Assignment 3: Convolutional Neural Networks with Pytorch

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

What is PyTorch?

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

Why?

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

PyTorch versions

This notebook assumes that you are using **PyTorch version >=1.0**. For example, recent `torch==1.13.0` is a good option to go.

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

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

Table of Contents

This assignment has 6 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: please implement the specific ResNet-10 architecture provided in this assignment and play around with it.
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 previous parts of the assignment we had to write our own code to download the CIFAR-20 dataset, preprocess it, and iterate through it in minibatches; PyTorch provides convenient tools to automate this process for us.

In [ ]:
```
# Add official website of pytorch

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.utils.data import sampler

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

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

In [ ]:
```
NUM_TRAIN = 49000
batch_size= 64

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

#==========================================================================#
# You should try changing the transform for the training data to include    #
```

```
# data augmentation such as RandmCrop and HorizontalFlip                  #
# when running the final part of the notebook where you have to achieve   #
# as high accuracy as possible on CIFAR-100.                              #
# Of course you will have to re-run this block for the effect to take place #
#==========================================================================#
train_transform = transform = T.Compose([
                T.ToTensor(),
                T.Normalize((0.5071, 0.4867, 0.4408), (0.2675, 0.2565, 0.2761))
            ])

# We set up a Dataset object for each split (train / val / test); Datasets load
# training examples one at a time, so we wrap each Dataset in a DataLoader which
# iterates through the Dataset and forms mini-batches. We divide the CIFAR-100
# training set into train and val sets by passing a Sampler object to the
# DataLoader telling how it should sample from the underlying Dataset.
cifar100_train = dset.CIFAR100('./datasets/cifar100', train=True, download=True,
                             transform=train_transform)
loader_train = DataLoader(cifar100_train, batch_size=batch_size, num_workers=2,
                          sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN)))

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

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

```
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. **You can run on GPU on datahub.**

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

In [ ]:

```
USE_GPU = True
num_class = 100
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 (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 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 [ ]:

```
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 pe

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 [ ]:

```python
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
    w1 = torch.zeros((50, hidden_layer_size), dtype=dtype)
    w2 = torch.zeros((hidden_layer_size, num_class), dtype=dtype)
    scores = two_layer_fc(x, [w1, w2])
    print(scores.size())  # you should see [64, 100]

two_layer_fc_test()
```

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

Barebones PyTorch: Three-Layer ConvNet

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

1. A convolutional layer (with bias) with `channel_1` filters, each with shape `KW1 x KH1`, and zero-padding of two
2. ReLU nonlinearity
3. A convolutional layer (with bias) with `channel_2` filters, each with shape `KW2 x KH2`, and zero-padding of one
4. ReLU nonlinearity
5. Fully-connected layer with bias, producing scores for C classes.

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

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

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

    Inputs:
    - x: A PyTorch Tensor of shape (N, 3, H, W) giving a minibatch of images
    - params: A list of PyTorch Tensors giving the weights and biases for the
      network; should contain the following:
      - conv_w1: PyTorch Tensor of shape (channel_1, 3, KH1, KW1) giving weights
        for the first convolutional layer
      - conv_b1: PyTorch Tensor of shape (channel_1,) giving biases for the first
        convolutional layer
      - conv_w2: PyTorch Tensor of shape (channel_2, channel_1, KH2, KW2) giving
        weights for the second convolutional layer
      - conv_b2: PyTorch Tensor of shape (channel_2,) giving biases for the second
        convolutional layer
      - fc_w: PyTorch Tensor giving weights for the fully-connected layer. Can you
        figure out what the shape should be?
      - fc_b: PyTorch Tensor giving biases for the fully-connected layer. Can you
        figure out what the shape should be?

    Returns:
```

```
    - scores: PyTorch Tensor of shape (N, C) giving classification scores for x
    """
    conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b = params
    scores = None
    ################################################################################
    # TODO: Implement the forward pass for the three-layer ConvNet.
    ################################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    x = F.conv2d(x, conv_w1, conv_b1, padding = 2)
    x = F.relu(x)
    x = F.conv2d(x, conv_w2, conv_b2, padding = 1)
    x = F.relu(x)
    x = flatten(x)
    scores = F.linear(x, fc_w.T, fc_b)

    # pass

    # *****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 [ ]:

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

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

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

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

Barebones PyTorch: Initialization

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

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

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

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

In [ ]:
```
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[ ]:
```
tensor([[ 0.1796, -0.5505, -0.0331,  0.1412,  0.2072],
        [ 0.2549,  0.5971, -0.1515,  0.4717, -1.0586],
        [-0.5405, -1.0834,  0.0437, -0.2138,  1.3759]], 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 [ ]:
```
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 * ac
```

BareBones PyTorch: Training Loop

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

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

In [ ]:
```
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 100-dimensional vector that represents the probability distribution over 100 classes.

You don't need to tune any hyperparameters, but you should see accuracies around 10% after training for one epoch.

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

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

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

```
Iteration 0, loss = 5.1590
Checking accuracy on the val set
Got 9 / 1000 correct (0.90%)

Iteration 100, loss = 4.3604
Checking accuracy on the val set
Got 88 / 1000 correct (8.80%)

Iteration 200, loss = 4.1589
Checking accuracy on the val set
Got 134 / 1000 correct (13.40%)

Iteration 300, loss = 3.7634
Checking accuracy on the val set
Got 126 / 1000 correct (12.60%)

Iteration 400, loss = 3.8531
Checking accuracy on the val set
Got 136 / 1000 correct (13.60%)

Iteration 500, loss = 3.6066
Checking accuracy on the val set
Got 159 / 1000 correct (15.90%)

Iteration 600, loss = 3.7873
Checking accuracy on the val set
Got 160 / 1000 correct (16.00%)

Iteration 700, loss = 3.5100
Checking accuracy on the val set
Got 170 / 1000 correct (17.00%)
```

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 you should see accuracies around 10% after training for one epoch.

In [ ]:
```
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 0, loss = 5.4948
Checking accuracy on the val set
Got 9 / 1000 correct (0.90%)

Iteration 100, loss = 4.3154
Checking accuracy on the val set
Got 65 / 1000 correct (6.50%)

Iteration 200, loss = 4.0879
Checking accuracy on the val set
Got 85 / 1000 correct (8.50%)

Iteration 300, loss = 4.0361
Checking accuracy on the val set
Got 117 / 1000 correct (11.70%)

Iteration 400, loss = 3.8655
Checking accuracy on the val set
Got 129 / 1000 correct (12.90%)

Iteration 500, loss = 3.7732
Checking accuracy on the val set
Got 122 / 1000 correct (12.20%)

Iteration 600, loss = 3.7655
Checking accuracy on the val set
Got 131 / 1000 correct (13.10%)

Iteration 700, loss = 3.6661
Checking accuracy on the val set
Got 148 / 1000 correct (14.80%)
```

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

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

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

To use the Module API, follow the steps below:

1. Subclass `nn.Module`. Give your network class an intuitive name like `TwoLayerFC`.
2. In the constructor `__init__()`, define all the layers you need as class attributes. Layer objects like `nn.Linear` and `nn.Conv2d` are themselves `nn.Module` subclasses and contain learnable parameters, so that you don't have to instantiate the raw tensors yourself. `nn.Module` will track these internal parameters for you. Refer to the doc to learn more about the dozens of builtin layers. **Warning**: don't forget to call the `super().__init__()` first!

3. In the `forward()` method, define the *connectivity* of your network. You should use the attributes defined in `__init__` as function calls that take tensor as input and output the "transformed" tensor. Do *not* create any new layers with learnable parameters in `forward()`! All of them must be declared upfront in `__init__`.

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

Module API: Two-Layer Network

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

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

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

def test_TwoLayerFC():
    input_size = 50
    x = torch.zeros((64, input_size), dtype=dtype)  # minibatch size 64, feature d
    model = TwoLayerFC(input_size, 42, num_class)
    scores = model(x)
    print(scores.size())  # you should see [64, 100]
test_TwoLayerFC()
```

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

Module API: Three-Layer ConvNet

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

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

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

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

After you implement the three-layer ConvNet, the `test_ThreeLayerConvNet` function will run your implementation; it should print `(64, 100)` for the shape of the output scores.

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

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

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



    def forward(self, x):
        scores = None
        ########################################################################
        # TODO: Implement the forward function for a 3-layer ConvNet. you     #
        # should use the layers you defined in __init__ and specify the       #
        # connectivity of those layers in forward()                           #
        ########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        x = self.conv_1(x)
        x = self.relu(x)
        x = self.conv_2(x)
        x = self.relu(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 64, image size
    model = ThreeLayerConvNet(in_channel=3, channel_1=32, channel_2=64, num_classe
    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 [ ]:
```
def check_accuracy_part34(loader, model):
    if loader.dataset.train:
        print('Checking accuracy on validation set')
    else:
        print('Checking accuracy on test set')
    num_correct = 0
```

```
    num_samples = 0
    model.eval()  # set model to evaluation mode
    with torch.no_grad():
        for x, y in loader:
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.long)
            scores = model(x)
            _, preds = scores.max(1)
            num_correct += (preds == y).sum()
            num_samples += preds.size(0)
        acc = float(num_correct) / num_samples
        print('Got %d / %d correct (%.2f)' % (num_correct, num_samples, 100 * acc))
        return acc
```

Module API: Training Loop

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

In [ ]:
```
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
    val_acc = []
    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 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('Epoch %d, Iteration %d, loss = %.4f' % (e, t, loss.item()))
                acc = check_accuracy_part34(loader_val, model)
                val_acc.append(acc)
                print()
    try:
```

```
        return val_acc
    except:
        pass
```

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 increasing during the training.

In [ ]:
```
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)
pass
```

```
Epoch 0, Iteration 0, loss = 5.4337
Checking accuracy on validation set
Got 18 / 1000 correct (1.80)

Epoch 0, Iteration 100, loss = 4.4669
Checking accuracy on validation set
Got 29 / 1000 correct (2.90)

Epoch 0, Iteration 200, loss = 4.3393
Checking accuracy on validation set
Got 50 / 1000 correct (5.00)

Epoch 0, Iteration 300, loss = 4.3237
Checking accuracy on validation set
Got 64 / 1000 correct (6.40)

Epoch 0, Iteration 400, loss = 4.2723
Checking accuracy on validation set
Got 72 / 1000 correct (7.20)

Epoch 0, Iteration 500, loss = 4.3540
Checking accuracy on validation set
Got 85 / 1000 correct (8.50)

Epoch 0, Iteration 600, loss = 4.0547
Checking accuracy on validation set
Got 92 / 1000 correct (9.20)

Epoch 0, Iteration 700, loss = 4.1615
Checking accuracy on validation set
Got 96 / 1000 correct (9.60)
```

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 better accuracy than the previous Two-Layer Network.

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

In [ ]:
```
learning_rate = 1e-3
channel_1 = 32
channel_2 = 64

model = None
optimizer = None
################################################################################
# TODO: Instantiate your ThreeLayerConvNet model and a corresponding optimizer #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
model = ThreeLayerConvNet(in_channel = 3, channel_1 = channel_1, channel_2 = chann
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)
pass
```

```
Epoch 0, Iteration 0, loss = 4.6908
Checking accuracy on validation set
Got 8 / 1000 correct (0.80)

Epoch 0, Iteration 100, loss = 4.0920
Checking accuracy on validation set
Got 73 / 1000 correct (7.30)

Epoch 0, Iteration 200, loss = 4.1153
Checking accuracy on validation set
Got 120 / 1000 correct (12.00)

Epoch 0, Iteration 300, loss = 3.7684
Checking accuracy on validation set
Got 123 / 1000 correct (12.30)

Epoch 0, Iteration 400, loss = 3.7082
Checking accuracy on validation set
Got 138 / 1000 correct (13.80)

Epoch 0, Iteration 500, loss = 3.4286
Checking accuracy on validation set
Got 145 / 1000 correct (14.50)

Epoch 0, Iteration 600, loss = 3.4506
Checking accuracy on validation set
Got 165 / 1000 correct (16.50)

Epoch 0, Iteration 700, loss = 3.7247
Checking accuracy on validation set
Got 163 / 1000 correct (16.30)
```

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, but you should see accuracies above 10% after training for one epoch.

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

hidden_layer_size = 4000
learning_rate = 1e-2

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

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

_ = train_part34(model, optimizer)
pass
```

```
Epoch 0, Iteration 0, loss = 4.6438
Checking accuracy on validation set
Got 10 / 1000 correct (1.00)

Epoch 0, Iteration 100, loss = 4.0299
Checking accuracy on validation set
Got 105 / 1000 correct (10.50)

Epoch 0, Iteration 200, loss = 3.6837
Checking accuracy on validation set
Got 125 / 1000 correct (12.50)

Epoch 0, Iteration 300, loss = 3.3482
Checking accuracy on validation set
Got 140 / 1000 correct (14.00)

Epoch 0, Iteration 400, loss = 3.6053
Checking accuracy on validation set
Got 157 / 1000 correct (15.70)

Epoch 0, Iteration 500, loss = 3.5416
Checking accuracy on validation set
Got 167 / 1000 correct (16.70)

Epoch 0, Iteration 600, loss = 3.3983
Checking accuracy on validation set
Got 183 / 1000 correct (18.30)

Epoch 0, Iteration 700, loss = 3.9882
Checking accuracy on validation set
Got 165 / 1000 correct (16.50)
```

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 accuracies above 12% after training for one epoch.

