final_ade20k_models

May 30, 2021

1 CS231n Final Project

2 Matthew Kaplan and Avery Rogers

3 mkaplan1@stanford.edu, averyr@stanford.edu

3.0.1 For our project, we explore whether appending a segmentation mask to the training data will lead to better scene classification performance versus vanilla scene classification methods.

Note: This was run on Colab Pro, with GPU runtime for cuda, and a High-RAM (25 GB) environment.

3.1 Part 0: Setup the Notebook, Load, and Preprocess the data.

[ ]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

# Enter the filename where the project
FOLDERNAME = 'CS231n Project/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Import sys so we can access utils files with imports.
# ie utils_ade20k in next code block.
import sys
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

[ ]:
```
# Run below cell to clear up RAM.
import gc
gc.collect()
```

[ ]:
```
59
```

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
```

```
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 torch.nn.functional as F
import torchvision.models as models

USE_GPU = True
dtype = torch.float32 # We will be using float throughout this tutorial.

if USE_GPU and torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

# Constant to control how frequently we print train loss.
print_every = 50
print('using device:', device)
```

```
using device: cuda
```

Here, we load the training, validation, and test data from Google Drive for speed and efficiency purposes. We had to pull the data from the dataset using a large dictionary that the MIT Researchers behind the dataset helped setup. For examples of how to parse through the dataset, see the ade20k_starter.ipynb notebook.

To see our work pulling training, validation, and test sets for the top 50 most popularly seen classes, see the Python scripts ade20k_scene_parser.py and load_ade20k_data.py scripts that we authored from scratch. This took up the bulk of our efforts up to the week following the milestone.

[ ]:
```
train_data_path = '/content/drive/MyDrive/CS231n Project/Datasets/x_train.npy'
x_train = torch.tensor(np.load(train_data_path)).double()

val_data_path = '/content/drive/MyDrive/CS231n Project/Datasets/x_val.npy'
x_val = torch.tensor(np.load(val_data_path)).double()

test_data_path = '/content/drive/MyDrive/CS231n Project/Datasets/x_test.npy'
x_test = torch.tensor(np.load(test_data_path)).double()

y_train_seg_path = '/content/drive/MyDrive/CS231n Project/Datasets/y_train_seg.
 ↪npy'
y_train_seg = torch.tensor(np.load(y_train_seg_path)).double()

y_val_seg_path = '/content/drive/MyDrive/CS231n Project/Datasets/y_val_seg.npy'
y_val_seg = torch.tensor(np.load(y_val_seg_path)).double()
```

```
y_test_seg_path = '/content/drive/MyDrive/CS231n Project/Datasets/y_test_seg.
 ↪npy'
y_test_seg = torch.tensor(np.load(y_test_seg_path)).double()

y_train_scenes_path = '/content/drive/MyDrive/CS231n Project/Datasets/
 ↪y_train_scenes.npy'
y_train_scenes = torch.tensor(np.load(y_train_scenes_path)).long()

y_val_scenes_path = '/content/drive/MyDrive/CS231n Project/Datasets/
 ↪y_val_scenes.npy'
y_val_scenes = torch.tensor(np.load(y_val_scenes_path)).long()

y_test_scenes_path = '/content/drive/MyDrive/CS231n Project/Datasets/
 ↪y_test_scenes.npy'
y_test_scenes = torch.tensor(np.load(y_test_scenes_path)).long()
```

[ ]:
```
# Verify shapes here.
print('x_train: {}'.format(x_train.shape))
print('x_val: {}'.format(x_val.shape))
print('x_test: {}'.format(x_test.shape))
print('y_train_seg: {}'.format(y_train_seg.shape))
print('y_val_seg: {}'.format(y_val_seg.shape))
print('y_test_seg: {}'.format(y_test_seg.shape))
print('y_train_scenes: {}'.format(y_train_scenes.shape))
print('y_val_scenes: {}'.format(y_val_scenes.shape))
print('y_test_scenes: {}'.format(y_test_scenes.shape))
```

```
x_train: torch.Size([13181, 128, 128, 3])
x_val: torch.Size([1369, 128, 128, 3])
x_test: torch.Size([1211, 128, 128, 3])
y_train_seg: torch.Size([13181, 128, 128, 1])
y_val_seg: torch.Size([1369, 128, 128, 1])
y_test_seg: torch.Size([1211, 128, 128, 1])
y_train_scenes: torch.Size([13181])
y_val_scenes: torch.Size([1369])
y_test_scenes: torch.Size([1211])
```

[ ]:
```
# WARNING: ONLY RUN THIS CELL ONCE

BATCH_SIZE = 64

# Preprocess the x_data
NUM_TRAIN = x_train.shape[0]
NUM_VAL = x_val.shape[0]
NUM_TEST = x_test.shape[0]
```

```
# Subtract mean
x_train_mean = torch.mean(x_train, dim=0)
x_train -= x_train_mean
x_val -= x_train_mean
x_test -= x_train_mean

# Divide by std (255)
x_train /= 255
x_val /= 255
x_test /= 255

# Permute the axes for pytorch layers
x_train = x_train.permute(0, 3, 1, 2)
x_val = x_val.permute(0, 3, 1, 2)
x_test = x_test.permute(0, 3, 1, 2)
y_train_seg = y_train_seg.permute(0, 3, 1, 2)
y_val_seg = y_val_seg.permute(0, 3, 1, 2)
y_test_seg = y_test_seg.permute(0, 3, 1, 2)

# Keep track of these values (will need to use to preprocess y)
y_train_mean = torch.mean(y_train_seg, dim=0)
# std
y_train_std = torch.std(y_train_seg, dim=0)
```

3.2 Part 1: Baseline Model

For this model, we want to map raw image pixels to a scene class (vanilla scene classification). This will give us a baseline to see if adding a segmentation mask to the input will improve performance. For our baseline model, we will use a pretrained deep network ResNet50.

[ ]:
```
resnet50 = models.resnet50(pretrained=True)
resnet50.fc = nn.Linear(2048, 50, bias=True)
print(resnet50)
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
```

```
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer2): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
```

```
track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (3): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
      (relu): ReLU(inplace=True)
    )
  )
  (layer3): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
```

```
    (3): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (4): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (5): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer4): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
```

```
track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=2048, out_features=50, bias=True)
)
```

The below functions had the high-level architecture provided by CS231n Course Staff in the PyTorch notebook from Assignment 2, but we made big changes to accomodate the different dataset and manually pulling random batches and getting accuracy on the different splits of data. This applies to the various check accuracy and train models that will appear later in the notebook in later parts.

[ ]:
```
def check_accuracy(x, y, model, input_string):
    num_correct = 0
    num_samples = 0
    model.eval()  # set model to evaluation mode
    if input_string == 'Test':
      num_iterations = int(NUM_TEST / BATCH_SIZE) + 1
    else:
      num_iterations = int(NUM_VAL / BATCH_SIZE) + 1
    with torch.no_grad():
        for i in range(num_iterations):
          # Need to get x and y
          if input_string == 'Train':
            batch_indices = np.random.choice(range(NUM_TRAIN), size=BATCH_SIZE,␣
↪replace=False)
            x_batch = x[batch_indices].type(dtype)
            y_batch = y[batch_indices].type(dtype).long()
          else:
            if i == num_iterations - 1:
              x_batch = x[num_samples:].type(dtype)
              y_batch = y[num_samples:].type(dtype).long()
            else:
              x_batch = x[num_samples:num_samples + BATCH_SIZE].type(dtype)
              y_batch = y[num_samples:num_samples + BATCH_SIZE].type(dtype).
↪long()
          x_batch = x_batch.to(device=device, dtype=dtype)  # move to device, e.
↪g. GPU
          y_batch = y_batch.to(device=device, dtype=torch.long)
          scores = model(x_batch)
          _, preds = scores.max(1)
          num_correct += (preds == y_batch).sum()
          num_samples += preds.size(0)
        acc = float(num_correct) / num_samples
        print('{}: Got {} / {} correct ({})'.format(input_string, num_correct,␣
↪num_samples, round(100 * acc, 2)))
```

Similar to the function above, the below functions had the high-level structure provided by the train34 function from Assignment 2 of CS231n, with major changes to accomodate manually pulling the mini-batches of data and accomodating the data structure specific to the ADE20k dataset we pulled.

[ ]:
```
def train_baseline(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
```

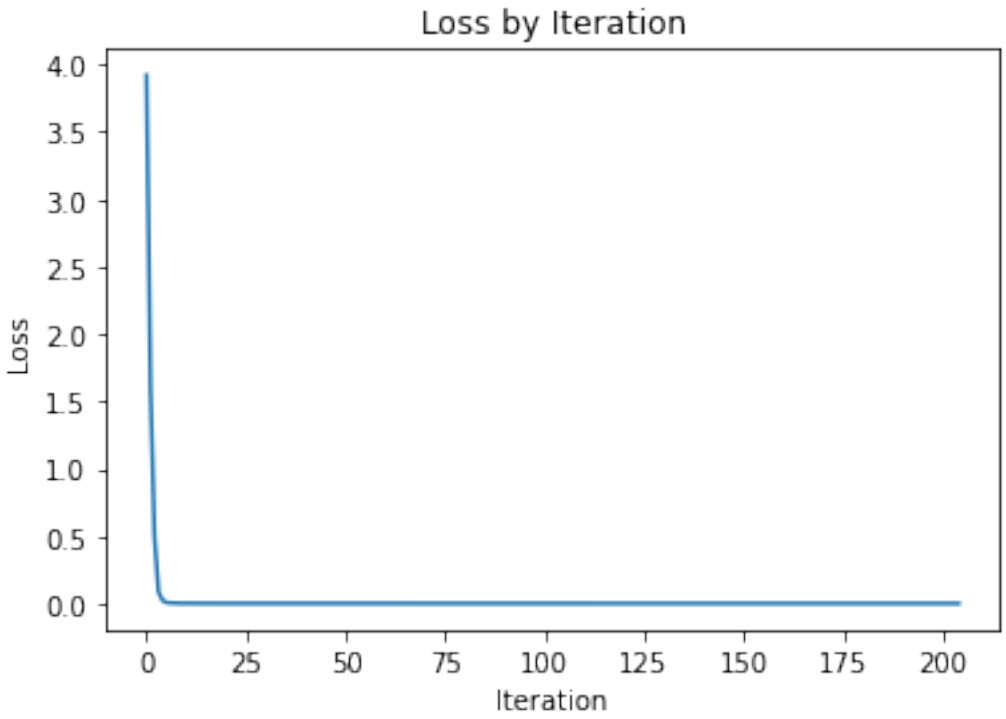
```
    - epochs: (Optional) A Python integer giving the number of epochs to train␣
↪for

    Returns: Nothing, but prints model accuracies during training.
    """
    loss_arr = []
    num_iter = []
    model = model.to(device=device)  # move the model parameters to CPU/GPU
    for e in range(epochs):
        for t in range(int(NUM_TRAIN / BATCH_SIZE)):
            num_iter.append(e * int(NUM_TRAIN / BATCH_SIZE) + t)
            batch_indices = np.random.choice(range(NUM_TRAIN), size=BATCH_SIZE,
                                             replace=False)
            x = x_train[batch_indices].type(dtype)
            y = y_train_scenes[batch_indices].type(dtype)
            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)

            loss_arr.append(loss)
            # Zero out all of the gradients for the variables which the␣
↪optimizer
            # will update.
            optimizer.zero_grad()

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

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

            if t % print_every == 0:
                print('Iteration %d, loss = %.4f' % (t, loss.item()))
                check_accuracy(x_train, y_train_scenes, model, 'Train')
                check_accuracy(x_val, y_val_scenes, model, 'Val')
                print()
    plt.plot(num_iter, loss_arr)
    plt.xlabel('Iteration')
    plt.ylabel('Loss')
    plt.title('Loss by Iteration')
```

[ ]:
```
def train_baseline_overfit(model, optimizer, epochs=1):
  loss_arr = []
  num_iter = []
  model = model.to(device=device)  # move the model parameters to CPU/GPU
  batch_indices = np.random.choice(range(NUM_TRAIN), size=BATCH_SIZE,
                                                        replace=False)
  x = x_train[batch_indices].type(dtype)
  y = y_train_scenes[batch_indices].type(dtype)
  for e in range(epochs):
    for t in range(int(NUM_TRAIN / BATCH_SIZE)):
        num_iter.append(e * int(NUM_TRAIN / BATCH_SIZE) + t)
        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)

        if (t % print_every == 0):
          _, preds = scores.max(1)
          preds = preds
          num_correct = (preds == y).sum()
          num_samples = preds.size(0)
          acc = float(num_correct) / num_samples
          print('{}: Got {} / {} correct ({})'.format('Train', num_correct,␣
↪num_samples, round(100 * acc, 2)))
          print('Loss = {}'.format(round(float(loss), 2)))


        loss_arr.append(loss)
        # Zero out all of the gradients for the variables which the optimizer
        # will update.
        optimizer.zero_grad()

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

        # Actually update the parameters of the model using the gradients
        # computed by the backwards pass.
        optimizer.step()
  plt.plot(num_iter, loss_arr)
  plt.xlabel('Iteration')
  plt.ylabel('Loss')
  plt.title('Loss by Iteration')
```

First, let's see if we can overfit ResNet50 on a small batch of training examples (just to verify the model can learn given our setup):

[ ]:
```
optimizer = optim.Adam(resnet50.parameters(), lr=.001)
train_baseline_overfit(resnet50, optimizer, epochs=1)
```

```
Train: Got 7 / 64 correct (10.94)
Loss = 3.92
Train: Got 64 / 64 correct (100.0)
Loss = 0.0
Train: Got 64 / 64 correct (100.0)
Loss = 0.0
Train: Got 64 / 64 correct (100.0)
Loss = 0.0
Train: Got 64 / 64 correct (100.0)
Loss = 0.0
```



The good news is that ResNet50 successfully overfit on a small batch of training examples. It's now time to train on all training examples using minibatch Adam updates. This will give us a baseline accuracy going forward:

[ ]:
```
optimizer = optim.Adam(resnet50.parameters(), lr=.001, weight_decay=1e-4)
train_baseline(resnet50, optimizer, epochs=10)
```

