# What's this PyTorch business?

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

For the last part of this assignment, though, we're going to leave behind your beautiful codebase and instead migrate to one of two popular deep learning frameworks: in this instance, PyTorch (or TensorFlow, if you switch over to that notebook).

### What is PyTorch?

PyTorch is a system for executing dynamic computational graphs over Tensor objects that

behave similarly as numpy ndarray. It comes with a powerful automatic differentiation engine that removes the need for manual back-propagation.

### Why?

Our code will now run on GPUs! Much faster training. When using a framework like PyTorch or TensorFlow you can harness the power of the GPU for your own custom neural network architectures without having to write CUDA code directly (which is beyond the scope of this class).

We want you to be ready to use one of these frameworks for your project so you can

experiment more efficiently than if you were writing every feature you want to use by hand.

We want you to stand on the shoulders of giants! TensorFlow and PyTorch are both

excellent frameworks that will make your lives a lot easier, and now that you understand their guts, you are free to use them :)

We want you to be exposed to the sort of deep learning code you might run into in academia or industry.

### PyTorch versions

This notebook assumes that you are using **PyTorch version 0.4**. Prior to this version, Tensors

had to be wrapped in Variable objects to be used in autograd; however Variables have now been deprecated. In addition 0.4 also separates a Tensor's datatype from its device, and uses numpy- style factories for constructing Tensors rather than directly invoking Tensor constructors.

How will I learn PyTorch?

Justin Johnson has made an excellent [tutorial](https://github.com/jcjohnson/pytorch-examples) for PyTorch.

You can also find the detailed [API doc](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/) is a much better place to ask than StackOverflow.

# Table of Contents

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

1. Preparation: we will use CIFAR-10 dataset.
2. Barebones PyTorch: we will work directly with the lowest-level PyTorch Tensors.
3. PyTorch Module API: we will use nn.Module to define arbitrary neural network architecture.
4. PyTorch Sequential API: we will use nn.Sequential to define a linear feed-forward network very conveniently.
5. CIFAR-10 open-ended challenge: please implement your own network to get as high

accuracy as possible on CIFAR-10. You can experiment with any layer, optimizer, hyperparameters or other advanced features.

Here is a table of comparison:

**API Flexibility Convenience**

|  |  |  |
| --- | --- | --- |
| Barebone | High | Low |
| nn.Module | High | Medium |
| nn.Sequential | Low | High |

# Part I. Preparation

First, we load the CIFAR-10 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-10 dataset, preprocess it, and iterate through it in minibatches; PyTorch provides convenient tools to automate this process for us.

In [1]:

**import** torch

**import** torch.nn **as** nn

**import** torch.optim **as** optim

**from** torch.utils.data **import** DataLoader

**from** torch.utils.data **import** sampler

**import** torchvision.datasets **as** dset

**import** torchvision.transforms **as** T

**import** numpy **as** np

In [2]:

NUM\_TRAIN **=** 49000

*# The torchvision.transforms package provides tools for preprocessing data # and for performing data augmentation; here we set up a transform to*

*# preprocess the data by subtracting the mean RGB value and dividing by the*

*# standard deviation of each RGB value; we've hardcoded the mean and std.*

transform **=** T**.**Compose([

T**.**ToTensor(),

T**.**Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))

])

*# We set up a Dataset object for each split (train / val / test); Datasets load # training examples one at a time, so we wrap each Dataset in a DataLoader which # iterates through the Dataset and forms minibatches. We divide the CIFAR-10*

*# training set into train and val sets by passing a Sampler object to the # DataLoader telling how it should sample from the underlying Dataset.*

cifar10\_train **=** dset**.**CIFAR10('./cs231n/datasets', train**=True**, download**=True**,

transform**=**transform)

loader\_train **=** DataLoader(cifar10\_train, batch\_size**=**64,

sampler**=**sampler**.**SubsetRandomSampler(range(NUM\_TRAIN)))

cifar10\_val **=** dset**.**CIFAR10('./cs231n/datasets', train**=True**, download**=True**, transform**=**transform)

loader\_val **=** DataLoader(cifar10\_val, batch\_size**=**64,

sampler**=**sampler**.**SubsetRandomSampler(range(NUM\_TRAIN, 50000))

cifar10\_test **=** dset**.**CIFAR10('./cs231n/datasets', train**=False**, download**=True**, transform**=**transform)

loader\_test **=** DataLoader(cifar10\_test, batch\_size**=**64)

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

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.

