Introduction to PyTorch

You've written a lot of code in this assignment to provide a whole host of neural network functionality. Dropout, Batch Norm, and 2D convolutions are some of the workhorses of deep learning in computer vision. You've also worked hard to make your code efficient and vectorized.

For the last part of this assignment, though, we're going to leave behind your beautiful codebase and instead migrate to one of two popular deep learning frameworks: in this instance, PyTorch.

Why do we use deep learning frameworks?

- 我们的代码现在可以在gpu上运行了!这将使我们的模型训练得更快。当使用像PyTorch、TensorFlow或Paddle这样的框架时，你可以为自己的自定义神经网络架构利用GPU的强大功能，而不必直接编写CUDA代码(这超出了本课程的范围)。
- 在这门课上，我们希望你准备好在你的项目中使用这些框架中的一个，这样你就可以比手工编写每个你想要使用的特性更有效地进行实验。
- 我们要你站在巨人的肩膀上！PyTorch、TensorFlow或Paddle都是很棒的框架，它们会让你的工作变得轻松很多，现在你已经理解了它们的精髓，你就可以自由使用它们了 :)
- 最后，我们希望你能够接触到你在学术界或工业界可能遇到的那种深度学习代码。

What is PyTorch?

PyTorch是一个用于在行为类似numpy ndarray的Tensor对象上执行动态计算图的系统。它附带了一个强大的自动微分引擎，消除了手动反向传播的需要。

How do I learn PyTorch?

我们的一位前导师Justin Johnson为PyTorch制作了一个非常棒的教程(https://github.com/jcjohnson/pytorch-examples)。

你也可以在这里找到详细的API文档。 如果你有API文档没有解决的其他问题，PyTorch论坛是一个比StackOverflow更好的提问场所。

Table of Contents

这个作业有5个部分。您将学习PyTorch的三个不同抽象层次，这将帮助您更好地理解它，并为最终项目做好准备。

1. Part I, Preparation: 我们将使用CIFAR-10数据集。

2. Part II, Barebones PyTorch: **Abstraction level 1**, 我们将直接使用最低级别的PyTorch张量。
3. Part III, PyTorch Module API: **Abstraction level 2**, 我们将使用 `nn.Module` 模块来定义任意的神经网络架构。
4. Part IV, PyTorch Sequential API: **Abstraction level 3**, 我们将使用 `nn.Sequential`，非常方便地定义线性前馈网络。
5. Part V, CIFAR-10开放挑战：请在CIFAR-10上实现你自己的网络，以获得尽可能高的精度。你可以尝试任何层，优化器，超参数或其他高级功能。越高的准确率得分越高。

Here is a table of comparison:

API	Flexibility	Convenience
Barebone	High	Low
`nn.Module`	High	Medium
`nn.Sequential`	Low	High

GPU

In [1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.utils.data import sampler

import torchvision.datasets as dset
import torchvision.transforms as T

import numpy as np

USE_GPU = True
dtype = torch.float32 # We will be using float throughout this tutorial.

if USE_GPU and torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

# Constant to control how frequently we print train loss.
print_every = 100
print('using device:', device)
```

```
using device: cuda
```

Part I. Preparation

现在，让我们加载CIFAR-10数据集。第一次这样做可能需要几分钟，但之后文件应该保持缓存状态。

在作业的前几部分，我们必须编写自己的代码来下载CIFAR-10数据集，预处理它，并在小批量中遍历它;PyTorch为我们提供了方便的工具来自动化这个过程。

In [2]:
```
NUM_TRAIN = 49000

# torchvision.transforms 包提供了用于预处理数据和执行数据增强的工具;
# 在这里，我们设置了一个变换，通过减去平均 RGB 值并除以每个 RGB 值的标准差来预处理
# 我们已经硬编码了平均值和标准差。
transform = T.Compose([
                T.ToTensor(),
                T.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
            ])

# 我们为每次拆分设置一个Dataset对象(train / val / test); 数据集每次加载一个训练示
# 因此我们将每个数据集包装在一个DataLoader中，该DataLoader在数据集中迭代并形成小批
# 我们将CIFAR-10训练集分为train集和val集，方法是向DataLoader传递一个Sampler对象，
# 告诉它应该如何从底层数据集进行采样。

cifar10_train = dset.CIFAR10('./datasets', train=True, download=True,
                             transform=transform)
loader_train = DataLoader(cifar10_train, batch_size=64,
                          sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN)))

cifar10_val = dset.CIFAR10('./datasets', train=True, download=True,
                           transform=transform)
loader_val = DataLoader(cifar10_val, batch_size=64,
                        sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN, 500

cifar10_test = dset.CIFAR10('./datasets', train=False, download=True,
                            transform=transform)
loader_test = DataLoader(cifar10_test, batch_size=64)
```

```
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
```

Part II. Barebones PyTorch

PyTorch附带了高级api，可以帮助我们方便地定义模型体系结构，我们将在本教程的第二部分中介绍。在本节中，我们将从最基本的PyTorch元素开始，以便更好地理解自动升级引擎。在这个练习之后，您将会更加欣赏高级模型API。

我们将从一个简单的全连接ReLU网络开始，该网络具有两个隐藏层，并且对CIFAR分类没有偏见。 这个实现使用PyTorch张量上的操作来计算正向传递，并使用PyTorch autograd来计算梯度。理解每一行很重要，因为您将在示例之后编写更难的版本。

当我们创建一个带有' requires_grad=True '的PyTorch张量时，涉及到这个张量的操作将不仅仅是计算值;他们还会在后台建立一个计算图，让我们可以很容易地通过图进行反向传播，计算一些张量关于下游损失的梯度。具体来说，如果x是一个带有' x.requires_grad == True'的张量，然后在反向传播' x.grad '将是另一个张量，保持x的梯度，相对于最后的标量损失。

PyTorch Tensors: Flatten Function

PyTorch张量在概念上类似于numpy数组:它是一个n维的数字网格，和numpy一样，PyTorch提供了许多函数来有效地操作张量。作为一个简单的例子，我们在下面提供了一个flatten函数，它可以重塑图像数据，以便在完全连接的神经网络中使用。

Recall that image data is typically stored in a Tensor of shape N x C x H x W, where:

- N is the number of datapoints
- C is the number of channels
- H is the height of the intermediate feature map in pixels
- W is the height of the intermediate feature map in pixels

当我们在做像二维卷积这样的事情时，这是表示数据的正确方式，这需要对中间特征彼此相对位置的空间理解。然而，当我们使用完全连接的仿射层处理图像时，我们希望每个数据点由单个向量表示——分隔数据的不同通道、行和列不再有用。所以，我们使用"flatten"操作将每个表示的"C x H x W"值折叠成一个长向量。下面的flatten函数首先从给定的一批数据中读入N、C、H和W值，然后返回该数据的"View"。"View"类似于numpy的"reshape"方法:它将x的维度重塑为N x ??,在哪里? ?可以是任何值(在本例中，它将是C x H x W，但我们不需要显式地指定它)。

In [3]:
```
def flatten(x):
    N = x.shape[0] # read in N, C, H, W
    return x.view(N, -1)  # "flatten" the C * H * W values into a single vector