In [ ]:
```
channel_1 = 32
channel_2 = 16
learning_rate = 1e-3

model = None
```

```
optimizer = None

################################################################################
# TODO: Rewrite the 2-layer ConvNet with bias from Part III with the           #
# Sequential API.                                                              #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
model = nn.Sequential(
    nn.Conv2d(3, channel_1, (5, 5), padding = 2),
    nn.ReLU(inplace=True),
    nn.Conv2d(channel_1, channel_2, (3,3), padding = 1),
    nn.ReLU(inplace = True),
    Flatten(),
    nn.Linear(channel_2 * 32 * 32, 100)
)
optimizer = optim.SGD(model.parameters(), lr=learning_rate,
                      momentum=0.9, nesterov=True)
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
################################################################################
#                                 END OF YOUR CODE
################################################################################

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

```
Epoch 0, Iteration 0, loss = 4.6209
Checking accuracy on validation set
Got 16 / 1000 correct (1.60)

Epoch 0, Iteration 100, loss = 4.3256
Checking accuracy on validation set
Got 62 / 1000 correct (6.20)

Epoch 0, Iteration 200, loss = 3.8887
Checking accuracy on validation set
Got 95 / 1000 correct (9.50)

Epoch 0, Iteration 300, loss = 3.7616
Checking accuracy on validation set
Got 122 / 1000 correct (12.20)

Epoch 0, Iteration 400, loss = 3.7505
Checking accuracy on validation set
Got 141 / 1000 correct (14.10)

Epoch 0, Iteration 500, loss = 3.6265
Checking accuracy on validation set
Got 157 / 1000 correct (15.70)

Epoch 0, Iteration 600, loss = 3.7126
Checking accuracy on validation set
Got 163 / 1000 correct (16.30)

Epoch 0, Iteration 700, loss = 3.4906
Checking accuracy on validation set
Got 184 / 1000 correct (18.40)
```

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 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:

model.png

In the architecture above, the down-sampling is performed in conv5_1. We recommend using the adam optimizer 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 [ ]:

```
########################################################################
# TODO: Implement the forward function for the Resnet specified        #
# above. HINT: You might need to create a helper class to              #
# define a Resnet block and then use that block here to create         #
# the resnet layers i.e. conv2_x, conv3_x, conv4_x and conv5_x         #
########################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

def conv(in_channels, out_channels, stride=1):
    return nn.Conv2d(in_channels, out_channels, kernel_size=(3, 3),
                     stride=stride, padding=1, bias=False)

# Residual block
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, bn, stride=1, downsample=None):
        super(ResidualBlock, self).__init__()
        self.bn = bn
        self.conv1 = conv(in_channels, out_channels, stride)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv(out_channels, out_channels)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = downsample

    def forward(self, x):
        residual = x
        out = self.conv1(x)
        if self.bn:
            out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        if self.bn:
            out = self.bn2(out)
        if self.downsample:
            residual = self.downsample(x)
        out += residual
        out = self.relu(out)
        return out

# ResNet
class ResNet(nn.Module):
    def __init__(self, block, layers, bn, num_classes=100):
        super(ResNet, self).__init__()
```

```
        self.in_channels = 64
        self.bn = bn
        self.conv = nn.Conv2d(3, 64, kernel_size=(7, 7), stride = 2, padding = 3, b
        self.relu = nn.ReLU(inplace=True)
        self.bns = nn.BatchNorm2d(64)
        self.maxpool = nn.MaxPool2d((3, 3), stride=2, padding=1)
        self.layer1 = self.make_layer(block, 64, layers[0])
        self.layer2 = self.make_layer(block, 128, layers[1])
        self.layer3 = self.make_layer(block, 256, layers[2])
        self.layer4 = self.make_layer(block, 512, layers[3], 2)
        self.avg_pool = nn.AvgPool2d((4, 4), stride=1)
        self.fc = nn.Linear(512, num_classes)

    def make_layer(self, block, out_channels, blocks, stride=1):
        downsample = None
        if ((stride != 1) or (self.in_channels != out_channels)):
            if not self.bn:
                downsample = nn.Sequential(
                    nn.Conv2d(self.in_channels, out_channels, kernel_size=(1, 1), s
            else :
                downsample = nn.Sequential(
                    nn.Conv2d(self.in_channels, out_channels, kernel_size=(1, 1), s
                    nn.BatchNorm2d(out_channels))
        layers = []
        layers.append(block(self.in_channels, out_channels, self.bn, stride, downsa
        self.in_channels = out_channels
        for i in range(1, blocks):
            layers.append(block(out_channels, out_channels, self.bn))
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.conv(x)
        if self.bn :
            out = self.bns(out)
        out = self.relu(out)
        out = self.maxpool(out)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = self.avg_pool(out)
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        return out

########################################################################
#                              END OF YOUR CODE                        #
########################################################################
```

In [ ]:

```
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(ResidualBlock, [2, 2, 2, 2], False).to(device)
optimizer = optim.Adam(model.parameters(), lr = learning_rate)

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

```
#                                 END OF YOUR CODE
################################################################################

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

```
Epoch 0, Iteration 0, loss = 4.5936
Checking accuracy on validation set
Got 10 / 1000 correct (1.00)

Epoch 0, Iteration 100, loss = 4.4705
Checking accuracy on validation set
Got 22 / 1000 correct (2.20)

Epoch 0, Iteration 200, loss = 4.1751
Checking accuracy on validation set
Got 37 / 1000 correct (3.70)

Epoch 0, Iteration 300, loss = 4.0760
Checking accuracy on validation set
Got 60 / 1000 correct (6.00)

Epoch 0, Iteration 400, loss = 3.8504
Checking accuracy on validation set
Got 84 / 1000 correct (8.40)

Epoch 0, Iteration 500, loss = 3.6796
Checking accuracy on validation set
Got 96 / 1000 correct (9.60)

Epoch 0, Iteration 600, loss = 3.9133
Checking accuracy on validation set
Got 125 / 1000 correct (12.50)

Epoch 0, Iteration 700, loss = 3.5847
Checking accuracy on validation set
Got 130 / 1000 correct (13.00)

Epoch 1, Iteration 0, loss = 3.4773
Checking accuracy on validation set
Got 133 / 1000 correct (13.30)

Epoch 1, Iteration 100, loss = 3.3310
Checking accuracy on validation set
Got 140 / 1000 correct (14.00)

Epoch 1, Iteration 200, loss = 3.4799
Checking accuracy on validation set
Got 175 / 1000 correct (17.50)

Epoch 1, Iteration 300, loss = 3.2424
Checking accuracy on validation set
Got 183 / 1000 correct (18.30)

Epoch 1, Iteration 400, loss = 3.5337
Checking accuracy on validation set
Got 186 / 1000 correct (18.60)

Epoch 1, Iteration 500, loss = 3.3049
Checking accuracy on validation set
Got 203 / 1000 correct (20.30)

Epoch 1, Iteration 600, loss = 2.9323
Checking accuracy on validation set
Got 228 / 1000 correct (22.80)

Epoch 1, Iteration 700, loss = 2.9287
Checking accuracy on validation set
Got 218 / 1000 correct (21.80)
```

```
Epoch 2, Iteration 0, loss = 2.9981
Checking accuracy on validation set
Got 227 / 1000 correct (22.70)

Epoch 2, Iteration 100, loss = 2.7804
Checking accuracy on validation set
Got 230 / 1000 correct (23.00)

Epoch 2, Iteration 200, loss = 2.8573
Checking accuracy on validation set
Got 229 / 1000 correct (22.90)

Epoch 2, Iteration 300, loss = 2.5132
Checking accuracy on validation set
Got 251 / 1000 correct (25.10)

Epoch 2, Iteration 400, loss = 2.7768
Checking accuracy on validation set
Got 254 / 1000 correct (25.40)

Epoch 2, Iteration 500, loss = 2.6480
Checking accuracy on validation set
Got 266 / 1000 correct (26.60)

Epoch 2, Iteration 600, loss = 2.6038
Checking accuracy on validation set
Got 285 / 1000 correct (28.50)

Epoch 2, Iteration 700, loss = 2.6356
Checking accuracy on validation set
Got 290 / 1000 correct (29.00)

Epoch 3, Iteration 0, loss = 2.3804
Checking accuracy on validation set
Got 294 / 1000 correct (29.40)

Epoch 3, Iteration 100, loss = 2.7495
Checking accuracy on validation set
Got 308 / 1000 correct (30.80)

Epoch 3, Iteration 200, loss = 2.3810
Checking accuracy on validation set
Got 315 / 1000 correct (31.50)

Epoch 3, Iteration 300, loss = 2.2264
Checking accuracy on validation set
Got 299 / 1000 correct (29.90)

Epoch 3, Iteration 400, loss = 2.5096
Checking accuracy on validation set
Got 299 / 1000 correct (29.90)

Epoch 3, Iteration 500, loss = 2.5629
Checking accuracy on validation set
Got 317 / 1000 correct (31.70)

Epoch 3, Iteration 600, loss = 2.5499
Checking accuracy on validation set
Got 310 / 1000 correct (31.00)

Epoch 3, Iteration 700, loss = 2.7235
Checking accuracy on validation set
Got 332 / 1000 correct (33.20)
```

```
Epoch 4, Iteration 0, loss = 2.2745
Checking accuracy on validation set
Got 325 / 1000 correct (32.50)

Epoch 4, Iteration 100, loss = 2.2457
Checking accuracy on validation set
Got 352 / 1000 correct (35.20)

Epoch 4, Iteration 200, loss = 2.4059
Checking accuracy on validation set
Got 333 / 1000 correct (33.30)

Epoch 4, Iteration 300, loss = 2.3160
Checking accuracy on validation set
Got 345 / 1000 correct (34.50)

Epoch 4, Iteration 400, loss = 2.5166
Checking accuracy on validation set
Got 350 / 1000 correct (35.00)

Epoch 4, Iteration 500, loss = 2.2794
Checking accuracy on validation set
Got 336 / 1000 correct (33.60)

Epoch 4, Iteration 600, loss = 2.2288
Checking accuracy on validation set
Got 328 / 1000 correct (32.80)

Epoch 4, Iteration 700, loss = 2.2540
Checking accuracy on validation set
Got 340 / 1000 correct (34.00)

Epoch 5, Iteration 0, loss = 2.0308
Checking accuracy on validation set
Got 357 / 1000 correct (35.70)

Epoch 5, Iteration 100, loss = 2.2814
Checking accuracy on validation set
Got 351 / 1000 correct (35.10)

Epoch 5, Iteration 200, loss = 1.7944
Checking accuracy on validation set
Got 350 / 1000 correct (35.00)

Epoch 5, Iteration 300, loss = 2.3405
Checking accuracy on validation set
Got 353 / 1000 correct (35.30)

Epoch 5, Iteration 400, loss = 2.0532
Checking accuracy on validation set
Got 341 / 1000 correct (34.10)

Epoch 5, Iteration 500, loss = 2.0315
Checking accuracy on validation set
Got 360 / 1000 correct (36.00)

Epoch 5, Iteration 600, loss = 2.2866
Checking accuracy on validation set
Got 356 / 1000 correct (35.60)

Epoch 5, Iteration 700, loss = 2.2899
Checking accuracy on validation set
Got 379 / 1000 correct (37.90)
```

```
Epoch 6, Iteration 0, loss = 2.2715
Checking accuracy on validation set
Got 367 / 1000 correct (36.70)

Epoch 6, Iteration 100, loss = 2.0020
Checking accuracy on validation set
Got 374 / 1000 correct (37.40)

Epoch 6, Iteration 200, loss = 2.0958
Checking accuracy on validation set
Got 388 / 1000 correct (38.80)

Epoch 6, Iteration 300, loss = 1.9465
Checking accuracy on validation set
Got 381 / 1000 correct (38.10)

Epoch 6, Iteration 400, loss = 1.8962
Checking accuracy on validation set
Got 375 / 1000 correct (37.50)

Epoch 6, Iteration 500, loss = 2.1367
Checking accuracy on validation set
Got 392 / 1000 correct (39.20)

Epoch 6, Iteration 600, loss = 1.8174
Checking accuracy on validation set
Got 384 / 1000 correct (38.40)

Epoch 6, Iteration 700, loss = 2.0282
Checking accuracy on validation set
Got 386 / 1000 correct (38.60)

Epoch 7, Iteration 0, loss = 1.7175
Checking accuracy on validation set
Got 370 / 1000 correct (37.00)

Epoch 7, Iteration 100, loss = 1.4601
Checking accuracy on validation set
Got 386 / 1000 correct (38.60)

Epoch 7, Iteration 200, loss = 1.7797
Checking accuracy on validation set
Got 402 / 1000 correct (40.20)

Epoch 7, Iteration 300, loss = 1.6920
Checking accuracy on validation set
Got 380 / 1000 correct (38.00)

