Assignment 2: 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 (http://pytorch.org/docs/stable/index.html)](http://pytorch.org/docs/stable/index.html) here. If you have other questions that are not addressed by the API docs, the [PyTorch forum (https://discuss.pytorch.org/)](https://discuss.pytorch.org/) 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. ResNet10 Implementation: we will implement ResNet10 from scratch given the architecture details
6. Part VI, 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 [1]:

```
# 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
```

In [2]:

```
NUM_TRAIN = 49000
batch_size= 64

# The torchvision.transforms package provides tools for preprocessi
# and for performing data augmentation; here we set up a transform
# preprocess the data by subtracting the mean RGB value and dividin
# standard deviation of each RGB value; we've hardcoded the mean an

#==================================================================
# You should try changing the transform for the training data to in
# data augmentation such as RandomCrop and HorizontalFlip
# when running the final part of the notebook where you have to ach
# as high accuracy as possible on CIFAR-100.
# Of course you will have to re-run this block for the effect to ta
#==================================================================
train_transform = transform = T.Compose([
                T.ToTensor(),
                T.Normalize((0.5071, 0.4867, 0.4408), (0.2675, 0.25
            ])

# We set up a Dataset object for each split (train / val / test); D
# training examples one at a time, so we wrap each Dataset in a Dat
# iterates through the Dataset and forms minibatches. We divide the
# training set into train and val sets by passing a Sampler object
# DataLoader telling how it should sample from the underlying Datas
cifar100_train = dset.CIFAR100('./datasets/cifar100', train=True, d
                             transform=train_transform)
loader_train = DataLoader(cifar100_train, batch_size=batch_size, nu
                          sampler=sampler.SubsetRandomSampler(range

cifar100_val = dset.CIFAR100('./datasets/cifar100', train=True, dow
                           transform=transform)
loader_val = DataLoader(cifar100_val, batch_size=batch_size, num_wo
                        sampler=sampler.SubsetRandomSampler(range(N

cifar100_test = dset.CIFAR100('./datasets/cifar100', train=False, d
                            transform=transform)
loader_test = DataLoader(cifar100_test, batch_size=batch_size, num_
```

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

You have an option to **use GPU by setting the flag to True below** (recommended). 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.

In [3]:

```
USE_GPU = True
num_class = 100
dtype = torch.float32 # we will be using float throughout this tuto

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 (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-100 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

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 width 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).

In [4]:

```
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 s

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.

In [5]:

```
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
    and the output layer will produce scores for C classes.

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

    Returns:
    - scores: A PyTorch Tensor of shape (N, C) giving classificatio
      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
    # w2 have requires_grad=True, operations involving these Tensor
    # PyTorch to build a computational graph, allowing automatic co
    # gradients. Since we are no longer implementing the backward p
    # 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, fe
```

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

In [6]:

```
def three_layer_convnet(x, params):
    """
    Performs the forward pass of a three-layer convolutional networ
    architecture defined above.

    Inputs:
    - x: A PyTorch Tensor of shape (N, 3, H, W) giving a minibatch 
    - params: A list of PyTorch Tensors giving the weights and bias
      network; should contain the following:
      - conv_w1: PyTorch Tensor of shape (channel_1, 3, KH1, KW1) g
        for the first convolutional layer
      - conv_b1: PyTorch Tensor of shape (channel_1,) giving biases
        convolutional layer
      - conv_w2: PyTorch Tensor of shape (channel_2, channel_1, KH2
        weights for the second convolutional layer
      - conv_b2: PyTorch Tensor of shape (channel_2,) giving biases
        convolutional layer
      - fc_w: PyTorch Tensor giving weights for the fully-connected
        figure out what the shape should be?
      - fc_b: PyTorch Tensor giving biases for the fully-connected 
        figure out what the shape should be?

    Returns:
    - scores: PyTorch Tensor of shape (N, C) giving classification 
    """
    conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b = params
    scores = None
    ###############################################################
    # TODO: Implement the forward pass for the three-layer ConvNet.
    ###############################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    #x = flatten(x)  # shape: [batch_size, C x H x W]
    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 = flatten(x)
    scores = x.mm(fc_w)+fc_b

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