def test_flatten():
    x = torch.arange(12).view(2, 1, 3, 2)
    print('Before flattening: ', x)
    print('After flattening: ', flatten(x))

test_flatten()
```

```
Before flattening:  tensor([[[[ 0,  1],
          [ 2,  3],
          [ 4,  5]]],


        [[[ 6,  7],
          [ 8,  9],
          [10, 11]]]])
After flattening:  tensor([[ 0,  1,  2,  3,  4,  5],
        [ 6,  7,  8,  9, 10, 11]])
```

Barebones PyTorch: Two-Layer Network

在这里，我们定义了一个函数two_layer_fc，它在一批图像数据上执行两层全连接ReLU网络的转发传递。在定义了正向传递之后，我们检查它不会崩溃，并且通过在网络中运行零来产生正确形状的输出。

你不需要在这里编写任何代码，但是阅读和理解实现是很重要的。

In [4]:
```
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 an
    # 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 w
    # 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 two_layer_fc_test():
    hidden_layer_size = 42
    x = torch.zeros((64, 50), dtype=dtype)  # minibatch size 64, feature dimensi
    w1 = torch.zeros((50, hidden_layer_size), dtype=dtype)
    w2 = torch.zeros((hidden_layer_size, 10), dtype=dtype)
    scores = two_layer_fc(x, [w1, w2])
    print(scores.size())  # you should see [64, 10]

two_layer_fc_test()
```

```
torch.Size([64, 10])
```

Barebones PyTorch: Three-Layer ConvNet

在这里，您将完成函数three_layer_convnet的实现，该函数将执行三层卷积网络的正向传递。与上面一样，我们可以通过在网络中传递零来立即测试我们的实现。网络应具有以下架构:

1. A convolutional layer (with bias) with `channel_1` filters, each with shape `KW1 x KH1`, and zero-padding of two
2. ReLU nonlinearity

3. A convolutional layer (with bias) with `channel_2` filters, each with shape `KW2 x KH2`, and zero-padding of one
4. ReLU nonlinearity
5. Fully-connected layer with bias, producing scores for C classes.

注意，在我们的全连接层之后，我们这里没有**softmax activation**: 这是因为PyTorch的交叉熵损失为你执行了softmax激活，并通过绑定该步骤使计算更有效。

HINT: For convolutions: http://pytorch.org/docs/stable/nn.html#torch.nn.functional.conv2d; pay attention to the shapes of convolutional filters!

In [5]:
```
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 firs
        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 seco
        convolutional layer
      - fc_w: PyTorch Tensor giving weights for the fully-connected layer. Can y
        figure out what the shape should be?
      - fc_b: PyTorch Tensor giving biases for the fully-connected layer. Can yo
        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)*****

    x = torch.nn.functional.conv2d(x, conv_w1, conv_b1, stride=1, padding=2)
    x = x.clamp(min = 0)
    x = torch.nn.functional.conv2d(x, conv_w2, conv_b2, stride=1, padding=1)
    x = x.clamp(min = 0)
    x = flatten(x)
    scores = x.mm(fc_w) + fc_b

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ################################################################################
    #                                 END OF YOUR CODE
    ################################################################################
    return scores
```

在定义了上面ConvNet的向前传递之后，运行以下单元格来测试您的实现。

当运行这个函数时，分数应该具有形状(64,10)。

In [6]:
```
def three_layer_convnet_test():
    x = torch.zeros((64, 3, 32, 32), dtype=dtype)  # minibatch size 64, image si

    conv_w1 = torch.zeros((6, 3, 5, 5), dtype=dtype)  # [out_channel, in_channel
    conv_b1 = torch.zeros((6,))  # out_channel
    conv_w2 = torch.zeros((9, 6, 3, 3), dtype=dtype)  # [out_channel, in_channel
    conv_b2 = torch.zeros((9,))  # out_channel

    # you must calculate the shape of the tensor after two conv layers, before t
    fc_w = torch.zeros((9 * 32 * 32, 10))
    fc_b = torch.zeros(10)

    scores = three_layer_convnet(x, [conv_w1, conv_b1, conv_w2, conv_b2, fc_w, f
    print(scores.size())  # you should see [64, 10]
three_layer_convnet_test()
```

```
torch.Size([64, 10])
```

Barebones PyTorch: Initialization

让我们编写两个实用程序方法来初始化模型的权重矩阵。

- `random_weight(shape)` 用Kaiming归一化方法初始化一个权张量。
- `zero_weight(shape)` 初始化一个全为0的权值张量。用于实例化偏置参数。

The `random_weight` function uses the Kaiming normal initialization method, described in:

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

In [7]:
```
def random_weight(shape):
    """
    Create random Tensors for weights; setting requires_grad=True means that we
    want to compute gradients for these Tensors during the backward pass.
    We use Kaiming normalization: sqrt(2 / fan_in)
    """
    if len(shape) == 2:  # FC weight
        fan_in = shape[0]
    else:
        fan_in = np.prod(shape[1:]) # conv weight [out_channel, in_channel, kH,
    # randn is standard normal distribution generator.
    w = torch.randn(shape, device=device, dtype=dtype) * np.sqrt(2. / fan_in)
    w.requires_grad = True
    return w

def zero_weight(shape):
    return torch.zeros(shape, device=device, dtype=dtype, requires_grad=True)

