x_np = np.arange(24).reshape((2, 3, 4))
        print('x_np:\n', x_np, '\n')

        # Run our computational graph to compute a concrete output value.
        # The first argument to sess.run tells TensorFlow which Tensor
        # we want it to compute the value of; the feed_dict specifies
        # values to plug into all placeholder nodes in the graph. The
        # resulting value of x_flat is returned from sess.run as a
        # numpy array.
        x_flat_np = sess.run(x_flat, feed_dict={x: x_np})
        print('x_flat_np:\n', x_flat_np, '\n')

        # We can reuse the same graph to perform the same computation
        # with different input data
        x_np = np.arange(12).reshape((2, 3, 2))
        print('x_np:\n', x_np, '\n')
        x_flat_np = sess.run(x_flat, feed_dict={x: x_np})
        print('x_flat_np:\n', x_flat_np)
test_flatten()
```

注释解释了tensorflow运算过程，大概就是先自己定义一个计算图，然后定义一些占位符留给那些暂时不知道的变量，一般来说是input这类东西，
每次运行一个计算图之前，先要reset一下
这里先定义了一个float32类型的x，然后对后面feed的这个x做flatten处理。

一 运行机制

TensorFlow的运行机制属于"定义"与"运行"相分离。从操作层面可以抽象成两种：构造模型和模型运行。

在讲解构建模型之前，需要讲解几个概念。在一个叫做"图"的容器中包括：

- 张量(tensor)：TensorFlow程序使用tensor数据结构来代表所有的数据，计算图中，操作间传递的数据都是tensor，你可以把TensorFlow tensor看做一个n维的数组或者列表。
- 变量(Variable)：常用于定义模型中的参数，是通过不断训练得到的值。比如权重和偏置。
- 占位符(placeholder)：输入变量的载体。也可以理解成定义函数时的参数。
- 图中的节点操作(op)：一个op获得0个或者多个Tensor，执行计算，产生0个或者多个Tensor。**op是描述张量中的运算关系，是网络中真正结构。**

一个TensorFlow图描述了计算的过程，为了进行计算，图必须在会话里启动，会话将图的op分发到诸如CPU或者GPU的设备上，同时提供执行op的方法，这些方法执行后，将产生的tensor返回，在python语言中，返回的tensor是numpy array对象，在C或者C++语言中，返回的tensor是tensorflow:Tensor实例。

session与图的交互过程中定义了以下两种数据的流向机制。

- 注入机制(feed):通过占位符向模式中传入数据。
- 取回机制(fetch)：从模式中取得结果。

觉得讲不好，上网查了一下。

```
x:  <class 'tensorflow.python.framework.ops.Tensor'> Tensor("Placeholder:0", dtype=float32, device=/device:CPU:0)
x_flat:  <class 'tensorflow.python.framework.ops.Tensor'> Tensor("Reshape:0", shape=(?, ?), dtype=float32, device=/device:CPU:0)

x_np:
 [[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]

x_flat_np:
 [[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11.]
 [12. 13. 14. 15. 16. 17. 18. 19. 20. 21. 22. 23.]]

x_np:
 [[[ 0  1]
  [ 2  3]
  [ 4  5]]

 [[ 6  7]
  [ 8  9]
  [10 11]]]

x_flat_np:
 [[ 0.  1.  2.  3.  4.  5.]
 [ 6.  7.  8.  9. 10. 11.]]
```

了解一下tensorflow的运行机制

```
In [13]: def two_layer_fc(x, params):
    """
    A fully-connected neural network; the architecture is:
    fully-connected layer -> ReLU -> fully connected layer.
    Note that we only need to define the forward pass here; TensorFlow will take
    care of computing the gradients for us.

    The input to the network will be a minibatch of data, of shape
    (N, d1, ..., dM) where d1 * ... * dM = D. The hidden layer will have H units,
    and the output layer will produce scores for C classes.

    Inputs:
    - x: A TensorFlow Tensor of shape (N, d1, ..., dM) giving a minibatch of
      input data.
    - params: A list [w1, w2] of TensorFlow Tensors giving weights for the
      network, where w1 has shape (D, H) and w2 has shape (H, C).

    Returns:
    - scores: A TensorFlow Tensor of shape (N, C) giving classification scores
      for the input data x.
    """
    w1, w2 = params  # Unpack the parameters
    x = flatten(x)   # Flatten the input; now x has shape (N, D)
    #x由 (N, d1,……, dk) 变成 (N,D)
    h = tf.nn.relu(tf.matmul(x, w1)) # Hidden layer: h has shape (N, H)
    #tf.neuralnetwork.relu……matmul为矩阵乘法
    scores = tf.matmul(h, w2)        # Compute scores of shape (N, C)
    return scores
```

太方便了，relu直接调用就行了，做这个作业顺便复习一下之前的东西

```
In [9]: def two_layer_fc_test():
    # TensorFlow's default computational graph is essentially a hidden global
    # variable. To avoid adding to this default graph when you rerun this cell,
    # we clear the default graph before constructing the graph we care about.
    tf.reset_default_graph()
    hidden_layer_size = 42

