What's this PyTorch business?

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 (or TensorFlow, if you choose to use that notebook).

What is PyTorch?

PyTorch is a system for executing dynamic computational graphs over Tensor objects that behave similarly as numpy ndarray. It comes with a powerful automatic differentiation engine that removes the need for manual back-propagation.

Why?

- Our code will now run on GPUs! Much faster training. When using a framework like PyTorch or TensorFlow you can harness the power of the GPU for your own custom neural network architectures without having to write CUDA code directly (which is beyond the scope of this class).
- We want you to be ready to use one of these frameworks for your project so you can experiment more efficiently than if you were writing every feature you want to use by hand.
- We want you to stand on the shoulders of giants! TensorFlow and PyTorch are both excellent frameworks that will make your lives a lot easier, and now that you understand their guts, you are free to use them :)
- We want you to be exposed to the sort of deep learning code you might run into in academia or industry.

PyTorch versions

This notebook assumes that you are using **PyTorch version 1.0**. In some of the previous versions (e.g. before 0.4), Tensors had to be wrapped in Variable objects to be used in autograd; however Variables have now been deprecated. In addition 1.0 also separates a Tensor's datatype from its device, and uses numpy-style factories for constructing Tensors rather than directly invoking Tensor constructors.

How will I learn PyTorch?

Justin Johnson has made an excellent tutorial for PyTorch.

You can also find the detailed API doc here. If you have other questions that are not addressed by the API docs, the PyTorch forum is a much better place to ask than StackOverflow.

Table of Contents

This assignment has 5 parts. You will learn PyTorch on **three different levels of abstraction**, which will help you understand it better and prepare you for the final project.

1. Part I, Preparation: we will use CIFAR-10 dataset.
2. Part II, Barebones PyTorch: **Abstraction level 1**, we will work directly with the lowest-level PyTorch Tensors.
3. Part III, PyTorch Module API: **Abstraction level 2**, we will use `nn.Module` to define arbitrary neural network architecture.
4. Part IV, PyTorch Sequential API: **Abstraction level 3**, we will use `nn.Sequential` to define a linear feed-forward network very conveniently.
5. Part V, CIFAR-10 open-ended challenge: please implement your own network to get as high accuracy as possible on CIFAR-10. You can experiment with any layer, optimizer, hyperparameters or other advanced features.

Here is a table of comparison:

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

Part I. Preparation

首先，我们加载CIFAR-10数据集。第一次执行可能会花费几分钟，但是之后文件应该存储在缓存中，不需要再次花费时间。

在之前的作业中，我们必须编写自己的代码来下载CIFAR-10数据集并对其进行预处理，然后以小批量的方式对其进行遍历。PyTorch为我们提供了方便的工具来自动执行此过程。

In [4]:
```
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
```

In [5]:
```
NUM_TRAIN = 49000

# The torchvision.transforms package provides tools for preprocessing data
# and for performing data augmentation; here we set up a transform to
# preprocess the data by subtracting the mean RGB value and dividing by the
# standard deviation of each RGB value; we've hardcoded the mean and std.
transform = T.Compose([
                T.ToTensor(),
                T.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
            ])

# We set up a Dataset object for each split (train / val / test); Datasets load
# training examples one at a time, so we wrap each Dataset in a DataLoader which
# iterates through the Dataset and forms minibatches. We divide the CIFAR-10
# training set into train and val sets by passing a Sampler object to the
# DataLoader telling how it should sample from the underlying Dataset.
cifar10_train = dset.CIFAR10('./daseCV/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('./daseCV/datasets', train=True, download=True,
                           transform=transform)
loader_val = DataLoader(cifar10_val, batch_size=64,
                        sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN, 50000)))

cifar10_test = dset.CIFAR10('./daseCV/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
```

你可以他通过**设置下面的flag来使用GPU**。本次作业并非一定使用GPU。请注意，如果您的计算机并没有安装CUDA，则 `torch.cuda.is_available()` 将返回False，并且本notebook将回退至CPU模式。

全局变量 `dtype` 和 `device` 将在整个作业中控制数据类型。

In [6]:
```
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 II. Barebones PyTorch

PyTorch附带了高级API，可帮助我们方便地定义模型架构，我们将在本教程的第二部分中介绍。在本节中，我们将从barebone PyTorch元素开始，以更好地了解autograd引擎。在完成本练习之后，您将更加喜欢高级模型API。

我们将从一个简单的全连接的ReLU网络开始，该网络具有两个隐藏层并且没有biases用以对CIFAR分类。此实现使用PyTorch Tensors上的运算来计算正向传播，并使用PyTorch autograd来计算梯度。理解每一行代码很重要，因为在示例之后您将编写一个更难的版本。

当我们使用 `requires_grad = True` 创建一个PyTorch Tensor时，涉及该Tensor的操作将不仅仅计算值。他们还建立一个计算图，使我们能够轻松地在该图中反向传播，以计算某些张量相对于下游loss的梯度。具体来说，如果x是张量同时设置 `x.requires_grad == True` ，那么在反向传播之后， `x.grad` 将会是另一个张量，其保存了x对于最终loss的梯度。

PyTorch Tensors: Flatten Function

PyTorch Tensor在概念上类似于numpy数组：它是一个n维数字网格，并且像numpy一样，PyTorch提供了许多功能来方便地在Tensor上进行操作。举一个简单的例子，我们提供一个 `flatten` 功能，该函数可以改变图像数据的形状以用于全连接神经网络。

回想一下，图像数据通常存储在形状为N x C x H x W的张量中，其中：

- N 是数据的数量
- C 是通道的数量
- H 是中间特征图的高度（以像素为单位）
- W 是中间特征图的宽度（以像素为单位）

当我们进行类似2D卷积的操作时，这是表示数据的正确方法，该操作需要对中间特征之间有所了解。但是，当我们使用全连接的仿射层来处理图像时，我们希望每个数据都由单个向量表示，不需要分离数据的不同通道以及行和列。因此，我们使用"flatten"操作将每个表示形式为 `C x H x W` 的值转换为单个长向量。下面的flatten函数首先从给定的一批数据中读取N，C，H和W值，然后返回该数据的"view"。""view"类似于numpy的"reshape"方法：将x的尺寸转换为N x ??，其中??允许为任何值（在这种情况下，它将为C x H x W，但我们无需明确指定）。

In [7]:
```
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 per imag