# create a weight of shape [3 x 5]
# you should see the type `torch.cuda.FloatTensor` if you use GPU.
```

```
# Otherwise it should be `torch.FloatTensor`
random_weight((3, 5))
```

Out[7]:
```
tensor([[-0.0097,  0.5370, -0.4325,  0.4246,  0.6793],
        [-0.6077,  2.1023, -1.5932, -0.5714, -0.7334],
        [-0.7410,  0.7438, -0.4543, -1.5367,  0.3243]], device='cuda:0',
       requires_grad=True)
```

Barebones PyTorch: Check Accuracy

当训练模型时，我们将使用以下函数来检查我们的模型在训练集或验证集上的准确性。

在检查精确度时，我们不需要计算任何梯度;因此，当我们计算分数时，我们不需要PyTorch为我们构建计算图。为了防止图被构建，我们在' torch.no_grad() '上下文管理器下进行计算。

In [8]:
```
def check_accuracy_part2(loader, model_fn, params):
    """
    Check the accuracy of a classification model.

    Inputs:
    - loader: A DataLoader for the data split we want to check
    - model_fn: A function that performs the forward pass of the model,
      with the signature scores = model_fn(x, params)
    - params: List of PyTorch Tensors giving parameters of the model

    Returns: Nothing, but prints the accuracy of the model
    """
    split = 'val' if loader.dataset.train else 'test'
    print('Checking accuracy on the %s set' % split)
    num_correct, num_samples = 0, 0
    with torch.no_grad():
        for x, y in loader:
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.int64)
            scores = model_fn(x, params)
            _, preds = scores.max(1)
            num_correct += (preds == y).sum()
            num_samples += preds.size(0)
        acc = float(num_correct) / num_samples
        print('Got %d / %d correct (%.2f%%)' % (num_correct, num_samples, 100 *
```

BareBones PyTorch: Training Loop

We can now set up a basic training loop to train our network. We will train the model using stochastic gradient descent without momentum. We will use `torch.functional.cross_entropy` to compute the loss; you can read about it here.

The training loop takes as input the neural network function, a list of initialized parameters (`[w1, w2]` in our example), and learning rate.

In [9]:
```
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()
```

BareBones PyTorch: Train a Two-Layer Network

现在我们准备运行训练循环。我们需要显式地为全连接权值' w1 '和' w2 '分配张量。

CIFAR的每个小批都有64个示例，因此张量形状为'[64,3,32,32]'。

扁平化后，' x '形状应该是'[64,3 * 32 * 32]'。这将是' w1 '的第一个维度的大小。 ' w1 '的第二个维度是隐藏层的大小，它也将是' w2 '的第一个维度。

最后，网络的输出是一个表示10个类的概率分布的10维向量。

你不需要调优任何超参数，但你应该看到准确性超过40%训练后的一个epoch。

In [12]:
```
hidden_layer_size = 4000
learning_rate = 1e-2
```

```
w1 = random_weight((3 * 32 * 32, hidden_layer_size))
w2 = random_weight((hidden_layer_size, 10))

train_part2(two_layer_fc, [w1, w2], learning_rate)
```

```
Iteration 0, loss = 3.7337
Checking accuracy on the val set
Got 133 / 1000 correct (13.30%)

Iteration 100, loss = 2.4485
Checking accuracy on the val set
Got 290 / 1000 correct (29.00%)

Iteration 200, loss = 2.0484
Checking accuracy on the val set
Got 378 / 1000 correct (37.80%)

Iteration 300, loss = 1.9386
Checking accuracy on the val set
Got 396 / 1000 correct (39.60%)

Iteration 400, loss = 1.9444
Checking accuracy on the val set
Got 428 / 1000 correct (42.80%)

Iteration 500, loss = 2.1081
Checking accuracy on the val set
Got 406 / 1000 correct (40.60%)

Iteration 600, loss = 1.4009
Checking accuracy on the val set
Got 436 / 1000 correct (43.60%)

Iteration 700, loss = 1.7375
Checking accuracy on the val set
Got 455 / 1000 correct (45.50%)
```

BareBones PyTorch: Training a ConvNet

在下文中，您应该使用上面定义的函数在CIFAR上训练三层卷积网络。网络应具有以下架构:

1. Convolutional layer (with bias) with 32 5x5 filters, with zero-padding of 2
2. ReLU
3. Convolutional layer (with bias) with 16 3x3 filters, with zero-padding of 1
4. ReLU
5. Fully-connected layer (with bias) to compute scores for 10 classes

你应该使用上面定义的' random_weight '函数初始化你的权重矩阵，你应该使用上面定义的' zero_weight '函数初始化你的偏置向量。

你不需要调优任何超参数，但如果一切正常，你应该在一个epoch后达到42%以上的精度。

In [13]:
```
learning_rate = 3e-3
```

```
channel_1 = 32
channel_2 = 16

conv_w1 = None
conv_b1 = None
conv_w2 = None
conv_b2 = None
fc_w = None
fc_b = None

################################################################################
# TODO: Initialize the parameters of a three-layer ConvNet.                    #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

conv_w1 = random_weight((channel_1, 3, 5, 5))
conv_b1 = zero_weight(channel_1)
conv_w2 = random_weight((channel_2, channel_1, 3, 3))
conv_b2 = zero_weight(channel_2)
fc_w = random_weight((32*32*channel_2, 10))
fc_b = zero_weight(10)

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

params = [conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b]
train_part2(three_layer_convnet, params, learning_rate)
```

```
Iteration 0, loss = 3.6305
Checking accuracy on the val set
Got 96 / 1000 correct (9.60%)

Iteration 100, loss = 1.9413
Checking accuracy on the val set
Got 341 / 1000 correct (34.10%)

Iteration 200, loss = 1.7457
Checking accuracy on the val set
Got 373 / 1000 correct (37.30%)

Iteration 300, loss = 1.6714
Checking accuracy on the val set
Got 423 / 1000 correct (42.30%)

Iteration 400, loss = 1.9223
Checking accuracy on the val set
Got 405 / 1000 correct (40.50%)

Iteration 500, loss = 1.3667
Checking accuracy on the val set
Got 437 / 1000 correct (43.70%)

Iteration 600, loss = 1.4911
Checking accuracy on the val set
Got 469 / 1000 correct (46.90%)