Epoch 7, Iteration 400, loss = 1.9559
Checking accuracy on validation set
Got 378 / 1000 correct (37.80)

Epoch 7, Iteration 500, loss = 1.6782
Checking accuracy on validation set
Got 397 / 1000 correct (39.70)

Epoch 7, Iteration 600, loss = 1.7092
Checking accuracy on validation set
Got 405 / 1000 correct (40.50)

Epoch 7, Iteration 700, loss = 2.1990
Checking accuracy on validation set
Got 396 / 1000 correct (39.60)
```

```
Epoch 8, Iteration 0, loss = 1.3531
Checking accuracy on validation set
Got 398 / 1000 correct (39.80)

Epoch 8, Iteration 100, loss = 1.4008
Checking accuracy on validation set
Got 413 / 1000 correct (41.30)

Epoch 8, Iteration 200, loss = 1.1633
Checking accuracy on validation set
Got 405 / 1000 correct (40.50)

Epoch 8, Iteration 300, loss = 1.2989
Checking accuracy on validation set
Got 369 / 1000 correct (36.90)

Epoch 8, Iteration 400, loss = 1.5887
Checking accuracy on validation set
Got 387 / 1000 correct (38.70)

Epoch 8, Iteration 500, loss = 1.4800
Checking accuracy on validation set
Got 378 / 1000 correct (37.80)

Epoch 8, Iteration 600, loss = 1.7862
Checking accuracy on validation set
Got 382 / 1000 correct (38.20)

Epoch 8, Iteration 700, loss = 1.3956
Checking accuracy on validation set
Got 375 / 1000 correct (37.50)

Epoch 9, Iteration 0, loss = 1.6034
Checking accuracy on validation set
Got 390 / 1000 correct (39.00)

Epoch 9, Iteration 100, loss = 1.4113
Checking accuracy on validation set
Got 386 / 1000 correct (38.60)

Epoch 9, Iteration 200, loss = 1.2030
Checking accuracy on validation set
Got 413 / 1000 correct (41.30)

Epoch 9, Iteration 300, loss = 1.0877
Checking accuracy on validation set
Got 410 / 1000 correct (41.00)

Epoch 9, Iteration 400, loss = 1.3642
Checking accuracy on validation set
Got 394 / 1000 correct (39.40)

Epoch 9, Iteration 500, loss = 1.4536
Checking accuracy on validation set
Got 405 / 1000 correct (40.50)

Epoch 9, Iteration 600, loss = 1.3838
Checking accuracy on validation set
Got 392 / 1000 correct (39.20)

Epoch 9, Iteration 700, loss = 1.1031
Checking accuracy on validation set
Got 393 / 1000 correct (39.30)
```

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.

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 [ ]:

```
learning_rate = 1e-3

model = None
optimizer = None

################################################################################
# TODO: InstantiateResnet with BatchNorm                                       #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
model = ResNet(ResidualBlock, [2, 2, 2, 2], True).to(device)
optimizer = optim.Adam(model.parameters(), lr = learning_rate)
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
################################################################################
#                                 END OF YOUR CODE
################################################################################

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

```
Epoch 0, Iteration 0, loss = 4.7145
Checking accuracy on validation set
Got 12 / 1000 correct (1.20)

Epoch 0, Iteration 100, loss = 4.0274
Checking accuracy on validation set
Got 69 / 1000 correct (6.90)

Epoch 0, Iteration 200, loss = 3.8886
Checking accuracy on validation set
Got 76 / 1000 correct (7.60)

Epoch 0, Iteration 300, loss = 3.5304
Checking accuracy on validation set
Got 113 / 1000 correct (11.30)

Epoch 0, Iteration 400, loss = 3.5342
Checking accuracy on validation set
Got 128 / 1000 correct (12.80)

Epoch 0, Iteration 500, loss = 3.5260
Checking accuracy on validation set
Got 157 / 1000 correct (15.70)

Epoch 0, Iteration 600, loss = 3.4020
Checking accuracy on validation set
Got 162 / 1000 correct (16.20)

Epoch 0, Iteration 700, loss = 3.2800
Checking accuracy on validation set
Got 205 / 1000 correct (20.50)

Epoch 1, Iteration 0, loss = 3.1158
Checking accuracy on validation set
Got 200 / 1000 correct (20.00)

Epoch 1, Iteration 100, loss = 3.2056
Checking accuracy on validation set
Got 203 / 1000 correct (20.30)

Epoch 1, Iteration 200, loss = 3.0411
Checking accuracy on validation set
Got 211 / 1000 correct (21.10)

Epoch 1, Iteration 300, loss = 2.6786
Checking accuracy on validation set
Got 249 / 1000 correct (24.90)

Epoch 1, Iteration 400, loss = 2.4999
Checking accuracy on validation set
Got 249 / 1000 correct (24.90)

Epoch 1, Iteration 500, loss = 2.8775
Checking accuracy on validation set
Got 281 / 1000 correct (28.10)

Epoch 1, Iteration 600, loss = 2.9202
Checking accuracy on validation set
Got 300 / 1000 correct (30.00)

Epoch 1, Iteration 700, loss = 2.7005
Checking accuracy on validation set
Got 303 / 1000 correct (30.30)
```

```
Epoch 2, Iteration 0, loss = 2.4331
Checking accuracy on validation set
Got 294 / 1000 correct (29.40)

Epoch 2, Iteration 100, loss = 2.4970
Checking accuracy on validation set
Got 343 / 1000 correct (34.30)

Epoch 2, Iteration 200, loss = 2.3497
Checking accuracy on validation set
Got 332 / 1000 correct (33.20)

Epoch 2, Iteration 300, loss = 2.3487
Checking accuracy on validation set
Got 354 / 1000 correct (35.40)

Epoch 2, Iteration 400, loss = 2.3719
Checking accuracy on validation set
Got 335 / 1000 correct (33.50)

Epoch 2, Iteration 500, loss = 2.5675
Checking accuracy on validation set
Got 329 / 1000 correct (32.90)

Epoch 2, Iteration 600, loss = 2.2938
Checking accuracy on validation set
Got 378 / 1000 correct (37.80)

Epoch 2, Iteration 700, loss = 2.2940
Checking accuracy on validation set
Got 340 / 1000 correct (34.00)

Epoch 3, Iteration 0, loss = 2.5159
Checking accuracy on validation set
Got 373 / 1000 correct (37.30)

Epoch 3, Iteration 100, loss = 2.3835
Checking accuracy on validation set
Got 386 / 1000 correct (38.60)

Epoch 3, Iteration 200, loss = 2.1807
Checking accuracy on validation set
Got 392 / 1000 correct (39.20)

Epoch 3, Iteration 300, loss = 2.2058
Checking accuracy on validation set
Got 410 / 1000 correct (41.00)

Epoch 3, Iteration 400, loss = 2.0527
Checking accuracy on validation set
Got 426 / 1000 correct (42.60)

Epoch 3, Iteration 500, loss = 2.6463
Checking accuracy on validation set
Got 403 / 1000 correct (40.30)

Epoch 3, Iteration 600, loss = 1.9219
Checking accuracy on validation set
Got 402 / 1000 correct (40.20)

Epoch 3, Iteration 700, loss = 2.1316
Checking accuracy on validation set
Got 427 / 1000 correct (42.70)
```

```
Epoch 4, Iteration 0, loss = 1.8777
Checking accuracy on validation set
Got 404 / 1000 correct (40.40)

Epoch 4, Iteration 100, loss = 1.8388
Checking accuracy on validation set
Got 426 / 1000 correct (42.60)

Epoch 4, Iteration 200, loss = 1.7796
Checking accuracy on validation set
Got 428 / 1000 correct (42.80)

Epoch 4, Iteration 300, loss = 2.1070
Checking accuracy on validation set
Got 432 / 1000 correct (43.20)

Epoch 4, Iteration 400, loss = 1.8111
Checking accuracy on validation set
Got 425 / 1000 correct (42.50)

Epoch 4, Iteration 500, loss = 2.0515
Checking accuracy on validation set
Got 422 / 1000 correct (42.20)

Epoch 4, Iteration 600, loss = 1.9382
Checking accuracy on validation set
Got 444 / 1000 correct (44.40)

Epoch 4, Iteration 700, loss = 1.8089
Checking accuracy on validation set
Got 431 / 1000 correct (43.10)

Epoch 5, Iteration 0, loss = 1.4933
Checking accuracy on validation set
Got 427 / 1000 correct (42.70)

Epoch 5, Iteration 100, loss = 1.3839
Checking accuracy on validation set
Got 437 / 1000 correct (43.70)

Epoch 5, Iteration 200, loss = 1.5637
Checking accuracy on validation set
Got 445 / 1000 correct (44.50)

Epoch 5, Iteration 300, loss = 1.6841
Checking accuracy on validation set
Got 477 / 1000 correct (47.70)

Epoch 5, Iteration 400, loss = 1.8578
Checking accuracy on validation set
Got 467 / 1000 correct (46.70)

Epoch 5, Iteration 500, loss = 1.7473
Checking accuracy on validation set
Got 474 / 1000 correct (47.40)

Epoch 5, Iteration 600, loss = 1.5993
Checking accuracy on validation set
Got 479 / 1000 correct (47.90)

Epoch 5, Iteration 700, loss = 1.5720
Checking accuracy on validation set
Got 469 / 1000 correct (46.90)
```

```
Epoch 6, Iteration 0, loss = 1.4385
Checking accuracy on validation set
Got 468 / 1000 correct (46.80)

Epoch 6, Iteration 100, loss = 1.4691
Checking accuracy on validation set
Got 473 / 1000 correct (47.30)

Epoch 6, Iteration 200, loss = 1.3744
Checking accuracy on validation set
Got 491 / 1000 correct (49.10)

Epoch 6, Iteration 300, loss = 1.3757
Checking accuracy on validation set
Got 483 / 1000 correct (48.30)

Epoch 6, Iteration 400, loss = 1.4825
Checking accuracy on validation set
Got 474 / 1000 correct (47.40)

Epoch 6, Iteration 500, loss = 1.0294
Checking accuracy on validation set
Got 455 / 1000 correct (45.50)

Epoch 6, Iteration 600, loss = 1.4314
Checking accuracy on validation set
Got 490 / 1000 correct (49.00)

Epoch 6, Iteration 700, loss = 1.1875
Checking accuracy on validation set
Got 469 / 1000 correct (46.90)

Epoch 7, Iteration 0, loss = 0.9715
Checking accuracy on validation set
Got 483 / 1000 correct (48.30)

Epoch 7, Iteration 100, loss = 0.9895
Checking accuracy on validation set
Got 483 / 1000 correct (48.30)

Epoch 7, Iteration 200, loss = 0.7152
Checking accuracy on validation set
Got 459 / 1000 correct (45.90)

Epoch 7, Iteration 300, loss = 0.9054
Checking accuracy on validation set
Got 477 / 1000 correct (47.70)

Epoch 7, Iteration 400, loss = 0.7260
Checking accuracy on validation set
Got 493 / 1000 correct (49.30)

Epoch 7, Iteration 500, loss = 1.2835
Checking accuracy on validation set
Got 497 / 1000 correct (49.70)

Epoch 7, Iteration 600, loss = 1.0763
Checking accuracy on validation set
Got 500 / 1000 correct (50.00)

Epoch 7, Iteration 700, loss = 1.1984
Checking accuracy on validation set
Got 481 / 1000 correct (48.10)
```

```
Epoch 8, Iteration 0, loss = 0.7199
Checking accuracy on validation set
Got 481 / 1000 correct (48.10)

Epoch 8, Iteration 100, loss = 0.6061
Checking accuracy on validation set
Got 504 / 1000 correct (50.40)

Epoch 8, Iteration 200, loss = 0.6082
Checking accuracy on validation set
Got 518 / 1000 correct (51.80)

Epoch 8, Iteration 300, loss = 0.8064
Checking accuracy on validation set
Got 507 / 1000 correct (50.70)

Epoch 8, Iteration 400, loss = 0.9768
Checking accuracy on validation set
Got 513 / 1000 correct (51.30)

Epoch 8, Iteration 500, loss = 1.0038
Checking accuracy on validation set
Got 485 / 1000 correct (48.50)

Epoch 8, Iteration 600, loss = 0.5953
Checking accuracy on validation set
Got 486 / 1000 correct (48.60)

Epoch 8, Iteration 700, loss = 1.1764
Checking accuracy on validation set
Got 530 / 1000 correct (53.00)

Epoch 9, Iteration 0, loss = 0.4498
Checking accuracy on validation set
Got 501 / 1000 correct (50.10)

Epoch 9, Iteration 100, loss = 0.5095
Checking accuracy on validation set
Got 503 / 1000 correct (50.30)

Epoch 9, Iteration 200, loss = 0.3707
Checking accuracy on validation set
Got 500 / 1000 correct (50.00)

Epoch 9, Iteration 300, loss = 0.4909
Checking accuracy on validation set
Got 506 / 1000 correct (50.60)

Epoch 9, Iteration 400, loss = 0.6644
Checking accuracy on validation set
Got 497 / 1000 correct (49.70)

Epoch 9, Iteration 500, loss = 0.4284
Checking accuracy on validation set
Got 498 / 1000 correct (49.80)

Epoch 9, Iteration 600, loss = 0.5112
Checking accuracy on validation set
Got 461 / 1000 correct (46.10)

Epoch 9, Iteration 700, loss = 0.6525
Checking accuracy on validation set
Got 498 / 1000 correct (49.80)
```

Discussion on BatchNorm

TODO: The batchnorm was capable of increasing the accuracy of the validation set up to 10%, which is a significant increase from the maximum of 40% without batchnorm. I believe this to be the case as batch normalization allows for frequent normalization of the data, which regularizes the outlier data from skewing our model too heavily after a convolution, thus improving performance on unseen data.