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

In [3]:

USE\_GPU **= True**

dtype **=** torch**.**float32 *# we will be using float throughout this tutorial*

**if** USE\_GPU **and** torch**.**cuda**.**is\_available(): device **=** torch**.**device('cuda')

**else**:

device **=** torch**.**device('cpu')

*# Constant to control how frequently we print train loss*

print\_every **=** 100

print('using device:', device)

using device: cuda

# Part II. Barebones PyTorch

PyTorch 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).

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 single vector per*

**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 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 5*

w1 **=** torch**.**zeros((50, hidden\_layer\_size), dtype**=**dtype) w2 **=** torch**.**zeros((hidden\_layer\_size, 10), dtype**=**dtype) scores **=** two\_layer\_fc(x, [w1, w2])

print(scores**.**size()) *# you should see [64, 10]*

two\_layer\_fc\_test()

torch.Size([64, 10])

### Barebones PyTorch: Three-Layer ConvNet

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.

**HINT**: For convolutions: <http://pytorch.org/docs/stable/nn.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 network with the architecture defined above.

Inputs:

* x: A PyTorch Tensor of shape (N, 3, H, W) giving a minibatch of images
* params: A list of PyTorch Tensors giving the weights and biases for the network; should contain the following:
  + conv\_w1: PyTorch Tensor of shape (channel\_1, 3, KH1, KW1) giving weights for the first convolutional layer
  + conv\_b1: PyTorch Tensor of shape (channel\_1,) giving biases for the first

convolutional layer

* + conv\_w2: PyTorch Tensor of shape (channel\_2, channel\_1, KH2, KW2) giving weights for the second convolutional layer
  + conv\_b2: PyTorch Tensor of shape (channel\_2,) giving biases for the second

convolutional layer

* + fc\_w: PyTorch Tensor giving weights for the fully-connected layer. Can you figure out what the shape should be?
  + fc\_b: PyTorch Tensor giving biases for the fully-connected layer. Can you

figure out what the shape should be?

Returns:

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

conv\_w1, conv\_b1, conv\_w2, conv\_b2, fc\_w, fc\_b **=** params scores **= None**

*################################################################################*

*# TODO: Implement the forward pass for the three-layer ConvNet. # ################################################################################*

conv1 **=** F**.**conv2d(x, weight**=**conv\_w1, bias**=**conv\_b1, padding**=**2) relu1 **=** F**.**relu(conv1)

conv2 **=** F**.**conv2d(relu1, weight**=**conv\_w2, bias**=**conv\_b2, padding**=**1)

relu2 **=** F**.**relu(conv2)

relu2\_flat **=** flatten(relu2)

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

*################################################################################ # 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, 10).

In [7]:

**def** three\_layer\_convnet\_test():

x **=** torch**.**zeros((64, 3, 32, 32), dtype**=**dtype) *# minibatch size 64, image size [*

conv\_w1 **=** torch**.**zeros((6, 3, 5, 5), dtype**=**dtype) *# [out\_channel, in\_channel, ke*

conv\_b1 **=** torch**.**zeros((6,)) *# out\_channel*

conv\_w2 **=** torch**.**zeros((9, 6, 3, 3), dtype**=**dtype) *# [out\_channel, in\_channel, ke*

conv\_b2 **=** torch**.**zeros((9,)) *# out\_channel*

*# you must calculate the shape of the tensor after two conv layers, before the f*

fc\_w **=** torch**.**zeros((9 **\*** 32 **\*** 32, 10)) fc\_b **=** torch**.**zeros(10)

scores **=** three\_layer\_convnet(x, [conv\_w1, conv\_b1, conv\_w2, conv\_b2, fc\_w, fc\_b] print(scores**.**size()) *# you should see [64, 10]*

three\_layer\_convnet\_test()

torch.Size([64, 10])