    # Scoping our computational graph setup code under a tf.device context
    # manager lets us tell TensorFlow where we want these Tensors to be
    # placed.
    with tf.device(device):
        # Set up a placehoder for the input of the network, and constant
        # zero Tensors for the network weights. Here we declare w1 and w2
        # using tf.zeros instead of tf.placeholder as we've seen before - this
        # means that the values of w1 and w2 will be stored in the computational
        # graph itself and will persist across multiple runs of the graph; in
        # particular this means that we don't have to pass values for w1 and w2
        # using a feed_dict when we eventually run the graph.
        x = tf.placeholder(tf.float32)
        w1 = tf.zeros((32 * 32 * 3, hidden_layer_size))
        w2 = tf.zeros((hidden_layer_size, 10))

        # Call our two_layer_fc function to set up the computational
        # graph for the forward pass of the network.
        scores = two_layer_fc(x, [w1, w2])
```

```python
    # Use numpy to create some concrete data that we will pass to the
    # computational graph for the x placeholder.
    x_np = np.zeros((64, 32, 32, 3))
    with tf.Session() as sess:
        # The calls to tf.zeros above do not actually instantiate the values
        # for w1 and w2; the following line tells TensorFlow to instantiate
        # the values of all Tensors (like w1 and w2) that live in the graph.
        sess.run(tf.global_variables_initializer())

        # Here we actually run the graph, using the feed_dict to pass the
        # value to bind to the placeholder for x; we ask TensorFlow to compute
        # the value of the scores Tensor, which it returns as a numpy array.
        scores_np = sess.run(scores, feed_dict={x: x_np})
        print(scores_np.shape)

two_layer_fc_test()
```

```
(64, 10)
```

这里加了一个size为42的隐藏层。

with是一个上下文机制。

tf.device可以把下文的计算图放在你希望放在的设备内，cpu_x，gpu_x。

session里面先用sess.run(tf.global_variables_initializer())将前面定义的一些tensor实例化，然后用sess.run运行计算图，将x_np灌进去，

这里只是做验证，参数全都为0

In [10]:
```python
def three_layer_convnet(x, params):
    """
    A three-layer convolutional network with the architecture described above.

    Inputs:
    - x: A TensorFlow Tensor of shape (N, H, W, 3) giving a minibatch of images
    - params: A list of TensorFlow Tensors giving the weights and biases for the
      network; should contain the following:
      - conv_w1: TensorFlow Tensor of shape (KH1, KW1, 3, channel_1) giving
        weights for the first convolutional layer.
      - conv_b1: TensorFlow Tensor of shape (channel_1,) giving biases for the
        first convolutional layer.
      - conv_w2: TensorFlow Tensor of shape (KH2, KW2, channel_1, channel_2)
        giving weights for the second convolutional layer
      - conv_b2: TensorFlow Tensor of shape (channel_2,) giving biases for the
        second convolutional layer.
      - fc_w: TensorFlow Tensor giving weights for the fully-connected layer.
        Can you figure out what the shape should be?
      - fc_b: TensorFlow Tensor giving biases for the fully-connected layer.
        Can you figure out what the shape should be?
    """
    conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b = params
    scores = None
    ############################################################################
    # TODO: Implement the forward pass for the three-layer ConvNet.            #
    ############################################################################
    with tf.Session() as sess:
        conv_out1 = tf.nn.conv2d(x,conv_w1,[1,1,1,1],"SAME",name="CONV1")
        conv_output1 = tf.nn.bias_add(conv_out1,conv_b1)
        relu_output1 = tf.nn.relu(conv_output1,name="RELU1")
        conv_out2 = tf.nn.conv2d(relu_output1,conv_w2,[1,1,1,1],"SAME",name="CONV2")
        conv_output2 = tf.nn.bias_add(conv_out2,conv_b2)
        relu_output2 = tf.nn.relu(conv_output2,name="RELU2")
        relu_output2 = tf.reshape(relu_output2,(tf.shape(relu_output2)[0],-1))
        scores = tf.matmul(relu_output2,fc_w)
        scores = tf.nn.bias_add(scores,fc_b)
    ############################################################################
    #                             END OF YOUR CODE                             #
    ############################################################################
    return scores
```

tf.nn.conv2d

```python
tf.nn.conv2d(
    input,
    filter,
    strides,
    padding,
    use_cudnn_on_gpu=True,
    data_format='NHWC',
    dilations=[1, 1, 1, 1],
    name=None
)
```

直接按顺序一步一步调用下来就好了......

In [11]:
```python
def three_layer_convnet_test():
    tf.reset_default_graph()

    with tf.device(device):
        x = tf.placeholder(tf.float32)
        conv_w1 = tf.zeros((5, 5, 3, 6))
        conv_b1 = tf.zeros((6,))
        conv_w2 = tf.zeros((3, 3, 6, 9))
        conv_b2 = tf.zeros((9,))
        fc_w = tf.zeros((32 * 32 * 9, 10))
        fc_b = tf.zeros((10,))
        params = [conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b]
        scores = three_layer_convnet(x, params)

    # Inputs to convolutional layers are 4-dimensional arrays with shape
    # [batch_size, height, width, channels]
    x_np = np.zeros((64, 32, 32, 3))

    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        scores_np = sess.run(scores, feed_dict={x: x_np})
        print('scores_np has shape: ', scores_np.shape)

with tf.device('/cpu:0'):
    three_layer_convnet_test()
```

```
scores_np has shape:  (64, 10)
```

We now define the `training_step` function which sets up the part of the computational graph that performs a single training step. This will take three basic steps:

1. Compute the loss
2. Compute the gradient of the loss with respect to all network weights
3. Make a weight update step using (stochastic) gradient descent.

整个训练过程就这三步

In [12]:
```python
def training_step(scores, y, params, learning_rate):
    """
    Set up the part of the computational graph which makes a training step.