Batch Size

In this exercise, we will study the effect of batch size on performance of Resnet. Specifically, you should try batch sizes of 32, 64 and 128 and describe the effect of varying batch size. You should also draw a graph showing the batch size on the x-axis and accuracy on the y-axis.

In [ ]:

```
batch_sizes = [32, 64, 128]
learning_rate = 1e-3
model = None
optimizer = None
import matplotlib.pyplot as plt

################################################################################
# TODO: Try Resnet with different batch sizes. Hint: You will need to         #
#  create a new dataloader with appropriate batch size for each experiment.    #
# You will also need to store the final accuracy for each experiment           #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
out = []
for batch in batch_sizes :
    del loader_train
    del loader_val
    del loader_test
    del model
    del optimizer
    loader_train = DataLoader(cifar100_train, batch_size=batch, num_workers=2,
                        sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN)))
    loader_val = DataLoader(cifar100_val, batch_size=batch, num_workers=2,
                        sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN, 5(
    loader_test = DataLoader(cifar100_test, batch_size=batch, num_workers=2)

    model = ResNet(ResidualBlock, [2, 2, 2, 2], True).to(device)
    optimizer = optim.Adam(model.parameters(), lr = learning_rate)

    val_accs = train_part34(model, optimizer, epochs=10)
    out.append(val_accs)
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
################################################################################
#                                 END OF YOUR CODE                             
################################################################################
```

```
Epoch 0, Iteration 0, loss = 4.5521
Checking accuracy on validation set
Got 8 / 1000 correct (0.80)

Epoch 0, Iteration 100, loss = 3.8260
Checking accuracy on validation set
Got 45 / 1000 correct (4.50)

Epoch 0, Iteration 200, loss = 4.5542
Checking accuracy on validation set
Got 48 / 1000 correct (4.80)

Epoch 0, Iteration 300, loss = 4.5389
Checking accuracy on validation set
Got 62 / 1000 correct (6.20)

Epoch 0, Iteration 400, loss = 4.0072
Checking accuracy on validation set
Got 92 / 1000 correct (9.20)

Epoch 0, Iteration 500, loss = 4.2262
Checking accuracy on validation set
Got 83 / 1000 correct (8.30)

Epoch 0, Iteration 600, loss = 3.5936
Checking accuracy on validation set
Got 90 / 1000 correct (9.00)

Epoch 0, Iteration 700, loss = 3.8218
Checking accuracy on validation set
Got 105 / 1000 correct (10.50)

Epoch 0, Iteration 800, loss = 3.2439
Checking accuracy on validation set
Got 122 / 1000 correct (12.20)

Epoch 0, Iteration 900, loss = 4.0166
Checking accuracy on validation set
Got 144 / 1000 correct (14.40)

Epoch 0, Iteration 1000, loss = 3.5632
Checking accuracy on validation set
Got 123 / 1000 correct (12.30)

Epoch 0, Iteration 1100, loss = 3.6051
Checking accuracy on validation set
Got 138 / 1000 correct (13.80)

Epoch 0, Iteration 1200, loss = 3.4651
Checking accuracy on validation set
Got 161 / 1000 correct (16.10)

Epoch 0, Iteration 1300, loss = 2.9663
Checking accuracy on validation set
Got 161 / 1000 correct (16.10)

Epoch 0, Iteration 1400, loss = 3.5483
Checking accuracy on validation set
Got 180 / 1000 correct (18.00)

Epoch 0, Iteration 1500, loss = 3.0659
Checking accuracy on validation set
Got 170 / 1000 correct (17.00)
```

```
Epoch 1, Iteration 0, loss = 2.5898
Checking accuracy on validation set
Got 198 / 1000 correct (19.80)

Epoch 1, Iteration 100, loss = 3.5207
Checking accuracy on validation set
Got 209 / 1000 correct (20.90)

Epoch 1, Iteration 200, loss = 3.2710
Checking accuracy on validation set
Got 210 / 1000 correct (21.00)

Epoch 1, Iteration 300, loss = 2.9709
Checking accuracy on validation set
Got 216 / 1000 correct (21.60)

Epoch 1, Iteration 400, loss = 2.7454
Checking accuracy on validation set
Got 229 / 1000 correct (22.90)

Epoch 1, Iteration 500, loss = 2.6944
Checking accuracy on validation set
Got 241 / 1000 correct (24.10)

Epoch 1, Iteration 600, loss = 3.5246
Checking accuracy on validation set
Got 237 / 1000 correct (23.70)

Epoch 1, Iteration 700, loss = 2.5296
Checking accuracy on validation set
Got 232 / 1000 correct (23.20)

Epoch 1, Iteration 800, loss = 2.8410
Checking accuracy on validation set
Got 229 / 1000 correct (22.90)

Epoch 1, Iteration 900, loss = 2.6921
Checking accuracy on validation set
Got 277 / 1000 correct (27.70)

Epoch 1, Iteration 1000, loss = 2.7465
Checking accuracy on validation set
Got 266 / 1000 correct (26.60)

Epoch 1, Iteration 1100, loss = 2.9031
Checking accuracy on validation set
Got 293 / 1000 correct (29.30)

Epoch 1, Iteration 1200, loss = 2.8926
Checking accuracy on validation set
Got 285 / 1000 correct (28.50)

Epoch 1, Iteration 1300, loss = 2.8090
Checking accuracy on validation set
Got 290 / 1000 correct (29.00)

Epoch 1, Iteration 1400, loss = 2.7117
Checking accuracy on validation set
Got 279 / 1000 correct (27.90)

Epoch 1, Iteration 1500, loss = 2.0673
Checking accuracy on validation set
Got 295 / 1000 correct (29.50)
```

```
Epoch 2, Iteration 0, loss = 2.5986
Checking accuracy on validation set
Got 316 / 1000 correct (31.60)

Epoch 2, Iteration 100, loss = 2.8180
Checking accuracy on validation set
Got 299 / 1000 correct (29.90)

Epoch 2, Iteration 200, loss = 2.4322
Checking accuracy on validation set
Got 319 / 1000 correct (31.90)

Epoch 2, Iteration 300, loss = 2.1683
Checking accuracy on validation set
Got 310 / 1000 correct (31.00)

Epoch 2, Iteration 400, loss = 2.0047
Checking accuracy on validation set
Got 345 / 1000 correct (34.50)

Epoch 2, Iteration 500, loss = 2.5327
Checking accuracy on validation set
Got 326 / 1000 correct (32.60)

Epoch 2, Iteration 600, loss = 2.3111
Checking accuracy on validation set
Got 331 / 1000 correct (33.10)

Epoch 2, Iteration 700, loss = 2.4433
Checking accuracy on validation set
Got 344 / 1000 correct (34.40)

Epoch 2, Iteration 800, loss = 3.0981
Checking accuracy on validation set
Got 353 / 1000 correct (35.30)

Epoch 2, Iteration 900, loss = 2.5513
Checking accuracy on validation set
Got 360 / 1000 correct (36.00)

Epoch 2, Iteration 1000, loss = 2.1232
Checking accuracy on validation set
Got 358 / 1000 correct (35.80)

Epoch 2, Iteration 1100, loss = 1.8168
Checking accuracy on validation set
Got 374 / 1000 correct (37.40)

Epoch 2, Iteration 1200, loss = 2.0406
Checking accuracy on validation set
Got 360 / 1000 correct (36.00)

Epoch 2, Iteration 1300, loss = 2.2751
Checking accuracy on validation set
Got 366 / 1000 correct (36.60)

Epoch 2, Iteration 1400, loss = 2.2281
Checking accuracy on validation set
Got 372 / 1000 correct (37.20)

Epoch 2, Iteration 1500, loss = 2.4589
Checking accuracy on validation set
Got 371 / 1000 correct (37.10)
```

```
Epoch 3, Iteration 0, loss = 2.0121
Checking accuracy on validation set
Got 356 / 1000 correct (35.60)

Epoch 3, Iteration 100, loss = 1.9386
Checking accuracy on validation set
Got 407 / 1000 correct (40.70)

Epoch 3, Iteration 200, loss = 2.1084
Checking accuracy on validation set
Got 405 / 1000 correct (40.50)

Epoch 3, Iteration 300, loss = 2.1779
Checking accuracy on validation set
Got 373 / 1000 correct (37.30)

Epoch 3, Iteration 400, loss = 2.6969
Checking accuracy on validation set
Got 373 / 1000 correct (37.30)

Epoch 3, Iteration 500, loss = 2.1072
Checking accuracy on validation set
Got 388 / 1000 correct (38.80)

Epoch 3, Iteration 600, loss = 2.1487
Checking accuracy on validation set
Got 416 / 1000 correct (41.60)

Epoch 3, Iteration 700, loss = 2.1773
Checking accuracy on validation set
Got 377 / 1000 correct (37.70)

Epoch 3, Iteration 800, loss = 2.2171
Checking accuracy on validation set
Got 389 / 1000 correct (38.90)

Epoch 3, Iteration 900, loss = 1.9767
Checking accuracy on validation set
Got 423 / 1000 correct (42.30)

Epoch 3, Iteration 1000, loss = 2.1909
Checking accuracy on validation set
Got 393 / 1000 correct (39.30)

Epoch 3, Iteration 1100, loss = 2.3674
Checking accuracy on validation set
Got 408 / 1000 correct (40.80)

Epoch 3, Iteration 1200, loss = 1.8673
Checking accuracy on validation set
Got 408 / 1000 correct (40.80)

Epoch 3, Iteration 1300, loss = 2.3127
Checking accuracy on validation set
Got 421 / 1000 correct (42.10)

Epoch 3, Iteration 1400, loss = 1.9287
Checking accuracy on validation set
Got 398 / 1000 correct (39.80)

Epoch 3, Iteration 1500, loss = 2.1599
Checking accuracy on validation set
Got 418 / 1000 correct (41.80)
```

```
Epoch 4, Iteration 0, loss = 1.7258
Checking accuracy on validation set
Got 415 / 1000 correct (41.50)

Epoch 4, Iteration 100, loss = 2.3608
Checking accuracy on validation set
Got 402 / 1000 correct (40.20)

Epoch 4, Iteration 200, loss = 1.8271
Checking accuracy on validation set
Got 432 / 1000 correct (43.20)

Epoch 4, Iteration 300, loss = 1.7993
Checking accuracy on validation set
Got 442 / 1000 correct (44.20)

Epoch 4, Iteration 400, loss = 2.0017
Checking accuracy on validation set
Got 440 / 1000 correct (44.00)

Epoch 4, Iteration 500, loss = 1.4971
Checking accuracy on validation set
Got 439 / 1000 correct (43.90)

Epoch 4, Iteration 600, loss = 1.7838
Checking accuracy on validation set
Got 437 / 1000 correct (43.70)

Epoch 4, Iteration 700, loss = 1.9016
Checking accuracy on validation set
Got 447 / 1000 correct (44.70)

Epoch 4, Iteration 800, loss = 1.7848
Checking accuracy on validation set
Got 432 / 1000 correct (43.20)

Epoch 4, Iteration 900, loss = 2.1104
Checking accuracy on validation set
Got 436 / 1000 correct (43.60)

Epoch 4, Iteration 1000, loss = 1.7860
Checking accuracy on validation set
Got 443 / 1000 correct (44.30)

Epoch 4, Iteration 1100, loss = 1.9848
Checking accuracy on validation set
Got 442 / 1000 correct (44.20)

Epoch 4, Iteration 1200, loss = 1.8165
Checking accuracy on validation set
Got 453 / 1000 correct (45.30)

Epoch 4, Iteration 1300, loss = 2.1645
Checking accuracy on validation set
Got 454 / 1000 correct (45.40)

Epoch 4, Iteration 1400, loss = 2.1366
Checking accuracy on validation set
Got 455 / 1000 correct (45.50)

Epoch 4, Iteration 1500, loss = 1.5142
Checking accuracy on validation set
Got 473 / 1000 correct (47.30)
```

```
Epoch 5, Iteration 0, loss = 1.5349
Checking accuracy on validation set
Got 468 / 1000 correct (46.80)

Epoch 5, Iteration 100, loss = 2.3825
Checking accuracy on validation set
Got 465 / 1000 correct (46.50)

Epoch 5, Iteration 200, loss = 1.5402
Checking accuracy on validation set
Got 456 / 1000 correct (45.60)

Epoch 5, Iteration 300, loss = 1.7763
Checking accuracy on validation set
Got 439 / 1000 correct (43.90)

Epoch 5, Iteration 400, loss = 1.5362
Checking accuracy on validation set
Got 473 / 1000 correct (47.30)

Epoch 5, Iteration 500, loss = 1.3766
Checking accuracy on validation set
Got 462 / 1000 correct (46.20)

Epoch 5, Iteration 600, loss = 2.2308
Checking accuracy on validation set
Got 480 / 1000 correct (48.00)

Epoch 5, Iteration 700, loss = 1.6654
Checking accuracy on validation set
Got 478 / 1000 correct (47.80)

Epoch 5, Iteration 800, loss = 1.7126
Checking accuracy on validation set
Got 470 / 1000 correct (47.00)

Epoch 5, Iteration 900, loss = 1.4712
Checking accuracy on validation set
Got 473 / 1000 correct (47.30)

Epoch 5, Iteration 1000, loss = 1.7381
Checking accuracy on validation set
Got 463 / 1000 correct (46.30)

Epoch 5, Iteration 1100, loss = 1.3519
Checking accuracy on validation set
Got 479 / 1000 correct (47.90)

Epoch 5, Iteration 1200, loss = 1.7110
Checking accuracy on validation set
Got 472 / 1000 correct (47.20)

Epoch 5, Iteration 1300, loss = 1.8647
Checking accuracy on validation set
Got 477 / 1000 correct (47.70)

Epoch 5, Iteration 1400, loss = 1.9877
Checking accuracy on validation set
Got 481 / 1000 correct (48.10)

Epoch 5, Iteration 1500, loss = 1.5419
Checking accuracy on validation set
Got 480 / 1000 correct (48.00)
```

```
Epoch 6, Iteration 0, loss = 1.4245
Checking accuracy on validation set
Got 474 / 1000 correct (47.40)

Epoch 6, Iteration 100, loss = 1.2440
Checking accuracy on validation set
Got 504 / 1000 correct (50.40)

Epoch 6, Iteration 200, loss = 1.2504
Checking accuracy on validation set
Got 507 / 1000 correct (50.70)

Epoch 6, Iteration 300, loss = 1.2232
Checking accuracy on validation set
Got 483 / 1000 correct (48.30)

Epoch 6, Iteration 400, loss = 1.3135
Checking accuracy on validation set
Got 476 / 1000 correct (47.60)

Epoch 6, Iteration 500, loss = 1.7003
Checking accuracy on validation set
Got 443 / 1000 correct (44.30)

Epoch 6, Iteration 600, loss = 1.4637
Checking accuracy on validation set
Got 477 / 1000 correct (47.70)

Epoch 6, Iteration 700, loss = 1.4552
Checking accuracy on validation set
Got 480 / 1000 correct (48.00)

Epoch 6, Iteration 800, loss = 1.2380
Checking accuracy on validation set
Got 469 / 1000 correct (46.90)

Epoch 6, Iteration 900, loss = 1.4909
Checking accuracy on validation set
Got 474 / 1000 correct (47.40)

Epoch 6, Iteration 1000, loss = 1.5161
Checking accuracy on validation set
Got 486 / 1000 correct (48.60)

Epoch 6, Iteration 1100, loss = 1.9225
Checking accuracy on validation set
Got 485 / 1000 correct (48.50)

Epoch 6, Iteration 1200, loss = 1.4708
Checking accuracy on validation set
Got 485 / 1000 correct (48.50)

Epoch 6, Iteration 1300, loss = 1.2981
Checking accuracy on validation set
Got 494 / 1000 correct (49.40)

Epoch 6, Iteration 1400, loss = 1.6003
Checking accuracy on validation set
Got 483 / 1000 correct (48.30)

Epoch 6, Iteration 1500, loss = 1.0799