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，我们定义一个函数 `two_layer_fc` ，该函数对一批图像数据执行两层全连接的ReLU网络的正向传播。定义正向传播后，我们通过将网络的值设置为0来检查其输出的形状来判断网络是否正确。

您无需在此处编写任何代码，但需要阅读并理解。

In [8]:
```
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 two_layer_fc_test():
    hidden_layer_size = 42
    x = torch.zeros((64, 50), dtype=dtype)  # minibatch size 64, feature dimension 50
    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 `three_layer_convnet` 函数，该函数将执行三层卷积网络的正向传播。像上面一样，我们通过将网络的值设置为0来检查其输出的形状来判断网络是否正确。网络应具有以下架构：

1. 具有 `channel_1` 滤波器的卷积层（带偏置），每个滤波器的形状均为 `KW1 x KH1`，zero-padding为2
2. 非线性ReLU
3. 具有 `channel_2` 滤波器的卷积层（带偏置），每个滤波器的形状均为 `KW2 x KH2`，zero-padding为1
4. 非线性ReLU
5. 具有偏差的全连接层，输出C类的分数。

请注意，在我们全连接层之后**没有softmax**：这是因为PyTorch的交叉熵损失会为您执行softmax，并通过捆绑该步骤可以使计算效率更高。

提示: 关于卷积: http://pytorch.org/docs/stable/nn.html#torch.nn.functional.conv2d; 注意卷积滤波器的形状!

In [9]:
```
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)*****
    out = F.conv2d(x, conv_w1, conv_b1, padding=2).relu()
    out = F.conv2d(out, conv_w2, conv_b2, padding=1).relu()
    assert out.shape[1] == conv_b2.shape[0]
    assert out.shape[2] == x.shape[-2] + 6 - conv_w1.shape[-2] - conv_w2.shape[-2] + 2
    assert out.shape[3] == x.shape[-1] + 6 - conv_w1.shape[-1] - conv_w2.shape[-1] + 2
    scores = F.linear(out.reshape(x.shape[0], -1), fc_w.transpose(1,0), fc_b)

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

在定义完上述ConvNet的正向传播之后，运行以下cell以测试您的代码。

运行此函数时，scores的形状为(64, 10)。

In [10]:
```
def three_layer_convnet_test():
    x = torch.zeros((64, 3, 32, 32), dtype=dtype)  # minibatch size 64, image size [3, 3

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

    # you must calculate the shape of the tensor after two conv layers, before the fully
    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, fc_b])
    print(scores.size())  # you should see [64, 10]
three_layer_convnet_test()
```

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

Barebones PyTorch: Initialization

让我们编写一些实用的方法来初始化模型的权重矩阵。

- `random_weight(shape)` 使用Kaiming归一化方法初始化权重tensor。
- `zero_weight(shape)` 用全零初始化权重tensor。主要用于实例化偏差。

`random_weight` 函数使用Kaiming归一化，具体描述如下：

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

In [11]:
```python
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, kW]
    # 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[11]:
```
tensor([[-0.0089,  0.4397,  0.7645,  0.0108,  1.0018],
        [-0.3989, -0.4889,  0.8773, -0.4337, -0.0585],
        [-1.0084,  0.4685,  0.5322, -0.1728, -0.3192]], device='cuda:0',
       requires_grad=True)
```

Barebones PyTorch: Check Accuracy

在训练模型时，我们将使用以下函数在训练或验证集上检查模型的准确性。

在检查准确性时，我们不需要计算任何梯度。当我们计算 scores 时，我们不需要PyTorch为我们构建计算图。为了防止构建图，我们将使用 `torch.no_grad()` 。

In [12]:
```python
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 * acc))
```

BareBones PyTorch: Training Loop

现在，我们可以使用一个基本的循环来训练我们的网络。我们将使用没有momentum的随机梯度下降训练模型，并使用 `torch.functional.cross_entropy` 来计算损失；您可以在此处阅读有关内容。

将初始化参数列表（在我们的示例中为 `[w1, w2]` ）和学习率作为神经网络函数训练的输入。

In [13]:
```
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

现在我们准备好运行训练循环。我们需要为全连接的权重 `w1` 和 `w2` 显式的分配tensors。

CIFAR的每个小批都有64个数据，因此tensor形状为 `[64, 3, 32, 32]` 。

展平后， `x` 形状应为 `[64, 3 * 32 * 32]` 。这将是 `w1` 的第一维尺寸。 `w1` 的第二维是隐藏层的大小，这同时也是 `w2` 的第一维。

最后，网络的输出是一个10维向量，代表10类的概率分布。

您无需调整任何超参数，但经过一个epoch的训练后，您应该会看到40％以上的准确度。

In [14]:
```
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 = 3.1854
Checking accuracy on the val set
Got 118 / 1000 correct (11.80%)

Iteration 100, loss = 2.0891
Checking accuracy on the val set
Got 314 / 1000 correct (31.40%)

Iteration 200, loss = 2.1777
Checking accuracy on the val set
Got 332 / 1000 correct (33.20%)

Iteration 300, loss = 1.8171
Checking accuracy on the val set
Got 375 / 1000 correct (37.50%)

Iteration 400, loss = 1.5074
Checking accuracy on the val set
Got 416 / 1000 correct (41.60%)

Iteration 500, loss = 1.7960
Checking accuracy on the val set
Got 422 / 1000 correct (42.20%)

Iteration 600, loss = 1.7488
Checking accuracy on the val set
Got 392 / 1000 correct (39.20%)

Iteration 700, loss = 1.3739
Checking accuracy on the val set
Got 421 / 1000 correct (42.10%)
```

BareBones PyTorch: Training a ConvNet

在下面，您应该使用上面定义的功能在CIFAR上训练三层卷积网络。网络应具有以下架构：

1. 带32 5x5滤波器的卷积层（带偏置），zero-padding为2
2. ReLU
3. 带16 3x3滤波器的卷积层（带偏置），zero-padding为1
4. ReLU
5. 全连接层（带偏置），可计算10个类别的scores

