Assignment 3: Convolutional Neural Networks with Pytorch

For this assignment, we're going to use one of most popular deep learning frameworks: PyTorch. And build our way through Convolutional Neural Networks.

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.

If you are running on datahub, you shouldn't face any problem.

You can also find the detailed PyTorch 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-100 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-100 open-ended challenge: please implement your own network to get as high accuracy as possible on CIFAR-100. 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

First, we load the CIFAR-100 dataset. This might take a couple minutes the first time you do it, but the files should stay cached after that.

In previous parts of the assignment we had to write our own code to download the CIFAR-20 dataset, preprocess it, and iterate through it in minibatches; PyTorch provides convenient tools to automate this process for us.

```
54 # Add official website of pytorch

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

```
NUM_TRAIN = 49000
batch_size= 32

# 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.

#================================================================================#
# You should try changing the transform for the training data to include    #
# data augmentation such as RandmCrop and HorizontalFlip                    #
# when running the final part of the notebook where you have to achieve     #
# as high accuracy as possible on CIFAR-100.                                #
# Of course you will have to re-run this block for the effect to take place #
#================================================================================#
train_transform = T.Compose([
                T.ToTensor(),
                T.Normalize((0.5071, 0.4867, 0.4408), (0.2675, 0.2565, 0.2761)),
                T.RandomCrop(32,padding=4),
                T.RandomHorizontalFlip(p=0.5),
                # T.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0.1)
                T.RandomAffine(5,(0,0.2),(0.9,1.1)),
            ])
transform = T.Compose([
                T.ToTensor(),
                T.Normalize((0.5071, 0.4867, 0.4408), (0.2675, 0.2565, 0.2761))
            ])
# 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-100
# training set into train and val sets by passing a Sampler object to the
# DataLoader telling how it should sample from the underlying Dataset.
cifar100_train = dset.CIFAR100('./datasets/cifar100', train=True, download=False,
                               transform=train_transform)
loader_train = DataLoader(cifar100_train, batch_size=batch_size, num_workers=2,
                          sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN)))

cifar100_val = dset.CIFAR100('./datasets/cifar100', train=True, download=False,
                             transform=transform)
loader_val = DataLoader(cifar100_val, batch_size=batch_size, num_workers=2,
                        sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN, 50000)))

cifar100_test = dset.CIFAR100('./datasets/cifar100', train=False, download=False,
                              transform=transform)
loader_test = DataLoader(cifar100_test, batch_size=batch_size, num_workers=2)
```

You have an option to **use GPU by setting the flag to True below**. It is not necessary to use GPU for this assignment. Note that if your computer does not have CUDA enabled, `torch.cuda.is_available()` will return False and this notebook will fallback to CPU mode. **You can run on GPU on datahub.**

The global variables `dtype` and `device` will control the data types throughout this assignment.

```
USE_GPU = True
num_class = 100
dtype = torch.float32 # we will be using float throughout this tutorial
print(torch.cuda.is_available())
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)
```

```
True
using device: cuda
```

```
print(cifar100_train[0][0].shape)
```

```
torch.Size([3, 32, 32])
```

Part II. Barebones PyTorch (10% of Grade)

PyTorch ships with high-level APIs to help us define model architectures conveniently, which we will cover in Part II of this tutorial. In this section, we will start with the barebone PyTorch elements to understand the autograd engine better. After this exercise, you will come to appreciate the high-level model API more.

We will start with a simple fully-connected ReLU network with two hidden layers and no biases for CIFAR classification. This implementation computes the forward pass using operations on PyTorch Tensors, and uses PyTorch autograd to compute gradients. It is important that you understand every line, because you will write a harder version after the example.

When we create a PyTorch Tensor with `requires_grad=True`, then operations involving that Tensor will not just compute values; they will also build up a computational graph in the background, allowing us to easily backpropagate through the graph to compute gradients of some Tensors with respect to a downstream loss. Concretely if x is a Tensor with `x.requires_grad == True` then after backpropagation `x.grad` will be another Tensor holding the gradient of x with respect to the scalar loss at the end.

PyTorch Tensors: Flatten Function

A PyTorch Tensor is conceptionally similar to a numpy array: it is an n-dimensional grid of numbers, and like numpy PyTorch provides many functions to efficiently operate on Tensors. As a simple example, we provide a `flatten` function below which reshapes image data for use in a fully-connected neural network.

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

This is the right way to represent the data when we are doing something like a 2D convolution, that needs spatial understanding of where the intermediate features are relative to each other. When we use fully connected affine layers to process the image, however, we want each datapoint to be represented by a single vector -- it's no longer useful to segregate the different channels, rows, and columns of the data. So, we use a "flatten" operation to collapse the `C x H x W` values per representation into a single long vector. The flatten function below first reads in the N, C, H, and W values from a given batch of data, and then returns a "view" of that data. "View" is analogous to numpy's "reshape" method: it reshapes x's dimensions to be N x ??, where ?? is allowed to be anything (in this case, it will be C x H x W, but we don't need to specify that explicitly).

14
```
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 image

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

Here we define a function `two_layer_fc` which performs the forward pass of a two-layer fully-connected ReLU network on a batch of image data. After defining the forward pass we check that it doesn't crash and that it produces outputs of the right shape by running zeros through the network.

You don't have to write any code here, but it's important that you read and understand the implementation.

15
```
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, num_class), dtype=dtype)
    scores = two_layer_fc(x, [w1, w2])
    print(scores.size())  # you should see [64, 100]

two_layer_fc_test()
```

```
torch.Size([64, 100])
```

Barebones PyTorch: Three-Layer ConvNet

Here you will complete the implementation of the function `three_layer_convnet`, which will perform the forward pass of a three-layer convolutional network. Like above, we can immediately test our implementation by passing zeros through the network. The network should have the following architecture:

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.

Note that we have **no softmax activation** here after our fully-connected layer: this is because PyTorch's cross entropy loss performs a softmax activation for you, and by bundling that step in makes computation more efficient.