### Barebones PyTorch: Initialization

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>

In [8]:

**def** random\_weight(shape): """

Create random Tensors for weights; setting requires\_grad=True means that we

want to compute gradients for these Tensors during the backward pass. We use Kaiming normalization: sqrt(2 / fan\_in)

"""

**if** len(shape) **==** 2: *# FC weight*

fan\_in **=** shape[0]

**else**:

fan\_in **=** np**.**prod(shape[1:]) *# conv weight [out\_channel, in\_channel, kH, kW] # randn is standard normal distribution generator.*

w **=** torch**.**randn(shape, device**=**device, dtype**=**dtype) **\*** np**.**sqrt(2. **/** fan\_in) w**.**requires\_grad **= True**

**return** w

**def** zero\_weight(shape):

**return** torch**.**zeros(shape, device**=**device, dtype**=**dtype, requires\_grad**=True**)

*# create a weight of shape [3 x 5]*

*# you should see the type `torch.cuda.FloatTensor` if you use GPU. # Otherwise it should be `torch.FloatTensor`*

random\_weight((3, 5))

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Out[8]: | tensor([[ 0.4123, | -0.3226, | 0.1429, | 0.3334, | 1.9862], |
|  | [ 0.3655, | 1.2438, | -1.0655, | 0.6376, | 1.3459], |
|  | [-0.1194, | -0.6314, | -0.5256, | 0.3198, | 0.5616]], 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.

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 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](http://pytorch.org/docs/stable/nn.html#cross-entropy).

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

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 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 40% 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, 10))

train\_part2(two\_layer\_fc, [w1, w2], learning\_rate)

Iteration 0, loss = 3.2817

Checking accuracy on the val set Got 125 / 1000 correct (12.50%)

Iteration 100, loss = 2.2560

Checking accuracy on the val set Got 320 / 1000 correct (32.00%)

Iteration 200, loss = 1.9334

Checking accuracy on the val set Got 360 / 1000 correct (36.00%)

Iteration 300, loss = 1.7549

Checking accuracy on the val set Got 334 / 1000 correct (33.40%)

Iteration 400, loss = 1.9019

Checking accuracy on the val set Got 402 / 1000 correct (40.20%)

Iteration 500, loss = 1.8647

Checking accuracy on the val set Got 425 / 1000 correct (42.50%)

Iteration 600, loss = 2.0816

Checking accuracy on the val set Got 409 / 1000 correct (40.90%)

Iteration 700, loss = 1.9150

Checking accuracy on the val set Got 416 / 1000 correct (41.60%)

### BareBones PyTorch: Training a ConvNet

In the below 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 10 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 42% 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. # ################################################################################*

conv\_w1 **=** random\_weight((channel\_1, 3, 5, 5)) conv\_b1 **=** zero\_weight((channel\_1,))

conv\_w2 **=** random\_weight((channel\_2, 32, 3, 3)) conv\_b2 **=** zero\_weight((channel\_2,))

fc\_w **=** random\_weight((channel\_2**\***32**\***32, 10)) fc\_b **=** zero\_weight((10,))

*################################################################################ # END OF YOUR CODE # ################################################################################*

params **=** [conv\_w1, conv\_b1, conv\_w2, conv\_b2, fc\_w, fc\_b] train\_part2(three\_layer\_convnet, params, learning\_rate)

Iteration 0, loss = 3.0587

Checking accuracy on the val set Got 96 / 1000 correct (9.60%)

Iteration 100, loss = 1.8284

Checking accuracy on the val set Got 348 / 1000 correct (34.80%)

Iteration 200, loss = 1.7841

Checking accuracy on the val set Got 389 / 1000 correct (38.90%)