Iteration 700, loss = 1.5586
Checking accuracy on the val set
Got 464 / 1000 correct (46.40%)
```

Part III. PyTorch Module API

Barebone PyTorch要求我们手动跟踪所有的参数张量。这对于只有几个张量的小型网络来说是很好的，但是在较大的网络中跟踪几十或几百个张量将非常不方便和容易出错。

PyTorch provides the `nn.Module` API for you to define arbitrary network architectures, while tracking every learnable parameters for you. In Part II, we implemented SGD ourselves. PyTorch also provides the `torch.optim` package that implements all the common optimizers, such as RMSProp, Adagrad, and Adam. It even supports approximate second-order methods like L-BFGS! You can refer to the doc for the exact specifications of each optimizer.

To use the Module API, follow the steps below:

1. Subclass `nn.Module`. Give your network class an intuitive name like `TwoLayerFC`.

2. In the constructor `__init__()`, define all the layers you need as class attributes. Layer objects like `nn.Linear` and `nn.Conv2d` are themselves `nn.Module` subclasses and contain learnable parameters, so that you don't have to instantiate the raw tensors yourself. `nn.Module` will track these

internal parameters for you. Refer to the doc to learn more about the dozens of builtin layers. **Warning**: don't forget to call the `super().__init__()` first!
3. In the `forward()` method, define the *connectivity* of your network. You should use the attributes defined in `__init__` as function calls that take tensor as input and output the "transformed" tensor. Do *not* create any new layers with learnable parameters in `forward()`! All of them must be declared upfront in `__init__`.

After you define your Module subclass, you can instantiate it as an object and call it just like the NN forward function in part II.

Module API: Two-Layer Network

Here is a concrete example of a 2-layer fully connected network:

In [14]:
```
class TwoLayerFC(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super().__init__()
        # assign layer objects to class attributes
        self.fc1 = nn.Linear(input_size, hidden_size)
        # nn.init package contains convenient initialization methods
        # http://pytorch.org/docs/master/nn.html#torch-nn-init
        nn.init.kaiming_normal_(self.fc1.weight)
        self.fc2 = nn.Linear(hidden_size, num_classes)
        nn.init.kaiming_normal_(self.fc2.weight)

    def forward(self, x):
        # forward always defines connectivity
        x = flatten(x)
        scores = self.fc2(F.relu(self.fc1(x)))
        return scores

def test_TwoLayerFC():
    input_size = 50
    x = torch.zeros((64, input_size), dtype=dtype)  # minibatch size 64, feature
    model = TwoLayerFC(input_size, 42, 10)
    scores = model(x)
    print(scores.size())  # you should see [64, 10]
test_TwoLayerFC()
```

```
torch.Size([64, 10])
```

Module API: Three-Layer ConvNet

现在轮到你实现一个3层ConvNet，然后是一个完全连接的层。网络架构应与第二部分相同:

1. Convolutional layer with `channel_1` 5x5 filters with zero-padding of 2
2. ReLU
3. Convolutional layer with `channel_2` 3x3 filters with zero-padding of 1
4. ReLU
5. Fully-connected layer to `num_classes` classes

You should initialize the weight matrices of the model using the Kaiming normal initialization method.

HINT: http://pytorch.org/docs/stable/nn.html#conv2d

After you implement the three-layer ConvNet, the `test_ThreeLayerConvNet` function will run your implementation; it should print `(64, 10)` for the shape of the output scores.

In [15]:
```
class ThreeLayerConvNet(nn.Module):
    def __init__(self, in_channel, channel_1, channel_2, num_classes):
        super().__init__()
        ########################################################################
        # TODO: Set up the layers you need for a three-layer ConvNet with the  #
        # architecture defined above.                                          #
        ########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        self.conv1 = nn.Conv2d(in_channel, channel_1, 5, 1, 2)
        self.conv2 = nn.Conv2d(channel_1, channel_2, 3, 1, 1)
        self.fc = nn.Linear(channel_2*32*32, num_classes)

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

    def forward(self, x):
        scores = None
        ########################################################################
        # TODO: Implement the forward function for a 3-layer ConvNet. you      #
        # should use the layers you defined in __init__ and specify the        #
        # connectivity of those layers in forward()                            #
        ########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = flatten(x)
        scores = self.fc(x)

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


def test_ThreeLayerConvNet():
    x = torch.zeros((64, 3, 32, 32), dtype=dtype)  # minibatch size 64, image si
    model = ThreeLayerConvNet(in_channel=3, channel_1=12, channel_2=8, num_class
    scores = model(x)
    print(scores.size())  # you should see [64, 10]
test_ThreeLayerConvNet()
```

```
torch.Size([64, 10])
```

Module API: Check Accuracy

给定验证集或测试集，我们可以检验神经网络的分类精度。

这个版本与第二部分的版本略有不同。您不再手动传递参数了。

In [17]:
```
def check_accuracy_part34(loader, model):
    if loader.dataset.train:
        print('Checking accuracy on validation set')
    else:
        print('Checking accuracy on test set')
    num_correct = 0
    num_samples = 0
    model.eval()  # set model to evaluation mode
    with torch.no_grad():
        for x, y in loader:
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.long)
            scores = model(x)
            _, preds = scores.max(1)
            num_correct += (preds == y).sum()
            num_samples += preds.size(0)
        acc = float(num_correct) / num_samples
        print('Got %d / %d correct (%.2f)' % (num_correct, num_samples, 100 * ac
```

Module API: Training Loop

We also use a slightly different training loop. Rather than updating the values of the weights ourselves, we use an Optimizer object from the `torch.optim` package, which abstract the notion of an optimization algorithm and provides implementations of most of the algorithms commonly used to optimize neural networks.

In [18]:
```
def train_part34(model, optimizer, epochs=1):
    """
    使用PyTorch模块API在CIFAR-10上训练模型。

    Inputs:
    - model: A PyTorch Module giving the model to train.
    - optimizer: An Optimizer object we will use to train the model
    - epochs: (Optional) A Python integer giving the number of epochs to train f

    Returns: Nothing, but prints model accuracies during training.
    """
    model = model.to(device=device)  # move the model parameters to CPU/GPU
    for e in range(epochs):
        for t, (x, y) in enumerate(loader_train):
            model.train()  # put model to training mode
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.long)

            scores = model(x)
            loss = F.cross_entropy(scores, y)