您应该使用上面定义的 `random_weight` 函数来初始化权重矩阵，并且使用上面的 `zero_weight` 函数来初始化偏差向量。

您无需调整任何超参数，但经过一个epoch的训练后，您应该会看到42%以上的准确度。

In [15]:
```
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((channel_2 * (32 - 3 - 5 + 2 + 6) * (32 - 3 - 5 + 2 + 6), 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 = 4.3981
Checking accuracy on the val set
Got 102 / 1000 correct (10.20%)

Iteration 100, loss = 1.7882
Checking accuracy on the val set
Got 339 / 1000 correct (33.90%)

Iteration 200, loss = 1.6487
Checking accuracy on the val set
Got 397 / 1000 correct (39.70%)

Iteration 300, loss = 1.6836
Checking accuracy on the val set
Got 413 / 1000 correct (41.30%)

Iteration 400, loss = 1.7750
Checking accuracy on the val set
Got 420 / 1000 correct (42.00%)

Iteration 500, loss = 1.6811
Checking accuracy on the val set
Got 462 / 1000 correct (46.20%)

Iteration 600, loss = 1.7349
Checking accuracy on the val set
Got 452 / 1000 correct (45.20%)
```

```
Iteration 700, loss = 1.3841
Checking accuracy on the val set
Got 472 / 1000 correct (47.20%)
```

Part III. PyTorch Module API

Barebone PyTorch要求我们手动跟踪所有参数的tensors。这对于具有几个tensors的小型网络倒是没什么问题，但是在较大的网络中跟踪数十个或数百个tensors将非常不方便且容易出错。

PyTorch为您提供 `nn.Module` API，以定义任意网络架构，同时为您跟踪每个可学习的参数。在Part II中，我们自己实现了SGD。PyTorch还提供了 `torch.optim` 软件包，该软件包实现了所有常见的优化器，例如RMSProp，Adagrad和Adam。它甚至支持近似二阶方法，例如L-BFGS！您可以参考doc 了解每个优化器的详细信息。

要使用Module API，请按照以下步骤操作：

1. 定义 `nn.Module` 的子类，并给您的类起一个直观的名称，例如 `TwoLayerFC` 。
2. 在构造函数 `__init__()` 中，将所有的层定义为类属性。像 `nn.Linear` 和 `nn.Conv2d` 这样的层对象本身就是 `nn.Module` 子类，并且包含可学习的参数，因此您不必自己实例化原始tensors。 `nn.Module` 将为您追踪这些内部参数。请参阅doc，以了解有关内置层的更多信息。**警告**：别忘了先调用 `super（）.__ init __（）` ！
3. 在 `forward()` 方法中，定义网络的*connectivity*。你应该使用 `__init__` 中定义的属性作为函数调用，把tensor作为输入，把“变换后的”tensor作为输出。。*不要在* `forward（）` 中创建任何带有可学习参数的新层！所有这些都必须在 `__init__` 中预先声明。

定义Module子类后，可以将其实例化为对象，然后像part II中的NN forward函数一样调用它。

Module API: Two-Layer Network

这是两层全连接网络的具体示例：

In [16]:
```
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 dimensi
    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

在完成全连接层之后接着完成你的三层的ConvNet。网络架构应与 Part II 相同：

1. 具有 `channel_1` 滤波器的卷积层（带偏置），每个滤波器的形状均为5x5，zero-padding为2
2. ReLU
3. 具有 `channel_2` 滤波器的卷积层（带偏置），每个滤波器的形状均为3x3，zero-padding为1
4. ReLU
5. 全连接层，输出 `num_classes` 类。

您应该使用Kaiming初始化方法初始化模型的权重矩阵。

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

在实现三层ConvNet之后， `test_ThreeLayerConvNet` 函数将运行您的代码；它应该输出形状为 `（64，10）` 的scores。

In [17]:
```
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.conv_1 = nn.Conv2d(in_channel, channel_1, kernel_size=(5,5), padding=2)
        nn.init.kaiming_normal_(self.conv_1.weight)
        self.relu_1 = nn.ReLU()
        self.conv_2 = nn.Conv2d(channel_1, channel_2, kernel_size=(3,3), padding=1)
        nn.init.kaiming_normal_(self.conv_2.weight)
        self.relu_2 = nn.ReLU()
        self.flatten = nn.Flatten()
        self.fc_lin = nn.Linear(channel_2*32*32, 10)

        # *****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.conv_1(x)
        x = self.relu_1(x)
        x = self.conv_2(x)
        x = self.relu_2(x)
        x = self.flatten(x)
        scores = self.fc_lin(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 size [3, 3
    model = ThreeLayerConvNet(in_channel=3, channel_1=12, channel_2=8, num_classes=10)
    scores = model(x)
    print(scores.size())  # you should see [64, 10]
test_ThreeLayerConvNet()
```

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

Module API: Check Accuracy

给定验证或测试集，我们可以检查神经网络的分类准确性。

此版本与part II中的版本略有不同。您不再需要手动传递参数。

In [18]:
```
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 * acc))
```

Module API: Training Loop

我们还使用了稍微不同的训练循环。我们不用自己更新权重的值，而是使用来自`torch.optim`包的Optimizer对象，该对象抽象了优化算法的概念，并实现了通常用于优化神经网络的大多数算法。

In [19]:
```
def train_part34(model, optimizer, epochs=1):
    """
    Train a model on CIFAR-10 using the PyTorch Module API.

    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 for

    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 optimizer
            # 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:
                model.eval()
                print('Iteration %d, loss = %.4f' % (t, loss.item()))
                check_accuracy_part34(loader_val, model)
                print()
```

Module API: Train a Two-Layer Network

现在我们准备好运行训练循环。与 part II相比，我们不再显式分配参数tensors。

只需将输入大小，隐藏层大小和类数（即输出大小）传递给 `TwoLayerFC` 的构造函数即可。

您还需要定义一个优化器来追踪 `TwoLayerFC` 内部的所有可学习参数。

您无需调整任何超参数，经过一个epoch的训练后，您应该会看到模型精度超过40%。

In [20]:
```
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 = 3.0793
Checking accuracy on validation set
Got 151 / 1000 correct (15.10)

Iteration 100, loss = 2.4668
Checking accuracy on validation set
Got 306 / 1000 correct (30.60)

Iteration 200, loss = 1.9122
Checking accuracy on validation set
Got 353 / 1000 correct (35.30)

Iteration 300, loss = 2.0285
Checking accuracy on validation set
Got 390 / 1000 correct (39.00)

Iteration 400, loss = 2.1427
Checking accuracy on validation set
Got 427 / 1000 correct (42.70)

Iteration 500, loss = 1.6551
Checking accuracy on validation set
Got 435 / 1000 correct (43.50)

Iteration 600, loss = 1.5800
```

```
Checking accuracy on validation set
Got 439 / 1000 correct (43.90)

Iteration 700, loss = 1.5001
Checking accuracy on validation set
Got 438 / 1000 correct (43.80)
```

Module API: Train a Three-Layer ConvNet

现在，您应该使用Module API在CIFAR上训练三层ConvNet。这看起来与训练两层网络非常相似！您无需调整任何超参数，但经过一个epoch的训练后，您应该达到45%以上水平的精度。

您应该使用没有动量的随机梯度下降法训练模型。

In [21]:
```
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(in_channel=3, channel_1=12, channel_2=8, num_classes=10)
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

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

train_part34(model, optimizer)
```

```
Iteration 0, loss = 2.3678
Checking accuracy on validation set
Got 141 / 1000 correct (14.10)

Iteration 100, loss = 2.0301
Checking accuracy on validation set
Got 323 / 1000 correct (32.30)

Iteration 200, loss = 1.8192
Checking accuracy on validation set
Got 376 / 1000 correct (37.60)

Iteration 300, loss = 1.7510
Checking accuracy on validation set
Got 392 / 1000 correct (39.20)

Iteration 400, loss = 1.8207
Checking accuracy on validation set
Got 404 / 1000 correct (40.40)

Iteration 500, loss = 1.6795
Checking accuracy on validation set
Got 428 / 1000 correct (42.80)

Iteration 600, loss = 1.7218
Checking accuracy on validation set
Got 442 / 1000 correct (44.20)
```

```
Iteration 700, loss = 1.5571
Checking accuracy on validation set
Got 458 / 1000 correct (45.80)
```

Part IV. PyTorch Sequential API

Part III介绍了PyTorch Module API，该API允许您定义任意可学习的层及其连接。

对于简单的模型，你需要经历3个步骤：子类 `nn.Module`，在 `__init__` 中定义各层，并在 `forward（）` 中逐个调用每一层。。那有没有更方便的方法？

幸运的是，PyTorch提供了一个名为 `nn.Sequential` 的容器模块，该模块将上述步骤合并为一个。它不如 `nn.Module` 灵活，因为您不能指定更复杂的拓扑结构，但是对于许多用例来说已经足够了。

Sequential API: Two-Layer Network

让我们看看如何用 `nn.Sequential` 重写之前的两层全连接网络示例，并使用上面定义的训练循环对其进行训练。

同样，您无需在此处调整任何超参数，但是经过一个epoch的训练后，您应该达到40%以上的准确性。

In [22]:
```
# 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 = 2.3432
Checking accuracy on validation set
Got 159 / 1000 correct (15.90)

Iteration 100, loss = 2.0467
Checking accuracy on validation set
Got 392 / 1000 correct (39.20)

Iteration 200, loss = 1.8387
Checking accuracy on validation set
Got 396 / 1000 correct (39.60)

Iteration 300, loss = 1.8488
Checking accuracy on validation set
Got 384 / 1000 correct (38.40)
```

```
Iteration 400, loss = 1.9878
Checking accuracy on validation set
Got 437 / 1000 correct (43.70)

Iteration 500, loss = 1.5803
Checking accuracy on validation set
Got 410 / 1000 correct (41.00)

Iteration 600, loss = 1.7201
Checking accuracy on validation set
Got 440 / 1000 correct (44.00)

Iteration 700, loss = 1.5870
Checking accuracy on validation set
Got 436 / 1000 correct (43.60)
```

Sequential API: Three-Layer ConvNet

在这里，您应该使用 `nn.Sequential` 来定义和训练三层ConvNet，其结构与我们在第三部分中使用的结构相同：

1. 带32 5x5滤波器的卷积层（带偏置），zero-padding为2
2. ReLU
3. 带16 3x3滤波器的卷积层（带偏置），zero-padding为1
4. ReLU
5. 全连接层（带偏置），可计算10个类别的分数

您应该使用上面定义的 `random_weight` 函数来初始化权重矩阵，并应该使用 `zero_weight` 函数来初始化偏差向量。

您应该使用Nesterov动量0.9的随机梯度下降来优化模型。

同样，您不需要调整任何超参数，但是经过一个epoch的训练，您应该会看到55%以上的准确性。

In [23]:
```python
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)*****

from collections import OrderedDict

model = nn.Sequential(OrderedDict([
    ('conv1', nn.Conv2d(3, channel_1, kernel_size=(5,5), padding=2, bias=True)),
    ('relu1', nn.ReLU()),
    ('conv2', nn.Conv2d(channel_1, channel_2, kernel_size=(3,3), padding=1)),
    ('relu2', nn.ReLU()),
    ('flatten', nn.Flatten()),
    ('fc_lin', nn.Linear(channel_2*32*32, 10))
])).to(device)

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 = 2.3169
Checking accuracy on validation set
Got 98 / 1000 correct (9.80)

Iteration 100, loss = 1.4671
Checking accuracy on validation set
Got 450 / 1000 correct (45.00)

Iteration 200, loss = 1.4937
Checking accuracy on validation set
Got 469 / 1000 correct (46.90)

Iteration 300, loss = 1.3541
Checking accuracy on validation set
Got 527 / 1000 correct (52.70)

Iteration 400, loss = 1.3351
Checking accuracy on validation set
Got 537 / 1000 correct (53.70)

Iteration 500, loss = 1.1839
Checking accuracy on validation set
Got 571 / 1000 correct (57.10)

Iteration 600, loss = 1.2437
Checking accuracy on validation set
Got 543 / 1000 correct (54.30)