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

```
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
    x = F.conv2d(x, conv_w1, bias=conv_b1,padding=2)
    x = F.relu(x)
    x = F.conv2d(x, conv_w2, bias=conv_b2,padding=1)
    x = F.relu(x)
    x = flatten(x)
    scores = x.mm(fc_w) + fc_b # nn.functional.linear(x, fc_w, bias=fc_b)
    # scores = None
    ################################################################################
    # TODO: Implement the forward pass for the three-layer ConvNet.                #
    ################################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

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

After defining the forward pass of the ConvNet above, run the following cell to test your implementation.

When you run this function, scores should have shape (64, 100).

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

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

    # you must calculate the shape of the tensor after two conv layers, before the fully-connected la
    fc_w = torch.zeros((9 * 32 * 32, num_class))
```

```
    fc_b = torch.zeros(num_class)

    scores = three_layer_convnet(x, [conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b])
    print(scores.size())  # you should see [64, 100]
three_layer_convnet_test()
```

```
torch.Size([64, 100])
```

Barebones PyTorch: Initialization

Let's write a couple utility methods to initialize the weight matrices for our models.

- `random_weight(shape)` initializes a weight tensor with the Kaiming normalization method.
- `zero_weight(shape)` initializes a weight tensor with all zeros. Useful for instantiating bias parameters.

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

```
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))
```

```
tensor([[-0.0540,  0.4889,  1.0840, -0.4864,  0.5212],
        [ 0.9080,  1.6614,  1.7636,  0.6107,  0.0716],
        [ 0.0958, -0.9291,  0.7358,  0.5205, -2.2374]], device='cuda:0',
       requires_grad=True)
```

Barebones PyTorch: Check Accuracy

When training the model we will use the following function to check the accuracy of our model on the training or validation sets.

When checking accuracy we don't need to compute any gradients; as a result we don't need PyTorch to build a computational graph for us when we compute scores. To prevent a graph from being built we scope our computation under a `torch.no_grad()` context manager.

```
42 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

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.

```
43 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

Now we are ready to run the training loop. We need to explicitly allocate tensors for the fully connected weights, `w1` and `w2`.

Each minibatch of CIFAR has 64 examples, so the tensor shape is `[64, 3, 32, 32]`.

After flattening, `x` shape should be `[64, 3 * 32 * 32]`. This will be the size of the first dimension of `w1`. The second dimension of `w1` is the hidden layer size, which will also be the first dimension of `w2`.

Finally, the output of the network is a 10-dimensional vector that represents the probability distribution over 10 classes.

You don't need to tune any hyperparameters but you should see accuracies above 15% after training for one epoch.

```
hidden_layer_size = 4000
learning_rate = 1e-2

w1 = random_weight((3 * 32 * 32, hidden_layer_size))
w2 = random_weight((hidden_layer_size, num_class))

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

```
Iteration 0, loss = 5.9374
Checking accuracy on the val set
Got 18 / 1000 correct (1.80%)

Iteration 100, loss = 4.2495
Checking accuracy on the val set
Got 76 / 1000 correct (7.60%)

Iteration 200, loss = 3.9595
Checking accuracy on the val set
Got 116 / 1000 correct (11.60%)

Iteration 300, loss = 3.8545
Checking accuracy on the val set
Got 124 / 1000 correct (12.40%)

Iteration 400, loss = 3.7451
Checking accuracy on the val set
```

```
Got 132 / 1000 correct (13.20%)

Iteration 500, loss = 3.7578
Checking accuracy on the val set
Got 160 / 1000 correct (16.00%)

Iteration 600, loss = 3.6868
Checking accuracy on the val set
Got 147 / 1000 correct (14.70%)

Iteration 700, loss = 3.6009
Checking accuracy on the val set
Got 169 / 1000 correct (16.90%)
```

BareBones PyTorch: Training a ConvNet

In the below cell you should use the functions defined above to train a three-layer convolutional network on CIFAR. The network should have the following architecture:

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 100 classes

You should initialize your weight matrices using the `random_weight` function defined above, and you should initialize your bias vectors using the `zero_weight` function above.

You don't need to tune any hyperparameters, but if everything works correctly you should achieve an accuracy above **12% after one epoch**.

```
*   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, conv_b1 = random_weight((channel_1,3,5,5)), random_weight((channel_1,))
    conv_w2, conv_b2 = random_weight((channel_2,channel_1,3,3)), random_weight((channel_2,))
    fc_w = random_weight((channel_2 * 32 * 32,100))
    fc_b = zero_weight((100,))

    # *****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)
```

Part III. PyTorch Module API (10% of Grade)

Barebone PyTorch requires that we track all the parameter tensors by hand. This is fine for small networks with a few tensors, but it would be extremely inconvenient and error-prone to track tens or hundreds of tensors in larger networks.

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:

```
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 dimension 50
    model = TwoLayerFC(input_size, 42, num_class)
    scores = model(x)
    print(scores.size())  # you should see [64, 100]
test_TwoLayerFC()
```

```
torch.Size([64, 100])
```

Module API: Three-Layer ConvNet

It's your turn to implement a 3-layer ConvNet followed by a fully connected layer. The network architecture should be the same as in Part II:

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.

```
class ThreeLayerConvNet(nn.Module):
    def __init__(self, in_channel, channel_1, channel_2, num_classes):
        super().__init__()

        self.conv_1 = nn.Conv2d(in_channel, channel_1, (5,5), padding=2)
        nn.init.kaiming_normal_(self.conv_1.weight)
        self.conv_2 = nn.Conv2d(channel_1, channel_2, (3,3), padding=1)
        nn.init.kaiming_normal_(self.conv_2.weight)

        self.fc1 = nn.Linear(channel_2 * 32 * 32, num_classes)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)


    def forward(self, x):

        ################################################################################
        # 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(x)
        x = self.conv_2(x)
        x = self.relu(x)
        x = flatten(x)
        # print(x.shape)
        scores = self.fc1(x)
        # scores = None
        # *****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, 32, 32]
    model = ThreeLayerConvNet(in_channel=3, channel_1=32, channel_2=64, num_classes=num_class)
    scores = model(x)
    print(scores.size())  # you should see [64, 100]
test_ThreeLayerConvNet()
```

```
torch.Size([64, 100])
```

Module API: Check Accuracy

Given the validation or test set, we can check the classification accuracy of a neural network.

This version is slightly different from the one in part II. You don't manually pass in the parameters anymore.

```
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))
    return acc
```

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.