```
    return scores
```

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).

In [7]:

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

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

    # you must calculate the shape of the tensor after two conv lay
    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, con
    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*

In [8]:

```
def random_weight(shape):
    """
    Create random Tensors for weights; setting requires_grad=True me
    want to compute gradients for these Tensors during the backward
    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_
    # randn is standard normal distribution generator.
    w = torch.randn(shape, device=device, dtype=dtype) * np.sqrt(2.
    w.requires_grad = True
    return w

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

# 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[8]:

```
tensor([[ 0.1471,  1.2627,  0.6928,  0.9917, -1.1141],
        [ 1.0998,  1.3971,  0.5445,  0.1644, -0.2566],
        [ 0.0222, -0.1655,  0.6048,  0.6477,  0.345
6]], 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

In [9]:

```
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 mo
      with the signature scores = model_fn(x, params)
    - params: List of PyTorch Tensors giving parameters of the mode

    Returns: 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,
            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_sa
    return 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](https://pytorch.org/docs/stable/nn.functional.html#torch.nn.functional.cross_entropy) (https://pytorch.org/docs/stable/nn.functional.html#torch.nn.functional.cross_entropy).

In [10]:

```
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
      It should have the signature scores = model_fn(x, params) whe
      PyTorch Tensor of image data, params is a list of PyTorch Ten
      model weights, and scores is a PyTorch Tensor of shape (N, C)
      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

    Returns: The accuracy of the model
    """
    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 c
        # graph has requires_grad=True and uses backpropagation to
        # gradient of the loss with respect to these Tensors, and s
        # gradients in the .grad attribute of each Tensor.
        loss.backward()

        # Update parameters. We don't want to backpropagate through
        # parameter updates, so we scope the updates under a torch.
        # context manager to prevent a computational graph from bei
        with torch.no_grad():
            for w in params:
                w -= learning_rate * w.grad

                # Manually zero the gradients after running the bac
                w.grad.zero_()

        if (t + 1) % print_every == 0:
            print('Iteration %d, loss = %.4f' % (t + 1, loss.item()
```

```
        check_accuracy_part2(loader_val, model_fn, params)
        print()
return check_accuracy_part2(loader_val, model_fn, params)
```

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 100-dimensional vector that represents the probability distribution over 100 classes.

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

In [11]:

```
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 100, loss = 4.1706
Checking accuracy on the val set
Got 86 / 1000 correct (8.60%)

Iteration 200, loss = 3.9730
Checking accuracy on the val set
Got 111 / 1000 correct (11.10%)

Iteration 300, loss = 3.7108
Checking accuracy on the val set
Got 125 / 1000 correct (12.50%)

Iteration 400, loss = 3.5200
Checking accuracy on the val set
Got 126 / 1000 correct (12.60%)

Iteration 500, loss = 3.5430
Checking accuracy on the val set
Got 141 / 1000 correct (14.10%)

Iteration 600, loss = 3.6325
Checking accuracy on the val set
Got 167 / 1000 correct (16.70%)

Iteration 700, loss = 3.7352
Checking accuracy on the val set
Got 155 / 1000 correct (15.50%)

Checking accuracy on the val set
Got 155 / 1000 correct (15.50%)
```

Out[11]:

```
0.155
```

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**.

In [12]:

```
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*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)
```

```
Iteration 100, loss = 4.0941
Checking accuracy on the val set
Got 84 / 1000 correct (8.40%)

Iteration 200, loss = 4.1192
Checking accuracy on the val set
Got 106 / 1000 correct (10.60%)

Iteration 300, loss = 4.1601
Checking accuracy on the val set
Got 115 / 1000 correct (11.50%)

Iteration 400, loss = 4.0073
Checking accuracy on the val set
Got 124 / 1000 correct (12.40%)

Iteration 500, loss = 3.6056
Checking accuracy on the val set
Got 133 / 1000 correct (13.30%)

Iteration 600, loss = 3.6941
Checking accuracy on the val set
Got 138 / 1000 correct (13.80%)

Iteration 700, loss = 3.6938
Checking accuracy on the val set
Got 140 / 1000 correct (14.00%)

Checking accuracy on the val set
Got 150 / 1000 correct (15.00%)
```

Out[12]:

```
0.15
```

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

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

Module API: Two-Layer Network

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

In [13]:

```
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 method
        # 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 siz
    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 (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

In [14]:

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

        self.conv_1 = nn.Conv2d(in_channel, channel_1, (5,5), paddi
        nn.init.kaiming_normal_(self.conv_1.weight)
        self.conv_2 = nn.Conv2d(channel_1, channel_2, (3,3), paddin
        nn.init.kaiming_normal_(self.conv_2.weight)

        self.fc1 = nn.Linear(65536, num_classes)
        self.relu = nn.ReLU(inplace=True)



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

        x = self.relu(self.conv_1(x))
        x = self.relu(self.conv_2(x))
        x = flatten(x)
        scores = self.fc1(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
    model = ThreeLayerConvNet(in_channel=3, channel_1=32, channel_2
    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.

In [15]:

```
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,
            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_samp
    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.

In [16]:

```
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 epoc

    Returns: The accuracy of the model
    """
    model = model.to(device=device)  # move the model parameters to
    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,
            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
            # will update.
            optimizer.zero_grad()

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

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

            if (t + 1) % print_every == 0:
                print('Epoch %d, Iteration %d, loss = %.4f' % (e, t
                check_accuracy_part34(loader_val, model)
                print()
    return check_accuracy_part34(loader_val, model)
```

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 8% after training for one epoch.

In [17]:

```
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 100, loss = 4.8119
Checking accuracy on validation set
Got 20 / 1000 correct (2.00)

Epoch 0, Iteration 200, loss = 4.6708
Checking accuracy on validation set
Got 44 / 1000 correct (4.40)

Epoch 0, Iteration 300, loss = 4.6364
Checking accuracy on validation set
Got 54 / 1000 correct (5.40)

Epoch 0, Iteration 400, loss = 4.2028
Checking accuracy on validation set
Got 60 / 1000 correct (6.00)

Epoch 0, Iteration 500, loss = 4.1251
Checking accuracy on validation set
Got 63 / 1000 correct (6.30)

Epoch 0, Iteration 600, loss = 4.1690
Checking accuracy on validation set
Got 69 / 1000 correct (6.90)

Epoch 0, Iteration 700, loss = 4.3534
Checking accuracy on validation set
Got 81 / 1000 correct (8.10)

Checking accuracy on validation set
Got 83 / 1000 correct (8.30)
```

Out[17]:

```
0.083
```

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.

In [18]:

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

model = None
optimizer = None
################################################################################
# TODO: Instantiate your ThreeLayerConvNet model and a correspondin
################################################################################
# *****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=1)
```

```
Epoch 0, Iteration 100, loss = 4.0868
Checking accuracy on validation set
Got 76 / 1000 correct (7.60)

Epoch 0, Iteration 200, loss = 4.2056
Checking accuracy on validation set
Got 107 / 1000 correct (10.70)

Epoch 0, Iteration 300, loss = 3.8746
Checking accuracy on validation set
Got 129 / 1000 correct (12.90)

Epoch 0, Iteration 400, loss = 4.0477
Checking accuracy on validation set
Got 132 / 1000 correct (13.20)

Epoch 0, Iteration 500, loss = 3.6928
Checking accuracy on validation set
Got 139 / 1000 correct (13.90)

Epoch 0, Iteration 600, loss = 3.5634
Checking accuracy on validation set
Got 146 / 1000 correct (14.60)

Epoch 0, Iteration 700, loss = 3.3084
Checking accuracy on validation set
Got 164 / 1000 correct (16.40)