Iteration 300, loss = 1.7071

Checking accuracy on the val set Got 425 / 1000 correct (42.50%)

Iteration 400, loss = 1.6384

Checking accuracy on the val set Got 415 / 1000 correct (41.50%)

Iteration 500, loss = 1.5265

Checking accuracy on the val set Got 438 / 1000 correct (43.80%)

Iteration 600, loss = 1.5509

Checking accuracy on the val set Got 465 / 1000 correct (46.50%)

Iteration 700, loss = 1.8430

Checking accuracy on the val set Got 476 / 1000 correct (47.60%)

# Part III. PyTorch Module API

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) 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) 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 methods #* [*http://pytorch.org/docs/master/nn.html#torch-nn-init*](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 dim*

model **=** TwoLayerFC(input\_size, 42, 10) scores **=** model(x)

print(scores**.**size()) *# you should see [64, 10]*

test\_TwoLayerFC()

torch.Size([64, 10])

### Module API: Three-Layer ConvNet

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.

In [14]:

**class** ThreeLayerConvNet(nn**.**Module):

**def** init (self, in\_channel, channel\_1, channel\_2, num\_classes): super()**.** init ()

*########################################################################*

*# TODO: Set up the layers you need for a three-layer ConvNet with the # # architecture defined above. #*

*########################################################################*

self**.**conv1 **=** nn**.**Conv2d(in\_channel, channel\_1, kernel\_size**=**5, padding**=**2, bias nn**.**init**.**kaiming\_normal\_(self**.**conv1**.**weight)

nn**.**init**.**constant\_(self**.**conv1**.**bias, 0)

self**.**conv2 **=** nn**.**Conv2d(channel\_1, channel\_2, kernel\_size**=**3, padding**=**1, bias**=** nn**.**init**.**kaiming\_normal\_(self**.**conv2**.**weight)

nn**.**init**.**constant\_(self**.**conv2**.**bias, 0)

self**.**fc **=** nn**.**Linear(channel\_2**\***32**\***32, num\_classes) nn**.**init**.**kaiming\_normal\_(self**.**fc**.**weight)

nn**.**init**.**constant\_(self**.**fc**.**bias, 0)

*######################################################################## # END OF YOUR CODE # ########################################################################*

**def** forward(self, x): scores **= None**

*########################################################################*

*# TODO: Implement the forward function for a 3-layer ConvNet. you # # should use the layers you defined in init and specify the # # connectivity of those layers in forward() # ########################################################################*

relu1 **=** F**.**relu(self**.**conv1(x))

relu2 **=** F**.**relu(self**.**conv2(relu1)) scores **=** self**.**fc(flatten(relu2))

*########################################################################*

*# END OF YOUR CODE # ########################################################################*

**return** scores

**def** test\_ThreeLayerConvNet():

x **=** torch**.**zeros((64, 3, 32, 32), dtype**=**dtype) *# minibatch size 64, image size [* model **=** ThreeLayerConvNet(in\_channel**=**3, channel\_1**=**12, channel\_2**=**8, num\_classes**=**1 scores **=** model(x)

print(scores**.**size()) *# you should see [64, 10]*

test\_ThreeLayerConvNet()

torch.Size([64, 10])

### 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, e.g. GPU*

y **=** y**.**to(device**=**device, dtype**=**torch**.**long) scores **=** model(x)

\_, preds **=** scores**.**max(1)

num\_correct **+=** (preds **==** y)**.**sum() num\_samples **+=** preds**.**size(0)

acc **=** float(num\_correct) **/** num\_samples

print('Got %d / %d correct (%.2f)' **%** (num\_correct, num\_samples, 100 **\*** acc))

### Module API: Training Loop

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 epochs to train for

Returns: Nothing, but prints model accuracies during training.