```
from torch.utils.tensorboard import SummaryWriter
import torchvision
```

```
def train_part34(model, optimizer, epochs=1, verbose=False):
    """
    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.
    """
    writer = SummaryWriter(log_dir='./log/')

    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:
                # acc = check_accuracy_part34(loader_val, model)
                if verbose:
                    print('Epoch %d, Iteration %d, loss = %.4f' % (e, t, loss.item()))
                    print()
                writer.add_scalar('Loss/train', loss, int(10*(e+t/800)) )
                # writer.add_scalar('Accuracy/test', acc, int(10*(e+t/800)) )
        scheduler.step()
    writer.close()
```

Module API: Train a Two-Layer Network

Now we are ready to run the training loop. In contrast to part II, we don't explicitly allocate parameter tensors anymore.

Simply pass the input size, hidden layer size, and number of classes (i.e. output size) to the constructor of `TwoLayerFC`.

You also need to define an optimizer that tracks all the learnable parameters inside `TwoLayerFC`.

You don't need to tune any hyperparameters, but you should see model accuracies above 9% after training for one epoch.

```
hidden_layer_size = 4000
learning_rate = 1e-3
model = TwoLayerFC(3 * 32 * 32, hidden_layer_size, num_class)
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

train_part34(model, optimizer)
```

```
Epoch 0, Iteration 0, loss = 5.3706
Checking accuracy on validation set
Got 9 / 1000 correct (0.90)

Epoch 0, Iteration 100, loss = 4.9797
Checking accuracy on validation set
Got 33 / 1000 correct (3.30)

Epoch 0, Iteration 200, loss = 4.4310
Checking accuracy on validation set
Got 53 / 1000 correct (5.30)

Epoch 0, Iteration 300, loss = 4.1894
Checking accuracy on validation set
Got 63 / 1000 correct (6.30)

Epoch 0, Iteration 400, loss = 4.3837
Checking accuracy on validation set
Got 79 / 1000 correct (7.90)

Epoch 0, Iteration 500, loss = 4.2321
Checking accuracy on validation set
Got 80 / 1000 correct (8.00)

Epoch 0, Iteration 600, loss = 3.9336
Checking accuracy on validation set
Got 89 / 1000 correct (8.90)

Epoch 0, Iteration 700, loss = 3.9524
Checking accuracy on validation set
Got 97 / 1000 correct (9.70)
```

Module API: Train a Three-Layer ConvNet

You should now use the Module API to train a three-layer ConvNet on CIFAR. This should look very similar to training the two-layer network! You don't need to tune any hyperparameters, but you should achieve **accuracy above 14% after training for one epoch**.

You should train the model using stochastic gradient descent without momentum.

```
learning_rate = 1e-3
channel_1 = 32
channel_2 = 64

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, num_class)
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, epochs=10)

Epoch 0, Iteration 0, loss = 4.6307
Checking accuracy on validation set
Got 11 / 1000 correct (1.10)

Epoch 0, Iteration 100, loss = 4.2524
Checking accuracy on validation set
Got 92 / 1000 correct (9.20)

Epoch 0, Iteration 200, loss = 3.8076
Checking accuracy on validation set
Got 114 / 1000 correct (11.40)

Epoch 0, Iteration 300, loss = 3.7414
Checking accuracy on validation set
Got 144 / 1000 correct (14.40)

Epoch 0, Iteration 400, loss = 3.7685
Checking accuracy on validation set
Got 145 / 1000 correct (14.50)

Epoch 0, Iteration 500, loss = 3.6771
Checking accuracy on validation set
Got 149 / 1000 correct (14.90)

Epoch 0, Iteration 600, loss = 3.7916
Checking accuracy on validation set
Got 162 / 1000 correct (16.20)

Epoch 0, Iteration 700, loss = 3.4535
Checking accuracy on validation set
Got 167 / 1000 correct (16.70)

Epoch 1, Iteration 0, loss = 3.9104
Checking accuracy on validation set
Got 178 / 1000 correct (17.80)

Epoch 1, Iteration 100, loss = 3.4871
Checking accuracy on validation set
Got 182 / 1000 correct (18.20)

Epoch 1, Iteration 200, loss = 3.3677
Checking accuracy on validation set
Got 184 / 1000 correct (18.40)

Epoch 1, Iteration 300, loss = 3.2507
Checking accuracy on validation set
Got 194 / 1000 correct (19.40)

Epoch 1, Iteration 400, loss = 3.1743
Checking accuracy on validation set
Got 201 / 1000 correct (20.10)

Epoch 1, Iteration 500, loss = 3.2543
Checking accuracy on validation set
Got 204 / 1000 correct (20.40)

Epoch 1, Iteration 600, loss = 3.4664
Checking accuracy on validation set
Got 199 / 1000 correct (19.90)

Epoch 1, Iteration 700, loss = 2.8647
Checking accuracy on validation set
Got 206 / 1000 correct (20.60)

Epoch 2, Iteration 0, loss = 3.2955
Checking accuracy on validation set
```

```
Got 207 / 1000 correct (20.70)

Epoch 2, Iteration 100, loss = 3.3614
Checking accuracy on validation set
Got 227 / 1000 correct (22.70)

Epoch 2, Iteration 200, loss = 3.4730
Checking accuracy on validation set
Got 221 / 1000 correct (22.10)

Epoch 2, Iteration 300, loss = 2.9703
Checking accuracy on validation set
Got 227 / 1000 correct (22.70)

Epoch 2, Iteration 400, loss = 2.9726
Checking accuracy on validation set
Got 243 / 1000 correct (24.30)

Epoch 2, Iteration 500, loss = 3.1383
Checking accuracy on validation set
Got 226 / 1000 correct (22.60)

Epoch 2, Iteration 600, loss = 2.9165
Checking accuracy on validation set
Got 237 / 1000 correct (23.70)

Epoch 2, Iteration 700, loss = 2.8607
Checking accuracy on validation set
Got 224 / 1000 correct (22.40)

Epoch 3, Iteration 0, loss = 2.9531
Checking accuracy on validation set
Got 235 / 1000 correct (23.50)

Epoch 3, Iteration 100, loss = 2.3275
Checking accuracy on validation set
Got 235 / 1000 correct (23.50)

Epoch 3, Iteration 200, loss = 2.8726
Checking accuracy on validation set
Got 248 / 1000 correct (24.80)

Epoch 3, Iteration 300, loss = 2.7435
Checking accuracy on validation set
Got 226 / 1000 correct (22.60)

Epoch 3, Iteration 400, loss = 3.2471
Checking accuracy on validation set
Got 250 / 1000 correct (25.00)