Checking accuracy on validation set
Got 488 / 1000 correct (48.80)
```

```
Epoch 7, Iteration 0, loss = 0.8511
Checking accuracy on validation set
Got 497 / 1000 correct (49.70)

Epoch 7, Iteration 100, loss = 0.9425
Checking accuracy on validation set
Got 506 / 1000 correct (50.60)

Epoch 7, Iteration 200, loss = 0.9075
Checking accuracy on validation set
Got 495 / 1000 correct (49.50)

Epoch 7, Iteration 300, loss = 0.9986
Checking accuracy on validation set
Got 493 / 1000 correct (49.30)

Epoch 7, Iteration 400, loss = 1.1669
Checking accuracy on validation set
Got 502 / 1000 correct (50.20)

Epoch 7, Iteration 500, loss = 1.1610
Checking accuracy on validation set
Got 498 / 1000 correct (49.80)

Epoch 7, Iteration 600, loss = 1.2849
Checking accuracy on validation set
Got 477 / 1000 correct (47.70)

Epoch 7, Iteration 700, loss = 1.4182
Checking accuracy on validation set
Got 478 / 1000 correct (47.80)

Epoch 7, Iteration 800, loss = 0.9454
Checking accuracy on validation set
Got 488 / 1000 correct (48.80)

Epoch 7, Iteration 900, loss = 0.7879
Checking accuracy on validation set
Got 495 / 1000 correct (49.50)

Epoch 7, Iteration 1000, loss = 1.2465
Checking accuracy on validation set
Got 508 / 1000 correct (50.80)

Epoch 7, Iteration 1100, loss = 1.4681
Checking accuracy on validation set
Got 476 / 1000 correct (47.60)

Epoch 7, Iteration 1200, loss = 1.7186
Checking accuracy on validation set
Got 485 / 1000 correct (48.50)

Epoch 7, Iteration 1300, loss = 1.0570
Checking accuracy on validation set
Got 482 / 1000 correct (48.20)

Epoch 7, Iteration 1400, loss = 1.2048
Checking accuracy on validation set
Got 473 / 1000 correct (47.30)

Epoch 7, Iteration 1500, loss = 0.7634
Checking accuracy on validation set
Got 485 / 1000 correct (48.50)
```

```
Epoch 8, Iteration 0, loss = 0.9962
Checking accuracy on validation set
Got 492 / 1000 correct (49.20)

Epoch 8, Iteration 100, loss = 0.7327
Checking accuracy on validation set
Got 507 / 1000 correct (50.70)

Epoch 8, Iteration 200, loss = 0.6177
Checking accuracy on validation set
Got 505 / 1000 correct (50.50)

Epoch 8, Iteration 300, loss = 0.7469
Checking accuracy on validation set
Got 485 / 1000 correct (48.50)

Epoch 8, Iteration 400, loss = 0.7593
Checking accuracy on validation set
Got 500 / 1000 correct (50.00)

Epoch 8, Iteration 500, loss = 0.7577
Checking accuracy on validation set
Got 479 / 1000 correct (47.90)

Epoch 8, Iteration 600, loss = 0.6339
Checking accuracy on validation set
Got 514 / 1000 correct (51.40)

Epoch 8, Iteration 700, loss = 0.6755
Checking accuracy on validation set
Got 465 / 1000 correct (46.50)

Epoch 8, Iteration 800, loss = 0.8210
Checking accuracy on validation set
Got 485 / 1000 correct (48.50)

Epoch 8, Iteration 900, loss = 0.3173
Checking accuracy on validation set
Got 505 / 1000 correct (50.50)

Epoch 8, Iteration 1000, loss = 0.6059
Checking accuracy on validation set
Got 501 / 1000 correct (50.10)

Epoch 8, Iteration 1100, loss = 1.0533
Checking accuracy on validation set
Got 487 / 1000 correct (48.70)

Epoch 8, Iteration 1200, loss = 0.8822
Checking accuracy on validation set
Got 490 / 1000 correct (49.00)

Epoch 8, Iteration 1300, loss = 0.7498
Checking accuracy on validation set
Got 493 / 1000 correct (49.30)

Epoch 8, Iteration 1400, loss = 0.9772
Checking accuracy on validation set
Got 491 / 1000 correct (49.10)

Epoch 8, Iteration 1500, loss = 0.9675
Checking accuracy on validation set
Got 490 / 1000 correct (49.00)
```

```
Epoch 9, Iteration 0, loss = 0.5710
Checking accuracy on validation set
Got 488 / 1000 correct (48.80)

Epoch 9, Iteration 100, loss = 0.7840
Checking accuracy on validation set
Got 512 / 1000 correct (51.20)

Epoch 9, Iteration 200, loss = 0.3722
Checking accuracy on validation set
Got 517 / 1000 correct (51.70)

Epoch 9, Iteration 300, loss = 0.3956
Checking accuracy on validation set
Got 490 / 1000 correct (49.00)

Epoch 9, Iteration 400, loss = 0.3545
Checking accuracy on validation set
Got 513 / 1000 correct (51.30)

Epoch 9, Iteration 500, loss = 0.4657
Checking accuracy on validation set
Got 498 / 1000 correct (49.80)

Epoch 9, Iteration 600, loss = 0.5057
Checking accuracy on validation set
Got 508 / 1000 correct (50.80)

Epoch 9, Iteration 700, loss = 0.2875
Checking accuracy on validation set
Got 481 / 1000 correct (48.10)

Epoch 9, Iteration 800, loss = 0.6182
Checking accuracy on validation set
Got 488 / 1000 correct (48.80)

Epoch 9, Iteration 900, loss = 0.7847
Checking accuracy on validation set
Got 503 / 1000 correct (50.30)

Epoch 9, Iteration 1000, loss = 0.7003
Checking accuracy on validation set
Got 486 / 1000 correct (48.60)

Epoch 9, Iteration 1100, loss = 0.5510
Checking accuracy on validation set
Got 478 / 1000 correct (47.80)

Epoch 9, Iteration 1200, loss = 0.6199
Checking accuracy on validation set
Got 477 / 1000 correct (47.70)

Epoch 9, Iteration 1300, loss = 0.4057
Checking accuracy on validation set
Got 489 / 1000 correct (48.90)

Epoch 9, Iteration 1400, loss = 0.3502
Checking accuracy on validation set
Got 488 / 1000 correct (48.80)

Epoch 9, Iteration 1500, loss = 0.9832
Checking accuracy on validation set
Got 505 / 1000 correct (50.50)
```

```
Epoch 0, Iteration 0, loss = 4.6812
Checking accuracy on validation set
Got 10 / 1000 correct (1.00)

Epoch 0, Iteration 100, loss = 4.0454
Checking accuracy on validation set
Got 61 / 1000 correct (6.10)

Epoch 0, Iteration 200, loss = 3.6703
Checking accuracy on validation set
Got 94 / 1000 correct (9.40)

Epoch 0, Iteration 300, loss = 3.6491
Checking accuracy on validation set
Got 131 / 1000 correct (13.10)

Epoch 0, Iteration 400, loss = 3.5639
Checking accuracy on validation set
Got 118 / 1000 correct (11.80)

Epoch 0, Iteration 500, loss = 3.3855
Checking accuracy on validation set
Got 151 / 1000 correct (15.10)

Epoch 0, Iteration 600, loss = 3.0528
Checking accuracy on validation set
Got 173 / 1000 correct (17.30)

Epoch 0, Iteration 700, loss = 3.2473
Checking accuracy on validation set
Got 152 / 1000 correct (15.20)

Epoch 1, Iteration 0, loss = 3.0978
Checking accuracy on validation set
Got 179 / 1000 correct (17.90)

Epoch 1, Iteration 100, loss = 3.0426
Checking accuracy on validation set
Got 218 / 1000 correct (21.80)

Epoch 1, Iteration 200, loss = 2.9125
Checking accuracy on validation set
Got 230 / 1000 correct (23.00)

Epoch 1, Iteration 300, loss = 2.9330
Checking accuracy on validation set
Got 199 / 1000 correct (19.90)

Epoch 1, Iteration 400, loss = 2.8200
Checking accuracy on validation set
Got 278 / 1000 correct (27.80)

Epoch 1, Iteration 500, loss = 2.8734
Checking accuracy on validation set
Got 262 / 1000 correct (26.20)

Epoch 1, Iteration 600, loss = 2.9020
Checking accuracy on validation set
Got 288 / 1000 correct (28.80)

Epoch 1, Iteration 700, loss = 2.3467
Checking accuracy on validation set
Got 279 / 1000 correct (27.90)
```

```
Epoch 2, Iteration 0, loss = 2.4128
Checking accuracy on validation set
Got 292 / 1000 correct (29.20)

Epoch 2, Iteration 100, loss = 2.8447
Checking accuracy on validation set
Got 303 / 1000 correct (30.30)

Epoch 2, Iteration 200, loss = 2.5269
Checking accuracy on validation set
Got 316 / 1000 correct (31.60)

Epoch 2, Iteration 300, loss = 2.8929
Checking accuracy on validation set
Got 322 / 1000 correct (32.20)

Epoch 2, Iteration 400, loss = 2.6949
Checking accuracy on validation set
Got 319 / 1000 correct (31.90)

Epoch 2, Iteration 500, loss = 1.9999
Checking accuracy on validation set
Got 338 / 1000 correct (33.80)

Epoch 2, Iteration 600, loss = 2.4238
Checking accuracy on validation set
Got 348 / 1000 correct (34.80)

Epoch 2, Iteration 700, loss = 2.2122
Checking accuracy on validation set
Got 352 / 1000 correct (35.20)

Epoch 3, Iteration 0, loss = 2.4220
Checking accuracy on validation set
Got 352 / 1000 correct (35.20)

Epoch 3, Iteration 100, loss = 1.9274
Checking accuracy on validation set
Got 359 / 1000 correct (35.90)

Epoch 3, Iteration 200, loss = 2.2059
Checking accuracy on validation set
Got 353 / 1000 correct (35.30)

Epoch 3, Iteration 300, loss = 2.6115
Checking accuracy on validation set
Got 349 / 1000 correct (34.90)

Epoch 3, Iteration 400, loss = 2.1696
Checking accuracy on validation set
Got 393 / 1000 correct (39.30)

Epoch 3, Iteration 500, loss = 2.0299
Checking accuracy on validation set
Got 405 / 1000 correct (40.50)

Epoch 3, Iteration 600, loss = 1.9603
Checking accuracy on validation set
Got 384 / 1000 correct (38.40)

Epoch 3, Iteration 700, loss = 1.9862
Checking accuracy on validation set
Got 384 / 1000 correct (38.40)
```

```
Epoch 4, Iteration 0, loss = 1.9213
Checking accuracy on validation set
Got 414 / 1000 correct (41.40)

Epoch 4, Iteration 100, loss = 1.4790
Checking accuracy on validation set
Got 397 / 1000 correct (39.70)

Epoch 4, Iteration 200, loss = 2.4788
Checking accuracy on validation set
Got 394 / 1000 correct (39.40)

Epoch 4, Iteration 300, loss = 1.8112
Checking accuracy on validation set
Got 427 / 1000 correct (42.70)

Epoch 4, Iteration 400, loss = 2.0810
Checking accuracy on validation set
Got 429 / 1000 correct (42.90)

Epoch 4, Iteration 500, loss = 1.7494
Checking accuracy on validation set
Got 430 / 1000 correct (43.00)

Epoch 4, Iteration 600, loss = 1.7287
Checking accuracy on validation set
Got 439 / 1000 correct (43.90)

Epoch 4, Iteration 700, loss = 2.2319
Checking accuracy on validation set
Got 434 / 1000 correct (43.40)

Epoch 5, Iteration 0, loss = 1.7987
Checking accuracy on validation set
Got 409 / 1000 correct (40.90)

Epoch 5, Iteration 100, loss = 1.3734
Checking accuracy on validation set
Got 450 / 1000 correct (45.00)

Epoch 5, Iteration 200, loss = 1.6755
Checking accuracy on validation set
Got 423 / 1000 correct (42.30)

Epoch 5, Iteration 300, loss = 1.6631
Checking accuracy on validation set
Got 460 / 1000 correct (46.00)

Epoch 5, Iteration 400, loss = 1.6980
Checking accuracy on validation set
Got 434 / 1000 correct (43.40)

Epoch 5, Iteration 500, loss = 1.8237
Checking accuracy on validation set
Got 468 / 1000 correct (46.80)

Epoch 5, Iteration 600, loss = 1.8266
Checking accuracy on validation set
Got 463 / 1000 correct (46.30)

Epoch 5, Iteration 700, loss = 1.5201
Checking accuracy on validation set
Got 447 / 1000 correct (44.70)
```

```
Epoch 6, Iteration 0, loss = 1.4595
Checking accuracy on validation set
Got 476 / 1000 correct (47.60)

Epoch 6, Iteration 100, loss = 1.5065
Checking accuracy on validation set
Got 457 / 1000 correct (45.70)

Epoch 6, Iteration 200, loss = 1.3775
Checking accuracy on validation set
Got 475 / 1000 correct (47.50)

Epoch 6, Iteration 300, loss = 1.5841
Checking accuracy on validation set
Got 459 / 1000 correct (45.90)

Epoch 6, Iteration 400, loss = 2.0794
Checking accuracy on validation set
Got 459 / 1000 correct (45.90)

Epoch 6, Iteration 500, loss = 1.4310
Checking accuracy on validation set
Got 469 / 1000 correct (46.90)

Epoch 6, Iteration 600, loss = 1.5857
Checking accuracy on validation set
Got 493 / 1000 correct (49.30)

Epoch 6, Iteration 700, loss = 1.5497
Checking accuracy on validation set
Got 471 / 1000 correct (47.10)

Epoch 7, Iteration 0, loss = 1.3450
Checking accuracy on validation set
Got 481 / 1000 correct (48.10)

Epoch 7, Iteration 100, loss = 1.1290
Checking accuracy on validation set
Got 505 / 1000 correct (50.50)

Epoch 7, Iteration 200, loss = 1.1551
Checking accuracy on validation set
Got 462 / 1000 correct (46.20)

Epoch 7, Iteration 300, loss = 1.1899
Checking accuracy on validation set
Got 483 / 1000 correct (48.30)

Epoch 7, Iteration 400, loss = 1.4443
Checking accuracy on validation set
Got 490 / 1000 correct (49.00)

Epoch 7, Iteration 500, loss = 1.2878
Checking accuracy on validation set
Got 505 / 1000 correct (50.50)

Epoch 7, Iteration 600, loss = 1.1175
Checking accuracy on validation set
Got 485 / 1000 correct (48.50)

Epoch 7, Iteration 700, loss = 1.1581
Checking accuracy on validation set
Got 505 / 1000 correct (50.50)
```

```
Epoch 8, Iteration 0, loss = 0.8849
Checking accuracy on validation set
Got 499 / 1000 correct (49.90)

Epoch 8, Iteration 100, loss = 0.9311
Checking accuracy on validation set
Got 525 / 1000 correct (52.50)

Epoch 8, Iteration 200, loss = 0.8891
Checking accuracy on validation set
Got 509 / 1000 correct (50.90)

Epoch 8, Iteration 300, loss = 0.9142
Checking accuracy on validation set
Got 490 / 1000 correct (49.00)

Epoch 8, Iteration 400, loss = 1.0170
Checking accuracy on validation set
Got 499 / 1000 correct (49.90)

Epoch 8, Iteration 500, loss = 1.0272
Checking accuracy on validation set
Got 487 / 1000 correct (48.70)

Epoch 8, Iteration 600, loss = 1.0908
Checking accuracy on validation set
Got 492 / 1000 correct (49.20)

Epoch 8, Iteration 700, loss = 1.0404
Checking accuracy on validation set
Got 492 / 1000 correct (49.20)

Epoch 9, Iteration 0, loss = 0.7242
Checking accuracy on validation set
Got 485 / 1000 correct (48.50)

Epoch 9, Iteration 100, loss = 0.6241
Checking accuracy on validation set
Got 526 / 1000 correct (52.60)

Epoch 9, Iteration 200, loss = 0.8815
Checking accuracy on validation set
Got 532 / 1000 correct (53.20)

Epoch 9, Iteration 300, loss = 0.6114
Checking accuracy on validation set
Got 519 / 1000 correct (51.90)

Epoch 9, Iteration 400, loss = 0.7224
Checking accuracy on validation set
Got 500 / 1000 correct (50.00)

Epoch 9, Iteration 500, loss = 0.7424
Checking accuracy on validation set
Got 480 / 1000 correct (48.00)

Epoch 9, Iteration 600, loss = 1.1702
Checking accuracy on validation set
Got 483 / 1000 correct (48.30)

Epoch 9, Iteration 700, loss = 0.5659
Checking accuracy on validation set
Got 501 / 1000 correct (50.10)
```