"""

model **=** model**.**to(device**=**device) *# move the model parameters to CPU/GPU*

**for** e **in** range(epochs):

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

model**.**train() *# put model to training mode*

x **=** x**.**to(device**=**device, dtype**=**dtype) *# move to device, e.g. GPU*

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

scores **=** model(x)

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

*# Zero out all of the gradients for the variables which the optimizer # will update.*

optimizer**.**zero\_grad()

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

loss**.**backward()

*# Actually update the parameters of the model using the gradients # computed by the backwards pass.*

optimizer**.**step()

**if** t **%** print\_every **==** 0:

print('Iteration %d, loss = %.4f' **%** (t, loss**.**item())) check\_accuracy\_part34(loader\_val, model)

print()

### 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 40% after training for one epoch.

In [17]:

hidden\_layer\_size **=** 4000 learning\_rate **=** 1e-2

model **=** TwoLayerFC(3 **\*** 32 **\*** 32, hidden\_layer\_size, 10)

optimizer **=** optim**.**SGD(model**.**parameters(), lr**=**learning\_rate)

train\_part34(model, optimizer)

Iteration 0, loss = 3.9216

Checking accuracy on validation set Got 147 / 1000 correct (14.70)

Iteration 100, loss = 2.6885

Checking accuracy on validation set Got 332 / 1000 correct (33.20)

Iteration 200, loss = 1.9235

Checking accuracy on validation set Got 377 / 1000 correct (37.70)

Iteration 300, loss = 1.8133

Checking accuracy on validation set Got 425 / 1000 correct (42.50)

Iteration 400, loss = 1.3607

Checking accuracy on validation set Got 429 / 1000 correct (42.90)

Iteration 500, loss = 2.1195

Checking accuracy on validation set Got 388 / 1000 correct (38.80)

Iteration 600, loss = 1.5725

Checking accuracy on validation set Got 434 / 1000 correct (43.40)

Iteration 700, loss = 1.5672

Checking accuracy on validation set Got 455 / 1000 correct (45.50)

### 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 above above 45% after training for one epoch.

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

In [18]:

learning\_rate **=** 3e-3 channel\_1 **=** 32

channel\_2 **=** 16

model **= None**

optimizer **= None**

*################################################################################*

*# TODO: Instantiate your ThreeLayerConvNet model and a corresponding optimizer # ################################################################################*

model **=** ThreeLayerConvNet(3, channel\_1, channel\_2, 10)

optimizer **=** optim**.**SGD(model**.**parameters(), lr**=**learning\_rate)

*################################################################################ # END OF YOUR CODE*

*################################################################################*

train\_part34(model, optimizer)

Iteration 0, loss = 4.1012

Checking accuracy on validation set Got 101 / 1000 correct (10.10)

Iteration 100, loss = 1.7691

Checking accuracy on validation set Got 347 / 1000 correct (34.70)

Iteration 200, loss = 1.8306

Checking accuracy on validation set Got 397 / 1000 correct (39.70)

Iteration 300, loss = 1.8571

Checking accuracy on validation set Got 409 / 1000 correct (40.90)

Iteration 400, loss = 1.5962

Checking accuracy on validation set Got 433 / 1000 correct (43.30)

Iteration 500, loss = 1.5841

Checking accuracy on validation set Got 444 / 1000 correct (44.40)

Iteration 600, loss = 1.6299

Checking accuracy on validation set Got 458 / 1000 correct (45.80)

Iteration 700, loss = 1.3607

Checking accuracy on validation set Got 476 / 1000 correct (47.60)

# Part IV. PyTorch Sequential API

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 40% accuracy after one epoch of training.