Epoch 3, Iteration 500, loss = 3.3182
Checking accuracy on validation set
Got 265 / 1000 correct (26.50)

Epoch 3, Iteration 600, loss = 2.8614
Checking accuracy on validation set
Got 257 / 1000 correct (25.70)

Epoch 3, Iteration 700, loss = 2.6419
Checking accuracy on validation set
Got 258 / 1000 correct (25.80)

Epoch 4, Iteration 0, loss = 2.4571
Checking accuracy on validation set
Got 253 / 1000 correct (25.30)

Epoch 4, Iteration 100, loss = 2.6165
Checking accuracy on validation set
Got 235 / 1000 correct (23.50)
```

```
Epoch 4, Iteration 200, loss = 3.0381
Checking accuracy on validation set
Got 254 / 1000 correct (25.40)

Epoch 4, Iteration 300, loss = 2.2488
Checking accuracy on validation set
Got 256 / 1000 correct (25.60)

Epoch 4, Iteration 400, loss = 2.5571
Checking accuracy on validation set
Got 250 / 1000 correct (25.00)

Epoch 4, Iteration 500, loss = 2.2724
Checking accuracy on validation set
Got 253 / 1000 correct (25.30)

Epoch 4, Iteration 600, loss = 2.8949
Checking accuracy on validation set
Got 275 / 1000 correct (27.50)

Epoch 4, Iteration 700, loss = 3.0051
Checking accuracy on validation set
Got 266 / 1000 correct (26.60)

Epoch 5, Iteration 0, loss = 2.4699
Checking accuracy on validation set
Got 265 / 1000 correct (26.50)

Epoch 5, Iteration 100, loss = 2.5320
Checking accuracy on validation set
Got 260 / 1000 correct (26.00)

Epoch 5, Iteration 200, loss = 2.4447
Checking accuracy on validation set
Got 266 / 1000 correct (26.60)

Epoch 5, Iteration 300, loss = 2.4716
Checking accuracy on validation set
Got 269 / 1000 correct (26.90)

Epoch 5, Iteration 400, loss = 2.2885
Checking accuracy on validation set
Got 267 / 1000 correct (26.70)

Epoch 5, Iteration 500, loss = 2.9361
Checking accuracy on validation set
Got 271 / 1000 correct (27.10)

Epoch 5, Iteration 600, loss = 2.6894
Checking accuracy on validation set
Got 275 / 1000 correct (27.50)

Epoch 5, Iteration 700, loss = 2.6154
Checking accuracy on validation set
Got 277 / 1000 correct (27.70)

Epoch 6, Iteration 0, loss = 2.4936
Checking accuracy on validation set
Got 276 / 1000 correct (27.60)

Epoch 6, Iteration 100, loss = 2.6612
Checking accuracy on validation set
Got 281 / 1000 correct (28.10)

Epoch 6, Iteration 200, loss = 2.2905
Checking accuracy on validation set
Got 271 / 1000 correct (27.10)
```

```
Epoch 6, Iteration 300, loss = 2.4576
Checking accuracy on validation set
Got 271 / 1000 correct (27.10)

Epoch 6, Iteration 400, loss = 2.5360
Checking accuracy on validation set
Got 277 / 1000 correct (27.70)

Epoch 6, Iteration 500, loss = 2.2093
Checking accuracy on validation set
Got 283 / 1000 correct (28.30)

Epoch 6, Iteration 600, loss = 2.6173
Checking accuracy on validation set
Got 276 / 1000 correct (27.60)

Epoch 6, Iteration 700, loss = 2.5639
Checking accuracy on validation set
Got 290 / 1000 correct (29.00)

Epoch 7, Iteration 0, loss = 2.1761
Checking accuracy on validation set
Got 296 / 1000 correct (29.60)

Epoch 7, Iteration 100, loss = 2.3117
Checking accuracy on validation set
Got 277 / 1000 correct (27.70)

Epoch 7, Iteration 200, loss = 2.3396
Checking accuracy on validation set
Got 279 / 1000 correct (27.90)

Epoch 7, Iteration 300, loss = 2.4789
Checking accuracy on validation set
Got 290 / 1000 correct (29.00)

Epoch 7, Iteration 400, loss = 2.2385
Checking accuracy on validation set
Got 278 / 1000 correct (27.80)

Epoch 7, Iteration 500, loss = 2.1992
Checking accuracy on validation set
Got 286 / 1000 correct (28.60)

Epoch 7, Iteration 600, loss = 2.2275
Checking accuracy on validation set
Got 281 / 1000 correct (28.10)

Epoch 7, Iteration 700, loss = 2.1755
Checking accuracy on validation set
Got 296 / 1000 correct (29.60)

Epoch 8, Iteration 0, loss = 2.0476
Checking accuracy on validation set
Got 278 / 1000 correct (27.80)

Epoch 8, Iteration 100, loss = 2.0544
Checking accuracy on validation set
Got 288 / 1000 correct (28.80)

Epoch 8, Iteration 200, loss = 2.1466
Checking accuracy on validation set
Got 297 / 1000 correct (29.70)

Epoch 8, Iteration 300, loss = 2.4655
Checking accuracy on validation set
Got 293 / 1000 correct (29.30)

Epoch 8, Iteration 400, loss = 1.9979
```

```
Checking accuracy on validation set
Got 282 / 1000 correct (28.20)

Epoch 8, Iteration 500, loss = 1.8594
Checking accuracy on validation set
Got 288 / 1000 correct (28.80)

Epoch 8, Iteration 600, loss = 2.1639
Checking accuracy on validation set
Got 299 / 1000 correct (29.90)

Epoch 8, Iteration 700, loss = 2.2210
Checking accuracy on validation set
Got 285 / 1000 correct (28.50)

Epoch 9, Iteration 0, loss = 1.8603
Checking accuracy on validation set
Got 303 / 1000 correct (30.30)

Epoch 9, Iteration 100, loss = 1.8706
Checking accuracy on validation set
Got 290 / 1000 correct (29.00)

Epoch 9, Iteration 200, loss = 2.0336
Checking accuracy on validation set
Got 285 / 1000 correct (28.50)

Epoch 9, Iteration 300, loss = 2.2610
Checking accuracy on validation set
Got 287 / 1000 correct (28.70)

Epoch 9, Iteration 400, loss = 2.2323
Checking accuracy on validation set
Got 294 / 1000 correct (29.40)