```
Iteration 0, loss = 10.4495
Train: Got 244 / 1408 correct (17.33)
Val: Got 185 / 1369 correct (13.51)
```

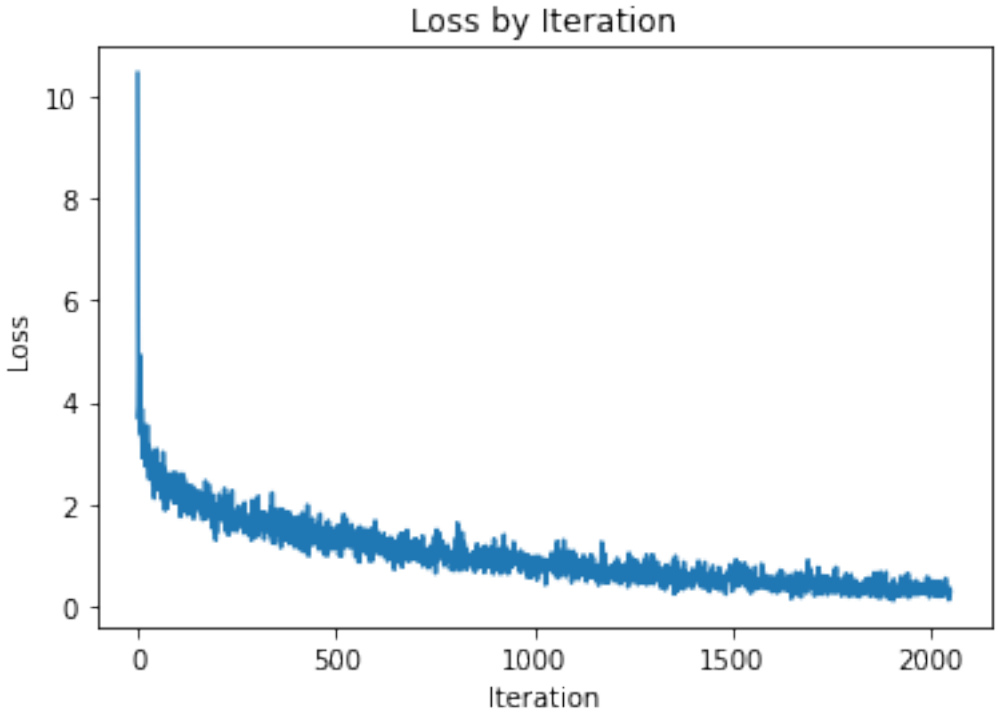
```
Iteration 50, loss = 2.5067
Train: Got 402 / 1408 correct (28.55)
Val: Got 375 / 1369 correct (27.39)

Iteration 100, loss = 2.4426
Train: Got 464 / 1408 correct (32.95)
Val: Got 374 / 1369 correct (27.32)

Iteration 150, loss = 2.0351
Train: Got 539 / 1408 correct (38.28)
Val: Got 389 / 1369 correct (28.41)

Iteration 200, loss = 1.9970
Train: Got 593 / 1408 correct (42.12)
Val: Got 421 / 1369 correct (30.75)

Iteration 0, loss = 2.1965
Train: Got 598 / 1408 correct (42.47)
Val: Got 426 / 1369 correct (31.12)

Iteration 50, loss = 1.7546
Train: Got 619 / 1408 correct (43.96)
Val: Got 463 / 1369 correct (33.82)

Iteration 100, loss = 1.7253
Train: Got 684 / 1408 correct (48.58)
Val: Got 475 / 1369 correct (34.7)

Iteration 150, loss = 1.6847
Train: Got 680 / 1408 correct (48.3)
Val: Got 466 / 1369 correct (34.04)

Iteration 200, loss = 1.4853
Train: Got 768 / 1408 correct (54.55)
Val: Got 534 / 1369 correct (39.01)

Iteration 0, loss = 1.3533
Train: Got 756 / 1408 correct (53.69)
Val: Got 512 / 1369 correct (37.4)

Iteration 50, loss = 1.8125
Train: Got 794 / 1408 correct (56.39)
Val: Got 543 / 1369 correct (39.66)

Iteration 100, loss = 1.4272
Train: Got 783 / 1408 correct (55.61)
Val: Got 515 / 1369 correct (37.62)
```

```
Iteration 150, loss = 1.5635
Train: Got 836 / 1408 correct (59.38)
Val: Got 552 / 1369 correct (40.32)

Iteration 200, loss = 1.3434
Train: Got 787 / 1408 correct (55.89)
Val: Got 453 / 1369 correct (33.09)

Iteration 0, loss = 1.0704
Train: Got 853 / 1408 correct (60.58)
Val: Got 592 / 1369 correct (43.24)

Iteration 50, loss = 1.2738
Train: Got 914 / 1408 correct (64.91)
Val: Got 548 / 1369 correct (40.03)

Iteration 100, loss = 0.9177
Train: Got 974 / 1408 correct (69.18)
Val: Got 623 / 1369 correct (45.51)

Iteration 150, loss = 1.0416
Train: Got 895 / 1408 correct (63.57)
Val: Got 580 / 1369 correct (42.37)

Iteration 200, loss = 0.9359
Train: Got 904 / 1408 correct (64.2)
Val: Got 553 / 1369 correct (40.39)

Iteration 0, loss = 1.2569
Train: Got 846 / 1408 correct (60.09)
Val: Got 548 / 1369 correct (40.03)

Iteration 50, loss = 1.1642
Train: Got 931 / 1408 correct (66.12)
Val: Got 557 / 1369 correct (40.69)

Iteration 100, loss = 0.9260
Train: Got 1003 / 1408 correct (71.24)
Val: Got 567 / 1369 correct (41.42)

Iteration 150, loss = 0.7855
Train: Got 917 / 1408 correct (65.13)
Val: Got 575 / 1369 correct (42.0)

Iteration 200, loss = 0.7536
Train: Got 975 / 1408 correct (69.25)
Val: Got 584 / 1369 correct (42.66)
```

```
Iteration 0, loss = 0.9293
Train: Got 1024 / 1408 correct (72.73)
Val: Got 641 / 1369 correct (46.82)

Iteration 50, loss = 0.8563
Train: Got 997 / 1408 correct (70.81)
Val: Got 572 / 1369 correct (41.78)

Iteration 100, loss = 0.6141
Train: Got 1030 / 1408 correct (73.15)
Val: Got 587 / 1369 correct (42.88)

Iteration 150, loss = 0.4721
Train: Got 1042 / 1408 correct (74.01)
Val: Got 558 / 1369 correct (40.76)

Iteration 200, loss = 0.5832
Train: Got 1101 / 1408 correct (78.2)
Val: Got 607 / 1369 correct (44.34)

Iteration 0, loss = 0.5639
Train: Got 1088 / 1408 correct (77.27)
Val: Got 587 / 1369 correct (42.88)

Iteration 50, loss = 0.7468
Train: Got 1103 / 1408 correct (78.34)
Val: Got 596 / 1369 correct (43.54)

Iteration 100, loss = 0.3237
Train: Got 1149 / 1408 correct (81.61)
Val: Got 630 / 1369 correct (46.02)

Iteration 150, loss = 0.5120
Train: Got 1147 / 1408 correct (81.46)
Val: Got 543 / 1369 correct (39.66)

Iteration 200, loss = 0.8654
Train: Got 1092 / 1408 correct (77.56)
Val: Got 594 / 1369 correct (43.39)

Iteration 0, loss = 0.5297
Train: Got 1054 / 1408 correct (74.86)
Val: Got 583 / 1369 correct (42.59)

Iteration 50, loss = 0.5367
Train: Got 1128 / 1408 correct (80.11)
Val: Got 591 / 1369 correct (43.17)
```

```
Iteration 100, loss = 0.3990
Train: Got 1076 / 1408 correct (76.42)
Val: Got 596 / 1369 correct (43.54)

Iteration 150, loss = 0.5388
Train: Got 1154 / 1408 correct (81.96)
Val: Got 588 / 1369 correct (42.95)

Iteration 200, loss = 0.4561
Train: Got 1139 / 1408 correct (80.89)
Val: Got 602 / 1369 correct (43.97)

Iteration 0, loss = 0.2993
Train: Got 1218 / 1408 correct (86.51)
Val: Got 631 / 1369 correct (46.09)

Iteration 50, loss = 0.9047
Train: Got 1158 / 1408 correct (82.24)
Val: Got 556 / 1369 correct (40.61)

Iteration 100, loss = 0.3145
Train: Got 1206 / 1408 correct (85.65)
Val: Got 611 / 1369 correct (44.63)

Iteration 150, loss = 0.3456
Train: Got 1244 / 1408 correct (88.35)
Val: Got 613 / 1369 correct (44.78)

Iteration 200, loss = 0.3111
Train: Got 1201 / 1408 correct (85.3)
Val: Got 558 / 1369 correct (40.76)

Iteration 0, loss = 0.2971
Train: Got 1151 / 1408 correct (81.75)
Val: Got 490 / 1369 correct (35.79)

Iteration 50, loss = 0.1971
Train: Got 1191 / 1408 correct (84.59)
Val: Got 567 / 1369 correct (41.42)

Iteration 100, loss = 0.2583
Train: Got 1233 / 1408 correct (87.57)
Val: Got 593 / 1369 correct (43.32)

Iteration 150, loss = 0.4147
Train: Got 1162 / 1408 correct (82.53)
Val: Got 497 / 1369 correct (36.3)
```

```
Iteration 200, loss = 0.1291
Train: Got 1184 / 1408 correct (84.09)
Val: Got 551 / 1369 correct (40.25)
```



[ ]:
```
# Check final validation accuracy:
check_accuracy(x_val, y_val_scenes, resnet50, 'Val')
```

```
Val: Got 571 / 1369 correct (41.71)
```

[ ]:
```
# Check final test accuracy:
check_accuracy(x_test, y_test_scenes, resnet50, 'Test')
```

```
Test: Got 645 / 1211 correct (53.26)
```

After manual hyperparameter tuning, training for 10 epochs on a learning rate of .001 and L2 Regularization strength of 1×10^{-4}, our baseline accuracy on the test set is 53.26%.

Note: Inspiration for .001 learning rate came from [1]:

[1]: S. Mishra, T. Yamasaki and H. Imaizumi, "Improving image classifiers for small datasets by learning rate adaptations," 2019 16th International Conference on Machine Vision Applications (MVA), 2019, pp. 1-6, doi: 10.23919/MVA.2019.8757890.

3.3 Part 2: Best Possible Performance Model (Gold Standard).

Here, we will concatenate the provided segmentation mask channel to the raw image pixel as an input. The segmentation masks have shape $N \times 1 \times H \times W$ and the raw images have shape $N \times 3 \times H \times W$. After concatenating on the channel axis, we will have an input of shape $N \times 4 \times H \times W$ which will serve as our input. We then train this input on a pretrained ResNet18 model that we cross validate, and the resulting scene classification accuracy serves as our gold standard for the accuracy our approach can achieve.

[ ]:
```
# First, undo preprocessing:
x_train *= 255
x_val *= 255
x_test *= 255

x_train += x_train_mean.permute((2, 0, 1))
x_val += x_train_mean.permute((2, 0, 1))
x_test += x_train_mean.permute((2, 0, 1))
```

[ ]:
```
# Watch out for RAM. RAM will be problematic potentially with all these
# permutations of the training examples.

img_seg_train = torch.cat((x_train, y_train_seg), dim=1)
img_seg_val = torch.cat((x_val, y_val_seg), dim=1)
img_seg_mean = torch.mean(img_seg_train, dim=0)
img_seg_std = torch.std(img_seg_train, dim=0)
img_seg_train -= img_seg_mean
img_seg_val -= img_seg_mean
img_seg_train /= img_seg_std
img_seg_val /= img_seg_std
```

[ ]:
```
class Flatten(nn.Module):
    def forward(self, x):
        N = x.shape[0]
        return x.reshape((N, -1))
```

At the start of our project, we tried to build our own model to achieve our gold standard. We end up using a cross-validated pretrained ResNet18 model to achieve this, but we still wanted to include the work we did below:

[ ]:
```
# Here is model:
input_channel = 3
channel_1 = 92
channel_2 = 64
channel_3 = 55
channel_4 = 45
input_linear = 1125
channel_5 = 32
hidden_1 = 750
hidden_2 = 250
num_classes = 50
```

```
model = nn.Sequential(
    nn.Conv2d(4, channel_1, 5, stride=1, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(2),
    nn.BatchNorm2d(channel_1),
    nn.Conv2d(channel_1, channel_2, 3, stride=1, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(2),
    nn.BatchNorm2d(channel_2),
    nn.Conv2d(channel_2, channel_3, 3, stride=1, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(2),
    nn.BatchNorm2d(channel_3),
    nn.Conv2d(channel_3, channel_4, 3, stride=1, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(3),
    nn.BatchNorm2d(channel_4),
    Flatten(),
    nn.Linear(input_linear, hidden_1),
    nn.ReLU(),
    nn.Dropout(p = .6),
    nn.BatchNorm1d(hidden_1),
    nn.Linear(hidden_1, hidden_2),
    nn.ReLU(),
    nn.Linear(hidden_2, num_classes),
)
```

[ ]:
```
def check_accuracy(x, y, model, input_string):
    num_correct = 0
    num_samples = 0
    model.eval()  # set model to evaluation mode
    if input_string == 'Test':
      num_iterations = int(NUM_TEST / BATCH_SIZE) + 1
    else:
      num_iterations = int(NUM_VAL / BATCH_SIZE) + 1
    with torch.no_grad():
        for i in range(num_iterations):
          # Need to get x and y
          if input_string == 'Train':
            batch_indices = np.random.choice(range(NUM_TRAIN), size=BATCH_SIZE,␣
 ↪replace=False)
            x_batch = x[batch_indices].type(dtype)
            y_batch = y[batch_indices].type(dtype).long()
          else:
            if i == num_iterations - 1:
              x_batch = x[num_samples:].type(dtype)
              y_batch = y[num_samples:].type(dtype).long()
```

```
            else:
              x_batch = x[num_samples:num_samples + BATCH_SIZE].type(dtype)
              y_batch = y[num_samples:num_samples + BATCH_SIZE].type(dtype).
↪long()
          x_batch = x_batch.to(device=device, dtype=dtype)  # move to device, e.
↪g. GPU
          y_batch = y_batch.to(device=device, dtype=torch.long)
          scores = model(x_batch)
          _, preds = scores.max(1)
          num_correct += (preds == y_batch).sum()
          num_samples += preds.size(0)
        acc = float(num_correct) / num_samples
        print('{}: Got {} / {} correct ({})'.format(input_string, num_correct,␣
↪num_samples, round(100 * acc, 2)))
        return round(100 * acc, 2)
```

[ ]:
```
def train_dual(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.
    """
    loss_arr = []
    num_iter = []
    model = model.to(device=device)  # move the model parameters to CPU/GPU
    for e in range(epochs):
        for t in range(int(NUM_TRAIN / BATCH_SIZE)):
            num_iter.append(e * int(NUM_TRAIN / BATCH_SIZE) + t)
            batch_indices = np.random.choice(range(NUM_TRAIN), size=BATCH_SIZE,
                                             replace=False)
            x = img_seg_train[batch_indices].type(dtype)
            y = y_train_scenes[batch_indices].type(dtype)
            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)

            loss_arr.append(loss)
```

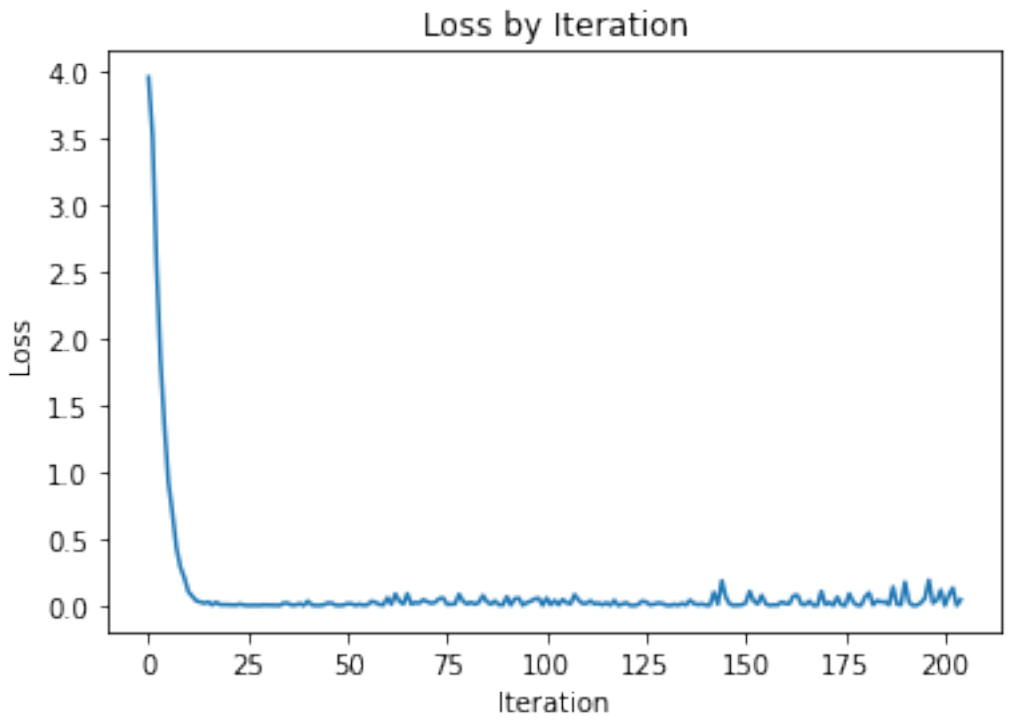
```
            # Zero out all of the gradients for the variables which the␣
↪optimizer
            # will update.
            optimizer.zero_grad()

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

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

            if t % print_every == 0:
                print('Iteration %d, loss = %.4f' % (t, loss.item()))
                check_accuracy(img_seg_train, y_train_scenes, model, 'Train')
                check_accuracy(img_seg_val, y_val_scenes, model, 'Val')
                print()
    plt.plot(num_iter, loss_arr)
    plt.xlabel('Iteration')
    plt.ylabel('Loss')
    plt.title('Loss by Iteration')
```

[ ]:
```
def train_dual_overfit(model, optimizer, epochs=1):
  loss_arr = []
  num_iter = []
  model = model.to(device=device)  # move the model parameters to CPU/GPU
  batch_indices = np.random.choice(range(NUM_TRAIN), size=BATCH_SIZE,
                                                        replace=False)
  x = img_seg_train[batch_indices].type(dtype)
  y = y_train_scenes[batch_indices].type(dtype)
  for e in range(epochs):
    for t in range(int(NUM_TRAIN / BATCH_SIZE)):
        num_iter.append(e * int(NUM_TRAIN / BATCH_SIZE) + t)
        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)

        if (t % print_every == 0):
          _, preds = scores.max(1)
          preds = preds
          num_correct = (preds == y).sum()
          num_samples = preds.size(0)
          acc = float(num_correct) / num_samples
```

```
        print('{}: Got {} / {} correct ({})'.format('Train', num_correct,␣
↪num_samples, round(100 * acc, 2)))
        print('Loss = {}'.format(round(float(loss), 2)))


      loss_arr.append(loss)
      # Zero out all of the gradients for the variables which the optimizer
      # will update.
      optimizer.zero_grad()

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

      # Actually update the parameters of the model using the gradients
      # computed by the backwards pass.
      optimizer.step()
 plt.plot(num_iter, loss_arr)
 plt.xlabel('Iteration')
 plt.ylabel('Loss')
 plt.title('Loss by Iteration')
```

[ ]:
```
optimizer = optimizer = optim.Adam(model.parameters(), lr=5e-3)
train_dual_overfit(model, optimizer, epochs=1)
```

```
Train: Got 2 / 64 correct (3.12)
Loss = 3.95
Train: Got 64 / 64 correct (100.0)
Loss = 0.01
Train: Got 62 / 64 correct (96.88)
Loss = 0.06
Train: Got 64 / 64 correct (100.0)
Loss = 0.02
Train: Got 64 / 64 correct (100.0)
Loss = 0.0
```



[ ]:
```
optimizer = optim.Adam(model.parameters(), lr=8e-3)
train_dual(model, optimizer, epochs=20)
```

```
Iteration 0, loss = 18.8047
Train: Got 274 / 1408 correct (19.46)
Val: Got 268 / 1369 correct (19.58)

Iteration 50, loss = 2.5936
Train: Got 517 / 1408 correct (36.72)
Val: Got 442 / 1369 correct (32.29)

Iteration 100, loss = 2.1734
Train: Got 525 / 1408 correct (37.29)
Val: Got 486 / 1369 correct (35.5)

Iteration 150, loss = 2.3676
Train: Got 563 / 1408 correct (39.99)
Val: Got 521 / 1369 correct (38.06)

Iteration 200, loss = 1.8704
Train: Got 587 / 1408 correct (41.69)
Val: Got 537 / 1369 correct (39.23)

Iteration 0, loss = 2.6057
```

```
Train: Got 588 / 1408 correct (41.76)
Val: Got 514 / 1369 correct (37.55)

Iteration 50, loss = 1.6593
Train: Got 660 / 1408 correct (46.88)
Val: Got 570 / 1369 correct (41.64)

Iteration 100, loss = 2.0947
Train: Got 646 / 1408 correct (45.88)
Val: Got 574 / 1369 correct (41.93)

Iteration 150, loss = 1.3912
Train: Got 677 / 1408 correct (48.08)
Val: Got 599 / 1369 correct (43.75)

Iteration 200, loss = 1.9805
Train: Got 680 / 1408 correct (48.3)
Val: Got 580 / 1369 correct (42.37)

Iteration 0, loss = 1.9124
Train: Got 668 / 1408 correct (47.44)
Val: Got 596 / 1369 correct (43.54)

Iteration 50, loss = 1.8592
Train: Got 696 / 1408 correct (49.43)
Val: Got 584 / 1369 correct (42.66)

Iteration 100, loss = 1.6148
Train: Got 712 / 1408 correct (50.57)
Val: Got 599 / 1369 correct (43.75)

Iteration 150, loss = 1.4705
Train: Got 691 / 1408 correct (49.08)
Val: Got 596 / 1369 correct (43.54)

Iteration 200, loss = 2.0167
Train: Got 735 / 1408 correct (52.2)
Val: Got 601 / 1369 correct (43.9)

Iteration 0, loss = 1.8109
Train: Got 707 / 1408 correct (50.21)
Val: Got 575 / 1369 correct (42.0)

Iteration 50, loss = 1.5571
Train: Got 742 / 1408 correct (52.7)
Val: Got 614 / 1369 correct (44.85)

Iteration 100, loss = 2.0471
```

```
Train: Got 712 / 1408 correct (50.57)
Val: Got 598 / 1369 correct (43.68)

Iteration 150, loss = 1.8652
Train: Got 738 / 1408 correct (52.41)
Val: Got 631 / 1369 correct (46.09)

Iteration 200, loss = 1.7885
Train: Got 757 / 1408 correct (53.76)
Val: Got 634 / 1369 correct (46.31)

Iteration 0, loss = 1.3393
Train: Got 747 / 1408 correct (53.05)
Val: Got 632 / 1369 correct (46.17)

Iteration 50, loss = 1.8306
Train: Got 785 / 1408 correct (55.75)
Val: Got 613 / 1369 correct (44.78)

Iteration 100, loss = 1.7391
Train: Got 771 / 1408 correct (54.76)
Val: Got 619 / 1369 correct (45.22)

Iteration 150, loss = 1.7090
Train: Got 761 / 1408 correct (54.05)
Val: Got 604 / 1369 correct (44.12)

Iteration 200, loss = 2.0660
Train: Got 691 / 1408 correct (49.08)
Val: Got 580 / 1369 correct (42.37)

Iteration 0, loss = 1.8908
Train: Got 736 / 1408 correct (52.27)
Val: Got 554 / 1369 correct (40.47)

Iteration 50, loss = 1.6851
Train: Got 765 / 1408 correct (54.33)
Val: Got 632 / 1369 correct (46.17)

Iteration 100, loss = 1.9767
Train: Got 721 / 1408 correct (51.21)
Val: Got 622 / 1369 correct (45.43)

Iteration 150, loss = 1.7066
Train: Got 774 / 1408 correct (54.97)
Val: Got 612 / 1369 correct (44.7)

Iteration 200, loss = 1.5088
```

```
Train: Got 799 / 1408 correct (56.75)
Val: Got 614 / 1369 correct (44.85)

Iteration 0, loss = 1.9504
Train: Got 790 / 1408 correct (56.11)
Val: Got 638 / 1369 correct (46.6)

Iteration 50, loss = 1.5225
Train: Got 763 / 1408 correct (54.19)
Val: Got 617 / 1369 correct (45.07)

Iteration 100, loss = 1.7904
Train: Got 852 / 1408 correct (60.51)
Val: Got 634 / 1369 correct (46.31)

Iteration 150, loss = 1.2019
Train: Got 829 / 1408 correct (58.88)
Val: Got 637 / 1369 correct (46.53)

Iteration 200, loss = 1.9299
Train: Got 803 / 1408 correct (57.03)
Val: Got 616 / 1369 correct (45.0)

Iteration 0, loss = 1.4436
Train: Got 811 / 1408 correct (57.6)
Val: Got 632 / 1369 correct (46.17)

Iteration 50, loss = 1.6453
Train: Got 805 / 1408 correct (57.17)
Val: Got 657 / 1369 correct (47.99)

Iteration 100, loss = 1.3525
Train: Got 822 / 1408 correct (58.38)
Val: Got 649 / 1369 correct (47.41)

Iteration 150, loss = 1.1471
Train: Got 834 / 1408 correct (59.23)
Val: Got 638 / 1369 correct (46.6)

Iteration 200, loss = 1.3254
Train: Got 823 / 1408 correct (58.45)
Val: Got 645 / 1369 correct (47.11)

Iteration 0, loss = 1.2797
Train: Got 802 / 1408 correct (56.96)
Val: Got 669 / 1369 correct (48.87)

Iteration 50, loss = 1.2446
```

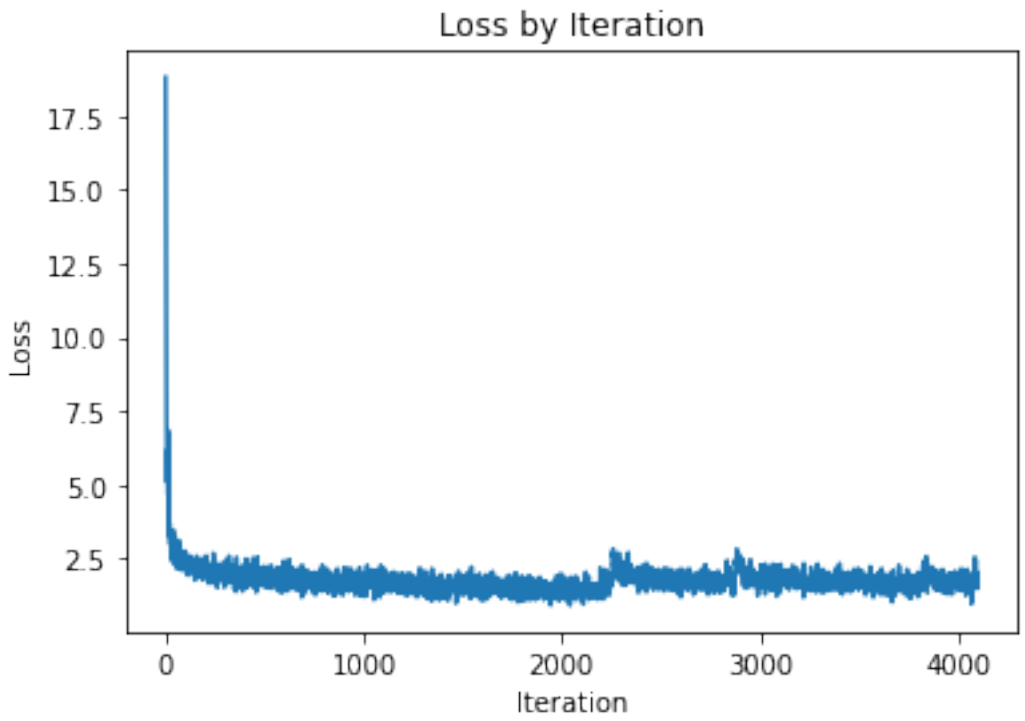
```
Train: Got 828 / 1408 correct (58.81)
Val: Got 663 / 1369 correct (48.43)

Iteration 100, loss = 1.4258
Train: Got 842 / 1408 correct (59.8)
Val: Got 658 / 1369 correct (48.06)

Iteration 150, loss = 1.4331
Train: Got 840 / 1408 correct (59.66)
Val: Got 645 / 1369 correct (47.11)

Iteration 200, loss = 1.8016
Train: Got 834 / 1408 correct (59.23)
Val: Got 620 / 1369 correct (45.29)

Iteration 0, loss = 1.2115
Train: Got 831 / 1408 correct (59.02)
Val: Got 645 / 1369 correct (47.11)

Iteration 50, loss = 1.7947
Train: Got 886 / 1408 correct (62.93)
Val: Got 649 / 1369 correct (47.41)

Iteration 100, loss = 1.3111
Train: Got 847 / 1408 correct (60.16)
Val: Got 651 / 1369 correct (47.55)

Iteration 150, loss = 2.0032
Train: Got 875 / 1408 correct (62.14)
Val: Got 645 / 1369 correct (47.11)

Iteration 200, loss = 1.3215
Train: Got 848 / 1408 correct (60.23)
Val: Got 665 / 1369 correct (48.58)

Iteration 0, loss = 1.4644
Train: Got 880 / 1408 correct (62.5)
Val: Got 691 / 1369 correct (50.47)

Iteration 50, loss = 1.3409
Train: Got 820 / 1408 correct (58.24)
Val: Got 632 / 1369 correct (46.17)

Iteration 100, loss = 1.2849
Train: Got 867 / 1408 correct (61.58)
Val: Got 641 / 1369 correct (46.82)

Iteration 150, loss = 1.3484
```

```
Train: Got 827 / 1408 correct (58.74)
Val: Got 604 / 1369 correct (44.12)

Iteration 200, loss = 1.5519
Train: Got 697 / 1408 correct (49.5)
Val: Got 503 / 1369 correct (36.74)

Iteration 0, loss = 2.0149
Train: Got 548 / 1408 correct (38.92)
Val: Got 437 / 1369 correct (31.92)

Iteration 50, loss = 1.7565
Train: Got 679 / 1408 correct (48.22)
Val: Got 555 / 1369 correct (40.54)

Iteration 100, loss = 1.9702
Train: Got 666 / 1408 correct (47.3)
Val: Got 570 / 1369 correct (41.64)

Iteration 150, loss = 2.2730
Train: Got 722 / 1408 correct (51.28)
Val: Got 599 / 1369 correct (43.75)

Iteration 200, loss = 1.9009
Train: Got 685 / 1408 correct (48.65)
Val: Got 566 / 1369 correct (41.34)

Iteration 0, loss = 2.1184
Train: Got 641 / 1408 correct (45.53)
Val: Got 556 / 1369 correct (40.61)

Iteration 50, loss = 1.6114
Train: Got 736 / 1408 correct (52.27)
Val: Got 581 / 1369 correct (42.44)

Iteration 100, loss = 1.8728
Train: Got 755 / 1408 correct (53.62)
Val: Got 605 / 1369 correct (44.19)

Iteration 150, loss = 1.7585
Train: Got 754 / 1408 correct (53.55)
Val: Got 610 / 1369 correct (44.56)

Iteration 200, loss = 1.5044
Train: Got 699 / 1408 correct (49.64)
Val: Got 612 / 1369 correct (44.7)

Iteration 0, loss = 1.7085
```

```
Train: Got 734 / 1408 correct (52.13)
Val: Got 614 / 1369 correct (44.85)

Iteration 50, loss = 1.4849
Train: Got 735 / 1408 correct (52.2)
Val: Got 606 / 1369 correct (44.27)

Iteration 100, loss = 1.9766
Train: Got 771 / 1408 correct (54.76)
Val: Got 622 / 1369 correct (45.43)

Iteration 150, loss = 1.5919
Train: Got 744 / 1408 correct (52.84)
Val: Got 619 / 1369 correct (45.22)

Iteration 200, loss = 1.9463
Train: Got 789 / 1408 correct (56.04)
Val: Got 602 / 1369 correct (43.97)

Iteration 0, loss = 1.9068
Train: Got 685 / 1408 correct (48.65)
Val: Got 566 / 1369 correct (41.34)

Iteration 50, loss = 2.0875
Train: Got 654 / 1408 correct (46.45)
Val: Got 526 / 1369 correct (38.42)

Iteration 100, loss = 1.8579
Train: Got 690 / 1408 correct (49.01)
Val: Got 592 / 1369 correct (43.24)

Iteration 150, loss = 1.9341
Train: Got 743 / 1408 correct (52.77)
Val: Got 600 / 1369 correct (43.83)

Iteration 200, loss = 1.6802
Train: Got 755 / 1408 correct (53.62)
Val: Got 600 / 1369 correct (43.83)

Iteration 0, loss = 1.7742
Train: Got 683 / 1408 correct (48.51)
Val: Got 569 / 1369 correct (41.56)

Iteration 50, loss = 2.1123
Train: Got 751 / 1408 correct (53.34)
Val: Got 605 / 1369 correct (44.19)

Iteration 100, loss = 2.1364
```

```
Train: Got 685 / 1408 correct (48.65)
Val: Got 579 / 1369 correct (42.29)

Iteration 150, loss = 1.9625
Train: Got 759 / 1408 correct (53.91)
Val: Got 629 / 1369 correct (45.95)

Iteration 200, loss = 1.6841
Train: Got 738 / 1408 correct (52.41)
Val: Got 609 / 1369 correct (44.49)

Iteration 0, loss = 1.7149
Train: Got 750 / 1408 correct (53.27)
Val: Got 615 / 1369 correct (44.92)

Iteration 50, loss = 1.7956
Train: Got 776 / 1408 correct (55.11)
Val: Got 631 / 1369 correct (46.09)

Iteration 100, loss = 1.8008
Train: Got 730 / 1408 correct (51.85)
Val: Got 588 / 1369 correct (42.95)

Iteration 150, loss = 1.9582
Train: Got 747 / 1408 correct (53.05)
Val: Got 607 / 1369 correct (44.34)

Iteration 200, loss = 1.8686
Train: Got 694 / 1408 correct (49.29)
Val: Got 538 / 1369 correct (39.3)

Iteration 0, loss = 1.5546
Train: Got 676 / 1408 correct (48.01)
Val: Got 563 / 1369 correct (41.12)

Iteration 50, loss = 1.7296
Train: Got 759 / 1408 correct (53.91)
Val: Got 615 / 1369 correct (44.92)

Iteration 100, loss = 1.4475
Train: Got 731 / 1408 correct (51.92)
Val: Got 559 / 1369 correct (40.83)

Iteration 150, loss = 1.7863
Train: Got 802 / 1408 correct (56.96)
Val: Got 588 / 1369 correct (42.95)

Iteration 200, loss = 1.7392
```

```
Train: Got 748 / 1408 correct (53.12)
Val: Got 607 / 1369 correct (44.34)

Iteration 0, loss = 1.5664
Train: Got 780 / 1408 correct (55.4)
Val: Got 620 / 1369 correct (45.29)

Iteration 50, loss = 1.6143
Train: Got 782 / 1408 correct (55.54)
Val: Got 595 / 1369 correct (43.46)

Iteration 100, loss = 1.6806
Train: Got 751 / 1408 correct (53.34)
Val: Got 601 / 1369 correct (43.9)

Iteration 150, loss = 1.9477
Train: Got 644 / 1408 correct (45.74)
Val: Got 567 / 1369 correct (41.42)

Iteration 200, loss = 1.8881
Train: Got 742 / 1408 correct (52.7)
Val: Got 592 / 1369 correct (43.24)

Iteration 0, loss = 1.8982
Train: Got 730 / 1408 correct (51.85)
Val: Got 573 / 1369 correct (41.86)

Iteration 50, loss = 1.6081
Train: Got 779 / 1408 correct (55.33)
Val: Got 627 / 1369 correct (45.8)

Iteration 100, loss = 1.8128
Train: Got 743 / 1408 correct (52.77)
Val: Got 578 / 1369 correct (42.22)

Iteration 150, loss = 1.5096
Train: Got 757 / 1408 correct (53.76)
Val: Got 601 / 1369 correct (43.9)

Iteration 200, loss = 1.4856
Train: Got 677 / 1408 correct (48.08)
Val: Got 581 / 1369 correct (42.44)
```



[ ]:
```
# Check final validation accuracy:
check_accuracy(img_seg_val, y_val_scenes, model, 'Val')
```

```
Val: Got 580 / 1369 correct (42.37)
```

[ ]:
```
42.37
```

[ ]:
```
# Check final test accuracy:
img_seg_test = torch.cat((x_test, y_test_seg), dim=1)
img_seg_test -= img_seg_mean
img_seg_test /= img_seg_std
check_accuracy(img_seg_test, y_test_scenes, model, 'Test')
```

```
Test: Got 576 / 1211 correct (47.56)
```

[ ]:
```
47.56
```

While our model we built was close to the baseline, the final accuracy did not beat the baseline so we had to pivot and use a pretrained model. Below we load ResNet18, cross-validate on epochs and regularization strength. We used ResNet18 because quicker training so easier cross validation.

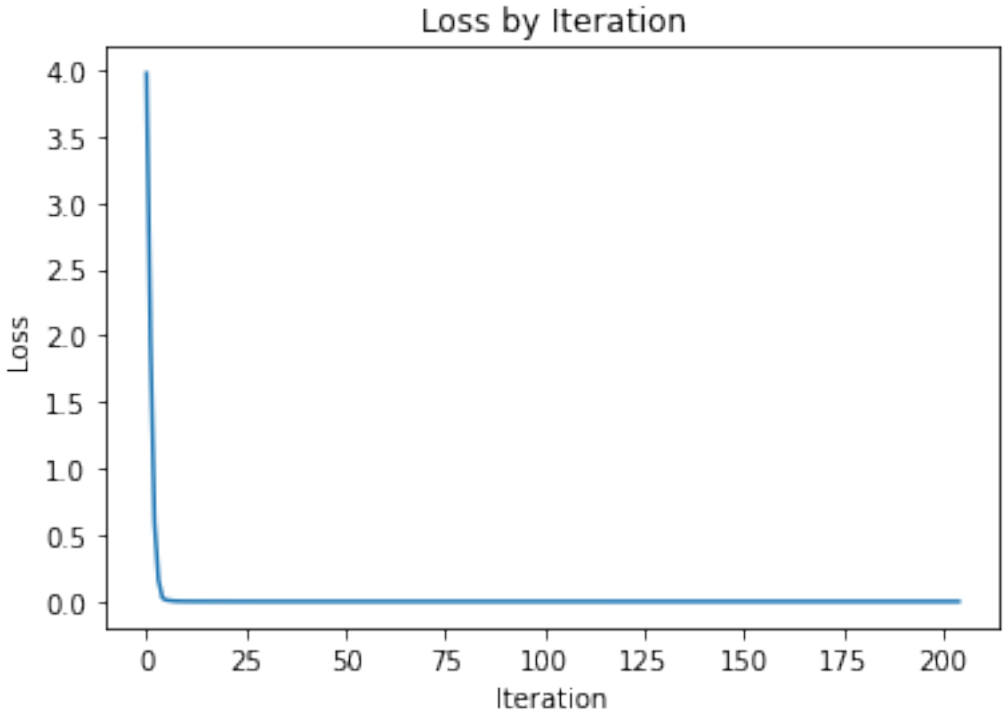
[ ]:
```
# resnet50 = models.resnet50(pretrained=True)
# resnet50.conv1 = nn.Conv2d(4, 64, kernel_size=(7, 7), stride=(2, 2),␣
 ↪padding=(3, 3), bias=False)
# resnet50.fc = nn.Linear(in_features=2048, out_features=50, bias=True)
```

[ ]:
```
# For quicker hyperparameter tuning
resnet18 = models.resnet18(pretrained=True)
# Adjust first layer to take in 4 channels instead of 3.
resnet18.conv1 = nn.Conv2d(4, 64, kernel_size=(7, 7), stride=(2, 2),␣
 ↪padding=(3, 3), bias=False)
# Adjust last layer to output 50 classes instead of 1000.
resnet18.fc = nn.Linear(in_features=512, out_features=50, bias=True)
```

Again, make sure we can overfit on the small batch of training examples.

[ ]:
```
optimizer = optim.Adam(resnet18.parameters(), lr=.001)
train_dual_overfit(resnet18, optimizer)
```

```
Train: Got 0 / 64 correct (0.0)
Loss = 3.97
Train: Got 64 / 64 correct (100.0)
Loss = 0.0
Train: Got 64 / 64 correct (100.0)
Loss = 0.0
Train: Got 64 / 64 correct (100.0)
Loss = 0.0
Train: Got 64 / 64 correct (100.0)
Loss = 0.0
```



Now, cross-validate the ResNet18 models on different numbers of epochs and regularization strengths (we found the model to be overfitting, but regularization didn't always translate to better validation performance so perhaps lowering epochs better means of preventing overfitting).

[ ]:
```
reg_strength = [3e-4, 5e-4, 7e-4]
epochs = [3, 5, 7]
best_model = None
best_accuracy = 0
best_epoch = None
for reg in reg_strength:
  for e in epochs:
    resnet18 = models.resnet18(pretrained=True)
    resnet18.conv1 = nn.Conv2d(4, 64, kernel_size=(7, 7), stride=(2, 2),␣
 ↪padding=(3, 3), bias=False)
    resnet18.fc = nn.Linear(in_features=512, out_features=50, bias=True)
    optimizer = optim.Adam(resnet18.parameters(), lr=.001, weight_decay=reg)
    train_dual(resnet18, optimizer, epochs=e)
    val_accuracy = check_accuracy(img_seg_val, y_val_scenes, resnet18, 'Val')
    if val_accuracy > best_accuracy:
      best_accuracy = val_accuracy
      best_model = resnet18
      best_epoch = e

print('Best accuracy is {}'.format(best_accuracy))
print('Best number of epochs is {}'.format(best_epoch))
```

```
Iteration 0, loss = 4.1686
Train: Got 111 / 1408 correct (7.88)
Val: Got 119 / 1369 correct (8.69)

Iteration 50, loss = 1.6299
Train: Got 600 / 1408 correct (42.61)
Val: Got 541 / 1369 correct (39.52)

Iteration 100, loss = 1.5864
Train: Got 660 / 1408 correct (46.88)
Val: Got 477 / 1369 correct (34.84)

Iteration 150, loss = 1.4557
Train: Got 723 / 1408 correct (51.35)
Val: Got 570 / 1369 correct (41.64)

Iteration 200, loss = 1.4583
Train: Got 811 / 1408 correct (57.6)
Val: Got 612 / 1369 correct (44.7)

Iteration 0, loss = 1.2664
Train: Got 751 / 1408 correct (53.34)
```

```
Val: Got 578 / 1369 correct (42.22)

Iteration 50, loss = 1.1697
Train: Got 833 / 1408 correct (59.16)
Val: Got 618 / 1369 correct (45.14)

Iteration 100, loss = 1.2037
Train: Got 894 / 1408 correct (63.49)
Val: Got 651 / 1369 correct (47.55)

Iteration 150, loss = 0.8438
Train: Got 866 / 1408 correct (61.51)
Val: Got 603 / 1369 correct (44.05)

Iteration 200, loss = 1.1836
Train: Got 827 / 1408 correct (58.74)
Val: Got 548 / 1369 correct (40.03)

Iteration 0, loss = 1.2123
Train: Got 847 / 1408 correct (60.16)
Val: Got 610 / 1369 correct (44.56)

Iteration 50, loss = 1.0867
Train: Got 891 / 1408 correct (63.28)
Val: Got 590 / 1369 correct (43.1)

Iteration 100, loss = 0.9428
Train: Got 936 / 1408 correct (66.48)
Val: Got 640 / 1369 correct (46.75)

Iteration 150, loss = 1.2749
Train: Got 950 / 1408 correct (67.47)
Val: Got 673 / 1369 correct (49.16)

Iteration 200, loss = 1.0447
Train: Got 942 / 1408 correct (66.9)
Val: Got 608 / 1369 correct (44.41)

Val: Got 601 / 1369 correct (43.9)
Iteration 0, loss = 4.3576
Train: Got 128 / 1408 correct (9.09)
Val: Got 89 / 1369 correct (6.5)

Iteration 50, loss = 2.2450
Train: Got 601 / 1408 correct (42.68)
Val: Got 561 / 1369 correct (40.98)

Iteration 100, loss = 1.6782
```

```
Train: Got 677 / 1408 correct (48.08)
Val: Got 591 / 1369 correct (43.17)

Iteration 150, loss = 1.5214
Train: Got 683 / 1408 correct (48.51)
Val: Got 592 / 1369 correct (43.24)

Iteration 200, loss = 1.5412
Train: Got 773 / 1408 correct (54.9)
Val: Got 610 / 1369 correct (44.56)

Iteration 0, loss = 1.4495
Train: Got 697 / 1408 correct (49.5)
Val: Got 544 / 1369 correct (39.74)

Iteration 50, loss = 1.3332
Train: Got 787 / 1408 correct (55.89)
Val: Got 592 / 1369 correct (43.24)

Iteration 100, loss = 1.1989
Train: Got 759 / 1408 correct (53.91)
Val: Got 598 / 1369 correct (43.68)

Iteration 150, loss = 1.0923
Train: Got 866 / 1408 correct (61.51)
Val: Got 663 / 1369 correct (48.43)

Iteration 200, loss = 1.0465
Train: Got 858 / 1408 correct (60.94)
Val: Got 615 / 1369 correct (44.92)

Iteration 0, loss = 1.3570
Train: Got 850 / 1408 correct (60.37)
Val: Got 640 / 1369 correct (46.75)

Iteration 50, loss = 1.0958
Train: Got 873 / 1408 correct (62.0)
Val: Got 638 / 1369 correct (46.6)

Iteration 100, loss = 1.0851
Train: Got 902 / 1408 correct (64.06)
Val: Got 625 / 1369 correct (45.65)

Iteration 150, loss = 1.0431
Train: Got 906 / 1408 correct (64.35)
Val: Got 596 / 1369 correct (43.54)

Iteration 200, loss = 1.2007
```

```
Train: Got 869 / 1408 correct (61.72)
Val: Got 616 / 1369 correct (45.0)

Iteration 0, loss = 1.0462
Train: Got 884 / 1408 correct (62.78)
Val: Got 594 / 1369 correct (43.39)

Iteration 50, loss = 0.8662
Train: Got 941 / 1408 correct (66.83)
Val: Got 604 / 1369 correct (44.12)

Iteration 100, loss = 0.8849
Train: Got 922 / 1408 correct (65.48)
Val: Got 643 / 1369 correct (46.97)

Iteration 150, loss = 0.8643
Train: Got 926 / 1408 correct (65.77)
Val: Got 587 / 1369 correct (42.88)

Iteration 200, loss = 0.7226
Train: Got 1024 / 1408 correct (72.73)
Val: Got 637 / 1369 correct (46.53)

Iteration 0, loss = 0.8048
Train: Got 1034 / 1408 correct (73.44)
Val: Got 630 / 1369 correct (46.02)

Iteration 50, loss = 1.2182
Train: Got 1008 / 1408 correct (71.59)
Val: Got 602 / 1369 correct (43.97)

Iteration 100, loss = 0.8412
Train: Got 1072 / 1408 correct (76.14)
Val: Got 673 / 1369 correct (49.16)

Iteration 150, loss = 0.8530
Train: Got 1055 / 1408 correct (74.93)
Val: Got 625 / 1369 correct (45.65)

Iteration 200, loss = 0.6978
Train: Got 1035 / 1408 correct (73.51)
Val: Got 641 / 1369 correct (46.82)

Val: Got 648 / 1369 correct (47.33)
Iteration 0, loss = 4.1372
Train: Got 88 / 1408 correct (6.25)
Val: Got 102 / 1369 correct (7.45)
```

```
Iteration 50, loss = 2.1655
Train: Got 663 / 1408 correct (47.09)
Val: Got 528 / 1369 correct (38.57)

Iteration 100, loss = 1.8239
Train: Got 679 / 1408 correct (48.22)
Val: Got 565 / 1369 correct (41.27)

Iteration 150, loss = 1.3867
Train: Got 743 / 1408 correct (52.77)
Val: Got 562 / 1369 correct (41.05)

Iteration 200, loss = 1.3785
Train: Got 727 / 1408 correct (51.63)
Val: Got 604 / 1369 correct (44.12)

Iteration 0, loss = 1.7214
Train: Got 782 / 1408 correct (55.54)
Val: Got 631 / 1369 correct (46.09)

Iteration 50, loss = 1.6226
Train: Got 793 / 1408 correct (56.32)
Val: Got 627 / 1369 correct (45.8)

Iteration 100, loss = 1.2079
Train: Got 795 / 1408 correct (56.46)
Val: Got 553 / 1369 correct (40.39)

Iteration 150, loss = 1.4342
Train: Got 868 / 1408 correct (61.65)
Val: Got 592 / 1369 correct (43.24)

Iteration 200, loss = 1.2133
Train: Got 807 / 1408 correct (57.32)
Val: Got 602 / 1369 correct (43.97)

Iteration 0, loss = 1.2952
Train: Got 896 / 1408 correct (63.64)
Val: Got 631 / 1369 correct (46.09)

Iteration 50, loss = 1.4181
Train: Got 902 / 1408 correct (64.06)
Val: Got 645 / 1369 correct (47.11)

Iteration 100, loss = 1.3253
Train: Got 902 / 1408 correct (64.06)
Val: Got 651 / 1369 correct (47.55)
```

```
Iteration 150, loss = 1.2229
Train: Got 963 / 1408 correct (68.39)
Val: Got 649 / 1369 correct (47.41)

Iteration 200, loss = 0.9203
Train: Got 960 / 1408 correct (68.18)
Val: Got 645 / 1369 correct (47.11)

Iteration 0, loss = 0.9881
Train: Got 912 / 1408 correct (64.77)
Val: Got 624 / 1369 correct (45.58)

Iteration 50, loss = 1.3187
Train: Got 905 / 1408 correct (64.28)
Val: Got 648 / 1369 correct (47.33)

Iteration 100, loss = 0.9519
Train: Got 905 / 1408 correct (64.28)
Val: Got 601 / 1369 correct (43.9)

Iteration 150, loss = 0.7947
Train: Got 957 / 1408 correct (67.97)
Val: Got 597 / 1369 correct (43.61)

Iteration 200, loss = 0.8933
Train: Got 957 / 1408 correct (67.97)
Val: Got 659 / 1369 correct (48.14)

Iteration 0, loss = 1.0855
Train: Got 972 / 1408 correct (69.03)
Val: Got 642 / 1369 correct (46.9)

Iteration 50, loss = 0.8309
Train: Got 964 / 1408 correct (68.47)
Val: Got 644 / 1369 correct (47.04)

Iteration 100, loss = 0.8283
Train: Got 1009 / 1408 correct (71.66)
Val: Got 628 / 1369 correct (45.87)

Iteration 150, loss = 0.9376
Train: Got 1052 / 1408 correct (74.72)
Val: Got 633 / 1369 correct (46.24)

Iteration 200, loss = 0.7009
Train: Got 1098 / 1408 correct (77.98)
Val: Got 680 / 1369 correct (49.67)
```

```
Iteration 0, loss = 1.0523
Train: Got 1018 / 1408 correct (72.3)
Val: Got 666 / 1369 correct (48.65)

Iteration 50, loss = 0.7245
Train: Got 1112 / 1408 correct (78.98)
Val: Got 681 / 1369 correct (49.74)

Iteration 100, loss = 0.6148
Train: Got 1082 / 1408 correct (76.85)
Val: Got 646 / 1369 correct (47.19)

Iteration 150, loss = 0.4900
Train: Got 1087 / 1408 correct (77.2)
Val: Got 656 / 1369 correct (47.92)

Iteration 200, loss = 0.5305
Train: Got 1129 / 1408 correct (80.18)
Val: Got 653 / 1369 correct (47.7)

Iteration 0, loss = 0.4614
Train: Got 1155 / 1408 correct (82.03)
Val: Got 659 / 1369 correct (48.14)

Iteration 50, loss = 0.5072
Train: Got 1050 / 1408 correct (74.57)
Val: Got 594 / 1369 correct (43.39)

Iteration 100, loss = 0.4607
Train: Got 1123 / 1408 correct (79.76)
Val: Got 666 / 1369 correct (48.65)

Iteration 150, loss = 0.6756
Train: Got 1186 / 1408 correct (84.23)
Val: Got 667 / 1369 correct (48.72)

Iteration 200, loss = 0.6456
Train: Got 1157 / 1408 correct (82.17)
Val: Got 621 / 1369 correct (45.36)

Val: Got 662 / 1369 correct (48.36)
Iteration 0, loss = 3.9704
Train: Got 93 / 1408 correct (6.61)
Val: Got 88 / 1369 correct (6.43)

Iteration 50, loss = 2.3757
Train: Got 603 / 1408 correct (42.83)
Val: Got 492 / 1369 correct (35.94)
```

```
Iteration 100, loss = 1.8471
Train: Got 695 / 1408 correct (49.36)
Val: Got 556 / 1369 correct (40.61)

Iteration 150, loss = 1.4220
Train: Got 628 / 1408 correct (44.6)
Val: Got 431 / 1369 correct (31.48)

Iteration 200, loss = 1.1599
Train: Got 826 / 1408 correct (58.66)
Val: Got 618 / 1369 correct (45.14)

Iteration 0, loss = 1.6055
Train: Got 765 / 1408 correct (54.33)
Val: Got 607 / 1369 correct (44.34)

Iteration 50, loss = 1.4837
Train: Got 822 / 1408 correct (58.38)
Val: Got 638 / 1369 correct (46.6)

Iteration 100, loss = 1.5700
Train: Got 856 / 1408 correct (60.8)
Val: Got 644 / 1369 correct (47.04)

Iteration 150, loss = 1.2697
Train: Got 784 / 1408 correct (55.68)
Val: Got 537 / 1369 correct (39.23)

Iteration 200, loss = 1.2933
Train: Got 920 / 1408 correct (65.34)
Val: Got 658 / 1369 correct (48.06)

Iteration 0, loss = 1.0001
Train: Got 841 / 1408 correct (59.73)
Val: Got 609 / 1369 correct (44.49)

Iteration 50, loss = 0.8660
Train: Got 929 / 1408 correct (65.98)
Val: Got 642 / 1369 correct (46.9)

Iteration 100, loss = 1.1189
Train: Got 920 / 1408 correct (65.34)
Val: Got 660 / 1369 correct (48.21)

Iteration 150, loss = 1.1376
Train: Got 927 / 1408 correct (65.84)
Val: Got 678 / 1369 correct (49.53)
```

```
Iteration 200, loss = 0.9180
Train: Got 883 / 1408 correct (62.71)
Val: Got 545 / 1369 correct (39.81)

Val: Got 653 / 1369 correct (47.7)
Iteration 0, loss = 4.1843
Train: Got 120 / 1408 correct (8.52)
Val: Got 90 / 1369 correct (6.57)

Iteration 50, loss = 1.9025
Train: Got 647 / 1408 correct (45.95)
Val: Got 586 / 1369 correct (42.8)

Iteration 100, loss = 1.8844
Train: Got 716 / 1408 correct (50.85)
Val: Got 580 / 1369 correct (42.37)

Iteration 150, loss = 1.7445
Train: Got 683 / 1408 correct (48.51)
Val: Got 582 / 1369 correct (42.51)

Iteration 200, loss = 1.6591
Train: Got 666 / 1408 correct (47.3)
Val: Got 540 / 1369 correct (39.44)

Iteration 0, loss = 1.4217
Train: Got 754 / 1408 correct (53.55)
Val: Got 630 / 1369 correct (46.02)

Iteration 50, loss = 1.3585
Train: Got 702 / 1408 correct (49.86)
Val: Got 531 / 1369 correct (38.79)

Iteration 100, loss = 1.5533
Train: Got 804 / 1408 correct (57.1)
Val: Got 637 / 1369 correct (46.53)

Iteration 150, loss = 1.2080
Train: Got 820 / 1408 correct (58.24)
Val: Got 658 / 1369 correct (48.06)

Iteration 200, loss = 1.3438
Train: Got 837 / 1408 correct (59.45)
Val: Got 641 / 1369 correct (46.82)

Iteration 0, loss = 1.0974
Train: Got 868 / 1408 correct (61.65)
```

```
Val: Got 644 / 1369 correct (47.04)

Iteration 50, loss = 1.1496
Train: Got 845 / 1408 correct (60.01)
Val: Got 585 / 1369 correct (42.73)

Iteration 100, loss = 1.0930
Train: Got 848 / 1408 correct (60.23)
Val: Got 625 / 1369 correct (45.65)

Iteration 150, loss = 1.1010
Train: Got 914 / 1408 correct (64.91)
Val: Got 605 / 1369 correct (44.19)

Iteration 200, loss = 0.7389
Train: Got 949 / 1408 correct (67.4)
Val: Got 618 / 1369 correct (45.14)

Iteration 0, loss = 1.1469
Train: Got 934 / 1408 correct (66.34)
Val: Got 624 / 1369 correct (45.58)

Iteration 50, loss = 1.2213
Train: Got 853 / 1408 correct (60.58)
Val: Got 532 / 1369 correct (38.86)

Iteration 100, loss = 0.6840
Train: Got 944 / 1408 correct (67.05)
Val: Got 654 / 1369 correct (47.77)

Iteration 150, loss = 0.5787
Train: Got 994 / 1408 correct (70.6)
Val: Got 679 / 1369 correct (49.6)

Iteration 200, loss = 0.6925
Train: Got 1001 / 1408 correct (71.09)
Val: Got 636 / 1369 correct (46.46)

Iteration 0, loss = 1.1165
Train: Got 974 / 1408 correct (69.18)
Val: Got 649 / 1369 correct (47.41)

Iteration 50, loss = 0.6769
Train: Got 982 / 1408 correct (69.74)
Val: Got 678 / 1369 correct (49.53)

Iteration 100, loss = 0.7751
Train: Got 1020 / 1408 correct (72.44)
```

```
Val: Got 673 / 1369 correct (49.16)

Iteration 150, loss = 0.7526
Train: Got 978 / 1408 correct (69.46)
Val: Got 648 / 1369 correct (47.33)

Iteration 200, loss = 0.7857
Train: Got 994 / 1408 correct (70.6)
Val: Got 635 / 1369 correct (46.38)

Val: Got 659 / 1369 correct (48.14)
Iteration 0, loss = 4.1323
Train: Got 67 / 1408 correct (4.76)
Val: Got 49 / 1369 correct (3.58)

Iteration 50, loss = 1.7898
Train: Got 662 / 1408 correct (47.02)
Val: Got 527 / 1369 correct (38.5)

Iteration 100, loss = 2.2295
Train: Got 674 / 1408 correct (47.87)
Val: Got 589 / 1369 correct (43.02)

Iteration 150, loss = 1.6675
Train: Got 714 / 1408 correct (50.71)
Val: Got 540 / 1369 correct (39.44)

Iteration 200, loss = 1.7449
Train: Got 759 / 1408 correct (53.91)
Val: Got 593 / 1369 correct (43.32)

Iteration 0, loss = 1.4262
Train: Got 776 / 1408 correct (55.11)
Val: Got 582 / 1369 correct (42.51)

Iteration 50, loss = 1.3763
Train: Got 784 / 1408 correct (55.68)
Val: Got 617 / 1369 correct (45.07)

Iteration 100, loss = 1.3466
Train: Got 774 / 1408 correct (54.97)
Val: Got 608 / 1369 correct (44.41)

Iteration 150, loss = 1.6450
Train: Got 784 / 1408 correct (55.68)
Val: Got 539 / 1369 correct (39.37)

Iteration 200, loss = 0.9727
```

```
Train: Got 857 / 1408 correct (60.87)
Val: Got 636 / 1369 correct (46.46)

Iteration 0, loss = 1.4689
Train: Got 795 / 1408 correct (56.46)
Val: Got 597 / 1369 correct (43.61)

Iteration 50, loss = 1.4247
Train: Got 858 / 1408 correct (60.94)
Val: Got 623 / 1369 correct (45.51)

Iteration 100, loss = 1.1947
Train: Got 867 / 1408 correct (61.58)
Val: Got 631 / 1369 correct (46.09)

Iteration 150, loss = 1.0257
Train: Got 909 / 1408 correct (64.56)
Val: Got 650 / 1369 correct (47.48)

Iteration 200, loss = 1.0435
Train: Got 921 / 1408 correct (65.41)
Val: Got 602 / 1369 correct (43.97)

Iteration 0, loss = 1.1473
Train: Got 905 / 1408 correct (64.28)
Val: Got 642 / 1369 correct (46.9)

Iteration 50, loss = 1.1221
Train: Got 895 / 1408 correct (63.57)
Val: Got 549 / 1369 correct (40.1)

Iteration 100, loss = 1.3490
Train: Got 943 / 1408 correct (66.97)
Val: Got 655 / 1369 correct (47.85)

Iteration 150, loss = 1.1878
Train: Got 914 / 1408 correct (64.91)
Val: Got 562 / 1369 correct (41.05)

Iteration 200, loss = 0.8494
Train: Got 943 / 1408 correct (66.97)
Val: Got 572 / 1369 correct (41.78)

Iteration 0, loss = 0.9659
Train: Got 937 / 1408 correct (66.55)
Val: Got 615 / 1369 correct (44.92)

Iteration 50, loss = 0.9014
```

```
Train: Got 872 / 1408 correct (61.93)
Val: Got 512 / 1369 correct (37.4)

Iteration 100, loss = 0.6731
Train: Got 977 / 1408 correct (69.39)
Val: Got 668 / 1369 correct (48.79)

Iteration 150, loss = 1.0973
Train: Got 983 / 1408 correct (69.82)
Val: Got 608 / 1369 correct (44.41)

Iteration 200, loss = 1.0042
Train: Got 972 / 1408 correct (69.03)
Val: Got 622 / 1369 correct (45.43)

Iteration 0, loss = 0.7671
Train: Got 1008 / 1408 correct (71.59)
Val: Got 629 / 1369 correct (45.95)

Iteration 50, loss = 1.0268
Train: Got 1018 / 1408 correct (72.3)
Val: Got 645 / 1369 correct (47.11)

Iteration 100, loss = 0.7744
Train: Got 1064 / 1408 correct (75.57)
Val: Got 664 / 1369 correct (48.5)

Iteration 150, loss = 0.6487
Train: Got 1026 / 1408 correct (72.87)
Val: Got 603 / 1369 correct (44.05)

Iteration 200, loss = 0.5211
Train: Got 1007 / 1408 correct (71.52)
Val: Got 619 / 1369 correct (45.22)

Iteration 0, loss = 0.5292
Train: Got 1024 / 1408 correct (72.73)
Val: Got 642 / 1369 correct (46.9)

Iteration 50, loss = 0.6765
Train: Got 1077 / 1408 correct (76.49)
Val: Got 600 / 1369 correct (43.83)

Iteration 100, loss = 0.6155
Train: Got 1080 / 1408 correct (76.7)
Val: Got 650 / 1369 correct (47.48)

Iteration 150, loss = 0.7363
```

```
Train: Got 1150 / 1408 correct (81.68)
Val: Got 636 / 1369 correct (46.46)

Iteration 200, loss = 0.5914
Train: Got 1027 / 1408 correct (72.94)
Val: Got 603 / 1369 correct (44.05)

Val: Got 612 / 1369 correct (44.7)
Iteration 0, loss = 4.3475
Train: Got 74 / 1408 correct (5.26)
Val: Got 65 / 1369 correct (4.75)

Iteration 50, loss = 2.1898
Train: Got 538 / 1408 correct (38.21)
Val: Got 457 / 1369 correct (33.38)

Iteration 100, loss = 1.4766
Train: Got 664 / 1408 correct (47.16)
Val: Got 651 / 1369 correct (47.55)

Iteration 150, loss = 1.5643
Train: Got 682 / 1408 correct (48.44)
Val: Got 578 / 1369 correct (42.22)

Iteration 200, loss = 1.7926
Train: Got 802 / 1408 correct (56.96)
Val: Got 607 / 1369 correct (44.34)

Iteration 0, loss = 1.8168
Train: Got 712 / 1408 correct (50.57)
Val: Got 591 / 1369 correct (43.17)

Iteration 50, loss = 1.7108
Train: Got 789 / 1408 correct (56.04)
Val: Got 593 / 1369 correct (43.32)

Iteration 100, loss = 1.3924
Train: Got 850 / 1408 correct (60.37)
Val: Got 614 / 1369 correct (44.85)

Iteration 150, loss = 1.5800
Train: Got 848 / 1408 correct (60.23)
Val: Got 648 / 1369 correct (47.33)

Iteration 200, loss = 1.2573
Train: Got 817 / 1408 correct (58.03)
Val: Got 579 / 1369 correct (42.29)
```

```
Iteration 0, loss = 1.0872
Train: Got 763 / 1408 correct (54.19)
Val: Got 523 / 1369 correct (38.2)

Iteration 50, loss = 1.3941
Train: Got 835 / 1408 correct (59.3)
Val: Got 567 / 1369 correct (41.42)

Iteration 100, loss = 1.0741
Train: Got 880 / 1408 correct (62.5)
Val: Got 635 / 1369 correct (46.38)

Iteration 150, loss = 1.1466
Train: Got 778 / 1408 correct (55.26)
Val: Got 558 / 1369 correct (40.76)

Iteration 200, loss = 1.2708
Train: Got 922 / 1408 correct (65.48)
Val: Got 582 / 1369 correct (42.51)

Val: Got 554 / 1369 correct (40.47)
Iteration 0, loss = 4.1955
Train: Got 56 / 1408 correct (3.98)
Val: Got 39 / 1369 correct (2.85)

Iteration 50, loss = 2.0482
Train: Got 566 / 1408 correct (40.2)
Val: Got 439 / 1369 correct (32.07)

Iteration 100, loss = 1.3078
Train: Got 695 / 1408 correct (49.36)
Val: Got 611 / 1369 correct (44.63)

Iteration 150, loss = 1.7353
Train: Got 738 / 1408 correct (52.41)
Val: Got 580 / 1369 correct (42.37)

Iteration 200, loss = 1.3759
Train: Got 835 / 1408 correct (59.3)
Val: Got 669 / 1369 correct (48.87)

Iteration 0, loss = 1.2363
Train: Got 793 / 1408 correct (56.32)
Val: Got 666 / 1369 correct (48.65)

Iteration 50, loss = 1.6972
Train: Got 789 / 1408 correct (56.04)
Val: Got 632 / 1369 correct (46.17)
```

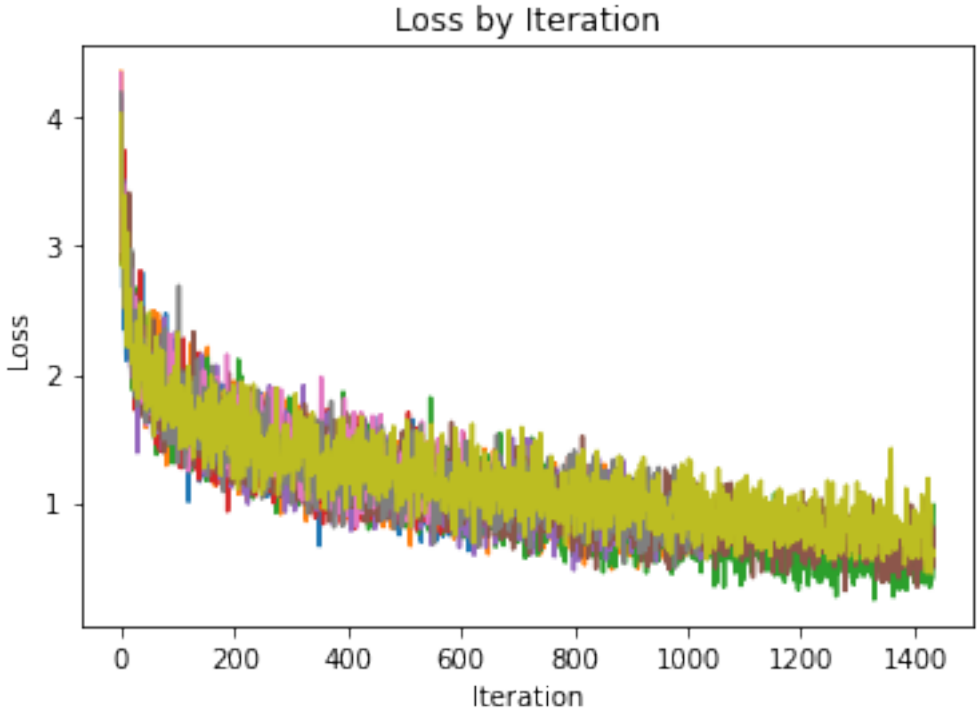
```
Iteration 100, loss = 1.1645
Train: Got 814 / 1408 correct (57.81)
Val: Got 639 / 1369 correct (46.68)

Iteration 150, loss = 1.2400
Train: Got 832 / 1408 correct (59.09)
Val: Got 610 / 1369 correct (44.56)

Iteration 200, loss = 1.2099
Train: Got 814 / 1408 correct (57.81)
Val: Got 591 / 1369 correct (43.17)

Iteration 0, loss = 1.5122
Train: Got 850 / 1408 correct (60.37)
Val: Got 561 / 1369 correct (40.98)

Iteration 50, loss = 1.6004
Train: Got 874 / 1408 correct (62.07)
Val: Got 627 / 1369 correct (45.8)

Iteration 100, loss = 1.5355
Train: Got 886 / 1408 correct (62.93)
Val: Got 623 / 1369 correct (45.51)

Iteration 150, loss = 1.1160
Train: Got 886 / 1408 correct (62.93)
Val: Got 623 / 1369 correct (45.51)

Iteration 200, loss = 1.1609
Train: Got 903 / 1408 correct (64.13)
Val: Got 609 / 1369 correct (44.49)

Iteration 0, loss = 1.0639
Train: Got 892 / 1408 correct (63.35)
Val: Got 598 / 1369 correct (43.68)

Iteration 50, loss = 1.2843
Train: Got 922 / 1408 correct (65.48)
Val: Got 635 / 1369 correct (46.38)

Iteration 100, loss = 1.1308
Train: Got 972 / 1408 correct (69.03)
Val: Got 651 / 1369 correct (47.55)

Iteration 150, loss = 0.9752
Train: Got 974 / 1408 correct (69.18)
Val: Got 669 / 1369 correct (48.87)
```

```
Iteration 200, loss = 0.8602
Train: Got 919 / 1408 correct (65.27)
Val: Got 599 / 1369 correct (43.75)

Iteration 0, loss = 1.1592
Train: Got 918 / 1408 correct (65.2)
Val: Got 618 / 1369 correct (45.14)

Iteration 50, loss = 0.9700
Train: Got 968 / 1408 correct (68.75)
Val: Got 626 / 1369 correct (45.73)

Iteration 100, loss = 0.9099
Train: Got 954 / 1408 correct (67.76)
Val: Got 658 / 1369 correct (48.06)

Iteration 150, loss = 1.0714
Train: Got 969 / 1408 correct (68.82)
Val: Got 621 / 1369 correct (45.36)

Iteration 200, loss = 0.9468
Train: Got 957 / 1408 correct (67.97)
Val: Got 618 / 1369 correct (45.14)

Val: Got 650 / 1369 correct (47.48)
Iteration 0, loss = 4.0319
Train: Got 87 / 1408 correct (6.18)
Val: Got 72 / 1369 correct (5.26)

Iteration 50, loss = 2.4852
Train: Got 582 / 1408 correct (41.34)
Val: Got 495 / 1369 correct (36.16)

Iteration 100, loss = 2.3267
Train: Got 743 / 1408 correct (52.77)
Val: Got 601 / 1369 correct (43.9)

Iteration 150, loss = 1.8558
Train: Got 686 / 1408 correct (48.72)
Val: Got 573 / 1369 correct (41.86)

Iteration 200, loss = 1.4507
Train: Got 772 / 1408 correct (54.83)
Val: Got 622 / 1369 correct (45.43)

Iteration 0, loss = 1.3124
Train: Got 759 / 1408 correct (53.91)
```

```
Val: Got 623 / 1369 correct (45.51)

Iteration 50, loss = 1.1806
Train: Got 823 / 1408 correct (58.45)
Val: Got 609 / 1369 correct (44.49)

Iteration 100, loss = 1.2297
Train: Got 832 / 1408 correct (59.09)
Val: Got 603 / 1369 correct (44.05)

Iteration 150, loss = 1.3637
Train: Got 814 / 1408 correct (57.81)
Val: Got 652 / 1369 correct (47.63)

Iteration 200, loss = 1.4233
Train: Got 843 / 1408 correct (59.87)
Val: Got 614 / 1369 correct (44.85)

Iteration 0, loss = 1.3491
Train: Got 826 / 1408 correct (58.66)
Val: Got 590 / 1369 correct (43.1)

Iteration 50, loss = 1.2895
Train: Got 803 / 1408 correct (57.03)
Val: Got 562 / 1369 correct (41.05)

Iteration 100, loss = 1.2886
Train: Got 813 / 1408 correct (57.74)
Val: Got 550 / 1369 correct (40.18)

Iteration 150, loss = 1.3091
Train: Got 872 / 1408 correct (61.93)
Val: Got 593 / 1369 correct (43.32)

Iteration 200, loss = 1.2637
Train: Got 831 / 1408 correct (59.02)
Val: Got 529 / 1369 correct (38.64)

Iteration 0, loss = 1.0391
Train: Got 890 / 1408 correct (63.21)
Val: Got 614 / 1369 correct (44.85)

Iteration 50, loss = 1.3067
Train: Got 931 / 1408 correct (66.12)
Val: Got 668 / 1369 correct (48.79)

Iteration 100, loss = 1.0838
Train: Got 871 / 1408 correct (61.86)
```

```
Val: Got 590 / 1369 correct (43.1)

Iteration 150, loss = 0.9897
Train: Got 929 / 1408 correct (65.98)
Val: Got 589 / 1369 correct (43.02)

Iteration 200, loss = 1.4132
Train: Got 881 / 1408 correct (62.57)
Val: Got 597 / 1369 correct (43.61)

Iteration 0, loss = 1.0089
Train: Got 980 / 1408 correct (69.6)
Val: Got 641 / 1369 correct (46.82)

Iteration 50, loss = 1.1576
Train: Got 997 / 1408 correct (70.81)
Val: Got 673 / 1369 correct (49.16)

Iteration 100, loss = 0.9193
Train: Got 953 / 1408 correct (67.68)
Val: Got 639 / 1369 correct (46.68)

Iteration 150, loss = 0.5956
Train: Got 920 / 1408 correct (65.34)
Val: Got 605 / 1369 correct (44.19)

Iteration 200, loss = 0.8859
Train: Got 914 / 1408 correct (64.91)
Val: Got 631 / 1369 correct (46.09)

Iteration 0, loss = 0.8011
Train: Got 949 / 1408 correct (67.4)
Val: Got 654 / 1369 correct (47.77)

Iteration 50, loss = 0.9016
Train: Got 985 / 1408 correct (69.96)
Val: Got 644 / 1369 correct (47.04)

Iteration 100, loss = 1.0007
Train: Got 958 / 1408 correct (68.04)
Val: Got 653 / 1369 correct (47.7)

Iteration 150, loss = 0.7489
Train: Got 954 / 1408 correct (67.76)
Val: Got 583 / 1369 correct (42.59)

Iteration 200, loss = 1.0263
Train: Got 1019 / 1408 correct (72.37)
```

```
Val: Got 619 / 1369 correct (45.22)

Iteration 0, loss = 0.5794
Train: Got 994 / 1408 correct (70.6)
Val: Got 622 / 1369 correct (45.43)

Iteration 50, loss = 1.1349
Train: Got 1005 / 1408 correct (71.38)
Val: Got 670 / 1369 correct (48.94)

Iteration 100, loss = 0.8874
Train: Got 1020 / 1408 correct (72.44)
Val: Got 636 / 1369 correct (46.46)

Iteration 150, loss = 0.6869
Train: Got 1017 / 1408 correct (72.23)
Val: Got 630 / 1369 correct (46.02)

Iteration 200, loss = 0.4970
Train: Got 1017 / 1408 correct (72.23)
Val: Got 676 / 1369 correct (49.38)

Val: Got 673 / 1369 correct (49.16)
Best accuracy is 49.16
Best number of epochs is 7
```



We found from trial and error that actually training on more epochs led to slightly better performance, so continue to train:

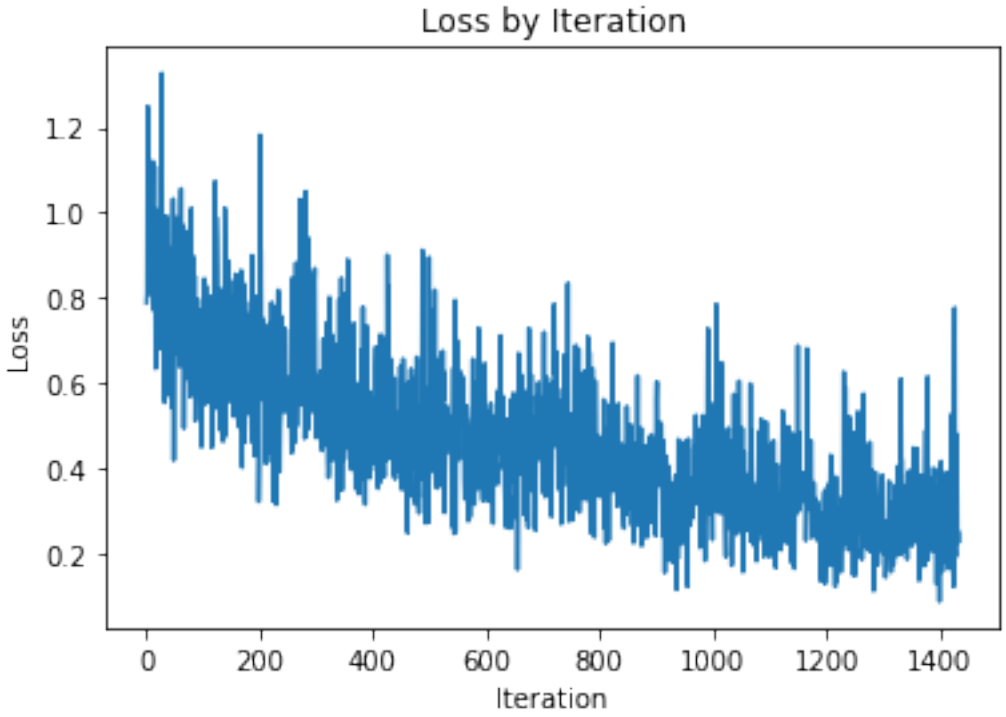
[ ]:
```
# Continue to train for double the epochs on best_model.
optimizer = optim.Adam(best_model.parameters(), lr=.001, weight_decay=.0005)
train_dual(best_model, optimizer, epochs=best_epoch)
```

```
Iteration 0, loss = 0.7886
Train: Got 1050 / 1408 correct (74.57)
Val: Got 643 / 1369 correct (46.97)

Iteration 50, loss = 0.8903
Train: Got 1050 / 1408 correct (74.57)
Val: Got 620 / 1369 correct (45.29)

Iteration 100, loss = 0.5983
Train: Got 1060 / 1408 correct (75.28)
Val: Got 606 / 1369 correct (44.27)

Iteration 150, loss = 0.6880
Train: Got 1084 / 1408 correct (76.99)
Val: Got 576 / 1369 correct (42.07)

Iteration 200, loss = 1.1823
Train: Got 1130 / 1408 correct (80.26)
Val: Got 658 / 1369 correct (48.06)

Iteration 0, loss = 0.7516
Train: Got 1090 / 1408 correct (77.41)
Val: Got 601 / 1369 correct (43.9)

Iteration 50, loss = 0.5296
Train: Got 1114 / 1408 correct (79.12)
Val: Got 614 / 1369 correct (44.85)

Iteration 100, loss = 0.5568
Train: Got 1121 / 1408 correct (79.62)
Val: Got 601 / 1369 correct (43.9)

Iteration 150, loss = 0.8903
Train: Got 1121 / 1408 correct (79.62)
Val: Got 635 / 1369 correct (46.38)

Iteration 200, loss = 0.6851
Train: Got 1176 / 1408 correct (83.52)
Val: Got 678 / 1369 correct (49.53)
```

```
Iteration 0, loss = 0.5314
Train: Got 1180 / 1408 correct (83.81)
Val: Got 648 / 1369 correct (47.33)

Iteration 50, loss = 0.4211
Train: Got 1130 / 1408 correct (80.26)
Val: Got 686 / 1369 correct (50.11)

Iteration 100, loss = 0.3765
Train: Got 1134 / 1408 correct (80.54)
Val: Got 573 / 1369 correct (41.86)

Iteration 150, loss = 0.5461
Train: Got 1226 / 1408 correct (87.07)
Val: Got 682 / 1369 correct (49.82)

Iteration 200, loss = 0.3431
Train: Got 1177 / 1408 correct (83.59)
Val: Got 688 / 1369 correct (50.26)

Iteration 0, loss = 0.4993
Train: Got 1165 / 1408 correct (82.74)
Val: Got 678 / 1369 correct (49.53)

Iteration 50, loss = 0.4304
Train: Got 1170 / 1408 correct (83.1)
Val: Got 613 / 1369 correct (44.78)

Iteration 100, loss = 0.2869
Train: Got 1168 / 1408 correct (82.95)
Val: Got 613 / 1369 correct (44.78)

Iteration 150, loss = 0.4476
Train: Got 1188 / 1408 correct (84.38)
Val: Got 629 / 1369 correct (45.95)

Iteration 200, loss = 0.3760
Train: Got 1233 / 1408 correct (87.57)
Val: Got 685 / 1369 correct (50.04)

Iteration 0, loss = 0.4505
Train: Got 1237 / 1408 correct (87.86)
Val: Got 667 / 1369 correct (48.72)

Iteration 50, loss = 0.4004
Train: Got 1228 / 1408 correct (87.22)
Val: Got 661 / 1369 correct (48.28)
```

```
Iteration 100, loss = 0.3654
Train: Got 1233 / 1408 correct (87.57)
Val: Got 649 / 1369 correct (47.41)

Iteration 150, loss = 0.5249
Train: Got 1231 / 1408 correct (87.43)
Val: Got 646 / 1369 correct (47.19)

Iteration 200, loss = 0.4433
Train: Got 1243 / 1408 correct (88.28)
Val: Got 600 / 1369 correct (43.83)

Iteration 0, loss = 0.2582
Train: Got 1257 / 1408 correct (89.28)
Val: Got 631 / 1369 correct (46.09)

Iteration 50, loss = 0.3060
Train: Got 1233 / 1408 correct (87.57)
Val: Got 627 / 1369 correct (45.8)

Iteration 100, loss = 0.2719
Train: Got 1267 / 1408 correct (89.99)
Val: Got 644 / 1369 correct (47.04)

Iteration 150, loss = 0.2370
Train: Got 1221 / 1408 correct (86.72)
Val: Got 612 / 1369 correct (44.7)

Iteration 200, loss = 0.2369
Train: Got 1264 / 1408 correct (89.77)
Val: Got 711 / 1369 correct (51.94)

Iteration 0, loss = 0.2243
Train: Got 1234 / 1408 correct (87.64)
Val: Got 691 / 1369 correct (50.47)

Iteration 50, loss = 0.3950
Train: Got 1264 / 1408 correct (89.77)
Val: Got 692 / 1369 correct (50.55)

Iteration 100, loss = 0.2805
Train: Got 1282 / 1408 correct (91.05)
Val: Got 682 / 1369 correct (49.82)

Iteration 150, loss = 0.1868
Train: Got 1266 / 1408 correct (89.91)
Val: Got 607 / 1369 correct (44.34)
```

```
Iteration 200, loss = 0.4548
Train: Got 1284 / 1408 correct (91.19)
Val: Got 664 / 1369 correct (48.5)
```



[ ]:
```
# Check final validation accuracy:
check_accuracy(img_seg_val, y_val_scenes, best_model, 'Val')
```

```
Val: Got 674 / 1369 correct (49.23)
```

[ ]:
```
49.23
```

[ ]:
```
# Check final test accuracy:
# Don't need first 3 lines if you trained my own model above.
check_accuracy(img_seg_test, y_test_scenes, best_model, 'Test')
```

```
Test: Got 727 / 1211 correct (60.03)
```

[ ]:
```
60.03
```

Here, we achieved our gold standard accuracy: 49.23% on the validation set and 60.03% on the test set! This is great news, as we see appending the segmentation mask to the training data can lead to better scene classification performance than our baseline if the segmentation mask is well constructed (we achieved roughly 7% better performance on both validation and test sets). Having a better gold standard is evidence that our approach has validity. We now try to beat our baseline by producing the segmentation mask instead of using the provided one below.

[ ]:
```
# torch.save(best_model, '/content/drive/MyDrive/CS231n Project/Datasets/
 ↪best_model.pt')
```

[ ]:
```
# For quick debugging, load a trained model from previous run to avoid waiting␣
 ↪for
# model to retrain. NOTE: in final run before submission this will NOT be used.
# best_model = torch.load('/content/drive/MyDrive/CS231n Project/Datasets/
 ↪best_model.pt')
```

[ ]:
```
# After performing above, run this cell until RAM
# in top right goes down.
img_seg_train = None
img_seg_val = None
img_seg_test = None
gc.collect()
```

[ ]: 13346

3.4 Part 3: Produce Segmentation Mask from Raw Image.

In this part, we use a Fully Connected ResNet50 model that can produce a segmentation mask from raw pixels. This segmentation mask will then serve as our input in our model in part 4.

[ ]:
```
# Re preprocess the x_data though it should already be permuted.
# Subtract mean
x_train_mean = torch.mean(x_train, dim=0)
x_train -= x_train_mean
x_val -= x_train_mean
x_test -= x_train_mean

# Divide by std (255)
x_train /= 255
x_val /= 255
x_test /= 255
```

[ ]:
```
# Now, figure out how many classes there are.
num_seg_classes = int(np.max(np.concatenate((np.unique(y_train_seg),
                            np.unique(y_val_seg), np.unique(y_test_seg)))) + 1)
print(num_seg_classes)
```

253

Here, we load a FCN ResNet50 model meant for segmentation.

[ ]:
```
# Load resnet segmentation model
seg_model = models.segmentation.fcn_resnet50(pretrained=False,
                      progress=True, num_classes=num_seg_classes, aux_loss=None)
```

[ ]:
```
def train_seg_overfit(model, optimizer, epochs=1):
  loss_arr = []
  num_iter = []
```

```
  model = model.to(device=device)  # move the model parameters to CPU/GPU
  batch_indices = np.random.choice(range(NUM_TRAIN), size=BATCH_SIZE,
                                                        replace=False)
  x = x_train[batch_indices].type(dtype)
  y = y_train_seg[batch_indices].type(dtype)
  y = y.reshape((y.shape[0], y.shape[2], y.shape[3]))
  x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
  y = y.to(device=device, dtype=torch.long)

  for e in range(epochs):
    for t in range(int(NUM_TRAIN / BATCH_SIZE)):
        num_iter.append(e * int(NUM_TRAIN / BATCH_SIZE) + t)
        model.train()  # put model to training mode

        scores = model(x)
        loss = F.cross_entropy(scores['out'], y, reduction='none')

        if (t % print_every == 0):
          print('Loss = {}'.format(round(float(loss.mean()), 2)))

        loss_arr.append(loss.mean())
        # Zero out all of the gradients for the variables which the optimizer
        # will update.
        optimizer.zero_grad()

        # This is the backwards pass: compute the gradient of the loss with
        # respect to each  parameter of the model.
        loss.backward(torch.ones(loss.shape).cuda())

        # Actually update the parameters of the model using the gradients
        # computed by the backwards pass.
        optimizer.step()
  plt.plot(num_iter, loss_arr)
  plt.xlabel('Iteration')
  plt.ylabel('Loss')
  plt.title('Loss by Iteration')
```

[ ]:
```
def train_seg(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.
    """
    loss_arr = []
    num_iter = []
    model = model.to(device=device)  # move the model parameters to CPU/GPU
    for e in range(epochs):
        for t in range(int(NUM_TRAIN / BATCH_SIZE)):
            num_iter.append(e * int(NUM_TRAIN / BATCH_SIZE) + t)
            batch_indices = np.random.choice(range(NUM_TRAIN), size=BATCH_SIZE,
                                                    replace=False)
            x = x_train[batch_indices].type(dtype)
            y = y_train_seg[batch_indices].type(dtype)
            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)

            y = y.reshape((y.shape[0], y.shape[2], y.shape[3]))

            scores = model(x)
            loss = F.cross_entropy(scores['out'], y, reduction='none')

            loss_arr.append(loss.mean())
            # Zero out all of the gradients for the variables which the␣
↪optimizer
            # will update.
            optimizer.zero_grad()

            # This is the backwards pass: compute the gradient of the loss with
            # respect to each  parameter of the model.
            loss.backward(torch.ones(loss.shape).cuda())

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

            if t % print_every == 0:
                print('Iteration %d, loss = %.4f' % (t, loss.mean()))
                print()
    plt.plot(num_iter, loss_arr)
    plt.xlabel('Iteration')
    plt.ylabel('Loss')
    plt.title('Loss by Iteration')
```

[ ]:
```
# optimizer = optim.Adam(seg_model.parameters(), lr=.001)
# train_seg_overfit(seg_model, optimizer)
```

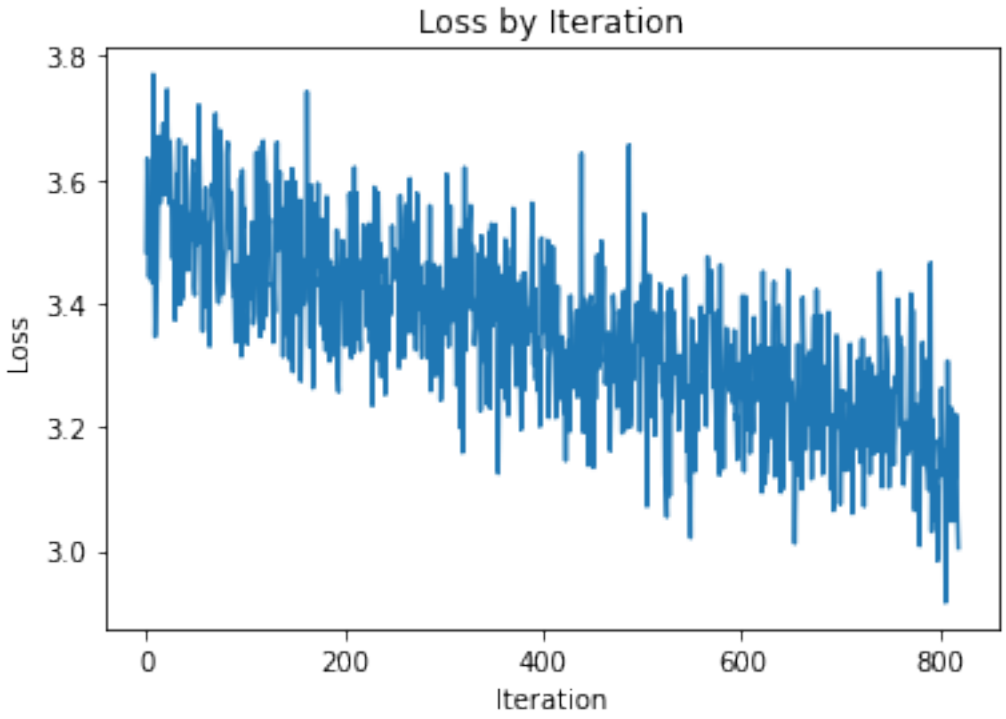
We trained the model using the code above. Training took roughly 30 mins an epoch so we had to train in bunches. You can see from the code below that we load the model that had been

trained for 4 epochs and we decided to train again for 4 epochs. This was necessary to not get kicked off runtime for using too many Colab resources.

[ ]:
```
seg_model = torch.load('/content/drive/MyDrive/CS231n Project/Datasets/
 ↪seg_model.pt')
```

[ ]:
```
# Train for 4 more epochs
optimizer = optim.Adam(seg_model.parameters(), lr=.001, weight_decay=1e-5)
train_seg(seg_model, optimizer, epochs=4)
```

```
Iteration 0, loss = 3.4828

Iteration 50, loss = 3.4849

Iteration 100, loss = 3.3970

Iteration 150, loss = 3.5966

Iteration 200, loss = 3.3314

Iteration 0, loss = 3.5779

Iteration 50, loss = 3.2957

Iteration 100, loss = 3.3302

Iteration 150, loss = 3.1253

Iteration 200, loss = 3.5021

Iteration 0, loss = 3.4917

Iteration 50, loss = 3.3966

Iteration 100, loss = 3.2166

Iteration 150, loss = 3.3100

Iteration 200, loss = 3.2531

Iteration 0, loss = 3.2423

Iteration 50, loss = 3.1640

Iteration 100, loss = 3.1660

Iteration 150, loss = 3.1734

Iteration 200, loss = 3.1085
```



We save after training so in future iterations we would not have to retrain.

[ ]:
```
# Save after more training.
torch.save(seg_model, '/content/drive/MyDrive/CS231n Project/Datasets/seg_model.
↪pt')
```

[ ]:
```
gc.collect()
```

[ ]:
```
2387
```

4 Part 4: Running Test Set Through Trained Model in Part 3 then Part 2 to Get Final Scene Classifcation Accuracy.

Put it together. Running the images through the trained model in part 3 then concatenating the output from that model with original image as an input into model from part 2 should score well.

Using the same network as part 2, plugging in the segmentation mask created in part 3 instead of the given segmentation masks should ideally outperform part 1. This will only outperform part 1 in 2 cases:

1) Part 2 outperforms Part 1 (it does) and \
2) The performance of the model in part 3 is good enough to produce a strong segmentation mask as input into the model (we will see).

First, we run the raw test images through the model from part 3 that produces the segmentation mask, and append this to the raw test images for input into the model from part 2.

[ ]:
```
seg_model.eval()
best_model.eval()
img_seg_input = torch.zeros((NUM_TEST, 1, x_test.shape[2], x_test.shape[3]))
# x_test is already preprocessed with -median and /std
num_iterations = int(NUM_TEST / BATCH_SIZE) + 1
num_samples = 0
for i in range(num_iterations):
  if i == num_iterations - 1:
    x_batch = x_test[num_samples:].type(dtype)
    y_batch = y_test_scenes[num_samples:].type(dtype).long()
  else:
    x_batch = x_test[num_samples:num_samples + BATCH_SIZE].type(dtype)
    y_batch = y_test_scenes[num_samples:num_samples + BATCH_SIZE].type(dtype).
 ↪long()
  x_batch = x_batch.to(device=device, dtype=dtype)  # move to device, e.g. GPU
  y_batch = y_batch.to(device=device, dtype=torch.long)
  scores = seg_model(x_batch)
  _, preds = torch.max(scores['out'], dim=1)
  if i == num_iterations - 1:
    img_seg_input[num_samples:,0,:,:] = preds
  else:
    img_seg_input[num_samples:num_samples +  BATCH_SIZE,0,:,:] = preds
  num_samples += BATCH_SIZE
# Feed through seg_model on eval mode to get NUM_TRAIN produced segmentation␣
 ↪masks
# Unprocess x_test, concatentate seg masks to x_test, then divde img_seg_mean␣
 ↪and
# img_seg_std. Feed through part 2 model and evaluate result.
```

[ ]:
```
# Unprocess x_test to press img_seg_input
x_test *= 255
x_test += x_train_mean
img_seg_input = torch.cat((x_test, img_seg_input), dim=1)
img_seg_input -= img_seg_mean
img_seg_input /= img_seg_std
```

Now, we run through the model from part 2 to obtain our final test accuracy:

[ ]:
```
# Put through model:
print('Final test accuracy: ')
print()
check_accuracy(img_seg_input, y_test_scenes, best_model, 'Test')
```

```
Final test accuracy:

Test: Got 585 / 1211 correct (48.31)
```

[ ]: `48.31`

As seen, we unfortunately did not beat the baseline of roughly 53%, but we were not far off at 48.31%. As we discuss in the writeup, we are skeptical that the segmentation mask actually had good data integrity (each pixel representing a unique class at that one channel). Considering our skepticism about the segmentation mask's underlying structure, our ability to get close to baseline allows us to still believe this approach can lead to better classification (as proven by part 2's superior performance to baseline).