            # Zero out all of the gradients for the variables which the optimize
```

```
            # will update.
            optimizer.zero_grad()

            # This is the backwards pass: compute the gradient of the loss with
            # respect to each  parameter of the model.
            loss.backward()

            # Actually update the parameters of the model using the gradients
            # computed by the backwards pass.
            optimizer.step()

            if t % print_every == 0:
                print('Iteration %d, loss = %.4f' % (t, loss.item()))
                check_accuracy_part34(loader_val, model)
                print()
```

Module API: Train a Two-Layer Network

现在我们准备运行训练循环。与第二部分相比，我们不再显式分配参数张量。

简单地将输入大小、隐藏层大小和类的数量(即输出大小)传递给' TwoLayerFC '的构造函数。

你还需要定义一个优化器来跟踪' TwoLayerFC '中的所有可学习参数。

您不需要调优任何超参数，但是您应该会看到在一个epoch的训练后模型的准确性超过40%。

In [19]:
```
hidden_layer_size = 4000
learning_rate = 1e-2
model = TwoLayerFC(3 * 32 * 32, hidden_layer_size, 10)
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

train_part34(model, optimizer)
```

```
Iteration 0, loss = 3.1989
Checking accuracy on validation set
Got 104 / 1000 correct (10.40)

Iteration 100, loss = 2.3239
Checking accuracy on validation set
Got 373 / 1000 correct (37.30)

Iteration 200, loss = 2.1270
Checking accuracy on validation set
Got 373 / 1000 correct (37.30)

Iteration 300, loss = 2.1504
Checking accuracy on validation set
Got 358 / 1000 correct (35.80)

Iteration 400, loss = 1.4748
Checking accuracy on validation set
Got 424 / 1000 correct (42.40)

Iteration 500, loss = 1.6731
Checking accuracy on validation set
Got 418 / 1000 correct (41.80)

Iteration 600, loss = 1.6290
Checking accuracy on validation set
Got 442 / 1000 correct (44.20)

Iteration 700, loss = 1.6193
Checking accuracy on validation set
Got 414 / 1000 correct (41.40)
```

Module API: Train a Three-Layer ConvNet

你现在应该使用模块API在CIFAR上训练三层ConvNet。这看起来应该非常类似于训练双层网络!你不需要调优任何超参数，但你应该达到45%以上的训练后一个时代。

你应该使用无动量的随机梯度下降来训练模型。

In [20]:
```
learning_rate = 3e-3
channel_1 = 32
channel_2 = 16

model = None
optimizer = None
################################################################################
# TODO: Instantiate your ThreeLayerConvNet model and a corresponding optimizer #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

model = ThreeLayerConvNet(3, channel_1, channel_2, 10)
optimizer = optim.SGD(model.parameters(), lr = learning_rate)
train_part34(model, optimizer, epochs=1)

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

```
##############################################################################

train_part34(model, optimizer)
```

```
Iteration 0, loss = 2.3227
Checking accuracy on validation set
Got 108 / 1000 correct (10.80)

Iteration 100, loss = 2.0137
Checking accuracy on validation set
Got 297 / 1000 correct (29.70)

Iteration 200, loss = 1.8606
Checking accuracy on validation set
Got 352 / 1000 correct (35.20)

Iteration 300, loss = 1.8494
Checking accuracy on validation set
Got 369 / 1000 correct (36.90)

Iteration 400, loss = 1.8042
Checking accuracy on validation set
Got 416 / 1000 correct (41.60)

Iteration 500, loss = 1.7413
Checking accuracy on validation set
Got 404 / 1000 correct (40.40)

Iteration 600, loss = 1.7719
Checking accuracy on validation set
Got 412 / 1000 correct (41.20)

Iteration 700, loss = 1.5279
Checking accuracy on validation set
Got 455 / 1000 correct (45.50)

Iteration 0, loss = 1.7942
Checking accuracy on validation set
Got 468 / 1000 correct (46.80)

Iteration 100, loss = 1.6910
Checking accuracy on validation set
Got 474 / 1000 correct (47.40)

Iteration 200, loss = 1.2976
Checking accuracy on validation set
Got 461 / 1000 correct (46.10)

Iteration 300, loss = 1.3676
Checking accuracy on validation set
Got 477 / 1000 correct (47.70)

Iteration 400, loss = 1.6848
Checking accuracy on validation set
Got 484 / 1000 correct (48.40)

Iteration 500, loss = 1.6594
Checking accuracy on validation set
Got 498 / 1000 correct (49.80)

Iteration 600, loss = 1.3867
Checking accuracy on validation set
Got 504 / 1000 correct (50.40)
```

```
Iteration 700, loss = 1.4801
Checking accuracy on validation set
Got 506 / 1000 correct (50.60)
```

Part IV. PyTorch Sequential API

第三部分介绍了PyTorch模块API，它允许您定义任意可学习层及其连通性。

For simple models like a stack of feed forward layers, you still need to go through 3 steps: subclass `nn.Module`, assign layers to class attributes in `__init__`, and call each layer one by one in `forward()`. Is there a more convenient way?

Fortunately, PyTorch provides a container Module called `nn.Sequential`, which merges the above steps into one. It is not as flexible as `nn.Module`, because you cannot specify more complex topology than a feed-forward stack, but it's good enough for many use cases.

Sequential API: Two-Layer Network

Let's see how to rewrite our two-layer fully connected network example with `nn.Sequential`, and train it using the training loop defined above.

Again, you don't need to tune any hyperparameters here, but you shoud achieve above 40% accuracy after one epoch of training.

