import torch.nn.functional as F  # useful stateless functions

def two\_layer\_fc(x, params):

    """

    A fully-connected neural networks; the architecture is:

    NN is fully connected -> ReLU -> fully connected layer.

    Note that this function only defines the forward pass;

    PyTorch will take care of the backward pass 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 PyTorch Tensor of shape (N, d1, ..., dM) giving a minibatch of

      input data.

    - params: A list [w1, w2] of PyTorch Tensors giving weights for the network;

      w1 has shape (D, H) and w2 has shape (H, C).

    Returns:

    - scores: A PyTorch Tensor of shape (N, C) giving classification scores for

      the input data x.

    """

    # first we flatten the image

    x = flatten(x)  # shape: [batch\_size, C x H x W]

    w1, w2 = params

    # Forward pass: compute predicted y using operations on Tensors. Since w1 and

    # w2 have requires\_grad=True, operations involving these Tensors will cause

    # PyTorch to build a computational graph, allowing automatic computation of

    # gradients. Since we are no longer implementing the backward pass by hand we

    # don't need to keep references to intermediate values.

    # you can also use `.clamp(min=0)`, equivalent to F.relu()

    x = F.relu(x.mm(w1))

    x = x.mm(w2)

    return x

def three\_layer\_convnet(x, params):

    """

    Performs the forward pass of a three-layer convolutional network with the

    architecture defined above.

    Inputs:

    - x: A PyTorch Tensor of shape (N, 3, H, W) giving a minibatch of images

    - params: A list of PyTorch Tensors giving the weights and biases for the

      network; should contain the following:

      - conv\_w1: PyTorch Tensor of shape (channel\_1, 3, KH1, KW1) giving weights

        for the first convolutional layer

      - conv\_b1: PyTorch Tensor of shape (channel\_1,) giving biases for the first

        convolutional layer

      - conv\_w2: PyTorch Tensor of shape (channel\_2, channel\_1, KH2, KW2) giving

        weights for the second convolutional layer

      - conv\_b2: PyTorch Tensor of shape (channel\_2,) giving biases for the second

        convolutional layer

      - fc\_w: PyTorch Tensor giving weights for the fully-connected layer. Can you

        figure out what the shape should be?

      - fc\_b: PyTorch Tensor giving biases for the fully-connected layer. Can you

        figure out what the shape should be?

    Returns:

    - scores: PyTorch Tensor of shape (N, C) giving classification scores for x

    """

    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.                #

    ################################################################################

    # \*\*\*\*\*START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)\*\*\*\*\*

# 通过conv2d完成卷积 torch.nn.functional.conv2d(input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1) → Tensor

    conv1 = F.conv2d(x, conv\_w1, conv\_b1, stride=1, padding=2)

    relu1 = F.relu(conv1)

    conv2 = F.conv2d(relu1, conv\_w2, conv\_b2, stride=1, padding=1)

    relu2 = F.relu(conv2)

    # 把高维的relu2转化为(N,?)的relu2\_flatten，并实现全连接

    relu2\_flatten = flatten(relu2)

    scores = relu2\_flatten.mm(fc\_w) + fc\_b

    # \*\*\*\*\*END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)\*\*\*\*\*

    ################################################################################

    #                                 END OF YOUR CODE                             #

    ################################################################################

    return scores

def train\_part2(model\_fn, params, learning\_rate):

    """

    Train a model on CIFAR-10.

    Inputs:

    - model\_fn: A Python function that performs the forward pass of the model.

      It should have the signature scores = model\_fn(x, params) where x is a

      PyTorch Tensor of image data, params is a list of PyTorch Tensors giving

      model weights, and scores is a PyTorch Tensor of shape (N, C) giving

      scores for the elements in x.

    - params: List of PyTorch Tensors giving weights for the model

    - learning\_rate: Python scalar giving the learning rate to use for SGD

    Returns: Nothing

    """

    for t, (x, y) in enumerate(loader\_train):

        # Move the data to the proper device (GPU or CPU)

        x = x.to(device=device, dtype=dtype)

        y = y.to(device=device, dtype=torch.long)

        # Forward pass: compute scores and loss

        scores = model\_fn(x, params)

        loss = F.cross\_entropy(scores, y)

        # Backward pass: PyTorch figures out which Tensors in the computational

        # graph has requires\_grad=True and uses backpropagation to compute the

        # gradient of the loss with respect to these Tensors, and stores the

        # gradients in the .grad attribute of each Tensor.

        loss.backward()

        # Update parameters. We don't want to backpropagate through the

        # parameter updates, so we scope the updates under a torch.no\_grad()

        # context manager to prevent a computational graph from being built.

        with torch.no\_grad():

            for w in params:

                w -= learning\_rate \* w.grad

                # Manually zero the gradients after running the backward pass

                w.grad.zero\_()

        if t % print\_every == 0:

            print('Iteration %d, loss = %.4f' % (t, loss.item()))

            check\_accuracy\_part2(loader\_val, model\_fn, params)

            print()