In [19]:

*# We need to wrap `flatten` function in a module in order to stack it # in nn.Sequential*

**class** Flatten(nn**.**Module): **def** forward(self, x): **return** flatten(x)

hidden\_layer\_size **=** 4000 learning\_rate **=** 1e-2

model **=** nn**.**Sequential( Flatten(),

nn**.**Linear(3 **\*** 32 **\*** 32, hidden\_layer\_size), nn**.**ReLU(),

nn**.**Linear(hidden\_layer\_size, 10),

)

*# you can use Nesterov momentum in optim.SGD*

optimizer **=** optim**.**SGD(model**.**parameters(), lr**=**learning\_rate,

momentum**=**0.9, nesterov**=True**)

train\_part34(model, optimizer)

Iteration 0, loss = 2.3569

Checking accuracy on validation set Got 159 / 1000 correct (15.90)

Iteration 100, loss = 1.9769

Checking accuracy on validation set Got 400 / 1000 correct (40.00)

Iteration 200, loss = 1.5860

Checking accuracy on validation set Got 400 / 1000 correct (40.00)

Iteration 300, loss = 1.7836

Checking accuracy on validation set Got 404 / 1000 correct (40.40)

Iteration 400, loss = 1.6260

Checking accuracy on validation set Got 448 / 1000 correct (44.80)

Iteration 500, loss = 1.7615

Checking accuracy on validation set Got 445 / 1000 correct (44.50)

Iteration 600, loss = 1.7117

Checking accuracy on validation set Got 446 / 1000 correct (44.60)

Iteration 700, loss = 1.8356

Checking accuracy on validation set Got 404 / 1000 correct (40.40)

### 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 10 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 55% after one epoch of training.

In [20]:

channel\_1 **=** 32

channel\_2 **=** 16

learning\_rate **=** 1e-2

model **= None**

optimizer **= None**

*################################################################################*

*# TODO: Rewrite the 3-layer ConvNet with bias from Part III with the # # Sequential API. #*

*################################################################################*

model **=** nn**.**Sequential(

nn**.**Conv2d(3, channel\_1, kernel\_size**=**5, padding**=**2), nn**.**ReLU(),

nn**.**Conv2d(channel\_1, channel\_2, kernel\_size**=**3, padding**=**1), nn**.**ReLU(),

Flatten(),

nn**.**Linear(channel\_2**\***32**\***32, 10),

)

optimizer **=** optim**.**SGD(model**.**parameters(), lr**=**learning\_rate,

momentum**=**0.9, nesterov**=True**)

*# Weight initialization*

*# Ref:* [*http://pytorch.org/docs/stable/nn.html#torch.nn.Module.apply*](http://pytorch.org/docs/stable/nn.html#torch.nn.Module.apply)

**def** init\_weights(m):

*# print(m)*

**if** type(m) **==** nn**.**Conv2d **or** type(m) **==** nn**.**Linear: random\_weight(m**.**weight**.**size())

zero\_weight(m**.**bias**.**size()) model**.**apply(init\_weights)

*################################################################################*

*# END OF YOUR CODE*

*################################################################################*

train\_part34(model, optimizer)

Iteration 0, loss = 2.3153

Checking accuracy on validation set Got 101 / 1000 correct (10.10)

Iteration 100, loss = 1.5728

Checking accuracy on validation set Got 449 / 1000 correct (44.90)

Iteration 200, loss = 1.4987

Checking accuracy on validation set Got 501 / 1000 correct (50.10)

Iteration 300, loss = 1.2147

Checking accuracy on validation set Got 517 / 1000 correct (51.70)

Iteration 400, loss = 1.1304

Checking accuracy on validation set Got 501 / 1000 correct (50.10)

Iteration 500, loss = 1.1010

Checking accuracy on validation set Got 544 / 1000 correct (54.40)

Iteration 600, loss = 1.3214

Checking accuracy on validation set Got 581 / 1000 correct (58.10)

Iteration 700, loss = 1.4248

Checking accuracy on validation set

Got 572 / 1000 correct (57.20)

# Part V. CIFAR-10 open-ended challenge

In this section, you can experiment with whatever ConvNet architecture you'd like on CIFAR-10.

Now it's your job to experiment with architectures, hyperparameters, loss functions, and optimizers to train a model that achieves **at least 70%** accuracy on the CIFAR-10 **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. One note: what we call in the class "spatial batch norm" is called "BatchNorm2D" in PyTorch.

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?

**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?

**Network architecture**: The network above has two layers of trainable parameters. 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) (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.