In [21]:
```
# We need to wrap `flatten` function in a module in order to stack it
# in nn.Sequential
class Flatten(nn.Module):
    def forward(self, x):
        return flatten(x)

hidden_layer_size = 4000
learning_rate = 1e-2

model = nn.Sequential(
    Flatten(),
    nn.Linear(3 * 32 * 32, hidden_layer_size),
    nn.ReLU(),
    nn.Linear(hidden_layer_size, 10),
)

# you can use Nesterov momentum in optim.SGD
optimizer = optim.SGD(model.parameters(), lr=learning_rate,
                      momentum=0.9, nesterov=True)

train_part34(model, optimizer)
```

```
Iteration 0, loss = 2.3042
Checking accuracy on validation set
Got 145 / 1000 correct (14.50)

Iteration 100, loss = 2.0466
Checking accuracy on validation set
Got 387 / 1000 correct (38.70)

Iteration 200, loss = 1.7456
Checking accuracy on validation set
Got 420 / 1000 correct (42.00)

Iteration 300, loss = 2.0917
Checking accuracy on validation set
Got 413 / 1000 correct (41.30)

Iteration 400, loss = 1.8159
Checking accuracy on validation set
Got 438 / 1000 correct (43.80)

Iteration 500, loss = 2.0000
Checking accuracy on validation set
Got 423 / 1000 correct (42.30)

Iteration 600, loss = 2.1194
Checking accuracy on validation set
Got 436 / 1000 correct (43.60)

Iteration 700, loss = 1.8402
Checking accuracy on validation set
Got 450 / 1000 correct (45.00)
```

Sequential API: Three-Layer ConvNet

这里你应该使用 `nn.Sequential` 定义和训练一个三层ConvNet，其体系结构与我们在第三部分中使用的相同:

1. Convolutional layer (with bias) with 32 5x5 filters, with zero-padding of 2
2. ReLU
3. Convolutional layer (with bias) with 16 3x3 filters, with zero-padding of 1
4. ReLU
5. Fully-connected layer (with bias) to compute scores for 10 classes

你可以使用默认的PyTorch权重初始化。

你应该使用带有Nesterov动量0.9的随机梯度下降来优化你的模型。

同样，您不需要调优任何超参数，但在一个训练周期后，您应该看到准确率超过55%。

In [22]:
```
channel_1 = 32
channel_2 = 16
learning_rate = 1e-2

model = None
optimizer = None
```

```
################################################################################
# TODO: Rewrite the 2-layer ConvNet with bias from Part III with the           #
# Sequential API.                                                              #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

model = nn.Sequential(
    nn.Conv2d(3, channel_1, 5, 1, 2),
    nn.ReLU(),
    nn.Conv2d(channel_1, channel_2, 3, 1, 1),
    nn.ReLU(),
    Flatten(),
    nn.Linear(channel_2*32*32, 10)
)

optimizer = optim.SGD(model.parameters(), lr = learning_rate,
                      momentum = 0.9, nesterov = True)

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

train_part34(model, optimizer)
```

```
Iteration 0, loss = 2.3329
Checking accuracy on validation set
Got 87 / 1000 correct (8.70)

Iteration 100, loss = 1.8282
Checking accuracy on validation set
Got 426 / 1000 correct (42.60)

Iteration 200, loss = 1.4421
Checking accuracy on validation set
Got 478 / 1000 correct (47.80)

Iteration 300, loss = 1.2983
Checking accuracy on validation set
Got 496 / 1000 correct (49.60)

Iteration 400, loss = 1.3239
Checking accuracy on validation set
Got 531 / 1000 correct (53.10)

Iteration 500, loss = 1.4000
Checking accuracy on validation set
Got 538 / 1000 correct (53.80)

Iteration 600, loss = 1.0529
Checking accuracy on validation set
Got 583 / 1000 correct (58.30)

Iteration 700, loss = 0.9892
Checking accuracy on validation set
Got 588 / 1000 correct (58.80)
```

Part V. CIFAR-10 开放性挑战

在本节中，您可以在CIFAR-10上试验你喜欢的任何ConvNet体系结构。

利用所学知识，尽可能提高准确率，越高的准确率得分越高。

最低要求：对结构、超参数、损失函数和优化器进行实验，以训练一个在10个epochs内在CIFAR-10 **验证集上实现至少70%**精度的模型。可以使用上面的check_accuracy和train函数。你可以用任意一个 `nn.Module` or `nn.Sequential` API。

在notebook的最后描述一下你做了什么。

以下是每个组件的官方API文档。注意:我们在类中称为“spatial batch norm”的东西在PyTorch中称为“BatchNorm2D”。

- Layers in torch.nn package: http://pytorch.org/docs/stable/nn.html
- Activations: http://pytorch.org/docs/stable/nn.html#non-linear-activations
- Loss functions: http://pytorch.org/docs/stable/nn.html#loss-functions
- Optimizers: http://pytorch.org/docs/stable/optim.html

你可以尝试的事情:

- **Filter size**: 上面我们使用的是5x5;更小的滤波器会更有效吗?
- **Number of filters**: 上面我们使用了32个滤波器。是多一点好还是少一点好?
- **Pooling vs Strided Convolution**: Do you use max pooling or just stride convolutions?
- **Batch normalization**: 尝试在卷积层之后添加spatial batch normalization，在affine layers之后添加vanilla batch normalization。你的网络训练速度更快吗?
- **Network architecture**: 上面的网络有两层可训练参数。有了深度网络，你能做得更好吗?可以尝试的良好架构包括:
  - [conv-relu-pool]xN -> [affine]xM -> [softmax or SVM]
  - [conv-relu-conv-relu-pool]xN -> [affine]xM -> [softmax or SVM]
  - [batchnorm-relu-conv]xN -> [affine]xM -> [softmax or SVM]
- **Global Average Pooling**: Instead of flattening and then having multiple affine layers, perform convolutions until your image gets small (7x7 or so) and then perform an average pooling operation to get to a 1x1 image picture (1, 1 , Filter#), which is then reshaped into a (Filter#) vector. This is used in Google's Inception Network (See Table 1 for their architecture).
- **Regularization**: 添加l2权重正则化，或者使用Dropout。

Tips for training

对于你尝试的每种网络体系结构，你都应该调优学习率和其他超参数。在这样做的时候，有几件重要的事情要记住:

- 如果参数运行良好，您应该在几百次迭代中看到改进
- 记住超参数调优的从粗到细的方法:首先测试大范围的超参数，只进行一些训练迭代，以找到有效的参数组合。
- 一旦找到了一些似乎有效的参数集，就可以围绕这些参数进行更细致的搜索。你可能需要训练更多的时间。

- 你应该使用验证集进行超参数搜索，并保存你的测试集，以便根据验证集选择的最佳参数对体系结构进行评估。

Going above and beyond

还有很多其他功能可以实现，以尝试和提高你的性能。

- 可选的优化器:您可以尝试Adam、Adagrad、RMSprop等。
- 可选激活函数，如leaky ReLU, parametric ReLU, ELU, or MaxOut。
- Model ensembles
- 数据增强
- New Architectures
  - ResNets where the input from the previous layer is added to the output.
  - DenseNets where inputs into previous layers are concatenated together.
  - This blog has an in-depth overview

Have fun and happy training!

In [23]:
```
################################################################################
# TODO:                                                                        #
# Experiment with any architectures, optimizers, and hyperparameters.          #
# Achieve AT LEAST 70% accuracy on the *validation set* within 10 epochs.      #
#                                                                              #
# Note that you can use the check_accuracy function to evaluate on either      #
# the test set or the validation set, by passing either loader_test or         #
# loader_val as the second argument to check_accuracy. You should not touch    #
# the test set until you have finished your architecture and  hyperparameter   #
# tuning, and only run the test set once at the end to report a final value.   #
################################################################################
model = None
optimizer = None

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

learning_rate = 1e-2
model = nn.Sequential(
    nn.Conv2d(3, 32, (3, 3), padding=1),
    nn.ReLU(),
    nn.Conv2d(32, 32, (3, 3), padding=1),
    nn.BatchNorm2d(32),
    nn.ReLU(),
    nn.MaxPool2d((2, 2)),
    nn.Conv2d(32, 64, (3, 3), padding=1),
    nn.ReLU(),
    nn.Conv2d(64, 64, (3, 3), padding=1),
    nn.BatchNorm2d(64),
    nn.ReLU(),
    nn.MaxPool2d((2, 2)),
    nn.Conv2d(64, 128, (3, 3), padding=1),
    nn.ReLU(),
    nn.Conv2d(128, 128, (3, 3), padding=1),
    nn.ReLU(),
    nn.Conv2d(128, 128, (3, 3), padding=1),
    nn.BatchNorm2d(128),
    nn.ReLU(),
```

```
    nn.MaxPool2d((2, 2)),
    Flatten(),
    nn.Linear(128 * 4 * 4, 512),
    nn.ReLU(),
    nn.Linear(512, 128),
    nn.ReLU(),
    nn.Linear(128, 10),
)
optimizer = optim.SGD(model.parameters(), lr=learning_rate,
                      momentum=0.9, nesterov=True)

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

# You should get at least 70% accuracy
train_part34(model, optimizer, epochs=10)
```

```
Iteration 0, loss = 2.2917
Checking accuracy on validation set
Got 105 / 1000 correct (10.50)

Iteration 100, loss = 1.5540
Checking accuracy on validation set
Got 366 / 1000 correct (36.60)

Iteration 200, loss = 1.1888
Checking accuracy on validation set
Got 466 / 1000 correct (46.60)

Iteration 300, loss = 1.3031
Checking accuracy on validation set
Got 356 / 1000 correct (35.60)

Iteration 400, loss = 1.1746
Checking accuracy on validation set
Got 563 / 1000 correct (56.30)

Iteration 500, loss = 1.2037
Checking accuracy on validation set
Got 558 / 1000 correct (55.80)

Iteration 600, loss = 0.9049
Checking accuracy on validation set
Got 653 / 1000 correct (65.30)

Iteration 700, loss = 0.9419
Checking accuracy on validation set
Got 604 / 1000 correct (60.40)

Iteration 0, loss = 0.9000
Checking accuracy on validation set
Got 561 / 1000 correct (56.10)

Iteration 100, loss = 0.8508
Checking accuracy on validation set
Got 705 / 1000 correct (70.50)

Iteration 200, loss = 0.6539
Checking accuracy on validation set
Got 664 / 1000 correct (66.40)

Iteration 300, loss = 0.7645
Checking accuracy on validation set
Got 672 / 1000 correct (67.20)

Iteration 400, loss = 0.5948
Checking accuracy on validation set
Got 711 / 1000 correct (71.10)

Iteration 500, loss = 0.8494
Checking accuracy on validation set
Got 671 / 1000 correct (67.10)

Iteration 600, loss = 0.8143
Checking accuracy on validation set
Got 706 / 1000 correct (70.60)
```

```
Iteration 700, loss = 0.5965
Checking accuracy on validation set
Got 737 / 1000 correct (73.70)

Iteration 0, loss = 0.7937
Checking accuracy on validation set
Got 644 / 1000 correct (64.40)

Iteration 100, loss = 0.6434
Checking accuracy on validation set
Got 713 / 1000 correct (71.30)

Iteration 200, loss = 0.5584
Checking accuracy on validation set
Got 721 / 1000 correct (72.10)

Iteration 300, loss = 0.8461
Checking accuracy on validation set
Got 750 / 1000 correct (75.00)

Iteration 400, loss = 0.6983
Checking accuracy on validation set
Got 740 / 1000 correct (74.00)

Iteration 500, loss = 0.5902
Checking accuracy on validation set
Got 725 / 1000 correct (72.50)

Iteration 600, loss = 0.9074
Checking accuracy on validation set
Got 762 / 1000 correct (76.20)

Iteration 700, loss = 0.5244
Checking accuracy on validation set
Got 777 / 1000 correct (77.70)

Iteration 0, loss = 0.4145
Checking accuracy on validation set
Got 774 / 1000 correct (77.40)

Iteration 100, loss = 0.3829
Checking accuracy on validation set
Got 788 / 1000 correct (78.80)

Iteration 200, loss = 0.4046
Checking accuracy on validation set
Got 779 / 1000 correct (77.90)

Iteration 300, loss = 0.3979
Checking accuracy on validation set
Got 775 / 1000 correct (77.50)

Iteration 400, loss = 0.7841
Checking accuracy on validation set
Got 759 / 1000 correct (75.90)

Iteration 500, loss = 0.3026
Checking accuracy on validation set
Got 797 / 1000 correct (79.70)
```

```
Iteration 600, loss = 0.2757
Checking accuracy on validation set
Got 810 / 1000 correct (81.00)

Iteration 700, loss = 0.5067
Checking accuracy on validation set
Got 794 / 1000 correct (79.40)

Iteration 0, loss = 0.4542
Checking accuracy on validation set
Got 806 / 1000 correct (80.60)

Iteration 100, loss = 0.4629
Checking accuracy on validation set
Got 790 / 1000 correct (79.00)

Iteration 200, loss = 0.3583
Checking accuracy on validation set
Got 789 / 1000 correct (78.90)

Iteration 300, loss = 0.4277
Checking accuracy on validation set
Got 793 / 1000 correct (79.30)

Iteration 400, loss = 0.4010
Checking accuracy on validation set
Got 787 / 1000 correct (78.70)

Iteration 500, loss = 0.2802
Checking accuracy on validation set
Got 809 / 1000 correct (80.90)

Iteration 600, loss = 0.4337
Checking accuracy on validation set
Got 800 / 1000 correct (80.00)

Iteration 700, loss = 0.4437
Checking accuracy on validation set
Got 809 / 1000 correct (80.90)

Iteration 0, loss = 0.2920
Checking accuracy on validation set
Got 813 / 1000 correct (81.30)

Iteration 100, loss = 0.3117
Checking accuracy on validation set
Got 813 / 1000 correct (81.30)

Iteration 200, loss = 0.2323
Checking accuracy on validation set
Got 815 / 1000 correct (81.50)

Iteration 300, loss = 0.4903
Checking accuracy on validation set
Got 825 / 1000 correct (82.50)

Iteration 400, loss = 0.3615
Checking accuracy on validation set
Got 808 / 1000 correct (80.80)
```

```
Iteration 500, loss = 0.5228
Checking accuracy on validation set
Got 809 / 1000 correct (80.90)

Iteration 600, loss = 0.3314
Checking accuracy on validation set
Got 790 / 1000 correct (79.00)

Iteration 700, loss = 0.2211
Checking accuracy on validation set
Got 819 / 1000 correct (81.90)

Iteration 0, loss = 0.3180
Checking accuracy on validation set
Got 803 / 1000 correct (80.30)

Iteration 100, loss = 0.2034
Checking accuracy on validation set
Got 819 / 1000 correct (81.90)

Iteration 200, loss = 0.3583
Checking accuracy on validation set
Got 815 / 1000 correct (81.50)

Iteration 300, loss = 0.3126
Checking accuracy on validation set
Got 838 / 1000 correct (83.80)

Iteration 400, loss = 0.2938
Checking accuracy on validation set
Got 827 / 1000 correct (82.70)

Iteration 500, loss = 0.2327
Checking accuracy on validation set
Got 832 / 1000 correct (83.20)

Iteration 600, loss = 0.2483
Checking accuracy on validation set
Got 796 / 1000 correct (79.60)

Iteration 700, loss = 0.3271
Checking accuracy on validation set
Got 814 / 1000 correct (81.40)

Iteration 0, loss = 0.2247
Checking accuracy on validation set
Got 823 / 1000 correct (82.30)

Iteration 100, loss = 0.2572
Checking accuracy on validation set
Got 811 / 1000 correct (81.10)

Iteration 200, loss = 0.1269
Checking accuracy on validation set
Got 832 / 1000 correct (83.20)

Iteration 300, loss = 0.2600
Checking accuracy on validation set
Got 815 / 1000 correct (81.50)
```

```
Iteration 400, loss = 0.1738
Checking accuracy on validation set
Got 814 / 1000 correct (81.40)

Iteration 500, loss = 0.2120
Checking accuracy on validation set
Got 817 / 1000 correct (81.70)

Iteration 600, loss = 0.2484
Checking accuracy on validation set
Got 834 / 1000 correct (83.40)

Iteration 700, loss = 0.2826
Checking accuracy on validation set
Got 841 / 1000 correct (84.10)

Iteration 0, loss = 0.1514
Checking accuracy on validation set
Got 842 / 1000 correct (84.20)

Iteration 100, loss = 0.1523
Checking accuracy on validation set
Got 841 / 1000 correct (84.10)

Iteration 200, loss = 0.1996
Checking accuracy on validation set
Got 832 / 1000 correct (83.20)

Iteration 300, loss = 0.2029
Checking accuracy on validation set
Got 816 / 1000 correct (81.60)

Iteration 400, loss = 0.2010
Checking accuracy on validation set
Got 835 / 1000 correct (83.50)

Iteration 500, loss = 0.2042
Checking accuracy on validation set
Got 834 / 1000 correct (83.40)

Iteration 600, loss = 0.1972
Checking accuracy on validation set
Got 830 / 1000 correct (83.00)

Iteration 700, loss = 0.2771
Checking accuracy on validation set
Got 840 / 1000 correct (84.00)

Iteration 0, loss = 0.1217
Checking accuracy on validation set
Got 842 / 1000 correct (84.20)

Iteration 100, loss = 0.0951
Checking accuracy on validation set
Got 842 / 1000 correct (84.20)

Iteration 200, loss = 0.1913
Checking accuracy on validation set
Got 823 / 1000 correct (82.30)
```

```
Iteration 300, loss = 0.1360
Checking accuracy on validation set
Got 828 / 1000 correct (82.80)

Iteration 400, loss = 0.2871
Checking accuracy on validation set
Got 818 / 1000 correct (81.80)

Iteration 500, loss = 0.1463
Checking accuracy on validation set
Got 843 / 1000 correct (84.30)

Iteration 600, loss = 0.2359
Checking accuracy on validation set
Got 839 / 1000 correct (83.90)

Iteration 700, loss = 0.2483
Checking accuracy on validation set
Got 795 / 1000 correct (79.50)
```

描述你做了什么

在下面的单元格中，你应该解释你做了什么，你实现的任何附加功能，和/或你在训练和评估网络过程中制作的任何图表。

Answer:

网络架构：
卷积层，激活函数，卷积层，批次归一化，激活函数，最大池化层(卷积核大小：3 * 3，激活函数RELU)
卷积层，激活函数，卷积层，批次归一化，激活函数，最大池化层(卷积核大小：3 * 3，激活函数RELU)
卷积层，激活函数，卷积层，激活函数，卷积层，批次归一化，激活函数，最大池化层(卷积核大小：3 * 3，激活函数RELU)
全连接层，激活函数，全连接层，激活函数，全连接层(激活函数RELU)
优化：SGD随机梯度下降

Test set -- run this only once

现在我们已经得到了满意的结果，我们在测试集中测试最终的模型(应该存储在best_model中)。想一想这与您的验证集准确性如何比较。

In [ ]:
```
best_model = model
check_accuracy_part34(loader_test, best_model)
```