Checking accuracy on validation set
Got 166 / 1000 correct (16.60)
```

Out[18]:

```
0.166
```

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.

In [19]:

```
# We need to wrap `flatten` function in a module in order to stack
# 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 100, loss = 3.4964
Checking accuracy on validation set
Got 110 / 1000 correct (11.00)

Epoch 0, Iteration 200, loss = 4.2437
Checking accuracy on validation set
Got 127 / 1000 correct (12.70)

Epoch 0, Iteration 300, loss = 3.4850
Checking accuracy on validation set
Got 151 / 1000 correct (15.10)

Epoch 0, Iteration 400, loss = 3.6443
Checking accuracy on validation set
Got 157 / 1000 correct (15.70)

Epoch 0, Iteration 500, loss = 3.5981
Checking accuracy on validation set
Got 166 / 1000 correct (16.60)

Epoch 0, Iteration 600, loss = 3.3760
Checking accuracy on validation set
Got 185 / 1000 correct (18.50)

Epoch 0, Iteration 700, loss = 3.5016
Checking accuracy on validation set
Got 166 / 1000 correct (16.60)

Checking accuracy on validation set
Got 176 / 1000 correct (17.60)
```

Out[19]:

```
0.176
```

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 14% after one epoch** of training.

In [20]:

```
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 th
# 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, 100)
)

optimizer = optim.SGD(model.parameters(), lr = learning_rate, momen

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

train_part34(model, optimizer, epochs=1)
```

```
Epoch 0, Iteration 100, loss = 4.3139
Checking accuracy on validation set
Got 56 / 1000 correct (5.60)

Epoch 0, Iteration 200, loss = 3.9606
Checking accuracy on validation set
Got 87 / 1000 correct (8.70)

Epoch 0, Iteration 300, loss = 3.8940
Checking accuracy on validation set
Got 116 / 1000 correct (11.60)

Epoch 0, Iteration 400, loss = 3.7851
Checking accuracy on validation set
Got 134 / 1000 correct (13.40)

Epoch 0, Iteration 500, loss = 3.3700
Checking accuracy on validation set
Got 136 / 1000 correct (13.60)

Epoch 0, Iteration 600, loss = 3.6859
Checking accuracy on validation set
Got 150 / 1000 correct (15.00)

Epoch 0, Iteration 700, loss = 3.7708
Checking accuracy on validation set
Got 166 / 1000 correct (16.60)

Checking accuracy on validation set
Got 178 / 1000 correct (17.80)
```

Out[20]:

```
0.178
```

Part V. Resnet10 Implementation (35% of Grade)

In this section, you will use the tools introduced above to implement the Resnet architecture. The Resnet architecture was introduced in: [https://arxiv.org/pdf/1512.03385.pdf (https://arxiv.org/pdf/1512.03385.pdf)](https://arxiv.org/pdf/1512.03385.pdf) and it has become one of the most popular architectures used for computer vision. The key feature of the resnet architecture is the presence of skip connections which allow for better gradient flow even for very deep networks. Therefore, unlike vanilla CNNs introduced above, we can effectively build Resnets models having more than 100 layers. However, for the purposes of this exercise we will be using a smaller Resnet-10 architecture shown in the diagram below:

layer name	output size	layer
conv1	16 x 16	7 x 7, 64, stride 2
conv2_x	8 x 8	3 x 3, maxpool, stride 2
		3 x 3, 64 3 x 3, 64
conv3_x	8 x 8	3 x 3, 128 3 x 3, 128
conv4_x	8 x 8	3 x 3, 256 3 x 3, 256
conv5_x	4 x 4	3 x 3, 512 3 x 3, 512
	1 x 1	average pool, 100-c Softmax

In the architecture above, the downsampling is performed in conv5_1. We recommend using the adam optimzer for training Resnet. You should see about *45%* accuracy in 10 epochs. The template below is based on the Module API but you are allowed to use other Pytorch APIs if you prefer.

In [23]:

```
class ResBlock(nn.Module):
    def __init__(self, in_ch, out_ch, filters, stride = 1, batch_no
        super(ResBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_ch, out_ch, filters, padding = 1,
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_ch, out_ch, filters, padding = 1
        self.bn1 = nn.BatchNorm2d(out_ch)
        self.bn2 = nn.BatchNorm2d(out_ch)
        self.bn3 = nn.BatchNorm2d(out_ch)
        self.batch_norm = batch_norm
        self.shortcut = nn.Sequential()
        if stride != 1 or in_ch != out_ch:
            self.shortcut = nn.Conv2d(in_ch, out_ch, 1, padding = 0

    def forward(self, x):
        a = self.shortcut(x)
        if self.batch_norm:
            a = self.bn3(a)
        x = self.conv1(x)
        if self.batch_norm:
            x = self.bn1(x)
        x = self.relu(x)
        x = self.conv2(x)
        if self.batch_norm:
            x = self.bn2(x)
        x = x + a
        x = self.relu(x)
        return x



class ResNet(nn.Module):
    def __init__(self, batch_norm = False):
        super(ResNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size = 7, padding = 3,
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, paddin
        self.conv_2x = ResBlock(64, 64, 3, batch_norm = batch_norm)
        self.conv_3x = ResBlock(64, 128, 3, batch_norm = batch_norm
        self.conv_4x = ResBlock(128, 256, 3, batch_norm = batch_nor
        self.conv_5x = ResBlock(256, 512, 3, stride = 2, batch_norm
        self.avgpool = nn.AvgPool2d(kernel_size = 4)
```

```
        self.linear = nn.Linear(512, 100)
        self.relu = nn.ReLU(inplace = True)
        self.bn = nn.BatchNorm2d(64)
        self.batch_norm = batch_norm

    def forward(self, x):
        x = self.conv1(x)
        if self.batch_norm:
            x = self.bn(x)
        x = self.relu(x)
        x = self.maxpool(x)
        x = self.conv_2x(x)
        x = self.conv_3x(x)
        x = self.conv_4x(x)
        x = self.conv_5x(x)
        x = self.avgpool(x)
        x = flatten(x)
        x = self.linear(x)
        return x
```

In [24]:

```
learning_rate = 1e-3

model = None
optimizer = None

################################################################################
# TODO: Instantiate and train Resnet-10.
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
model = ResNet(batch_norm = False)

optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Train the model

print_every = 100
train_part34(model, optimizer, epochs=10)

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

```
Checking accuracy on validation set
Got 401 / 1000 correct (40.10)

Epoch 9, Iteration 600, loss = 1.2252
Checking accuracy on validation set
```

BatchNorm

Now you will also introduce the Batch-Normalization layer within the Resnet architecture implemented above. Please add a batch normalization layer after each conv in your network before applying the activation function (i.e. the order should be conv->BatchNorm->Relu). Please read the section 3.4 from the Resnet paper (https://arxiv.org/pdf/1512.03385.pdf (https://arxiv.org/pdf/1512.03385.pdf)).

Feel free to re-use the Resnet class that you have implemented above by introducing a boolean flag for batch normalization.

After trying out batch-norm, please discuss the performance comparison between Resnet with BatchNorm and without BatchNorm and possible reasons for why one performs better than the other.

In [25]:

```
learning_rate = 1e-3

model = None
optimizer = None

################################################################################
# TODO: Instantiate and train Resnet-10.
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
model = ResNet(batch_norm = True)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Train the model
print_every = 500
train_part34(model, optimizer, epochs=10)

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

```
Epoch 0, Iteration 500, loss = 3.5256
Checking accuracy on validation set
Got 190 / 1000 correct (19.00)

Epoch 1, Iteration 500, loss = 2.8393
Checking accuracy on validation set
Got 297 / 1000 correct (29.70)

Epoch 2, Iteration 500, loss = 2.3445
Checking accuracy on validation set
Got 384 / 1000 correct (38.40)