    Inputs:
    - scores: TensorFlow Tensor of shape (N, C) giving classification scores for
      the model.
    - y: TensorFlow Tensor of shape (N,) giving ground-truth labels for scores;
      y[i] == c means that c is the correct class for scores[i].
    - params: List of TensorFlow Tensors giving the weights of the model
    - learning_rate: Python scalar giving the learning rate to use for gradient
      descent step.

    Returns:
    - loss: A TensorFlow Tensor of shape () (scalar) giving the loss for this
      batch of data; evaluating the loss also performs a gradient descent step
      on params (see above).
    """
    # First compute the loss; the first line gives losses for each example in
    # the minibatch, and the second averages the losses acros the batch
    losses = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=scores)
    loss = tf.reduce_mean(losses)

    # Compute the gradient of the loss with respect to each parameter of the the
    # network. This is a very magical function call: TensorFlow internally
    # traverses the computational graph starting at loss backward to each element
    # of params, and uses backpropagation to figure out how to compute gradients;
    # it then adds new operations to the computational graph which compute the
    # requested gradients, and returns a list of TensorFlow Tensors that will
    # contain the requested gradients when evaluated.
    grad_params = tf.gradients(loss, params)
    #loss 对params里面的值分别求导

    # Make a gradient descent step on all of the model parameters.
    new_weights = []
    for w, grad_w in zip(params, grad_params):
        new_w = tf.assign_sub(w, learning_rate * grad_w)
        new_weights.append(new_w)

    # Insert a control dependency so that evaluting the loss causes a weight
    # update to happen; see the discussion above.
    with tf.control_dependencies(new_weights):
        return tf.identity(loss)
    # control_dependencies:执行完new_weights后才能执行下面的内容，
    # tf.identify (loss) 依赖于with后面的内容
    # identify是一个op操作表示赋值，control_dependencies只有当里面是op时才会生效
```

tf.nn.sparse_softmax_cross_entropy_with_logits
这个API先做一个one-hot处理，例如一个样本的标签为3，one-hot格式为[0,0,0,1,......,0]，然后计算softmax和cross_entropy

转为one-hot格式之后就该计算我们的cross-entropy了，公式如下：

$$H_{y'}(y) = -\sum_i y'_i \log(y_i)$$

其中 y'_i 为label中的第i个值，y_i 为经softmax归一化输出的vector中的对应分量，由此可以看出，当分类越准确时，y_i 所对应的分量就会越接近于1，从而 $H_{y'}(y)$ 的值也就会越小。

贴张图，方便记忆

连减去平均值都定义好了......

gradients（a,b）如果a为list，以len为2为例，则计算(a1对b求导+a2对b求导）。如果b为list，则分别计算a对b1求导，a对b2求导。

In [13]:
```
def train_part2(model_fn, init_fn, learning_rate):
    """
    Train a model on CIFAR-10.

    Inputs:
    - model_fn: A Python function that performs the forward pass of the model
      using TensorFlow; it should have the following signature:
      scores = model_fn(x, params) where x is a TensorFlow Tensor giving a
      minibatch of image data, params is a list of TensorFlow Tensors holding
      the model weights, and scores is a TensorFlow Tensor of shape (N, C)
      giving scores for all elements of x.
    - init_fn: A Python function that initializes the parameters of the model.
      It should have the signature params = init_fn() where params is a list
      of TensorFlow Tensors holding the (randomly initialized) weights of the
      model.
    - learning_rate: Python float giving the learning rate to use for SGD.
    """
    # First clear the default graph
    tf.reset_default_graph()
    is_training = tf.placeholder(tf.bool, name='is_training')
    # Set up the computational graph for performing forward and backward passes,
    # and weight updates.
    with tf.device(device):
        # Set up placeholders for the data and labels
        x = tf.placeholder(tf.float32, [None, 32, 32, 3])
        y = tf.placeholder(tf.int32, [None])
        params = init_fn()           # Initialize the model parameters
        scores = model_fn(x, params) # Forward pass of the model
        loss = training_step(scores, y, params, learning_rate)

    # Now we actually run the graph many times using the training data
    with tf.Session() as sess:
        # Initialize variables that will live in the graph
        sess.run(tf.global_variables_initializer())
        for t, (x_np, y_np) in enumerate(train_dset):
            # Run the graph on a batch of training data; recall that asking
            # TensorFlow to evaluate loss will cause an SGD step to happen.
            feed_dict = {x: x_np, y: y_np}
            loss_np = sess.run(loss, feed_dict=feed_dict)

            # Periodically print the loss and check accuracy on the val set
            if t % print_every == 0:
                print('Iteration %d, loss = %.4f' % (t, loss_np))
                check_accuracy(sess, val_dset, x, scores, is_training)
```

In [16]:
```
def check_accuracy(sess, dset, x, scores, is_training=None):
    """
    Check accuracy on a classification model.