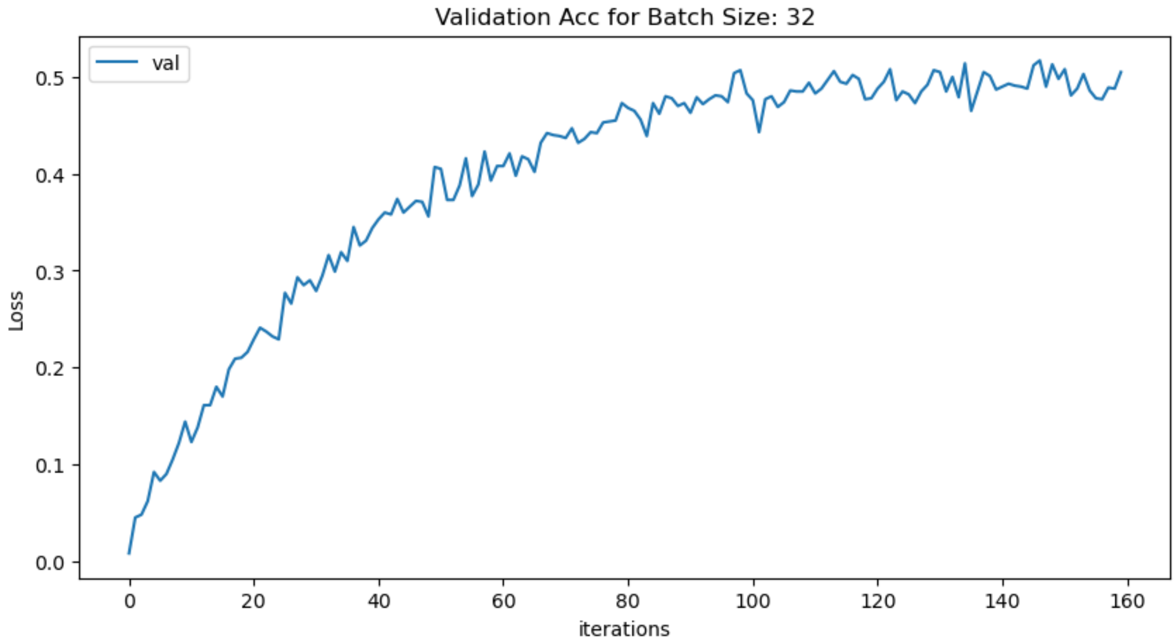
```
Epoch 0, Iteration 0, loss = 4.7556
Checking accuracy on validation set
Got 7 / 1000 correct (0.70)

Epoch 0, Iteration 100, loss = 3.6253
Checking accuracy on validation set
Got 85 / 1000 correct (8.50)

Epoch 0, Iteration 200, loss = 3.4608
Checking accuracy on validation set
Got 143 / 1000 correct (14.30)

Epoch 0, Iteration 300, loss = 3.1931
Checking accuracy on validation set
Got 177 / 1000 correct (17.70)

Epoch 1, Iteration 0, loss = 3.0513
Checking accuracy on validation set
Got 198 / 1000 correct (19.80)

Epoch 1, Iteration 100, loss = 3.0544
Checking accuracy on validation set
Got 216 / 1000 correct (21.60)

Epoch 1, Iteration 200, loss = 3.0928
Checking accuracy on validation set
Got 232 / 1000 correct (23.20)

Epoch 1, Iteration 300, loss = 2.9521
Checking accuracy on validation set
Got 283 / 1000 correct (28.30)

Epoch 2, Iteration 0, loss = 2.6691
Checking accuracy on validation set
Got 251 / 1000 correct (25.10)

Epoch 2, Iteration 100, loss = 2.5106
Checking accuracy on validation set
Got 324 / 1000 correct (32.40)

Epoch 2, Iteration 200, loss = 2.5741
Checking accuracy on validation set
Got 350 / 1000 correct (35.00)

Epoch 2, Iteration 300, loss = 2.2599
Checking accuracy on validation set
Got 321 / 1000 correct (32.10)

Epoch 3, Iteration 0, loss = 2.0427
Checking accuracy on validation set
Got 365 / 1000 correct (36.50)

Epoch 3, Iteration 100, loss = 2.5467
Checking accuracy on validation set
Got 352 / 1000 correct (35.20)

Epoch 3, Iteration 200, loss = 2.2486
Checking accuracy on validation set
Got 381 / 1000 correct (38.10)

Epoch 3, Iteration 300, loss = 2.2145
Checking accuracy on validation set
Got 418 / 1000 correct (41.80)
```