Iteration 700, loss = 1.1172
Checking accuracy on validation set
Got 575 / 1000 correct (57.50)
```

Part V. CIFAR-10 open-ended challenge

在本节中，您可以尝试在CIFAR-10上使用任何ConvNet架构。

现在，您的工作就是尝试使用不同的架构、超参数、损失函数和优化器，以训练出在CIFAR-10上运行10个epoch内的使得 **验证集** 上 **至少达到70%** 精度的模型。你可以使用上面的check_accuracy和train函数。也可以使用 `nn.Module` 或 `nn.Sequential` API。

描述您在本notebook末尾所做的事情。

这是每个组件的官方API文档。需要注意的是：在PyTorch中"spatial batch norm"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

Things you might try:

- **过滤器大小**：上面我们使用了5x5的大小；较小的过滤器会更有效吗？
- **过滤器数量**：上面我们使用了32个过滤器。多点更好还是少一点更好？
- **Pooling vs Strided Convolution**: 您使用 max pooling还是stride convolutions？
- **Batch normalization**: 尝试在卷积层之后添加空间批处理归一化，并在affine layers之后添加批归一化。您的网络训练速度会更快吗？
- **网络架构**: 上面的网络具有两层可训练的参数。深度网络可以做得更好吗？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**: 不将图片转变为向量而是有多个仿射层并执行卷积直到图像变小（大约7x7），然后执行平均池化操作以获取1x1图像图片(1, 1 , Filter#)，然后将其变换为为(Filter#)向量。在Google's Inception Network中使用了它（其结构请参见表1）。
- **正则化**：添加l2权重正则化，或者使用Dropout。

Tips for training

对于尝试的每种网络结构，您都应该调整学习速率和其他超参数。进行此操作时，需要牢记一些重要事项：

- 如果参数运行良好，则应在几百次迭代中看到改进
- 请记住，从粗略到精细的超参数调整方法：首先测试大范围的超参数，只需要几个训练迭代就可以找到有效的参数组合。
- 找到一些似乎有效的参数后，请在这些参数周围进行更精细的搜索。您可能需要训练更多的epochs。
- 您应该使用验证集进行超参数搜索，并保存测试集，以便根据验证集选择的最佳参数评估网络结构。

Going above and beyond

If you are feeling adventurous there are many other features you can implement to try and improve your performance. You are **not required** to implement any of these, but don't miss the fun if you have time! 如果您喜欢冒险，可以使用许多其他功能来尝试并提高性能。下面**不是不须**完成的，但如果有时间，请不要错过！

- 替代的优化器：您可以尝试Adam，Adagrad，RMSprop等。
- 替代激活函数，例如leaky ReLU，parametric ReLU，ELU或MaxOut。
- 集成学习
- 数据增强
- 新架构
  - 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 [34]:
```
################################################################################
# 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)*****

class BigConv(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        # output * 32 16 16
        self.pack_1 = nn.Sequential(
            nn.Conv2d(3, 32, (3, 3), padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 32, (3, 3), padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(32),
            nn.MaxPool2d((2, 2), stride=2),
            nn.Dropout(p=0.25)
        )
        # output * 64 8 8
        self.pack_2 = nn.Sequential(
            nn.Conv2d(32, 64, (3, 3), padding=1),
            nn.ReLU(),
            nn.Conv2d(64, 64, (3, 3), padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(64),
            nn.MaxPool2d((2, 2), stride=2),
            nn.Dropout(p=0.25)
        )
        # output * 128 4 4
        self.pack_3 = nn.Sequential(
            nn.Conv2d(64, 128, (3, 3), padding=1),
            nn.ReLU(),
            nn.Conv2d(128, 128, (3, 3), padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(128),
            nn.MaxPool2d((2, 2), stride=2),
            nn.Dropout(p=0.25)
        )
        # output * 10
        self.pack_4 = nn.Sequential(
            nn.Flatten(start_dim=-3, end_dim=-1),
            nn.Linear(128 * 4 * 4, 128),
            nn.Dropout(p=0.25),
            nn.Linear(128, 10)
        )

    def forward(self, x):
        x = self.pack_1(x)
        x = self.pack_2(x)
        x = self.pack_3(x)
        x = self.pack_4(x)
        return x

model = BigConv()
optimizer = optim.Adam(model.parameters())

# *****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=20)
```

```
Iteration 0, loss = 2.6360
Checking accuracy on validation set
Got 133 / 1000 correct (13.30)

Iteration 100, loss = 1.6598
Checking accuracy on validation set
Got 405 / 1000 correct (40.50)

Iteration 200, loss = 1.4868
Checking accuracy on validation set
Got 472 / 1000 correct (47.20)

Iteration 300, loss = 1.2604
Checking accuracy on validation set
Got 511 / 1000 correct (51.10)

Iteration 400, loss = 1.4456
Checking accuracy on validation set
Got 570 / 1000 correct (57.00)

Iteration 500, loss = 1.1915
Checking accuracy on validation set
Got 606 / 1000 correct (60.60)

Iteration 600, loss = 1.0796
Checking accuracy on validation set
Got 652 / 1000 correct (65.20)

Iteration 700, loss = 1.0937
Checking accuracy on validation set
Got 631 / 1000 correct (63.10)

Iteration 0, loss = 1.1758
Checking accuracy on validation set
Got 649 / 1000 correct (64.90)

Iteration 100, loss = 1.0251
Checking accuracy on validation set
Got 661 / 1000 correct (66.10)

Iteration 200, loss = 0.9425
Checking accuracy on validation set
Got 695 / 1000 correct (69.50)

Iteration 300, loss = 1.0042
Checking accuracy on validation set
Got 683 / 1000 correct (68.30)

Iteration 400, loss = 1.0625
Checking accuracy on validation set
Got 714 / 1000 correct (71.40)

Iteration 500, loss = 0.6212
Checking accuracy on validation set
Got 701 / 1000 correct (70.10)

Iteration 600, loss = 0.9563
Checking accuracy on validation set
Got 701 / 1000 correct (70.10)

Iteration 700, loss = 0.6887
Checking accuracy on validation set
Got 726 / 1000 correct (72.60)

Iteration 0, loss = 0.7948
Checking accuracy on validation set
```

```
Got 730 / 1000 correct (73.00)

Iteration 100, loss = 0.9468
Checking accuracy on validation set
Got 757 / 1000 correct (75.70)

Iteration 200, loss = 1.0138
Checking accuracy on validation set
Got 745 / 1000 correct (74.50)