Epoch 3, Iteration 500, loss = 2.1642
Checking accuracy on validation set
Got 437 / 1000 correct (43.70)

Epoch 4, Iteration 500, loss = 1.7465
Checking accuracy on validation set
Got 474 / 1000 correct (47.40)

Epoch 5, Iteration 500, loss = 1.2117
Checking accuracy on validation set
Got 489 / 1000 correct (48.90)

Epoch 6, Iteration 500, loss = 1.4403
Checking accuracy on validation set
Got 490 / 1000 correct (49.00)

Epoch 7, Iteration 500, loss = 1.0983
Checking accuracy on validation set
Got 516 / 1000 correct (51.60)

Epoch 8, Iteration 500, loss = 0.5093
Checking accuracy on validation set
Got 504 / 1000 correct (50.40)

Epoch 9, Iteration 500, loss = 0.4026
Checking accuracy on validation set
Got 525 / 1000 correct (52.50)

Checking accuracy on validation set
Got 534 / 1000 correct (53.40)
```

Out[25]:

```
0.534
```

Discussion on BatchNorm

Batch Normalization tends to better predict the data in terms of accuracy. This is due to the fact that the Batch Normailzation helps in keeping the distribution of the input constant, by reducing internal covariate shift and helping in mitigating exploading or vanishing gradient problem. It also has an regularization effect on the model, thus reducing overfitting. Also, the dependence on the initial parameters of the model are reduced by doing batchnorm.

Batch Size

In this exercise, we will study the effect of batch size on performance of ResNet (with BatchNorm).

Specifically, you should try batch sizes of 32, 64 and 128 and describe the effect of varying batch size. You should print the validation accuracy of using each batch size in different rows.

After trying out different batch size, please discuss the effect of different batch sizes and possible reasons for that (either they are showing some trend or not).

In [31]:

```
print_every = 9999
batch_sizes = [32, 64, 128]
learning_rate = 1e-3
model = None
optimizer = None

################################################################################
# TODO: Try Resnet with different batch sizes. Hint: You will need
#  create a new dataloader with appropriate batch size for each exp
# You will also need to store the final accuracy for each experimen
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
for batches in batch_sizes:
    loader_train = DataLoader(cifar100_train, batch_size=batches, n
                              sampler=sampler.SubsetRandomSampler(r

    cifar100_val = dset.CIFAR100('./datasets/cifar100', train=True,
                                transform=transform)
    loader_val = DataLoader(cifar100_val, batch_size=batches, num_w
                            sampler=sampler.SubsetRandomSampler(ran

    cifar100_test = dset.CIFAR100('./datasets/cifar100', train=Fals
                                 transform=transform)
    loader_test = DataLoader(cifar100_test, batch_size=batches, num_
    model = ResNet(batch_norm = True)
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    # Train the model
    print('For Batch Size: ',batches)
    train_part34(model, optimizer, epochs=10)

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
################################################################################
#                                  END OF YOUR CODE
################################################################################
print_every = 100
```

```
Files already downloaded and verified
Files already downloaded and verified
For Batch Size:  32
Checking accuracy on validation set
Got 550 / 1000 correct (55.00)
Files already downloaded and verified
Files already downloaded and verified
For Batch Size:  64
Checking accuracy on validation set
Got 527 / 1000 correct (52.70)
Files already downloaded and verified
Files already downloaded and verified
For Batch Size:  128
Checking accuracy on validation set
Got 502 / 1000 correct (50.20)
```

Discuss effect of Batch Size

A batch size of 32 was performing better than batch sizes of 64 and 128. This might be due to the fact that a larger batch size of 64 and 128 may show a larger variablity between different batches thus making it hard for the model to converge.

Part VI. CIFAR-100 open-ended challenge (25% of Grade)

In this section, you can experiment with whatever ConvNet architecture you'd like on CIFAR-100 **except Resnet** because we already tried it.

Now it's your job to experiment with architectures, hyperparameters, loss functions, and optimizers to train a model that achieves **at least 48%** accuracy on the CIFAR-100 **validation** 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 (http://pytorch.org/docs/stable/nn.html)
- Activations: http://pytorch.org/docs/stable/nn.html#non-linear-activations (http://pytorch.org/docs/stable/nn.html#non-linear-activations)
- Loss functions: http://pytorch.org/docs/stable/nn.html#loss-functions (http://pytorch.org/docs/stable/nn.html#loss-functions)
- Optimizers: http://pytorch.org/docs/stable/optim.html (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](https://arxiv.org/abs/1512.00567) (https://arxiv.org/abs/1512.00567) (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
  - DenseNets (https://arxiv.org/abs/1608.06993) where inputs into previous layers are concatenated together.

In [24]:

```
class ConvBlock(nn.Module):
    def __init__(self, in_ch, out_ch, kernel_size = 3, padding = 1,
        super(ConvBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_ch, out_ch, kernel_size = kernel_
        self.relu = nn.ReLU()
        self.bn1 = nn.BatchNorm2d(out_ch)

    def forward(self, x):
        return self.bn1(self.relu(self.conv1(x)))

class IncBlock(nn.Module):
    def __init__(self, in_ch, out_1x1, red_3x3, out_3x3, red_5x5, o
        super(IncBlock, self).__init__()
        self.conv_1 = ConvBlock(in_ch, out_1x1, kernel_size = 1, pa
        self.conv_2 = nn.Sequential(
            ConvBlock(in_ch, red_3x3, kernel_size = 1, padding =0),
            ConvBlock(red_3x3, out_3x3, kernel_size = 3, stride = 1
        )
        self.conv_3 = nn.Sequential(
            ConvBlock(in_ch, red_5x5, kernel_size = 1, padding = 0)
            ConvBlock(red_5x5, out_5x5, kernel_size = 3, padding =
        )
        self.conv_4 = nn.Sequential(
            nn.MaxPool2d(kernel_size = 3, stride = 1, padding = 1),
            ConvBlock(in_ch, out_1x1pool, kernel_size = 1, padding
        )

    def forward(self, x):
        x = torch.cat([self.conv_1(x), self.conv_2(x), self.conv_3(
        return x

class IncNet(nn.Module):
    def __init__(self):
        super(IncNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size = 5, padding = 2,
        self.maxpool1 = nn.MaxPool2d(kernel_size = 3, padding = 1,
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU()

        self.conv2 = nn.Conv2d(64, 192, kernel_size = 3, padding =
        #self.maxpool2 = nn.MaxPool2d(kernel_size = 3, padding = 1,
        self.bn2 = nn.BatchNorm2d(192)
```

```
        self.inc_2 = nn.Sequential(IncBlock(192, 64, 96, 128, 16, 3
        self.inc_3 = nn.Sequential(IncBlock(256, 128, 128, 192, 32,

        self.maxpool3 = nn.MaxPool2d(kernel_size = 3, stride = 2, p

        self.inc_4 = nn.Sequential(IncBlock(480, 192, 96, 208, 16,
        self.inc_5 = nn.Sequential(IncBlock(512, 160, 112, 224, 24,

        self.avgpool = nn.AvgPool2d(kernel_size=3, stride=2)
        self.dropout1 = nn.Dropout(p = 0.2)
        self.fc1 = nn.Linear(4608, 100)
#         self.dropout2 = nn.Dropout(p = 0.2)
#         self.fc2 = nn.Linear(512, 100)

    def forward(self, x):
        x = self.conv1(x)
        x = self.maxpool1(x)
        x = self.relu(self.bn1(x))
        x = self.conv2(x)
        #x = self.maxpool2(x)
        x = self.relu(self.bn2(x))
        x = self.inc_2(x)
        x = self.inc_3(x)
        x = self.maxpool3(x)
        x = self.inc_4(x)
        x = self.inc_5(x)
        x = self.avgpool(x)
        x = flatten(x)
        x = self.dropout1(x)
        x = self.fc1(x)
#         x = self.dropout2(x)
#         x = self.fc2(x)
        return x
```

In [25]:

```
batch_size = 64
learning_rate = 1e-3

################################################################################
# TODO:
# Experiment with any architectures, optimizers, and hyperparameter
# Achieve AT LEAST 48% accuracy on the *validation set* within 10 e
#
# Note that you can use the check_accuracy function to evaluate on
# the test set or the validation set, by passing either loader_test
# loader_val as the second argument to check_accuracy. You should n
# the test set until you have finished your architecture and  hyper
# tuning, and only run the test set once at the end to report a fin
################################################################################
loader_train = DataLoader(cifar100_train, batch_size=batch_size, nu
                              sampler=sampler.SubsetRandomSampler(r

cifar100_val = dset.CIFAR100('./datasets/cifar100', train=True, dow
                               transform=transform)

loader_val = DataLoader(cifar100_val, batch_size=batch_size, num_wo
                            sampler=sampler.SubsetRandomSampler(ran

cifar100_test = dset.CIFAR100('./datasets/cifar100', train=False, d
                                transform=transform)

loader_test = DataLoader(cifar100_test, batch_size=batch_size, num_
model = None
optimizer = None

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
model = IncNet()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

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

# You should get at least 48% accuracy.
print_every = 600
train_part34(model, optimizer, epochs=10)
```

```
#print_every = 100
```

```
Files already downloaded and verified
Files already downloaded and verified
Epoch 0, Iteration 600, loss = 2.8793
Checking accuracy on validation set
Got 300 / 1000 correct (30.00)

Epoch 1, Iteration 600, loss = 2.1213
Checking accuracy on validation set
Got 432 / 1000 correct (43.20)

Epoch 2, Iteration 600, loss = 1.5602
Checking accuracy on validation set
Got 468 / 1000 correct (46.80)

Epoch 3, Iteration 600, loss = 1.5129
Checking accuracy on validation set
Got 515 / 1000 correct (51.50)

Epoch 4, Iteration 600, loss = 1.3944
Checking accuracy on validation set
Got 509 / 1000 correct (50.90)

Epoch 5, Iteration 600, loss = 1.1480
Checking accuracy on validation set
Got 532 / 1000 correct (53.20)

Epoch 6, Iteration 600, loss = 0.7440
Checking accuracy on validation set
Got 536 / 1000 correct (53.60)

Epoch 7, Iteration 600, loss = 0.7698
Checking accuracy on validation set
Got 545 / 1000 correct (54.50)

Epoch 8, Iteration 600, loss = 0.4932
Checking accuracy on validation set
Got 527 / 1000 correct (52.70)

Epoch 9, Iteration 600, loss = 0.4412
Checking accuracy on validation set
Got 544 / 1000 correct (54.40)
```

```
Checking accuracy on validation set
Got 535 / 1000 correct (53.50)
```

Out[35]:

```
0.535
```

Describe what you did (10% 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.

I tried to use a modified version of the InceptionNet (GoogleNet) for this exercise. I tried to modify the values of the layer sizes that could cater to the needs of this dataset which has a very small dimenision compared to the orginal dataset they used. I used 2x Inception Blocks (4 in total). The model is as follows:

```
    1) convulution layer - 1 reduce the dim of orginal image to half (16 x 16)
    2) Maxpooling - reduce the dim of orginal image to one fourth (8 x 8)
    3) convulution layer - 2
    4) Inception Block - 2a
    5) Inception Block - 2b
    6) Maxpooling - reduce the dim of orginal image to one eight (4 x 4)
    7) Inception Block - 3a
    8) Inception Block - 3b
    9) AvgPooling - reduce dim to (2 x 2)
    10) Dropout layer to avoid overfitting
    11) Fully connected layer to predict the output
```

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

In [26]:

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

```
Checking accuracy on test set
Got 5454 / 10000 correct (54.54)
```

Out[26]:

```
0.5454
```