Epoch 9, Iteration 500, loss = 1.9580
Checking accuracy on validation set
Got 305 / 1000 correct (30.50)

Epoch 9, Iteration 600, loss = 2.1724
Checking accuracy on validation set
Got 300 / 1000 correct (30.00)

Epoch 9, Iteration 700, loss = 2.2970
Checking accuracy on validation set
Got 298 / 1000 correct (29.80)
```

Part IV. PyTorch Sequential API (10% of Grade)

Part III introduced the PyTorch Module API, which allows you to define arbitrary learnable layers and their connectivity.

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 17% accuracy after one epoch of training.

71
```
# 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, num_class),
)

# 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)
```

```
Epoch 0, Iteration 0, loss = 4.6178
Checking accuracy on validation set
Got 15 / 1000 correct (1.50)

Epoch 0, Iteration 100, loss = 3.7430
Checking accuracy on validation set
Got 115 / 1000 correct (11.50)

Epoch 0, Iteration 200, loss = 3.5820
Checking accuracy on validation set
Got 128 / 1000 correct (12.80)

Epoch 0, Iteration 300, loss = 3.7992
Checking accuracy on validation set
Got 141 / 1000 correct (14.10)

Epoch 0, Iteration 400, loss = 3.5358
Checking accuracy on validation set
Got 139 / 1000 correct (13.90)

Epoch 0, Iteration 500, loss = 3.8043
Checking accuracy on validation set
Got 155 / 1000 correct (15.50)

Epoch 0, Iteration 600, loss = 3.6303
Checking accuracy on validation set
Got 160 / 1000 correct (16.00)

Epoch 0, Iteration 700, loss = 3.3456
Checking accuracy on validation set
Got 164 / 1000 correct (16.40)
```

Sequential API: Three-Layer ConvNet

Here you should use `nn.Sequential` to define and train a three-layer ConvNet with the same architecture we used in Part III:

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 100 classes

You should initialize your weight matrices using the `random_weight` function defined above, and you should initialize your bias vectors using the `zero_weight` function above.

You should optimize your model using stochastic gradient descent with Nesterov momentum 0.9.

Again, you don't need to tune any hyperparameters but you should see **accuracy above 20% after one epoch** of training.

```
channel_1 = 32
channel_2 = 16
learning_rate = 1e-3

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,(5,5),padding=2),
    nn.ReLU(),
    nn.Conv2d(channel_1,channel_2,(3,3),padding=1),
    nn.ReLU(),
    Flatten(),
    nn.Linear(channel_2*32*32,num_class)
)
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, epochs=1)
```

```
Epoch 0, Iteration 0, loss = 4.6118
Checking accuracy on validation set
Got 7 / 1000 correct (0.70)

Epoch 0, Iteration 100, loss = 4.3096
Checking accuracy on validation set
Got 47 / 1000 correct (4.70)
```

```
Epoch 0, Iteration 200, loss = 4.0023
Checking accuracy on validation set
Got 95 / 1000 correct (9.50)

Epoch 0, Iteration 300, loss = 3.8132
Checking accuracy on validation set
Got 99 / 1000 correct (9.90)

Epoch 0, Iteration 400, loss = 3.8537
Checking accuracy on validation set
Got 135 / 1000 correct (13.50)

Epoch 0, Iteration 500, loss = 3.7771
Checking accuracy on validation set
Got 135 / 1000 correct (13.50)

Epoch 0, Iteration 600, loss = 3.7188
Checking accuracy on validation set
Got 151 / 1000 correct (15.10)

Epoch 0, Iteration 700, loss = 3.4768
Checking accuracy on validation set
Got 185 / 1000 correct (18.50)
```

Part V. CIFAR-10 open-ended challenge (50% Grade)

In this section, you can experiment with whatever ConvNet architecture you'd like on CIFAR-100.

Now it's your job to experiment with architectures, hyperparameters, loss functions, and optimizers to train a model that achieves **at least 55%** accuracy on the CIFAR-100 **test** set within 10 epochs. You can use the check_accuracy and train functions from above. You can use either `nn.Module` or `nn.Sequential` API.

Describe what you did at the end of this notebook.

Here are the official API documentation for each component.

- 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:

- **Filter size**: Above we used 5x5; would smaller filters be more efficient?
- **Adam Optimizer**: Above we used SGD optimizer, would an Adam optimizer do better?
- **Number of filters**: Above we used 32 filters. Do more or fewer do better?
- **Pooling vs Strided Convolution**: Do you use max pooling or just stride convolutions?

- **Batch normalization**: Try adding spatial batch normalization after convolution layers and vanilla batch normalization after affine layers. Do your networks train faster? You can also try out LayerNorm and GroupNorm.
- **Network architecture**: Can you do better with a deep network? Good architectures to try include:
  - [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**: Add l2 weight regularization, or perhaps use Dropout.

Tips for training

For each network architecture that you try, you should tune the learning rate and other hyperparameters. When doing this there are a couple important things to keep in mind:

- If the parameters are working well, you should see improvement within a few hundred iterations
- Remember the coarse-to-fine approach for hyperparameter tuning: start by testing a large range of hyperparameters for just a few training iterations to find the combinations of parameters that are working at all.
- Once you have found some sets of parameters that seem to work, search more finely around these parameters. You may need to train for more epochs.
- You should use the validation set for hyperparameter search, and save your test set for evaluating your architecture on the best parameters as selected by the validation set.

Want more improvements?

There are many other features you can implement to try and improve your performance.

- Alternative optimizers: you can try Adam, Adagrad, RMSprop, etc.
- Alternative activation functions such as leaky ReLU, parametric ReLU, ELU, or MaxOut.
- Model ensembles
- Data augmentation
- New Architectures
  - ResNets where the input from the previous layer is added to the output.
  - DenseNets where inputs into previous layers are concatenated together.

Have fun and may the gradients be with you!