    Inputs:
    - sess: A TensorFlow Session that will be used to run the graph
    - dset: A Dataset object on which to check accuracy
    - x: A TensorFlow placeholder Tensor where input images should be fed
    - scores: A TensorFlow Tensor representing the scores output from the
      model; this is the Tensor we will ask TensorFlow to evaluate.

    Returns: Nothing, but prints the accuracy of the model
    """
    num_correct, num_samples = 0, 0
    for x_batch, y_batch in dset:
        feed_dict = {x: x_batch, is_training: 0}
        scores_np = sess.run(scores, feed_dict=feed_dict)
        y_pred = scores_np.argmax(axis=1)
        num_samples += x_batch.shape[0]
        num_correct += (y_pred == y_batch).sum()
    acc = float(num_correct) / num_samples
    print('Got %d / %d correct (%.2f%%)' % (num_correct, num_samples, 100 * acc))
```

这两部分没什么好说的，注意check_accuracy没有reset计算图，在上一个def里面调用，所以用的还是train_part2的计算图

```
In [17]: def kaiming_normal(shape):
             if len(shape) == 2:
                 fan_in, fan_out = shape[0], shape[1]
             elif len(shape) == 4:
                 fan_in, fan_out = np.prod(shape[:3]), shape[3]
             return tf.random_normal(shape) * np.sqrt(2.0 / fan_in)
```

```
In [*]: def two_layer_fc_init():
            """
            Initialize the weights of a two-layer network, for use with the
            two_layer_network function defined above.

            Inputs: None

            Returns: A list of:
            - w1: TensorFlow Variable giving the weights for the first layer
            - w2: TensorFlow Variable giving the weights for the second layer
            """
            hidden_layer_size = 4000
            w1 = tf.Variable(kaiming_normal((3 * 32 * 32, 4000)))
            w2 = tf.Variable(kaiming_normal((4000, 10)))
            return [w1, w2]

        learning_rate = 1e-2
        train_part2(two_layer_fc, two_layer_fc_init, learning_rate)
```

这里使用了一个新的初始化方法。论文地址

[1] He et al, *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification *, ICCV 2015, https://arxiv.org/abs/1502.01852

CSDN解释https://blog.csdn.net/blood0604/article/details/73927710

通过一系列的推导，得出了针对ReLUs的初始化方法：假设卷积核的大小为k*k，在第L层有c个filter（或者channels），L层的filter_out size（或者L+1层的卷积核个数）为d（c L=d L-1）。那么L层weight的大小为k*k*c*d。

$$\frac{1}{2}n_l Var[w_l] = 1, \quad \forall l.$$

对于第L层的初始化方法为：将该层的权重从高斯分布中采样进行初始化，高斯分布的均值为0，标准差（std）为

$\sqrt{2/n_l}$。其中nL=k*k*c。对于所有层采用相同的初始化方法。

还需要注意的是，tf.random.normal和np.random.normal输入的参数顺序不一样，具体可查百度。

还有就是在tensorflow内，可学习的参数用tf.variable定义

```
In [27]: def three_layer_convnet_init():
             """
             Initialize the weights of a Three-Layer ConvNet, for use with the
             three_layer_convnet function defined above.

             Inputs: None

             Returns a list containing:
             - conv_w1: TensorFlow Variable giving weights for the first conv layer
             - conv_b1: TensorFlow Variable giving biases for the first conv layer
             - conv_w2: TensorFlow Variable giving weights for the second conv layer
             - conv_b2: TensorFlow Variable giving biases for the second conv layer
             - fc_w: TensorFlow Variable giving weights for the fully-connected layer
             - fc_b: TensorFlow Variable giving biases for the fully-connected layer
             """
             params = None
             ############################################################################
             # TODO: Initialize the parameters of the three-layer network.              #
             ############################################################################
             conv_w1 = tf.Variable(kaiming_normal((5,5,3,32)))
             conv_b1 = tf.Variable(tf.zeros(32,))
             conv_w2 = tf.Variable(kaiming_normal((3,3,32,16)))
             conv_b2 = tf.Variable(tf.zeros(16,))
             fc_w = tf.Variable(kaiming_normal((16*32*32,10)))
             fc_b = tf.Variable(tf.zeros(10,))
             params = (conv_w1,conv_b1,conv_w2,conv_b2,fc_w,fc_b)
             ############################################################################
             #                             END OF YOUR CODE                             #
             ############################################################################
```

```
    return params

learning_rate = 3e-3
train_part2(three_layer_convnet, three_layer_convnet_init, learning_rate)
```

```
Iteration 0, loss = 2.7416
Got 98 / 1000 correct (9.80%)
Iteration 100, loss = 1.8426
Got 358 / 1000 correct (35.80%)
Iteration 200, loss = 1.6481
Got 403 / 1000 correct (40.30%)
Iteration 300, loss = 1.5809
Got 408 / 1000 correct (40.80%)
Iteration 400, loss = 1.6264
Got 453 / 1000 correct (45.30%)
Iteration 500, loss = 1.6708
Got 458 / 1000 correct (45.80%)
Iteration 600, loss = 1.5757
Got 479 / 1000 correct (47.90%)
Iteration 700, loss = 1.6837
Got 475 / 1000 correct (47.50%)
```

以上使用的都是低级的API，方便理解网络是怎么工作的，下面开始使用高级的API

In [25]:
```
class TwoLayerFC(tf.keras.Model):
    def __init__(self, hidden_size, num_classes):
        super().__init__()
        initializer = tf.variance_scaling_initializer(scale=2.0)
        self.fc1 = tf.layers.Dense(hidden_size, activation=tf.nn.relu,
                                   kernel_initializer=initializer)
        self.fc2 = tf.layers.Dense(num_classes,
                                   kernel_initializer=initializer)
    def call(self, x, training=None):
        x = tf.layers.flatten(x)
        x = self.fc1(x)
        x = self.fc2(x)
        return x
```