Iteration 300, loss = 0.8649
Checking accuracy on validation set
Got 755 / 1000 correct (75.50)

Iteration 400, loss = 0.7637
Checking accuracy on validation set
Got 764 / 1000 correct (76.40)

Iteration 500, loss = 0.8929
Checking accuracy on validation set
Got 750 / 1000 correct (75.00)

Iteration 600, loss = 0.9235
Checking accuracy on validation set
Got 760 / 1000 correct (76.00)

Iteration 700, loss = 0.7515
Checking accuracy on validation set
Got 785 / 1000 correct (78.50)

Iteration 0, loss = 0.7784
Checking accuracy on validation set
Got 771 / 1000 correct (77.10)

Iteration 100, loss = 0.9068
Checking accuracy on validation set
Got 782 / 1000 correct (78.20)

Iteration 200, loss = 0.7217
Checking accuracy on validation set
Got 779 / 1000 correct (77.90)

Iteration 300, loss = 0.6748
Checking accuracy on validation set
Got 774 / 1000 correct (77.40)

Iteration 400, loss = 0.8174
Checking accuracy on validation set
Got 771 / 1000 correct (77.10)

Iteration 500, loss = 0.7913
Checking accuracy on validation set
Got 780 / 1000 correct (78.00)

Iteration 600, loss = 0.7568
Checking accuracy on validation set
Got 783 / 1000 correct (78.30)

Iteration 700, loss = 0.4711
Checking accuracy on validation set
Got 799 / 1000 correct (79.90)

Iteration 0, loss = 0.6546
Checking accuracy on validation set
Got 765 / 1000 correct (76.50)
```

```
Iteration 100, loss = 0.5852
Checking accuracy on validation set
Got 808 / 1000 correct (80.80)

Iteration 200, loss = 0.5380
Checking accuracy on validation set
Got 801 / 1000 correct (80.10)

Iteration 300, loss = 0.5153
Checking accuracy on validation set
Got 793 / 1000 correct (79.30)

Iteration 400, loss = 0.8168
Checking accuracy on validation set
Got 780 / 1000 correct (78.00)

Iteration 500, loss = 0.6439
Checking accuracy on validation set
Got 801 / 1000 correct (80.10)

Iteration 600, loss = 0.5506
Checking accuracy on validation set
Got 801 / 1000 correct (80.10)

Iteration 700, loss = 0.7322
Checking accuracy on validation set
Got 800 / 1000 correct (80.00)

Iteration 0, loss = 0.6914
Checking accuracy on validation set
Got 786 / 1000 correct (78.60)

Iteration 100, loss = 0.6665
Checking accuracy on validation set
Got 798 / 1000 correct (79.80)

Iteration 200, loss = 0.7187
Checking accuracy on validation set
Got 805 / 1000 correct (80.50)

Iteration 300, loss = 0.4184
Checking accuracy on validation set
Got 817 / 1000 correct (81.70)

Iteration 400, loss = 0.4432
Checking accuracy on validation set
Got 804 / 1000 correct (80.40)

Iteration 500, loss = 0.6268
Checking accuracy on validation set
Got 829 / 1000 correct (82.90)

Iteration 600, loss = 0.5040
Checking accuracy on validation set
Got 794 / 1000 correct (79.40)

Iteration 700, loss = 0.4792
Checking accuracy on validation set
Got 802 / 1000 correct (80.20)

Iteration 0, loss = 0.5366
Checking accuracy on validation set
Got 809 / 1000 correct (80.90)

Iteration 100, loss = 0.6428
Checking accuracy on validation set
```

```
Got 824 / 1000 correct (82.40)

Iteration 200, loss = 0.3999
Checking accuracy on validation set
Got 817 / 1000 correct (81.70)

Iteration 300, loss = 0.5813
Checking accuracy on validation set
Got 819 / 1000 correct (81.90)

Iteration 400, loss = 0.5011
Checking accuracy on validation set
Got 809 / 1000 correct (80.90)

Iteration 500, loss = 0.3607
Checking accuracy on validation set
Got 806 / 1000 correct (80.60)

Iteration 600, loss = 0.4229
Checking accuracy on validation set
Got 812 / 1000 correct (81.20)

Iteration 700, loss = 0.4387
Checking accuracy on validation set
Got 791 / 1000 correct (79.10)

Iteration 0, loss = 0.4604
Checking accuracy on validation set
Got 825 / 1000 correct (82.50)

Iteration 100, loss = 0.2376
Checking accuracy on validation set
Got 796 / 1000 correct (79.60)

Iteration 200, loss = 0.4001
Checking accuracy on validation set
Got 823 / 1000 correct (82.30)

Iteration 300, loss = 0.3688
Checking accuracy on validation set
Got 807 / 1000 correct (80.70)

Iteration 400, loss = 0.3126
Checking accuracy on validation set
Got 820 / 1000 correct (82.00)

Iteration 500, loss = 0.4591
Checking accuracy on validation set
Got 817 / 1000 correct (81.70)

Iteration 600, loss = 0.4308
Checking accuracy on validation set
Got 826 / 1000 correct (82.60)

Iteration 700, loss = 0.6394
Checking accuracy on validation set
Got 828 / 1000 correct (82.80)

Iteration 0, loss = 0.4829
Checking accuracy on validation set
Got 809 / 1000 correct (80.90)

Iteration 100, loss = 0.4804
Checking accuracy on validation set
Got 828 / 1000 correct (82.80)
```

```
Iteration 200, loss = 0.3331
Checking accuracy on validation set
Got 823 / 1000 correct (82.30)

Iteration 300, loss = 0.4586
Checking accuracy on validation set
Got 810 / 1000 correct (81.00)

Iteration 400, loss = 0.2515
Checking accuracy on validation set
Got 819 / 1000 correct (81.90)

Iteration 500, loss = 0.2776
Checking accuracy on validation set
Got 829 / 1000 correct (82.90)

Iteration 600, loss = 0.5459
Checking accuracy on validation set
Got 831 / 1000 correct (83.10)

Iteration 700, loss = 0.4804
Checking accuracy on validation set
Got 806 / 1000 correct (80.60)

Iteration 0, loss = 0.4301
Checking accuracy on validation set
Got 823 / 1000 correct (82.30)