```
################################################################################
# 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
from functools import partial

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
class Conv2dAuto(nn.Conv2d):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.padding =  (self.kernel_size[0] // 2, self.kernel_size[1] // 2) # dynamic add padding ba

conv3x3 = partial(Conv2dAuto, kernel_size=3, bias=False)

# noinspection PyTypeChecker
def activation_func(activation):
    return  nn.ModuleDict([
        ['relu', nn.ReLU(inplace=True)],
        ['leaky_relu', nn.LeakyReLU(negative_slope=0.01, inplace=True)],
        ['selu', nn.SELU(inplace=True)],
        ['none', nn.Identity()]
    ])[activation]

class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, activation='relu'):
        super().__init__()
        self.in_channels, self.out_channels, self.activation = in_channels, out_channels, activation
        self.blocks = nn.Identity()
        self.activate = activation_func(activation)
        self.shortcut = nn.Identity()

    def forward(self, x):
        residual = x
        if self.should_apply_shortcut: residual = self.shortcut(x)
        x = self.blocks(x)
        x += residual
        x = self.activate(x)
        return x

    @property
    def should_apply_shortcut(self):
        return self.in_channels != self.out_channels

class ResNetResidualBlock(ResidualBlock):
    def __init__(self, in_channels, out_channels, expansion=1, downsampling=1, conv=conv3x3, *args, *
        super().__init__(in_channels, out_channels, *args, **kwargs)
        self.expansion, self.downsampling, self.conv = expansion, downsampling, conv
        self.shortcut = nn.Sequential(
            nn.Conv2d(self.in_channels, self.expanded_channels, kernel_size=1,
                      stride=self.downsampling, bias=False),
            nn.BatchNorm2d(self.expanded_channels)) if self.should_apply_shortcut else None


    @property
    def expanded_channels(self):
        return self.out_channels * self.expansion
```

```
    @property
    def should_apply_shortcut(self):
        return self.in_channels != self.expanded_channels


def conv_bn(in_channels, out_channels, conv, *args, **kwargs):
    return nn.Sequential(conv(in_channels, out_channels, *args, **kwargs), nn.BatchNorm2d(out_channel


class ResNetBasicBlock(ResNetResidualBlock):
    """
    Basic ResNet block composed by two layers of 3x3conv/batchnorm/activation
    """
    expansion = 1
    def __init__(self, in_channels, out_channels, *args, **kwargs):
        super().__init__(in_channels, out_channels, *args, **kwargs)
        self.blocks = nn.Sequential(
            conv_bn(self.in_channels, self.out_channels, conv=self.conv, bias=False, stride=self.down
            activation_func(self.activation),
            conv_bn(self.out_channels, self.expanded_channels, conv=self.conv, bias=False),
        )


class ResNetBottleNeckBlock(ResNetResidualBlock):
    expansion = 4
    def __init__(self, in_channels, out_channels, *args, **kwargs):
        super().__init__(in_channels, out_channels, expansion=4, *args, **kwargs)
        self.blocks = nn.Sequential(
           conv_bn(self.in_channels, self.out_channels, self.conv, kernel_size=1),
             activation_func(self.activation),
             conv_bn(self.out_channels, self.out_channels, self.conv, kernel_size=3, stride=self.down
             activation_func(self.activation),
             conv_bn(self.out_channels, self.expanded_channels, self.conv, kernel_size=1),
        )


class ResNetLayer(nn.Module):
    """
    A ResNet layer composed by `n` blocks stacked one after the other
    """
    def __init__(self, in_channels, out_channels, block=ResNetBasicBlock, n=1, *args, **kwargs):
        super().__init__()
        # 'We perform downsampling directly by convolutional layers that have a stride of 2.'
        downsampling = 2 if in_channels != out_channels else 1
        self.blocks = nn.Sequential(
            block(in_channels , out_channels, *args, **kwargs, downsampling=downsampling),
            *[block(out_channels * block.expansion,
                    out_channels, downsampling=1, *args, **kwargs) for _ in range(n - 1)]
        )

    def forward(self, x):
        x = self.blocks(x)
        return x


class ResNetEncoder(nn.Module):
    """
    ResNet encoder composed by layers with increasing features.
    """
    def __init__(self, in_channels=3, blocks_sizes=[64, 128, 256, 512], deepths=[2,2,2,2],
                 activation='relu', block=ResNetBasicBlock, *args, **kwargs):
        super().__init__()
        self.blocks_sizes = blocks_sizes

        self.gate = nn.Sequential(
            nn.Conv2d(in_channels, self.blocks_sizes[0], kernel_size=7, stride=2, padding=3, bias=Fal
            nn.BatchNorm2d(self.blocks_sizes[0]),
            activation_func(activation),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
```

```
        )

        self.in_out_block_sizes = list(zip(blocks_sizes, blocks_sizes[1:]))
        self.blocks = nn.ModuleList([
            ResNetLayer(blocks_sizes[0], blocks_sizes[0], n=deepths[0], activation=activation,
                        block=block,*args, **kwargs),
            *[ResNetLayer(in_channels * block.expansion,
                          out_channels, n=n, activation=activation,
                          block=block, *args, **kwargs)
              for (in_channels, out_channels), n in zip(self.in_out_block_sizes, deepths[1:])]
        ])


    def forward(self, x):
        x = self.gate(x)
        for block in self.blocks:
            x = block(x)
        return x


class ResnetDecoder(nn.Module):
    """
    This class represents the tail of ResNet. It performs a global pooling and maps the output to the
    correct class by using a fully connected layer.
    """
    def __init__(self, in_features, n_classes):
        super().__init__()
        self.avg = nn.AdaptiveAvgPool2d((1, 1))
        self.decoder = nn.Sequential(
                                    nn.Linear(in_features, n_classes))

    def forward(self, x):
        x = self.avg(x)
        x = x.view(x.size(0), -1)
        x = self.decoder(x)
        return x


class ResNet(nn.Module):

    def __init__(self, in_channels, n_classes, *args, **kwargs):
        super().__init__()
        self.encoder = ResNetEncoder(in_channels, *args, **kwargs)
        self.decoder = ResnetDecoder(self.encoder.blocks[-1].blocks[-1].expanded_channels, n_classes)

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x

def resnet18(in_channels, n_classes):
    return ResNet(in_channels, n_classes, block=ResNetBasicBlock, deepths=[2, 2, 2, 2])

def resnet34(in_channels, n_classes):
    return ResNet(in_channels, n_classes, block=ResNetBasicBlock, deepths=[3, 4, 6, 3])

model = resnet18(3, 100)
learning_rate = 1e-1
gamma = 0.7
weight_decay = 1e-4
# optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, betas=(0.9, 0.999), eps=1e-08, w
optimizer = optim.SGD(model.parameters(), lr=learning_rate,weight_decay=weight_decay, momentum=0.9)
scheduler = optim.lr_scheduler.ExponentialLR(optimizer, gamma, last_epoch=-1, verbose=False)


print("resnet 18")
print("learning_rate, ", learning_rate)
print("weight_decay, ", weight_decay)
print("ExponentialLR, ", gamma)
```

```
train_part34(model, optimizer, epochs=100)
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
################################################################################
#                                END OF YOUR CODE
################################################################################

# You should get at least 55% accuracy. We were able to get 60% by Epoch 10.


resnet 18
learning_rate,  0.001
weight_decay,  0.0001
ExponentialLR,  0.9
Epoch 0, Iteration 0, loss = 4.7906
Checking accuracy on validation set
Got 13 / 1000 correct (1.30)

Epoch 0, Iteration 100, loss = 3.9949
Checking accuracy on validation set
Got 101 / 1000 correct (10.10)

Epoch 0, Iteration 200, loss = 3.9023
Checking accuracy on validation set
Got 134 / 1000 correct (13.40)

Epoch 0, Iteration 300, loss = 3.5162
Checking accuracy on validation set
Got 156 / 1000 correct (15.60)

Epoch 1, Iteration 0, loss = 3.4837
Checking accuracy on validation set
Got 191 / 1000 correct (19.10)

Epoch 1, Iteration 100, loss = 3.5059
Checking accuracy on validation set
Got 200 / 1000 correct (20.00)

Epoch 1, Iteration 200, loss = 3.0991
Checking accuracy on validation set
Got 220 / 1000 correct (22.00)

Epoch 1, Iteration 300, loss = 3.1837
Checking accuracy on validation set
Got 226 / 1000 correct (22.60)

Epoch 2, Iteration 0, loss = 3.0231
Checking accuracy on validation set
Got 222 / 1000 correct (22.20)

Epoch 2, Iteration 100, loss = 2.9361
Checking accuracy on validation set
Got 253 / 1000 correct (25.30)

Epoch 2, Iteration 200, loss = 3.0117
Checking accuracy on validation set
Got 267 / 1000 correct (26.70)

Epoch 2, Iteration 300, loss = 3.0547
Checking accuracy on validation set
Got 279 / 1000 correct (27.90)

Epoch 3, Iteration 0, loss = 2.7401
Checking accuracy on validation set
Got 287 / 1000 correct (28.70)

Epoch 3, Iteration 100, loss = 2.7154
Checking accuracy on validation set
Got 294 / 1000 correct (29.40)
```

```
Epoch 3, Iteration 200, loss = 2.7333
Checking accuracy on validation set
Got 322 / 1000 correct (32.20)

Epoch 3, Iteration 300, loss = 2.7635
Checking accuracy on validation set
Got 302 / 1000 correct (30.20)

Epoch 4, Iteration 0, loss = 2.6933
Checking accuracy on validation set
Got 294 / 1000 correct (29.40)

Epoch 4, Iteration 100, loss = 2.6062
Checking accuracy on validation set
Got 337 / 1000 correct (33.70)

Epoch 4, Iteration 200, loss = 3.0130
Checking accuracy on validation set
Got 330 / 1000 correct (33.00)

Epoch 4, Iteration 300, loss = 2.4761
Checking accuracy on validation set
Got 333 / 1000 correct (33.30)

Epoch 5, Iteration 0, loss = 2.9703
Checking accuracy on validation set
Got 340 / 1000 correct (34.00)

Epoch 5, Iteration 100, loss = 2.5857
Checking accuracy on validation set
Got 335 / 1000 correct (33.50)

Epoch 5, Iteration 200, loss = 2.8873
Checking accuracy on validation set
Got 342 / 1000 correct (34.20)

Epoch 5, Iteration 300, loss = 2.6696
Checking accuracy on validation set
Got 347 / 1000 correct (34.70)

Epoch 6, Iteration 0, loss = 2.4116
Checking accuracy on validation set
Got 356 / 1000 correct (35.60)

Epoch 6, Iteration 100, loss = 2.5355
Checking accuracy on validation set
Got 361 / 1000 correct (36.10)

Epoch 6, Iteration 200, loss = 2.7867
Checking accuracy on validation set
Got 372 / 1000 correct (37.20)

Epoch 6, Iteration 300, loss = 2.6288
Checking accuracy on validation set
Got 363 / 1000 correct (36.30)

Epoch 7, Iteration 0, loss = 2.3346
Checking accuracy on validation set
Got 378 / 1000 correct (37.80)

Epoch 7, Iteration 100, loss = 2.3080
Checking accuracy on validation set
Got 368 / 1000 correct (36.80)

Epoch 7, Iteration 200, loss = 2.1426
Checking accuracy on validation set
Got 369 / 1000 correct (36.90)
```

```
Epoch 7, Iteration 300, loss = 2.2942
Checking accuracy on validation set
Got 386 / 1000 correct (38.60)

Epoch 8, Iteration 0, loss = 2.3376
Checking accuracy on validation set
Got 366 / 1000 correct (36.60)

Epoch 8, Iteration 100, loss = 2.2548
Checking accuracy on validation set
Got 382 / 1000 correct (38.20)

Epoch 8, Iteration 200, loss = 2.3493
Checking accuracy on validation set
Got 387 / 1000 correct (38.70)

Epoch 8, Iteration 300, loss = 2.1076
Checking accuracy on validation set
Got 395 / 1000 correct (39.50)

Epoch 9, Iteration 0, loss = 2.2336
Checking accuracy on validation set
Got 392 / 1000 correct (39.20)

Epoch 9, Iteration 100, loss = 1.8633
Checking accuracy on validation set
Got 388 / 1000 correct (38.80)

Epoch 9, Iteration 200, loss = 2.2216
Checking accuracy on validation set
Got 395 / 1000 correct (39.50)

Epoch 9, Iteration 300, loss = 1.9951
Checking accuracy on validation set
Got 391 / 1000 correct (39.10)

Epoch 10, Iteration 0, loss = 2.2243
Checking accuracy on validation set
Got 404 / 1000 correct (40.40)

Epoch 10, Iteration 100, loss = 2.2242
Checking accuracy on validation set
Got 406 / 1000 correct (40.60)

Epoch 10, Iteration 200, loss = 2.4440
Checking accuracy on validation set
Got 419 / 1000 correct (41.90)

Epoch 10, Iteration 300, loss = 2.1318
Checking accuracy on validation set
Got 417 / 1000 correct (41.70)

Epoch 11, Iteration 0, loss = 2.5736
Checking accuracy on validation set
Got 423 / 1000 correct (42.30)

Epoch 11, Iteration 100, loss = 2.0911
Checking accuracy on validation set
Got 408 / 1000 correct (40.80)

Epoch 11, Iteration 200, loss = 2.0459
Checking accuracy on validation set
Got 411 / 1000 correct (41.10)

Epoch 11, Iteration 300, loss = 2.0767
Checking accuracy on validation set
Got 408 / 1000 correct (40.80)

Epoch 12, Iteration 0, loss = 2.1727
```

```
Checking accuracy on validation set
Got 407 / 1000 correct (40.70)

Epoch 12, Iteration 100, loss = 2.1559
Checking accuracy on validation set
Got 423 / 1000 correct (42.30)

Epoch 12, Iteration 200, loss = 2.1215
Checking accuracy on validation set
Got 427 / 1000 correct (42.70)

Epoch 12, Iteration 300, loss = 2.2073
Checking accuracy on validation set
Got 421 / 1000 correct (42.10)

Epoch 13, Iteration 0, loss = 1.9387
Checking accuracy on validation set
Got 413 / 1000 correct (41.30)

Epoch 13, Iteration 100, loss = 2.1851
Checking accuracy on validation set
Got 417 / 1000 correct (41.70)

Epoch 13, Iteration 200, loss = 2.0995
Checking accuracy on validation set
Got 421 / 1000 correct (42.10)

Epoch 13, Iteration 300, loss = 1.9652
Checking accuracy on validation set
Got 408 / 1000 correct (40.80)

Epoch 14, Iteration 0, loss = 2.1165
Checking accuracy on validation set
Got 424 / 1000 correct (42.40)

Epoch 14, Iteration 100, loss = 1.8092
Checking accuracy on validation set
Got 419 / 1000 correct (41.90)

Epoch 14, Iteration 200, loss = 2.1153
Checking accuracy on validation set
Got 414 / 1000 correct (41.40)

Epoch 14, Iteration 300, loss = 2.3237
Checking accuracy on validation set
Got 423 / 1000 correct (42.30)
```

Describe what you did (20% of Grade)

In the cell below you should write an explanation of what you did, any additional features that you implemented, and/or any graphs that you made in the process of training and evaluating your network.

```
117 import torch
    import torch.nn as nn
    import torch.nn.functional as F
    import torch.nn.init as init

    from torch.autograd import Variable

    def _weights_init(m):
        classname = m.__class__.__name__
        #print(classname)
```

```
    if isinstance(m, nn.Linear) or isinstance(m, nn.Conv2d):
        init.kaiming_normal_(m.weight)

class LambdaLayer(nn.Module):
    def __init__(self, lambd):
        super(LambdaLayer, self).__init__()
        self.lambd = lambd

    def forward(self, x):
        return self.lambd(x)


class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, in_planes, planes, stride=1, option='A'):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=3, stride=stride, padding=1, bias=False
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != planes:
            if option == 'A':
                """
                For CIFAR10 ResNet paper uses option A.
                """
                self.shortcut = LambdaLayer(lambda x:
                                            F.pad(x[:, :, ::2, ::2], (0, 0, 0, 0, planes//4, planes//4
            elif option == 'B':
                self.shortcut = nn.Sequential(
                     nn.Conv2d(in_planes, self.expansion * planes, kernel_size=1, stride=stride, bias
                     nn.BatchNorm2d(self.expansion * planes)
                )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = F.relu(out)
        return out


class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=100):
        super(ResNet, self).__init__()
        self.in_planes = 16

        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(16)
        self.layer1 = self._make_layer(block, 16, num_blocks[0], stride=1)
        self.layer2 = self._make_layer(block, 32, num_blocks[1], stride=2)
        self.layer3 = self._make_layer(block, 64, num_blocks[2], stride=2)
        self.linear = nn.Linear(64, num_classes)

        self.apply(_weights_init)

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1]*(num_blocks-1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_planes, planes, stride))
            self.in_planes = planes * block.expansion

        return nn.Sequential(*layers)

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
```

```
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = F.avg_pool2d(out, out.size()[3])
        out = out.view(out.size(0), -1)
        out = self.linear(out)
        return out


def resnet20():
    return ResNet(BasicBlock, [3, 3, 3])


model = resnet20()
learning_rate = 0.2
gamma = 0.8
weight_decay = 1e-4
# optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, betas=(0.9, 0.999), eps=1e-08, w
optimizer = optim.SGD(model.parameters(), lr=learning_rate,weight_decay=weight_decay, momentum=0.9)
scheduler = optim.lr_scheduler.ExponentialLR(optimizer, gamma, last_epoch=-1, verbose=False)


print("resnet 18")
print("learning_rate, ", learning_rate)
print("weight_decay, ", weight_decay)
print("ExponentialLR, ", gamma)

train_part34(model, optimizer, epochs=10)
```

```
resnet 18
learning_rate,  0.2
weight_decay,  0.0001
ExponentialLR,  0.8
```

TODO: Describe what you did

1. Plain conv network does not work good for me, so I try ResNet. The first version is in the previous cell. I greedily tune parameters like learning rate, weight decay rate and learning rate decay rate, to explore the limit of the ResNet architecture. However, this architecture just gives around 44% acc.

2. I read the ResNet paper, and find that the common ResNet is not designed for cifar challenge. Instead, it works for imagenet challenge. Cifar is too small(32*32), and will lose a lot of detail through the pooling operation. The author actually proposed another version of ResNet. I adopt the cifar version, and then searched for the optimal paramters.

3. ResNet allows very large learning rate(0.1) for shallow networks. (20 and 34 layers). So I use large learning rate with exponential decay, and train for 100 epoches to test the limit of the ResNet20 network.

4. The val accuarcy and loss curve shows that the model converges after 50 epoches. It can meet our 55% requirement after 20 epoches. However, our goal is to train it in 10 epoches. So we should use more aggressive optimization method. I double the initial learning rate, and use smaller batch size(32), so the update will be more aggressive but more unstable.

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.

```
111 best_model = model
    check_accuracy_part34(loader_test, best_model)
```

```
    Checking accuracy on test set
    Got 5892 / 10000 correct (58.92)
```

```
111 0.5892
```