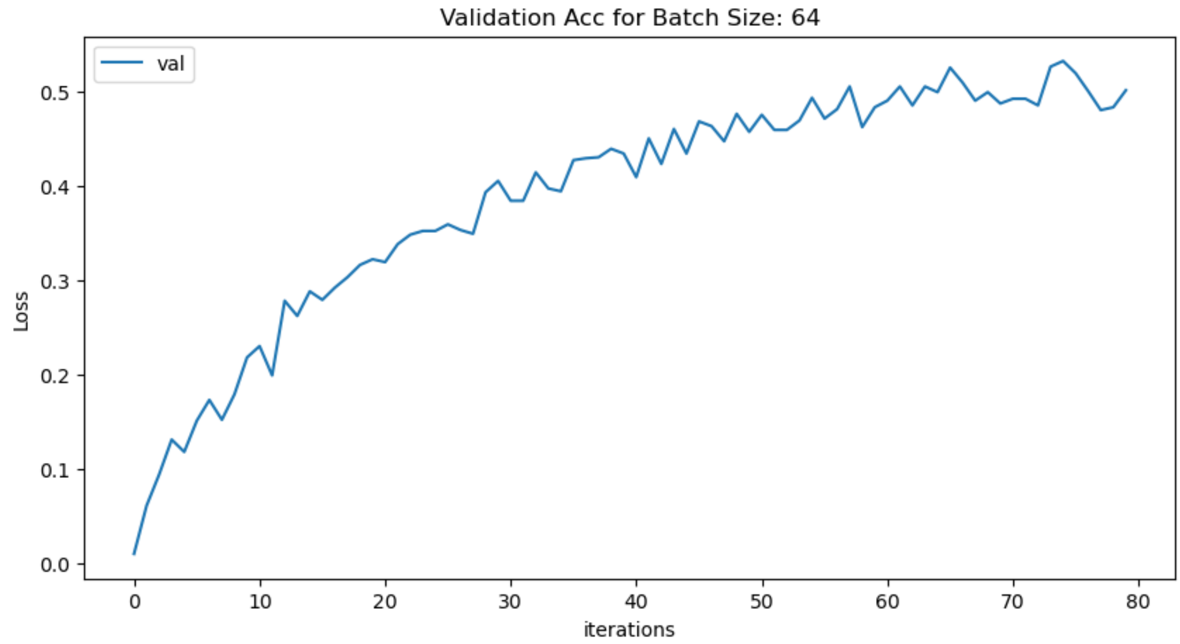
```
Epoch 4, Iteration 0, loss = 2.0365
Checking accuracy on validation set
Got 394 / 1000 correct (39.40)

Epoch 4, Iteration 100, loss = 1.9024
Checking accuracy on validation set
Got 421 / 1000 correct (42.10)

Epoch 4, Iteration 200, loss = 1.7373
Checking accuracy on validation set
Got 395 / 1000 correct (39.50)

Epoch 4, Iteration 300, loss = 1.9476
Checking accuracy on validation set
Got 430 / 1000 correct (43.00)

Epoch 5, Iteration 0, loss = 1.4082
Checking accuracy on validation set
Got 434 / 1000 correct (43.40)

Epoch 5, Iteration 100, loss = 1.4724
Checking accuracy on validation set
Got 402 / 1000 correct (40.20)

Epoch 5, Iteration 200, loss = 1.5636
Checking accuracy on validation set
Got 449 / 1000 correct (44.90)

Epoch 5, Iteration 300, loss = 1.9013
Checking accuracy on validation set
Got 431 / 1000 correct (43.10)

Epoch 6, Iteration 0, loss = 1.4162
Checking accuracy on validation set
Got 461 / 1000 correct (46.10)

Epoch 6, Iteration 100, loss = 1.4935
Checking accuracy on validation set
Got 448 / 1000 correct (44.80)

Epoch 6, Iteration 200, loss = 1.4225
Checking accuracy on validation set
Got 456 / 1000 correct (45.60)

Epoch 6, Iteration 300, loss = 1.3674
Checking accuracy on validation set
Got 461 / 1000 correct (46.10)

Epoch 7, Iteration 0, loss = 1.2198
Checking accuracy on validation set
Got 476 / 1000 correct (47.60)

Epoch 7, Iteration 100, loss = 1.1922
Checking accuracy on validation set
Got 457 / 1000 correct (45.70)

Epoch 7, Iteration 200, loss = 1.2249
Checking accuracy on validation set
Got 503 / 1000 correct (50.30)

Epoch 7, Iteration 300, loss = 1.4181
Checking accuracy on validation set
Got 481 / 1000 correct (48.10)
```

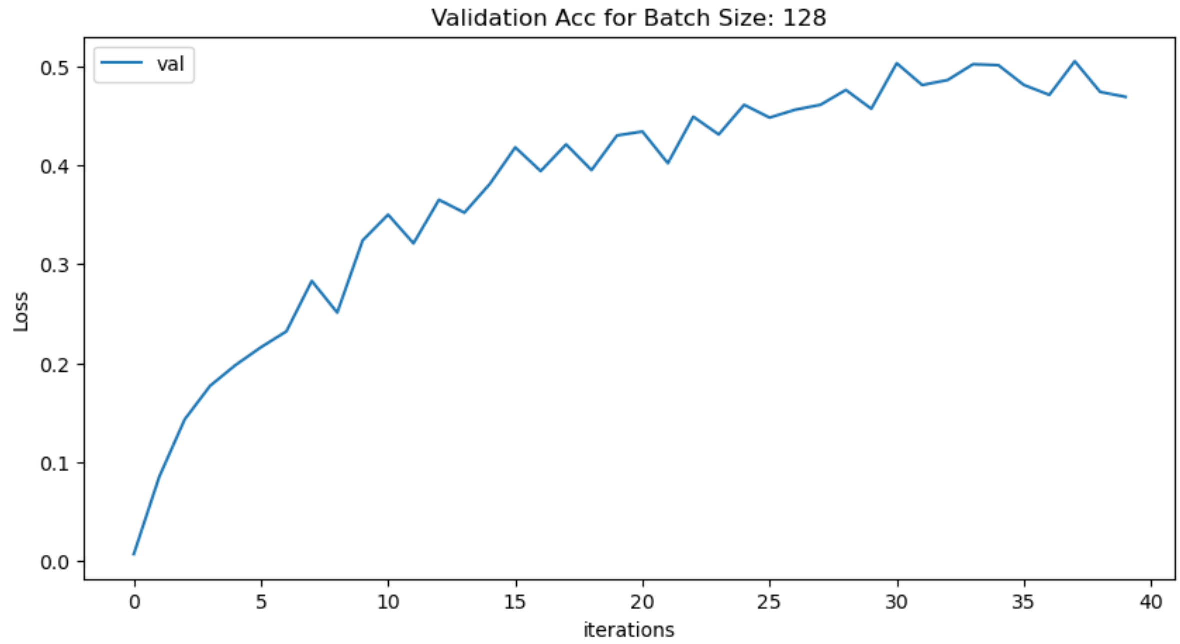
```
Epoch 8, Iteration 0, loss = 1.0907
Checking accuracy on validation set
Got 486 / 1000 correct (48.60)

Epoch 8, Iteration 100, loss = 1.0645
Checking accuracy on validation set
Got 502 / 1000 correct (50.20)

Epoch 8, Iteration 200, loss = 0.7803
Checking accuracy on validation set
Got 501 / 1000 correct (50.10)

Epoch 8, Iteration 300, loss = 0.8583
Checking accuracy on validation set
Got 481 / 1000 correct (48.10)

Epoch 9, Iteration 0, loss = 0.6400
Checking accuracy on validation set
Got 471 / 1000 correct (47.10)

Epoch 9, Iteration 100, loss = 0.8396
Checking accuracy on validation set
Got 505 / 1000 correct (50.50)

Epoch 9, Iteration 200, loss = 0.8774
Checking accuracy on validation set
Got 474 / 1000 correct (47.40)

Epoch 9, Iteration 300, loss = 0.8922
Checking accuracy on validation set
Got 469 / 1000 correct (46.90)
```

In [ ]:

```
for i, acc in enumerate(out) :
    # print(acc)
    plt.figure(figsize=(10,5))
    plt.title("Validation Acc for Batch Size: " + str(batch_sizes[i]))
    plt.plot(acc,label="val")
    plt.xlabel("iterations")
    plt.ylabel("Loss")
    plt.legend()
    plt.show()
```







Discuss effect of Batch Size

TODO: Batch size did not seem to have a significant or noticeable affect on the performance of the model, however it did seem to have a significant effect on the training time, as the training time became much slower at lower batch sizes. This may be due to the optimizer that we are using.

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 50%** 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
- Activations: http://pytorch.org/docs/stable/nn.html#non-linear-activations
- Loss functions: http://pytorch.org/docs/stable/nn.html#loss-functions
- Optimizers: http://pytorch.org/docs/stable/optim.html

Things you might try:

- **Filter size**: Above we used 5x5; would smaller filters be more efficient?
- **Adam Optimizer**: Above we used SGD optimizer, would an Adam optimizer do better?
- **Number of filters**: Above we used 32 filters. Do more or fewer do better?
- **Pooling vs Strided Convolution**: Do you use max pooling or just stride convolutions?
- **Batch normalization**: Try adding spatial batch normalization after convolution layers and vanilla batch normalization after affine layers. Do your networks train faster? You can also try out LayerNorm and GroupNorm.
- **Network architecture**: Can you do better with a deep network? Good architectures to try include:
  - [conv-relu-pool]xN -> [affine]xM -> [softmax or SVM]
  - [conv-relu-conv-relu-pool]xN -> [affine]xM -> [softmax or SVM]
  - [batchnorm-relu-conv]xN -> [affine]xM -> [softmax or SVM]
- **Global Average Pooling**: Instead of flattening and then having multiple affine layers, perform convolutions until your image gets small (7x7 or so) and then perform an average pooling operation to get to a 1x1 image picture (1, 1 , Filter#), which is then reshaped into a (Filter#) vector. This is used in Google's Inception Network (See Table 1 for their architecture).
- **Regularization**: Add l2 weight regularization, or perhaps use Dropout.

Tips for training

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

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

- You should use the validation set for hyperparameter search, and save your test set for evaluating your architecture on the best parameters as selected by the validation set.

Want more improvements?

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

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

Have fun and may the gradients be with you!

In [ ]:

```
# Add official website of pytorch

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.utils.data import sampler

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

import numpy as np
import torch.nn.functional as F  # useful stateless functions

NUM_TRAIN = 49000
batch_size= 64

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

#============================================================================#
# You should try changing the transform for the training data to include    #
# data augmentation such as RandmCrop and HorizontalFlip                    #
# when running the final part of the notebook where you have to achieve     #
# as high accuracy as possible on CIFAR-100.                                #
# Of course you will have to re-run this block for the effect to take place #
#============================================================================#
train_transform = transform = T.Compose([
                T.ToTensor(),
                T.Normalize((0.5071, 0.4867, 0.4408), (0.2675, 0.2565, 0.2761))
            ])

# We set up a Dataset object for each split (train / val / test); Datasets load
# training examples one at a time, so we wrap each Dataset in a DataLoader which
# iterates through the Dataset and forms mini-batches. We divide the CIFAR-100
# training set into train and val sets by passing a Sampler object to the
# DataLoader telling how it should sample from the underlying Dataset.
cifar100_train = dset.CIFAR100('./datasets/cifar100', train=True, download=True,
                             transform=train_transform)
loader_train = DataLoader(cifar100_train, batch_size=batch_size, num_workers=2,
                          sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN)))
```

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

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

USE_GPU = True
num_class = 100
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)

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

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

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
    val_acc = []
    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 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('Epoch %d, Iteration %d, loss = %.4f' % (e, t, loss.item()))
                acc = check_accuracy_part34(loader_val, model)
                val_acc.append(acc)
                print()
    try:
        return val_acc, model
    except:
        pass
```

```
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
using device: cuda
```

In [ ]:

```
class Inception(nn.Module):
    def __init__(self, input_channels, n1x1, n3x3_reduce, n3x3, n5x5_reduce, n5x5,
        super().__init__()
        self.b1 = nn.Sequential(
            nn.Conv2d(input_channels, n1x1, kernel_size=(1, 1)),
            nn.BatchNorm2d(n1x1),
            nn.ReLU(inplace=True)
        )

        self.b2 = nn.Sequential(
            nn.Conv2d(input_channels, n3x3_reduce, kernel_size=(1, 1)),
            nn.BatchNorm2d(n3x3_reduce),
            nn.ReLU(inplace=True),
            nn.Conv2d(n3x3_reduce, n3x3, kernel_size=(3, 3), padding=1),
            nn.BatchNorm2d(n3x3),
            nn.ReLU(inplace=True)
        )

        self.b3 = nn.Sequential(
            nn.Conv2d(input_channels, n5x5_reduce, kernel_size=(1, 1)),
            nn.BatchNorm2d(n5x5_reduce),
            nn.ReLU(inplace=True),
            nn.Conv2d(n5x5_reduce, n5x5, kernel_size=(5, 5), padding=2),
            nn.BatchNorm2d(n5x5),
            nn.ReLU(inplace=True)
        )

        self.b4 = nn.Sequential(
            nn.MaxPool2d((3, 3), stride=1, padding=1),
            nn.Conv2d(input_channels, pool_proj, kernel_size=1),
            nn.BatchNorm2d(pool_proj),
            nn.ReLU(inplace=True)
        )
```

```python
    def forward(self, x):
        return torch.cat([self.b1(x), self.b2(x), self.b3(x), self.b4(x)], dim=1)


class GoogLeNet(nn.Module):

    def __init__(self, num_class=100):
        super().__init__()
        self.layer = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=(3, 3), padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=(3, 3), padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 192, kernel_size=(3, 3), padding=1, bias=False),
            nn.BatchNorm2d(192),
            nn.ReLU(inplace=True),
        )

        self.a3 = Inception(192, 64, 96, 128, 16, 32, 32)
        self.b3 = Inception(256, 128, 128, 192, 32, 96, 64)

        self.mp = nn.MaxPool2d((3, 3), stride=2, padding=1)

        self.a4 = Inception(480, 192, 96, 208, 16, 48, 64)
        self.b4 = Inception(512, 160, 112, 224, 24, 64, 64)
        self.c4 = Inception(512, 128, 128, 256, 24, 64, 64)
        self.d4 = Inception(512, 112, 144, 288, 32, 64, 64)
        self.e4 = Inception(528, 256, 160, 320, 32, 128, 128)

        self.a5 = Inception(832, 256, 160, 320, 32, 128, 128)
        self.b5 = Inception(832, 384, 192, 384, 48, 128, 128)

        self.ap = nn.AdaptiveAvgPool2d((1, 1))
        self.dropout = nn.Dropout2d(p=0.4)
        self.fc = nn.Linear(1024, num_class)

    def forward(self, x):
        x = self.layer(x)

        x = self.mp(x)

        x = self.a3(x)
        x = self.b3(x)

        x = self.mp(x)

        x = self.a4(x)
        x = self.b4(x)
        x = self.c4(x)
        x = self.d4(x)
        x = self.e4(x)

        x = self.mp(x)

        x = self.a5(x)
        x = self.b5(x)

        x = self.ap(x)
        x = self.dropout(x)
        x = x.view(x.size()[0], -1)
        x = self.fc(x)
```

```
        return x
```

In [ ]:

```
################################################################################
# TODO:                                                                        #
# Experiment with any architectures, optimizers, and hyperparameters.          #
# Achieve AT LEAST 52% 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.   #
################################################################################
from matplotlib import pyplot as plt
device = torch.device('cuda')
learning_rate = 1e-3
model = None
optimizer = None

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
model = GoogLeNet().to(device)
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 52% accuracy.
gnet, best_model = train_part34(model, optimizer, epochs=10)
```

```
Epoch 0, Iteration 0, loss = 4.6034
Checking accuracy on validation set
Got 12 / 1000 correct (1.20)

Epoch 0, Iteration 100, loss = 4.1190
Checking accuracy on validation set
Got 50 / 1000 correct (5.00)

Epoch 0, Iteration 200, loss = 4.2251
Checking accuracy on validation set
Got 70 / 1000 correct (7.00)

Epoch 0, Iteration 300, loss = 4.0041
Checking accuracy on validation set
Got 99 / 1000 correct (9.90)

Epoch 0, Iteration 400, loss = 3.7956
Checking accuracy on validation set
Got 90 / 1000 correct (9.00)

Epoch 0, Iteration 500, loss = 3.7432
Checking accuracy on validation set
Got 105 / 1000 correct (10.50)

Epoch 0, Iteration 600, loss = 3.3255
Checking accuracy on validation set
Got 148 / 1000 correct (14.80)

Epoch 0, Iteration 700, loss = 3.3667
Checking accuracy on validation set
Got 185 / 1000 correct (18.50)

Epoch 1, Iteration 0, loss = 3.2067
Checking accuracy on validation set
Got 158 / 1000 correct (15.80)

Epoch 1, Iteration 100, loss = 3.1602
Checking accuracy on validation set
Got 180 / 1000 correct (18.00)

Epoch 1, Iteration 200, loss = 2.9808
Checking accuracy on validation set
Got 210 / 1000 correct (21.00)

Epoch 1, Iteration 300, loss = 2.5936
Checking accuracy on validation set
Got 243 / 1000 correct (24.30)

Epoch 1, Iteration 400, loss = 2.8843
Checking accuracy on validation set
Got 250 / 1000 correct (25.00)

Epoch 1, Iteration 500, loss = 2.8817
Checking accuracy on validation set
Got 239 / 1000 correct (23.90)

Epoch 1, Iteration 600, loss = 2.4402
Checking accuracy on validation set
Got 270 / 1000 correct (27.00)

Epoch 1, Iteration 700, loss = 2.1320
Checking accuracy on validation set
Got 309 / 1000 correct (30.90)
```

```
Epoch 2, Iteration 0, loss = 2.2196
Checking accuracy on validation set
Got 301 / 1000 correct (30.10)

Epoch 2, Iteration 100, loss = 2.5661
Checking accuracy on validation set
Got 297 / 1000 correct (29.70)

Epoch 2, Iteration 200, loss = 2.4526
Checking accuracy on validation set
Got 353 / 1000 correct (35.30)

Epoch 2, Iteration 300, loss = 2.1228
Checking accuracy on validation set
Got 344 / 1000 correct (34.40)

Epoch 2, Iteration 400, loss = 2.0270
Checking accuracy on validation set
Got 382 / 1000 correct (38.20)

Epoch 2, Iteration 500, loss = 1.9594
Checking accuracy on validation set
Got 385 / 1000 correct (38.50)

Epoch 2, Iteration 600, loss = 2.1416
Checking accuracy on validation set
Got 400 / 1000 correct (40.00)

Epoch 2, Iteration 700, loss = 2.3350
Checking accuracy on validation set
Got 386 / 1000 correct (38.60)

Epoch 3, Iteration 0, loss = 1.7800
Checking accuracy on validation set
Got 404 / 1000 correct (40.40)

Epoch 3, Iteration 100, loss = 1.6499
Checking accuracy on validation set
Got 388 / 1000 correct (38.80)

Epoch 3, Iteration 200, loss = 1.6657
Checking accuracy on validation set
Got 413 / 1000 correct (41.30)

Epoch 3, Iteration 300, loss = 1.7696
Checking accuracy on validation set
Got 435 / 1000 correct (43.50)

Epoch 3, Iteration 400, loss = 1.7867
Checking accuracy on validation set
Got 435 / 1000 correct (43.50)

Epoch 3, Iteration 500, loss = 1.9792
Checking accuracy on validation set
Got 415 / 1000 correct (41.50)

Epoch 3, Iteration 600, loss = 1.7166
Checking accuracy on validation set
Got 459 / 1000 correct (45.90)

Epoch 3, Iteration 700, loss = 2.2118
Checking accuracy on validation set
Got 428 / 1000 correct (42.80)
```

```
Epoch 4, Iteration 0, loss = 1.7293
Checking accuracy on validation set
Got 460 / 1000 correct (46.00)

Epoch 4, Iteration 100, loss = 1.8044
Checking accuracy on validation set
Got 457 / 1000 correct (45.70)

Epoch 4, Iteration 200, loss = 1.6939
Checking accuracy on validation set
Got 463 / 1000 correct (46.30)

Epoch 4, Iteration 300, loss = 1.5705
Checking accuracy on validation set
Got 469 / 1000 correct (46.90)

Epoch 4, Iteration 400, loss = 1.8756
Checking accuracy on validation set
Got 471 / 1000 correct (47.10)

Epoch 4, Iteration 500, loss = 1.7158
Checking accuracy on validation set
Got 485 / 1000 correct (48.50)

Epoch 4, Iteration 600, loss = 1.8145
Checking accuracy on validation set
Got 496 / 1000 correct (49.60)

Epoch 4, Iteration 700, loss = 1.5088
Checking accuracy on validation set
Got 481 / 1000 correct (48.10)

Epoch 5, Iteration 0, loss = 1.3161
Checking accuracy on validation set
Got 500 / 1000 correct (50.00)

Epoch 5, Iteration 100, loss = 1.7933
Checking accuracy on validation set
Got 519 / 1000 correct (51.90)

Epoch 5, Iteration 200, loss = 1.3881
Checking accuracy on validation set
Got 512 / 1000 correct (51.20)

Epoch 5, Iteration 300, loss = 1.2345
Checking accuracy on validation set
Got 500 / 1000 correct (50.00)

Epoch 5, Iteration 400, loss = 1.3172
Checking accuracy on validation set
Got 551 / 1000 correct (55.10)

Epoch 5, Iteration 500, loss = 1.3747
Checking accuracy on validation set
Got 524 / 1000 correct (52.40)

Epoch 5, Iteration 600, loss = 1.5698
Checking accuracy on validation set
Got 502 / 1000 correct (50.20)

Epoch 5, Iteration 700, loss = 1.4454
Checking accuracy on validation set
Got 501 / 1000 correct (50.10)
```

```
Epoch 6, Iteration 0, loss = 1.5631
Checking accuracy on validation set
Got 532 / 1000 correct (53.20)

Epoch 6, Iteration 100, loss = 1.0490
Checking accuracy on validation set
Got 521 / 1000 correct (52.10)

Epoch 6, Iteration 200, loss = 1.6105
Checking accuracy on validation set
Got 519 / 1000 correct (51.90)

Epoch 6, Iteration 300, loss = 1.2281
Checking accuracy on validation set
Got 535 / 1000 correct (53.50)

Epoch 6, Iteration 400, loss = 1.1212
Checking accuracy on validation set
Got 508 / 1000 correct (50.80)

Epoch 6, Iteration 500, loss = 1.2327
Checking accuracy on validation set
Got 550 / 1000 correct (55.00)

Epoch 6, Iteration 600, loss = 1.0500
Checking accuracy on validation set
Got 551 / 1000 correct (55.10)

Epoch 6, Iteration 700, loss = 0.9925
Checking accuracy on validation set
Got 534 / 1000 correct (53.40)

Epoch 7, Iteration 0, loss = 1.0480
Checking accuracy on validation set
Got 501 / 1000 correct (50.10)

Epoch 7, Iteration 100, loss = 0.9066
Checking accuracy on validation set
Got 542 / 1000 correct (54.20)

Epoch 7, Iteration 200, loss = 1.1470
Checking accuracy on validation set
Got 558 / 1000 correct (55.80)

Epoch 7, Iteration 300, loss = 1.1010
Checking accuracy on validation set
Got 548 / 1000 correct (54.80)

Epoch 7, Iteration 400, loss = 1.1127
Checking accuracy on validation set
Got 554 / 1000 correct (55.40)

Epoch 7, Iteration 500, loss = 1.0775
Checking accuracy on validation set
Got 550 / 1000 correct (55.00)

Epoch 7, Iteration 600, loss = 1.0713
Checking accuracy on validation set
Got 574 / 1000 correct (57.40)

Epoch 7, Iteration 700, loss = 1.1424
Checking accuracy on validation set
Got 568 / 1000 correct (56.80)
```

```
Epoch 8, Iteration 0, loss = 0.8704
Checking accuracy on validation set
Got 545 / 1000 correct (54.50)

Epoch 8, Iteration 100, loss = 0.8237
Checking accuracy on validation set
Got 562 / 1000 correct (56.20)

Epoch 8, Iteration 200, loss = 1.2278
Checking accuracy on validation set
Got 553 / 1000 correct (55.30)

Epoch 8, Iteration 300, loss = 0.8180
Checking accuracy on validation set
Got 560 / 1000 correct (56.00)

Epoch 8, Iteration 400, loss = 0.8334
Checking accuracy on validation set
Got 565 / 1000 correct (56.50)

Epoch 8, Iteration 500, loss = 0.9141
Checking accuracy on validation set
Got 563 / 1000 correct (56.30)

Epoch 8, Iteration 600, loss = 1.2058
Checking accuracy on validation set
Got 553 / 1000 correct (55.30)

Epoch 8, Iteration 700, loss = 1.2206
Checking accuracy on validation set
Got 559 / 1000 correct (55.90)

Epoch 9, Iteration 0, loss = 0.9621
Checking accuracy on validation set
Got 542 / 1000 correct (54.20)

Epoch 9, Iteration 100, loss = 0.7697
Checking accuracy on validation set
Got 574 / 1000 correct (57.40)

Epoch 9, Iteration 200, loss = 0.9294
Checking accuracy on validation set
Got 563 / 1000 correct (56.30)

Epoch 9, Iteration 300, loss = 0.8053
Checking accuracy on validation set
Got 535 / 1000 correct (53.50)

Epoch 9, Iteration 400, loss = 0.6958
Checking accuracy on validation set
Got 560 / 1000 correct (56.00)

Epoch 9, Iteration 500, loss = 0.6906
Checking accuracy on validation set
Got 567 / 1000 correct (56.70)

Epoch 9, Iteration 600, loss = 0.7044
Checking accuracy on validation set
Got 571 / 1000 correct (57.10)

Epoch 9, Iteration 700, loss = 0.8370
Checking accuracy on validation set
Got 561 / 1000 correct (56.10)
```

In [ ]:
```
plt.figure(figsize=(10,5))
plt.title("Validation Loss for Batch Size: " + str(64))
plt.plot(gnet,label="val")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

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.

TODO: I implemented the GoogLeNet, which uses Inception blocks incorporated with 1x1 convoutions to reduce dimensionality, thus reducing computational needs, along with average pooling, in my case using adaptive average pooling to simplify implementation, which decreases the number of trainable parameters to improve accuracy even further. I created a separate class for the Inception block, which consists of concatenated 1x1 convolution, 3x3 convolution, 5x5 convolution and a 3x3 max pooling layers, all respectively stacked with a 1x1 convolution, excluding obviously the first 1x1 convolution. All of the channel sizes, kernel sizes, pool proj and stride values were taken directly from the GoogLeNet paper's picture of the architecture, and the initial convolutional layer for the feature map was separately designed for the Cifar-100 dataset as its image size is different from the ImageNet dataset originally used to train in the paper. Even without the use of residual blocks, this architecture was able to reach an accuracy well above 50%. The other hyperparameters were played around with, but best performance was given with the original set, so I ended up using Adam optimizer, 1e-3 learning rate and 64 batch size.

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

```
Checking accuracy on test set
Got 5595 / 10000 correct (55.95)
```

Out[ ]:
```
0.5595
```

The test accuracy is relatively close to the validation accuracy, which suggests a well trained model that reflects the training process well.