```
def test_TwoLayerFC():
    """ A small unit test to exercise the TwoLayerFC model above. """
    tf.reset_default_graph()
    input_size, hidden_size, num_classes = 50, 42, 10

    # As usual in TensorFlow, we first need to define our computational graph.
    # To this end we first construct a TwoLayerFC object, then use it to construct
    # the scores Tensor.
    model = TwoLayerFC(hidden_size, num_classes)
    with tf.device(device):
        x = tf.zeros((64, input_size))
        scores = model(x)

    # Now that our computational graph has been defined we can run the graph
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        scores_np = sess.run(scores)
        print(scores_np.shape)

test_TwoLayerFC()
```

Super().i]__init__()会先调用父类，这里为tf.keras.model。
initializer = tf.variance_scaling_initializer(scale=2.0)这一行为前面看到的kaiming的初始化方法。
然后在tf.layers.Dense中使用了该初始化方法，kernel_initializer为权重矩阵的初始化。
（这个API过时了，现在最好用tf.keras.layers.Dense）

```
In [26]: def two_layer_fc_functional(inputs, hidden_size, num_classes):
             initializer = tf.variance_scaling_initializer(scale=2.0)
             flattened_inputs = tf.layers.flatten(inputs)
             fc1_output = tf.layers.dense(flattened_inputs, hidden_size, activation=tf.nn.relu,
                                          kernel_initializer=initializer)
             scores = tf.layers.dense(fc1_output, num_classes,
                                      kernel_initializer=initializer)
             return scores

         def test_two_layer_fc_functional():
             """ A small unit test to exercise the TwoLayerFC model above. """
             tf.reset_default_graph()
             input_size, hidden_size, num_classes = 50, 42, 10

             # As usual in TensorFlow, we first need to define our computational graph.
             # To this end we first construct a two layer network graph by calling the
             # two_layer_network() function. This function constructs the computation
             # graph and outputs the score tensor.
             with tf.device(device):
                 x = tf.zeros((64, input_size))
                 scores = two_layer_fc_functional(x, hidden_size, num_classes)

             # Now that our computational graph has been defined we can run the graph
             with tf.Session() as sess:
                 sess.run(tf.global_variables_initializer())
                 scores_np = sess.run(scores)
                 print(scores_np.shape)

         test_two_layer_fc_functional()
```

```
WARNING:tensorflow:From <ipython-input-26-e2eb30b3f3fa>:5: dense (from tensorflow.python.layers.core) is deprecated and will be removed in a future version.
Instructions for updating:
Use keras.layers.dense instead.
(64, 10)
```

两种方法对比，其实是一样的

```
In [32]: class ThreeLayerConvNet(tf.keras.Model):
             def __init__(self, channel_1, channel_2, num_classes):
                 super().__init__()
                 ########################################################################
                 # TODO: Implement the __init__ method for a three-layer ConvNet. You   #
                 # should instantiate layer objects to be used in the forward pass.     #
                 ########################################################################
                 initializer = tf.variance_scaling_initializer(scale = 2.0)
                 self.conv1 = tf.layers.Conv2D(filters=channel_1,kernel_size=(5,5),
                                               strides=(1,1),padding='same',
                                               activation=tf.nn.relu,
                                               kernel_initializer=initializer)
                 self.conv2 = tf.layers.Conv2D(filters=channel_2,kernel_size=(3,3),
                                               strides=(1,1),padding='same',
                                               activation=tf.nn.relu,
                                               kernel_initializer=initializer)
                 self.fc = tf.layers.Dense(units=num_classes,kernel_initializer=initializer)
                 ########################################################################
                 #                           END OF YOUR CODE                           #
                 ########################################################################

             def call(self, x, training=None):
                 scores = None
                 ########################################################################
                 # TODO: Implement the forward pass for a three-layer ConvNet. You      #
                 # should use the layer objects defined in the __init__ method.         #
                 ########################################################################
                 x = self.conv1(x)
                 x = self.conv2(x)
                 scores = self.fc(tf.layers.flatten(x))
                 ########################################################################
                 #                           END OF YOUR CODE                           #
                 ########################################################################
                 return scores
```

Padding='same'表示zero填充，若为valid表示不填充。

```
In [31]: def test_ThreeLayerConvNet():
    tf.reset_default_graph()

    channel_1, channel_2, num_classes = 12, 8, 10
    model = ThreeLayerConvNet(channel_1, channel_2, num_classes)
    with tf.device(device):
        x = tf.zeros((64, 3, 32, 32))
        scores = model(x)

    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        scores_np = sess.run(scores)
        print(scores_np.shape)

test_ThreeLayerConvNet()
```

```
(64, 10)
```

调用测试一下。

```
In [34]: def train_part34(model_init_fn, optimizer_init_fn, num_epochs=1):
    """
    Simple training loop for use with models defined using tf.keras. It trains
    a model for one epoch on the CIFAR-10 training set and periodically checks
    accuracy on the CIFAR-10 validation set.

    Inputs:
    - model_init_fn: A function that takes no parameters; when called it
      constructs the model we want to train: model = model_init_fn()
    - optimizer_init_fn: A function which takes no parameters; when called it
      constructs the Optimizer object we will use to optimize the model:
      optimizer = optimizer_init_fn()
    - num_epochs: The number of epochs to train for

    Returns: Nothing, but prints progress during trainingn
    """
    tf.reset_default_graph()
    with tf.device(device):
        # Construct the computational graph we will use to train the model. We
        # use the model_init_fn to construct the model, declare placeholders for
        # the data and labels
        x = tf.placeholder(tf.float32, [None, 32, 32, 3])
        y = tf.placeholder(tf.int32, [None])

        # We need a place holder to explicitly specify if the model is in the training
        # phase or not. This is because a number of layers behaves differently in
        # training and in testing, e.g., dropout and batch normalization.
        # We pass this variable to the computation graph through feed_dict as shown below.
        is_training = tf.placeholder(tf.bool, name='is_training')

        # Use the model function to build the forward pass.
        scores = model_init_fn(x, is_training)

        # Compute the loss like we did in Part II
        loss = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=scores)
        loss = tf.reduce_mean(loss)

        # Use the optimizer_fn to construct an Optimizer, then use the optimizer
        # to set up the training step. Asking TensorFlow to evaluate the
        # train_op returned by optimizer.minimize(loss) will cause us to make a
        # single update step using the current minibatch of data.

        # Note that we use tf.control_dependencies to force the model to run
        # the tf.GraphKeys.UPDATE_OPS at each training step. tf.GraphKeys.UPDATE_OPS
        # holds the operators that update the states of the network.
        # For example, the tf.layers.batch_normalization function adds the running mean
        # and variance update operators to tf.GraphKeys.UPDATE_OPS.
        optimizer = optimizer_init_fn()
        update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
        with tf.control_dependencies(update_ops):
            train_op = optimizer.minimize(loss)
```

```python
    # Now we can run the computational graph many times to train the model.
    # When we call sess.run we ask it to evaluate train_op, which causes the
    # model to update.
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        t = 0
        for epoch in range(num_epochs):
            print('Starting epoch %d' % epoch)
            for x_np, y_np in train_dset:
                feed_dict = {x: x_np, y: y_np, is_training:1}
                loss_np, _ = sess.run([loss, train_op], feed_dict=feed_dict)
                if t % print_every == 0:
                    print('Iteration %d, loss = %.4f' % (t, loss_np))
                    check_accuracy(sess, val_dset, x, scores, is_training=is_training)
                    print()
                t += 1
```

In [37]:
```python
hidden_size, num_classes = 4000, 10
learning_rate = 1e-2

def model_init_fn(inputs, is_training):
    return TwoLayerFC(hidden_size, num_classes)(inputs)

def optimizer_init_fn():
    return tf.train.GradientDescentOptimizer(learning_rate)

train_part34(model_init_fn, optimizer_init_fn)
```

scores调用了传入的model_init_fn，看下面该函数的定义，第一个括号对应TwoLayerFC的初始化的参数，第二个括号内为传入的inputs即x。
求loss。使用优化，这里的调用是梯度下降。然后注意下面三行，在计算图中，有的API比如tf.layers.batch_normalization会自动更新参数（Mean and variance），这个更新操作被存在UPDATE_OPS里面，update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)该语句表示先获得这些操作的op，然后下面有一个依赖关系，即先更新参数，再去用optimizer去优化losshttps://blog.csdn.net/huitailangyz/article/details/85015611

```
Starting epoch 0
Iteration 0, loss = 3.2466
Got 128 / 1000 correct (12.80%)

Iteration 100, loss = 1.8806
Got 371 / 1000 correct (37.10%)

Iteration 200, loss = 1.5007
Got 397 / 1000 correct (39.70%)

Iteration 300, loss = 1.8556
Got 398 / 1000 correct (39.80%)

Iteration 400, loss = 1.7928
Got 413 / 1000 correct (41.30%)

Iteration 500, loss = 1.7941
Got 446 / 1000 correct (44.60%)

Iteration 600, loss = 1.8809
Got 439 / 1000 correct (43.90%)

Iteration 700, loss = 1.9619
Got 438 / 1000 correct (43.80%)
```

运行结果

In [38]:
```python
hidden_size, num_classes = 4000, 10
learning_rate = 1e-2

def model_init_fn(inputs, is_training):
    return two_layer_fc_functional(inputs, hidden_size, num_classes)

def optimizer_init_fn():
    return tf.train.GradientDescentOptimizer(learning_rate)

train_part34(model_init_fn, optimizer_init_fn)
```

```
Starting epoch 0
Iteration 0, loss = 2.8403
Got 91 / 1000 correct (9.10%)

Iteration 100, loss = 1.8861
Got 362 / 1000 correct (36.20%)

Iteration 200, loss = 1.5436
Got 389 / 1000 correct (38.90%)

Iteration 300, loss = 1.8643
Got 365 / 1000 correct (36.50%)

Iteration 400, loss = 1.7693
Got 413 / 1000 correct (41.30%)

Iteration 500, loss = 1.7708
Got 425 / 1000 correct (42.50%)

Iteration 600, loss = 1.8917
Got 413 / 1000 correct (41.30%)

Iteration 700, loss = 1.9328
Got 447 / 1000 correct (44.70%)
```

两种方法的对比

```
In [40]: learning_rate = 3e-3
channel_1, channel_2, num_classes = 32, 16, 10

def model_init_fn(inputs, is_training):
    model = None
    ############################################################################
    # TODO: Complete the implementation of model_fn.                           #
    ############################################################################
    model = ThreeLayerConvNet(channel_1, channel_2, num_classes)
    ############################################################################
    #                            END OF YOUR CODE                              #
    ############################################################################
    return model(inputs)

def optimizer_init_fn():
    optimizer = None
    ############################################################################
    # TODO: Complete the implementation of model_fn.                           #
    ############################################################################
    optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                           momentum=0.9, use_nesterov=True)
    ############################################################################
    #                            END OF YOUR CODE                              #
    ############################################################################
    return optimizer

train_part34(model_init_fn, optimizer_init_fn)
```

这里使用三层网络，优化器为momentumSGD，use nesterov

```
Starting epoch 0
Iteration 0, loss = 2.8013
Got 133 / 1000 correct (13.30%)

Iteration 100, loss = 1.6081
Got 460 / 1000 correct (46.00%)

Iteration 200, loss = 1.2867
Got 481 / 1000 correct (48.10%)

Iteration 300, loss = 1.4354
Got 497 / 1000 correct (49.70%)

Iteration 400, loss = 1.3034
Got 511 / 1000 correct (51.10%)

Iteration 500, loss = 1.4733
Got 530 / 1000 correct (53.00%)

Iteration 600, loss = 1.3792
Got 550 / 1000 correct (55.00%)

Iteration 700, loss = 1.1496
Got 570 / 1000 correct (57.00%)
```

```
In [41]: learning_rate = 1e-2

def model_init_fn(inputs, is_training):
    input_shape = (32, 32, 3)
    hidden_layer_size, num_classes = 4000, 10
    initializer = tf.variance_scaling_initializer(scale=2.0)
    layers = [
        tf.layers.Flatten(input_shape=input_shape),
        tf.layers.Dense(hidden_layer_size, activation=tf.nn.relu,
                        kernel_initializer=initializer),
        tf.layers.Dense(num_classes, kernel_initializer=initializer),
    ]
    model = tf.keras.Sequential(layers)
    return model(inputs)

def optimizer_init_fn():
    return tf.train.GradientDescentOptimizer(learning_rate)

train_part34(model_init_fn, optimizer_init_fn)
```

```
Starting epoch 0
Iteration 0, loss = 2.7341
Got 126 / 1000 correct (12.60%)

Iteration 100, loss = 1.8776
Got 382 / 1000 correct (38.20%)

Iteration 200, loss = 1.4742
Got 381 / 1000 correct (38.10%)

Iteration 300, loss = 1.7330
Got 361 / 1000 correct (36.10%)

Iteration 400, loss = 1.7399
Got 412 / 1000 correct (41.20%)

Iteration 500, loss = 1.8389
Got 430 / 1000 correct (43.00%)

Iteration 600, loss = 1.7967
Got 435 / 1000 correct (43.50%)

Iteration 700, loss = 2.0145
Got 434 / 1000 correct (43.40%)
```

使用Keras API　主流

```
In [48]: def model_init_fn(inputs, is_training):
    model = None
    ############################################################################
    # TODO: Construct a three-layer ConvNet using tf.keras.Sequential.         #
    ############################################################################
    input_shape = (32,32,3)
    initializer = tf.variance_scaling_initializer(scale=2.0)
    layers = [tf.layers.Conv2D(input_shape=input_shape,filters=16,kernel_size=(5,5),
                                strides=(1,1),padding='same',activation=tf.nn.relu,
                                kernel_initializer=initializer),
              tf.layers.Conv2D(filters=32,kernel_size=(3,3),
                                strides=(1,1),padding='same',activation=tf.nn.relu,
                                kernel_initializer=initializer),
              tf.layers.Flatten(),
              tf.layers.Dense(units=10,kernel_initializer=initializer)
              ]
    model = tf.keras.Sequential(layers)
    ############################################################################
    #                            END OF YOUR CODE                              #
    ############################################################################
    return model(inputs)

learning_rate = 5e-4
```

```
def optimizer_init_fn():
    optimizer = None
    ############################################################################
    # TODO: Complete the implementation of model_fn.                           #
    ############################################################################
    optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                           momentum=0.9,use_nesterov=True)
    ############################################################################
    #                            END OF YOUR CODE                              #
    ############################################################################
    return optimizer

train_part34(model_init_fn, optimizer_init_fn)
```

这里用了一个Sequential（layers）

```
Starting epoch 0
Iteration 0, loss = 3.7231
Got 87 / 1000 correct (8.70%)

Iteration 100, loss = 1.8941
Got 408 / 1000 correct (40.80%)

Iteration 200, loss = 1.4370
Got 449 / 1000 correct (44.90%)

Iteration 300, loss = 1.4716
Got 467 / 1000 correct (46.70%)

Iteration 400, loss = 1.5022
Got 482 / 1000 correct (48.20%)

Iteration 500, loss = 1.6189
Got 507 / 1000 correct (50.70%)

Iteration 600, loss = 1.5774
Got 518 / 1000 correct (51.80%)

Iteration 700, loss = 1.5095
Got 529 / 1000 correct (52.90%)
```

运行结果

最后的训练，我用了GPU，环境配置有点麻烦，可以参考https://www.cnblogs.com/wanyu416/p/9536853.html注意版本号，这篇博文的版本号好像比较老旧，我这里是tensorflow-gpu1.13.1　然后CUDA是10.1，cudnn就配套CUDA就行。

模型因为要用GPU就没在他给的代码区里面写，自己在下面加的，上面的让他空着就行

```
device = '/device:GPU:0'
print_every = 700
num_epochs = 10
# train_part34(model_init_fn, optimizer_init_fn, num_epochs)
model = None
optimizer = None
input_shape = (32,32,3)
learning_rate = 1e-2
channel_1,channel_2,num_classes = 128,256,10
filter_1,filter_2,filter_3 = (5,5),(3,3),(1,1)

tf.reset_default_graph()
```

```
with tf.device(device):
    initializer = tf.variance_scaling_initializer(scale=2.0)

    layers = [tf.keras.layers.Conv2D(channel_1,filter_1,(1,1),'same',
                      use_bias=True,bias_initializer=tf.zeros_initializer(),
                      activation=tf.nn.relu,kernel_initializer=initializer),
              tf.keras.layers.BatchNormalization(),

              tf.keras.layers.Conv2D(channel_2,filter_2,(1,1),'same',
                      use_bias=True,bias_initializer=tf.zeros_initializer(),
                      activation=tf.nn.relu,kernel_initializer=initializer),
              tf.keras.layers.BatchNormalization(),
              tf.keras.layers.MaxPool2D(strides=2),

              tf.keras.layers.Conv2D(channel_2,filter_2,(1,1),'same',
                      use_bias=True,bias_initializer=tf.zeros_initializer(),
                      activation=tf.nn.relu,kernel_initializer=initializer),
              tf.keras.layers.BatchNormalization(),

              tf.keras.layers.Conv2D(channel_1,filter_1,(1,1),'same',
                      use_bias=True,bias_initializer=tf.zeros_initializer(),
                      activation=tf.nn.relu,kernel_initializer=initializer),
              tf.keras.layers.BatchNormalization(),
              tf.keras.layers.MaxPool2D(strides=2),

              tf.keras.layers.Flatten(),
              tf.keras.layers.Dense(num_classes,kernel_initializer=initializer,
                              kernel_regularizer=tf.keras.regularizers.l2(0.01)),
              tf.keras.layers.BatchNormalization(),
              tf.keras.layers.Softmax(),
             ]
```

```
    tf.keras.backend.clear_session()

    model = tf.keras.Sequential(layers)

#     optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
    optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                           momentum=0.9, use_nesterov=True)

    model.compile(optimizer,"sparse_categorical_crossentropy",metrics=['accuracy'])

    model.fit(X_train, y_train, batch_size=32, epochs=10, validation_data=(X_val, y_val))
```

```
Train on 49000 samples, validate on 1000 samples
Epoch 1/10
49000/49000 [==============================] - 120s 2ms/sample - loss: 1.2931 - acc: 0.5911 - val_loss: 1.0582 - val_acc: 0.6700
Epoch 2/10
49000/49000 [==============================] - 119s 2ms/sample - loss: 0.9914 - acc: 0.7012 - val_loss: 0.9209 - val_acc: 0.7540
Epoch 3/10
49000/49000 [==============================] - 122s 2ms/sample - loss: 0.8992 - acc: 0.7473 - val_loss: 0.8122 - val_acc: 0.7760
Epoch 4/10
49000/49000 [==============================] - 122s 2ms/sample - loss: 0.8337 - acc: 0.7813 - val_loss: 0.8121 - val_acc: 0.7900
Epoch 5/10
49000/49000 [==============================] - 123s 3ms/sample - loss: 0.7897 - acc: 0.8041 - val_loss: 0.7704 - val_acc: 0.8110
Epoch 6/10
49000/49000 [==============================] - 122s 2ms/sample - loss: 0.7407 - acc: 0.8264 - val_loss: 0.8542 - val_acc: 0.7990
Epoch 7/10
49000/49000 [==============================] - 123s 3ms/sample - loss: 0.7073 - acc: 0.8433 - val_loss: 0.8140 - val_acc: 0.8050
Epoch 8/10
49000/49000 [==============================] - 121s 2ms/sample - loss: 0.6652 - acc: 0.8637 - val_loss: 0.8166 - val_acc: 0.8120
Epoch 9/10
49000/49000 [==============================] - 121s 2ms/sample - loss: 0.6283 - acc: 0.8786 - val_loss: 0.7925 - val_acc: 0.8320
Epoch 10/10
49000/49000 [==============================] - 121s 2ms/sample - loss: 0.5970 - acc: 0.8922 - val_loss: 0.8647 - val_acc: 0.8260
```

模型：Conv-BN-Conv-BN-Maxpooling-Conv-BN-Conv-BN-Maxpooling-FC-BN-loss

tf.keras.backend.clear_session()该语句是为了重复调用模型

model.compile，第一个参数为优化器，第二个参数为loss，第三个参数为评估标准。

Model.fit详情查看百度。