Iteration 100, loss = 0.4929
Checking accuracy on validation set
Got 838 / 1000 correct (83.80)

Iteration 200, loss = 0.2703
Checking accuracy on validation set
Got 826 / 1000 correct (82.60)

Iteration 300, loss = 0.2653
Checking accuracy on validation set
Got 819 / 1000 correct (81.90)

Iteration 400, loss = 0.3907
Checking accuracy on validation set
Got 831 / 1000 correct (83.10)

Iteration 500, loss = 0.6052
Checking accuracy on validation set
Got 833 / 1000 correct (83.30)

Iteration 600, loss = 0.4178
Checking accuracy on validation set
Got 840 / 1000 correct (84.00)

Iteration 700, loss = 0.6958
Checking accuracy on validation set
Got 831 / 1000 correct (83.10)

Iteration 0, loss = 0.4304
Checking accuracy on validation set
Got 830 / 1000 correct (83.00)

Iteration 100, loss = 0.3932
Checking accuracy on validation set
Got 825 / 1000 correct (82.50)

Iteration 200, loss = 0.5808
Checking accuracy on validation set
```

```
Got 836 / 1000 correct (83.60)

Iteration 300, loss = 0.3326
Checking accuracy on validation set
Got 848 / 1000 correct (84.80)

Iteration 400, loss = 0.3799
Checking accuracy on validation set
Got 826 / 1000 correct (82.60)

Iteration 500, loss = 0.2710
Checking accuracy on validation set
Got 840 / 1000 correct (84.00)

Iteration 600, loss = 0.3954
Checking accuracy on validation set
Got 824 / 1000 correct (82.40)

Iteration 700, loss = 0.4141
Checking accuracy on validation set
Got 825 / 1000 correct (82.50)

Iteration 0, loss = 0.4708
Checking accuracy on validation set
Got 825 / 1000 correct (82.50)

Iteration 100, loss = 0.4948
Checking accuracy on validation set
Got 826 / 1000 correct (82.60)

Iteration 200, loss = 0.4149
Checking accuracy on validation set
Got 843 / 1000 correct (84.30)

Iteration 300, loss = 0.3387
Checking accuracy on validation set
Got 838 / 1000 correct (83.80)

Iteration 400, loss = 0.3840
Checking accuracy on validation set
Got 832 / 1000 correct (83.20)

Iteration 500, loss = 0.3702
Checking accuracy on validation set
Got 832 / 1000 correct (83.20)

Iteration 600, loss = 0.1974
Checking accuracy on validation set
Got 854 / 1000 correct (85.40)

Iteration 700, loss = 0.2064
Checking accuracy on validation set
Got 838 / 1000 correct (83.80)

Iteration 0, loss = 0.2209
Checking accuracy on validation set
Got 832 / 1000 correct (83.20)

Iteration 100, loss = 0.3390
Checking accuracy on validation set
Got 847 / 1000 correct (84.70)

Iteration 200, loss = 0.2789
Checking accuracy on validation set
Got 841 / 1000 correct (84.10)
```

```
Iteration 300, loss = 0.4591
Checking accuracy on validation set
Got 843 / 1000 correct (84.30)

Iteration 400, loss = 0.3995
Checking accuracy on validation set
Got 837 / 1000 correct (83.70)

Iteration 500, loss = 0.3104
Checking accuracy on validation set
Got 831 / 1000 correct (83.10)

Iteration 600, loss = 0.3802
Checking accuracy on validation set
Got 832 / 1000 correct (83.20)

Iteration 700, loss = 0.3008
Checking accuracy on validation set
Got 843 / 1000 correct (84.30)

Iteration 0, loss = 0.1627
Checking accuracy on validation set
Got 836 / 1000 correct (83.60)

Iteration 100, loss = 0.3271
Checking accuracy on validation set
Got 837 / 1000 correct (83.70)

Iteration 200, loss = 0.2992
Checking accuracy on validation set
Got 829 / 1000 correct (82.90)

Iteration 300, loss = 0.2249
Checking accuracy on validation set
Got 829 / 1000 correct (82.90)

Iteration 400, loss = 0.2743
Checking accuracy on validation set
Got 844 / 1000 correct (84.40)

Iteration 500, loss = 0.3601
Checking accuracy on validation set
Got 853 / 1000 correct (85.30)

Iteration 600, loss = 0.1835
Checking accuracy on validation set
Got 856 / 1000 correct (85.60)

Iteration 700, loss = 0.3157
Checking accuracy on validation set
Got 829 / 1000 correct (82.90)

Iteration 0, loss = 0.4585
Checking accuracy on validation set
Got 845 / 1000 correct (84.50)

Iteration 100, loss = 0.2256
Checking accuracy on validation set
Got 845 / 1000 correct (84.50)

Iteration 200, loss = 0.3731
Checking accuracy on validation set
Got 832 / 1000 correct (83.20)

Iteration 300, loss = 0.3724
Checking accuracy on validation set
```

```
Got 823 / 1000 correct (82.30)

Iteration 400, loss = 0.3933
Checking accuracy on validation set
Got 844 / 1000 correct (84.40)

Iteration 500, loss = 0.3081
Checking accuracy on validation set
Got 843 / 1000 correct (84.30)

Iteration 600, loss = 0.3135
Checking accuracy on validation set
Got 835 / 1000 correct (83.50)

Iteration 700, loss = 0.1834
Checking accuracy on validation set
Got 839 / 1000 correct (83.90)

Iteration 0, loss = 0.3219
Checking accuracy on validation set
Got 841 / 1000 correct (84.10)

Iteration 100, loss = 0.3474
Checking accuracy on validation set
Got 839 / 1000 correct (83.90)

Iteration 200, loss = 0.4103
Checking accuracy on validation set
Got 833 / 1000 correct (83.30)

Iteration 300, loss = 0.1337
Checking accuracy on validation set
Got 833 / 1000 correct (83.30)

Iteration 400, loss = 0.3261
Checking accuracy on validation set
Got 834 / 1000 correct (83.40)

Iteration 500, loss = 0.1213
Checking accuracy on validation set
Got 839 / 1000 correct (83.90)

Iteration 600, loss = 0.3802
Checking accuracy on validation set
Got 828 / 1000 correct (82.80)

Iteration 700, loss = 0.4460
Checking accuracy on validation set
Got 859 / 1000 correct (85.90)

Iteration 0, loss = 0.4001
Checking accuracy on validation set
Got 839 / 1000 correct (83.90)

Iteration 100, loss = 0.3044
Checking accuracy on validation set
Got 852 / 1000 correct (85.20)

Iteration 200, loss = 0.2701
Checking accuracy on validation set
Got 845 / 1000 correct (84.50)

Iteration 300, loss = 0.1810
Checking accuracy on validation set
Got 839 / 1000 correct (83.90)
```

```
Iteration 400, loss = 0.3284
Checking accuracy on validation set
Got 841 / 1000 correct (84.10)

Iteration 500, loss = 0.2562
Checking accuracy on validation set
Got 860 / 1000 correct (86.00)

Iteration 600, loss = 0.3704
Checking accuracy on validation set
Got 848 / 1000 correct (84.80)

Iteration 700, loss = 0.4615
Checking accuracy on validation set
Got 848 / 1000 correct (84.80)

Iteration 0, loss = 0.1839
Checking accuracy on validation set
Got 838 / 1000 correct (83.80)

Iteration 100, loss = 0.1816
Checking accuracy on validation set
Got 849 / 1000 correct (84.90)

Iteration 200, loss = 0.2083
Checking accuracy on validation set
Got 841 / 1000 correct (84.10)

Iteration 300, loss = 0.2401
Checking accuracy on validation set
Got 841 / 1000 correct (84.10)

Iteration 400, loss = 0.4460
Checking accuracy on validation set
Got 842 / 1000 correct (84.20)

Iteration 500, loss = 0.2891
Checking accuracy on validation set
Got 840 / 1000 correct (84.00)

Iteration 600, loss = 0.4302
Checking accuracy on validation set
Got 845 / 1000 correct (84.50)

Iteration 700, loss = 0.1762
Checking accuracy on validation set
Got 849 / 1000 correct (84.90)

Iteration 0, loss = 0.1484
Checking accuracy on validation set
Got 843 / 1000 correct (84.30)

Iteration 100, loss = 0.2872
Checking accuracy on validation set
Got 841 / 1000 correct (84.10)

Iteration 200, loss = 0.2141
Checking accuracy on validation set
Got 851 / 1000 correct (85.10)

Iteration 300, loss = 0.5723
Checking accuracy on validation set
Got 839 / 1000 correct (83.90)

Iteration 400, loss = 0.1724
Checking accuracy on validation set
```

```
Got 838 / 1000 correct (83.80)

Iteration 500, loss = 0.3240
Checking accuracy on validation set
Got 860 / 1000 correct (86.00)

Iteration 600, loss = 0.4539
Checking accuracy on validation set
Got 851 / 1000 correct (85.10)

Iteration 700, loss = 0.2328
Checking accuracy on validation set
Got 842 / 1000 correct (84.20)

Iteration 0, loss = 0.2952
Checking accuracy on validation set
Got 827 / 1000 correct (82.70)

Iteration 100, loss = 0.2198
Checking accuracy on validation set
Got 839 / 1000 correct (83.90)

Iteration 200, loss = 0.2678
Checking accuracy on validation set
Got 857 / 1000 correct (85.70)

Iteration 300, loss = 0.1838
Checking accuracy on validation set
Got 829 / 1000 correct (82.90)

Iteration 400, loss = 0.1964
Checking accuracy on validation set
Got 852 / 1000 correct (85.20)

Iteration 500, loss = 0.2667
Checking accuracy on validation set
Got 839 / 1000 correct (83.90)

Iteration 600, loss = 0.3304
Checking accuracy on validation set
Got 853 / 1000 correct (85.30)

Iteration 700, loss = 0.3756
Checking accuracy on validation set
Got 849 / 1000 correct (84.90)
```

描述下你做了什么

在下面的单元格中，你应该解释你做了什么，你实现了什么额外的功能，和/或你在训练和评估你的网络的过程中做了什么。。

简单实现了一个卷积神经网络, 在命名为BigConv的网络结构中.

每个pack层将通道数加倍, 将图片的长宽两维减半. 当维数降低到一定程度, 使用全连接层直接输出.

每个pack层除了维数外, 结构完全相同, 都是Conv-Relu-Conv-Relu-BatchNorm-MaxPool-Dropout

训练策略上没有任何特殊之处, 直接使用Adam默认参数训练. 甚至没有调整超参数.

Test set -- run this only once

Now that we've gotten a result we're happy with, we test our final model on the test set (which you should store in best_model). Think about how this compares to your validation set accuracy. 现在我们已经获得了满意的结果，我们在测试集上测试最终模型（您应该将其存储在best_model中）。考虑一下这与你在验证集上的准确性相比如何。

In [35]:
```
best_model = model
check_accuracy_part34(loader_test, best_model)

num_correct = 0
num_samples = 0
best_model.eval()   # set model to evaluation mode
with torch.no_grad():
    for x, y in loader_test:
        x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
        y = y.to(device=device, dtype=torch.long)
        scores = best_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 * acc))
```

```
Checking accuracy on test set
Got 8375 / 10000 correct (83.75)
Got 8375 / 10000 correct (83.75)
```

提醒：运行完下面代码之后，点击下面的submit，然后去leaderboard上查看你的成绩。本模型对应的成绩在phase2的leaderboard中。

In [36]:
```
import os
#输出格式
def output_file(acc, phase_id=2):
    path=os.getcwd()
    if not os.path.exists(path + '/output/phase_{}'.format(phase_id)):
        os.makedirs(path + '/output/phase_{}'.format(phase_id))
    path=path + '/output/phase_{}/accuracy.txt'.format(phase_id)
    with open(path,'w+') as f:
        f.write(str(acc))

def zip_fun(phase_id=2):
    path=os.getcwd()
    output_path = path + '/output'
    files = os.listdir(output_path)
    for _file in files:
        if _file.find('zip') != -1:
            os.remove(output_path + '/' + _file)
    newpath=path+'/output/phase_{}'.format(phase_id)
    os.chdir(newpath)
    cmd = 'zip ../accuracy_phase_{}.zip accuracy.txt'.format(phase_id)
    os.system(cmd)
    os.chdir(path)
output_file(acc)
zip_fun()
```