### Going above and beyond

If you are feeling adventurous there are many other features you can implement to try and improve your performance. You are **not required** to implement any of these, but don't miss the fun if you have time!

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](https://arxiv.org/abs/1512.03385) where the input from the previous layer is added to the output. [DenseNets](https://arxiv.org/abs/1608.06993) where inputs into previous layers are concatenated together. [This blog has an in-depth overview](https://chatbotslife.com/resnets-highwaynets-and-densenets-oh-my-9bb15918ee32)

### Have fun and happy training!

In [21]: *################################################################################ # 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**

*# A 4-layer convolutional network*

*# (conv -> batchnorm -> relu -> maxpool) \* 3 -> fc*

layer1 **=** nn**.**Sequential(

nn**.**Conv2d(3, 16, kernel\_size**=**5, padding**=**2), nn**.**BatchNorm2d(16),

nn**.**ReLU(),

nn**.**MaxPool2d(2)

)

layer2 **=** nn**.**Sequential(

nn**.**Conv2d(16, 32, kernel\_size**=**3, padding**=**1), nn**.**BatchNorm2d(32),

nn**.**ReLU(),

nn**.**MaxPool2d(2)

)

layer3 **=** nn**.**Sequential(

nn**.**Conv2d(32, 64, kernel\_size**=**3, padding**=**1), nn**.**BatchNorm2d(64),

nn**.**ReLU(),

nn**.**MaxPool2d(2)

)

layer4 **=** nn**.**Sequential( nn**.**Dropout(0.3),

nn**.**Linear(64**\***4**\***4, 10), nn**.**ReLU(),

nn**.**Linear(10,10)

)

model **=** nn**.**Sequential( layer1,

layer2, layer3,

Flatten(), layer4

)

learning\_rate **=** 1.2e-3

optimizer **=** optim**.**Adam(model**.**parameters(), lr**=**learning\_rate)

*# Print training status every epoch: set print\_every to a large number*

print\_every **=** 10000

*################################################################################ # END OF YOUR CODE*

*################################################################################*

*# You should get at least 70% accuracy*

train\_part34(model, optimizer, epochs**=**10)

Iteration 0, loss = 2.3417

Checking accuracy on validation set Got 87 / 1000 correct (8.70)

Iteration 0, loss = 1.1359

Checking accuracy on validation set Got 546 / 1000 correct (54.60)

Iteration 0, loss = 0.8376

Checking accuracy on validation set Got 638 / 1000 correct (63.80)

Iteration 0, loss = 0.9256

Checking accuracy on validation set Got 665 / 1000 correct (66.50)

Iteration 0, loss = 0.9029

Checking accuracy on validation set Got 669 / 1000 correct (66.90)

Iteration 0, loss = 1.0565

Checking accuracy on validation set Got 686 / 1000 correct (68.60)

Iteration 0, loss = 0.7262

Checking accuracy on validation set Got 677 / 1000 correct (67.70)

Iteration 0, loss = 0.7773

Checking accuracy on validation set

Got 699 / 1000 correct (69.90)

Iteration 0, loss = 0.8339

Checking accuracy on validation set Got 706 / 1000 correct (70.60)

Iteration 0, loss = 0.9034

Checking accuracy on validation set Got 738 / 1000 correct (73.80)

Describe what you did

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.

TODO: Describe what you did

I firstly understood the network and did some tests about the network, finally I managed to increase the test accuracy to 73.08%. For this increase, I increased my layer amount, that can be seen above as "layer 4". I increased my learning rate. After the convolutional layers, I added a dropout layer to avoid the overfitting, which also gave me the chance to increase the learning rate since the chance of overfitting is decreased. Lastly, I put a ReLU layer to relatively increase the complex learning of the network.

## 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.

In [22]:

best\_model **=** model

check\_accuracy\_part34(loader\_test, best\_model)

Checking accuracy on test set

Got 7308 / 10000 correct (73.08)

In [ ]: