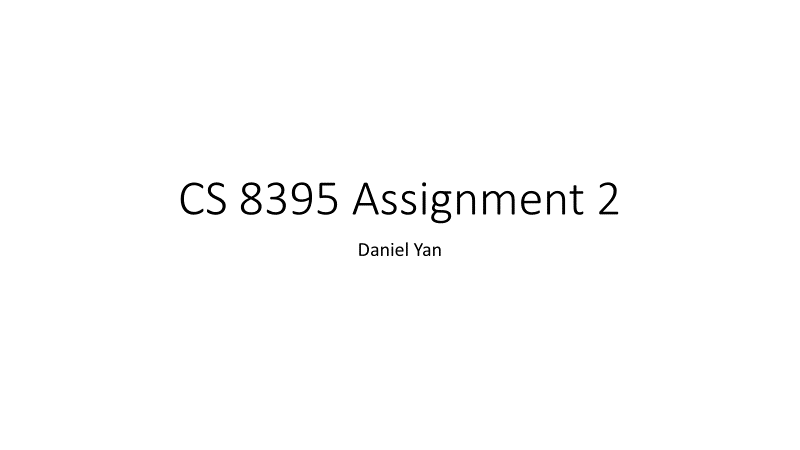
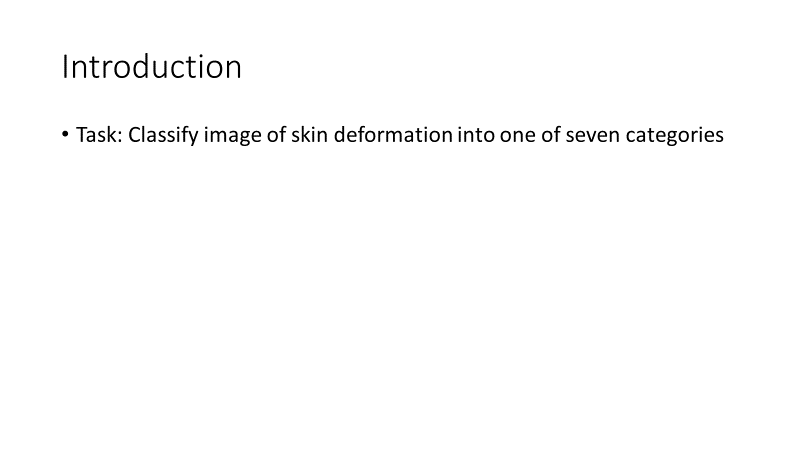
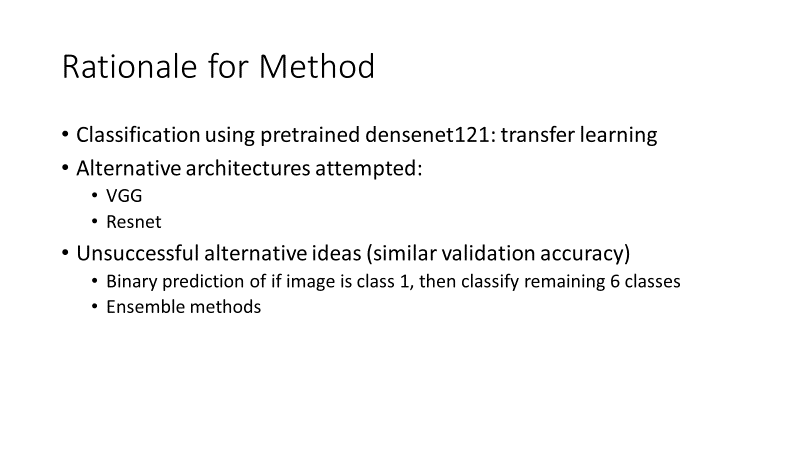
Slide 1



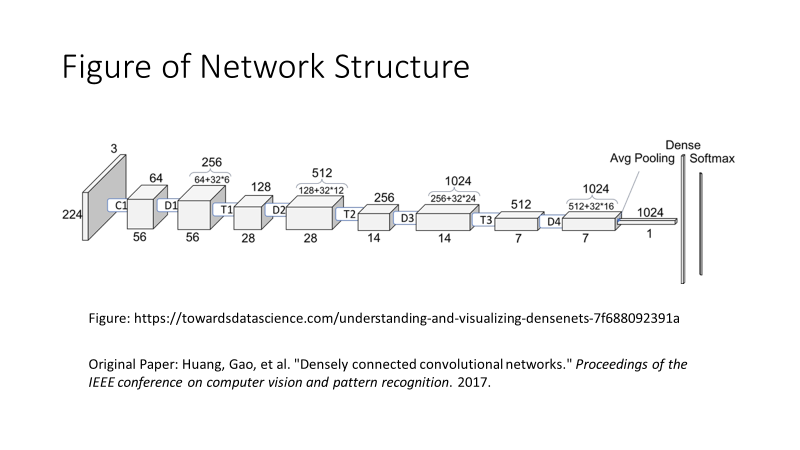
Slide 2



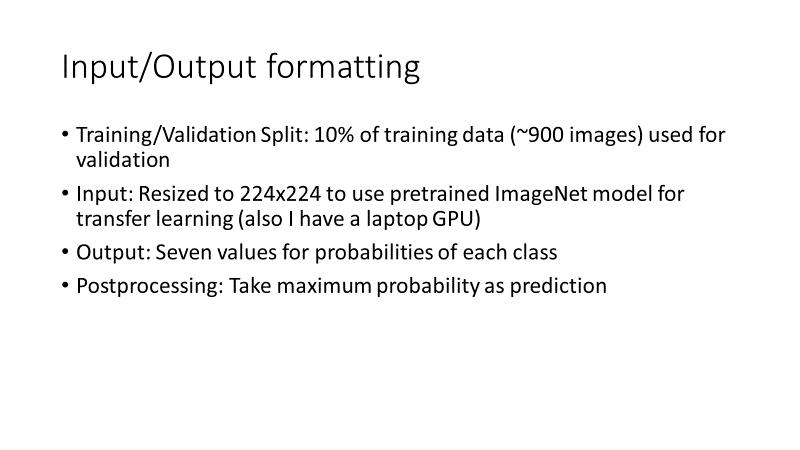
Slide 3



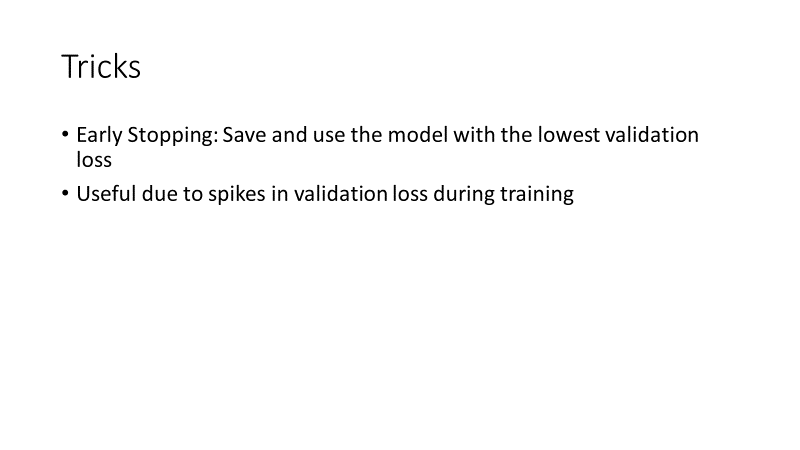
Slide 4



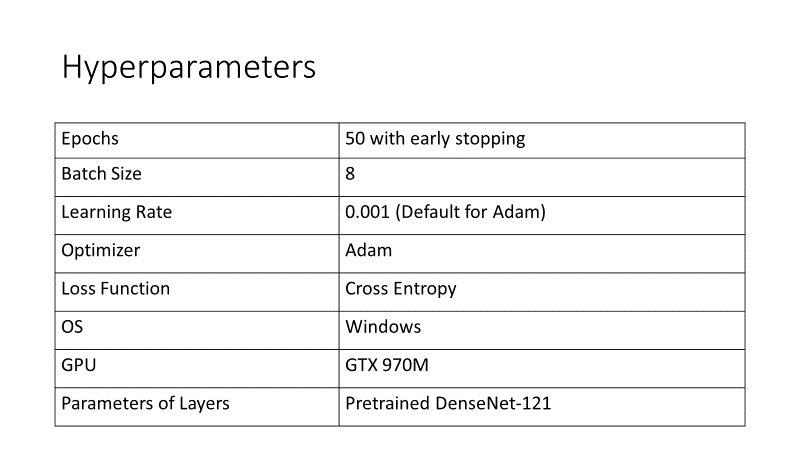
Slide 5



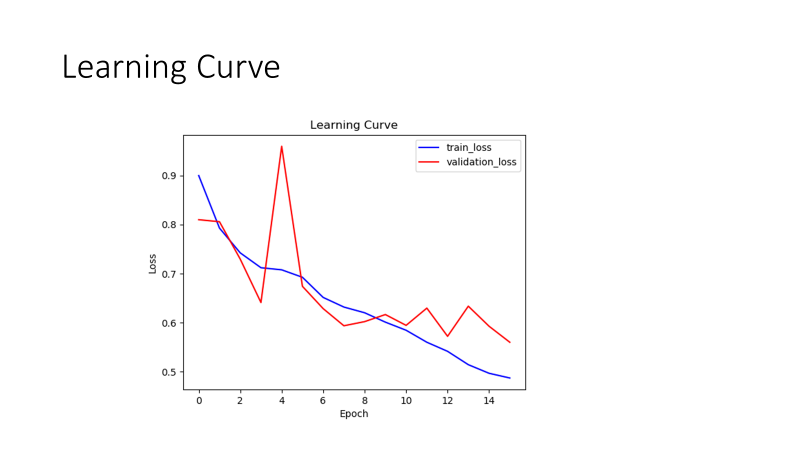
Slide 6



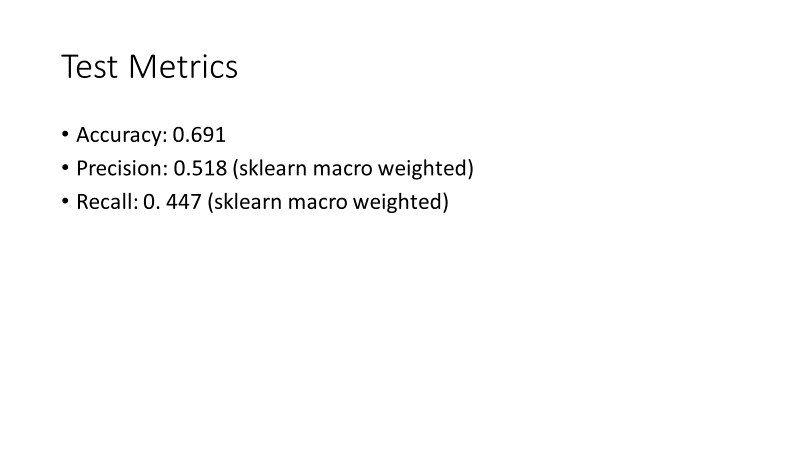
Slide 7



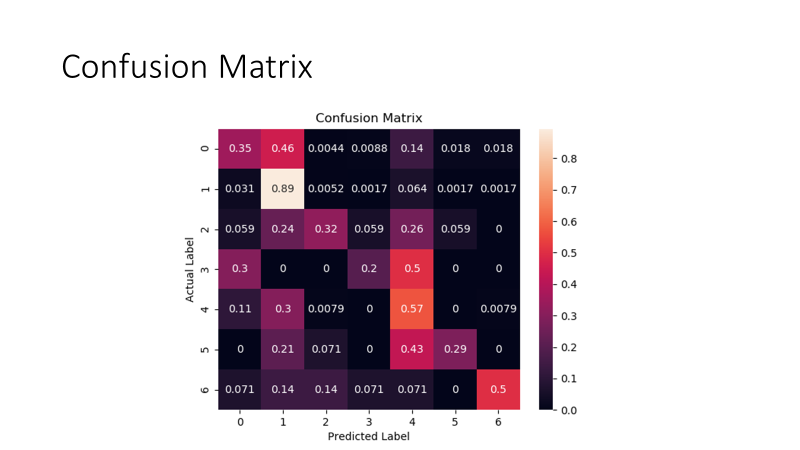
Slide 8



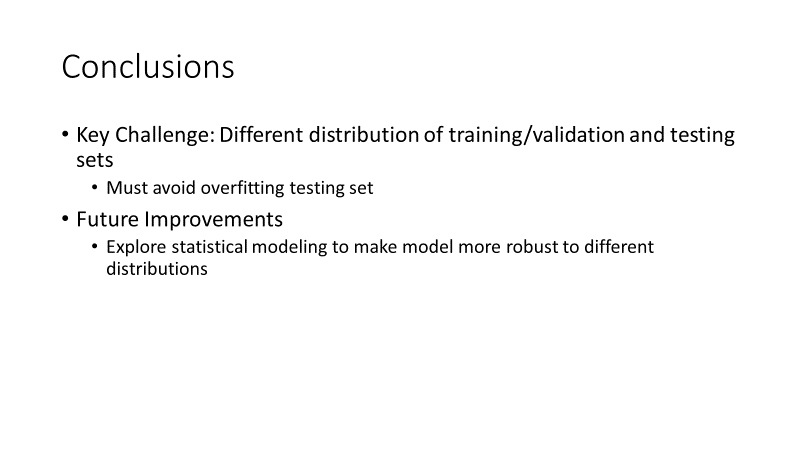
Slide 9



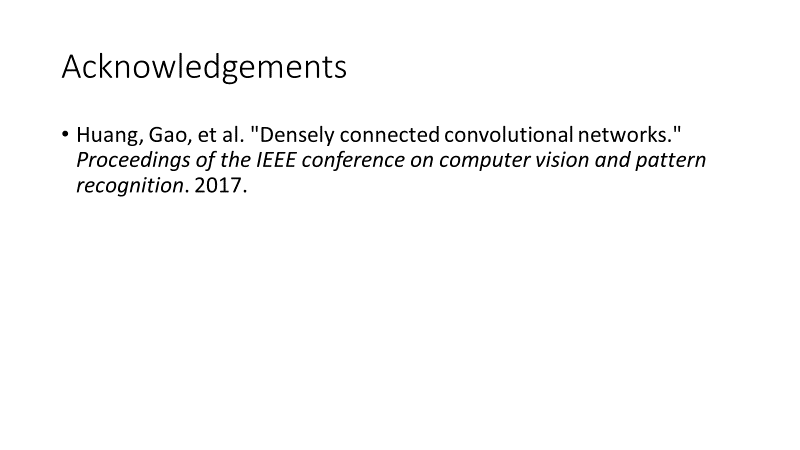
Slide 10



Slide 11



Slide 12



**Code**

generate\_labels.py:

*# Author: Daniel Yan  
# Email: daniel.yan@vanderbilt.edu  
# Description: Quick script to reformat the labels for training.  
  
# Imports***import** pandas **as** pd  
  
*# Function to generate numerical label from one-hot***def** generate\_numerical\_label(row):  
 **if** row[**"MEL"**] == 1:  
 **return** 0  
 **elif** row[**"NV"**] == 1:  
 **return** 1  
 **elif** row[**"BCC"**] == 1:  
 **return** 2  
 **elif** row[**"AKIEC"**] == 1:  
 **return** 3  
 **elif** row[**"BKL"**] == 1:  
 **return** 4  
 **elif** row[**"DF"**] == 1:  
 **return** 5  
 **else**:  
 **return** 6  
  
*# Function to subtract 1 from labels 2-6 for the set without class 1***def** relabel\_no\_class\_1(row):  
 **if** row[**"label"**] == 0:  
 **return** 0  
 **else**:  
 **return** row[**"label"**] - 1  
  
*# Function to subtract 1 from labels 2-6 for the set without class 1***def** binary\_1(row):  
 **if** row[**"label"**] == 1:  
 **return** 1  
 **else**:  
 **return** 0  
  
*# Load in labels files.*train\_labels\_df = pd.read\_csv(**"../data/labels/Train\_labels.csv"**, sep=**","**)  
test\_labels\_df = pd.read\_csv(**"../data/labels/Test\_labels.csv"**, sep=**","**)  
  
*# Append a .jpg for the file name*train\_labels\_df[**"image"**] = train\_labels\_df[**"image"**] + **".jpg"**test\_labels\_df[**"image"**] = test\_labels\_df[**"image"**] + **".jpg"***# Add new column for integer value for the label*train\_labels\_df[**"label"**] = train\_labels\_df.apply(generate\_numerical\_label, axis=1)  
test\_labels\_df[**"label"**] = test\_labels\_df.apply(generate\_numerical\_label, axis=1)  
  
*# Get total number of training and testing instances*num\_train\_images = train\_labels\_df.shape[0]  
num\_test\_images = test\_labels\_df.shape[0]  
*# Print out fraction of images for each label***for** label **in** [**"MEL"**,**"NV"**,**"BCC"**,**"AKIEC"**,**"BKL"**,**"DF"**,**"VASC"**]:  
 *# Get number of instances with that label* train\_instances = train\_labels\_df[label].sum()  
 test\_instances = test\_labels\_df[label].sum()  
 *# Print out fraction of instances* print(**"Percentage of label "**, label)  
 print(**"Training: "**, float(train\_instances/num\_train\_images))  
 print(**"Testing: "**, float(test\_instances/num\_test\_images))  
  
*# Drop original one-hot label columns*train\_labels\_df = train\_labels\_df.drop(columns=[**"MEL"**,**"NV"**,**"BCC"**,**"AKIEC"**,**"BKL"**,**"DF"**,**"VASC"**])  
test\_labels\_df = test\_labels\_df.drop(columns=[**"MEL"**,**"NV"**,**"BCC"**,**"AKIEC"**,**"BKL"**,**"DF"**,**"VASC"**])  
*# Store the label names*train\_labels\_df.to\_csv(**"../data/labels/formatted\_train\_labels.csv"**, sep=**"\t"**, index=**False**, header=**False**)  
test\_labels\_df.to\_csv(**"../data/labels/formatted\_test\_labels.csv"**, sep=**"\t"**, index=**False**, header=**False**)

image\_resize.py

*# Author: Daniel Yan  
# Email: daniel.yan@vanderbilt.edu  
# Description: Quick script to resize images to 224x224 to use torchvision models.***from** PIL **import** Image  
**import** os, sys  
  
*# Constants*OLD\_PATH = **"../data/original/"**NEW\_PATH = **"../data/resized224/"***# Resize training images***for** file\_name **in** os.listdir(OLD\_PATH+**"train"**):  
 *# Open image* old\_image = Image.open(OLD\_PATH+**"train/"**+file\_name)  
 *# Resize image* new\_image = old\_image.resize((224, 224), Image.ANTIALIAS)  
 *# Save image* new\_image.save(NEW\_PATH+**"train/"**+file\_name)  
  
*# Resize testing images***for** file\_name **in** os.listdir(OLD\_PATH+**"test"**):  
 *# Open image* old\_image = Image.open(OLD\_PATH+**"test/"**+file\_name)  
 *# Resize image* new\_image = old\_image.resize((224, 224), Image.ANTIALIAS)  
 *# Save image* new\_image.save(NEW\_PATH+**"test/"**+file\_name)

train.py

*# Author: Daniel Yan  
# Email: daniel.yan@vanderbilt.edu  
# Description: Train densenet for image classification.  
  
# Imports for Pytorch***from** \_\_future\_\_ **import** print\_function  
**import** argparse  
**from** matplotlib **import** pyplot **as** plt  
**import** numpy **as** np  
**import** pandas **as** pd  
**import** os  
**import** torch  
**from** torch.utils.data **import** Dataset, DataLoader  
**import** torch.nn **as** nn  
**import** torch.nn.functional **as** F  
**import** torch.optim **as** optim  
**from** torchvision **import** transforms, models  
**from** torch.optim.lr\_scheduler **import** StepLR  
**from** skimage **import** io  
  
*# Constants for the name of the model to save to*MODEL\_NAME = **"densenet\_pretrained"***# Class for the dataset***class** ImagesDataset(Dataset):  
 **def** \_\_init\_\_(self, csv\_file, root\_dir, transform=**None**):  
 *"""  
 Args:  
 csv\_file (string): Path to the csv file with annotations.  
 root\_dir (string): Directory with all the images.  
 transform (callable, optional): Optional transform to be applied  
 on a sample.  
 """* self.labels\_df = pd.read\_csv(csv\_file, sep=**"\t"**, header=**None**)  
 self.root\_dir = root\_dir  
 self.transform = transform  
  
 **def** \_\_len\_\_(self):  
 **return** len(self.labels\_df)  
  
 **def** \_\_getitem\_\_(self, idx):  
 **if** torch.is\_tensor(idx):  
 idx = idx.tolist()  
  
 img\_name = os.path.join(self.root\_dir,  
 self.labels\_df.iloc[idx, 0])  
 image = io.imread(img\_name)  
 label = self.labels\_df.iloc[idx, 1:]  
 sample = {**'image'**: image, **'label'**: label}  
  
 **if** self.transform:  
 sample = self.transform(sample)  
  
 **return** sample  
  
**class** ToTensor(object):  
 *"""Convert ndarrays in sample to Tensors."""* **def** \_\_call\_\_(self, sample):  
 image, label = sample[**'image'**], sample[**'label'**]  
 *# Normalize images with mean and standard deviation for pretrained models* normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],  
 std=[0.229, 0.224, 0.225])  
 in\_transform = transforms.Compose([normalize])  
 *# swap color axis because  
 # numpy image: H x W x C  
 # torch image: C X H X W* image = image.transpose((2, 0, 1))  
 image = torch.from\_numpy(image).float()  
 image = in\_transform(image)  
 *# Format label as torch tensor* label = torch.from\_numpy(np.array(label).astype(int))  
 **return** {**'image'**: image,  
 **'label'**: label}  
  
  
**def** train(args, model, device, train\_loader, optimizer, epoch, train\_losses):  
 *# Specify that we are in training phase* model.train()  
 *# Total Train Loss* total\_loss = 0  
 *# Iterate through all minibatches.* **for** batch\_index, batch\_sample **in** enumerate(train\_loader):  
 *# Send training data and the training labels to GPU/CPU* data, target = batch\_sample[**"image"**].to(device, dtype=torch.float32), batch\_sample[**"label"**].to(device, dtype=torch.long)  
 *# Zero the gradients carried over from previous step* optimizer.zero\_grad()  
 *# Get the label* target = target[:, 0]  
 *# Obtain the predictions from forward propagation* output = model(data)  
 *# Compute the cross entropy for the loss.* loss = F.cross\_entropy(output, target)  
 total\_loss += loss.item()  
 *# Perform backward propagation to compute the negative gradient, and  
 # update the gradients with optimizer.step()* loss.backward()  
 optimizer.step()  
 *# Update training error and add to accumulation of training loss over time.* train\_error = total\_loss / len(train\_loader)  
 train\_losses.append(train\_error)  
 *# Print output if epoch is finished* print(**'Train Epoch: {} \tAverage Loss: {:.6f}'**.format(epoch, train\_error))  
  
  
  
**def** test(args, model, device, test\_loader, test\_losses):  
 *# Specify that we are in evaluation phase* model.eval()  
 *# Set the loss and number of correct instances initially to 0.* test\_loss = 0  
 *# Set no correct predictions initially* correct = 0  
 *# No gradient calculation because we are in testing phase.* **with** torch.no\_grad():  
 *# For each testing example, we run forward  
 # propagation to calculate the  
 # testing prediction. Update the total loss  
 # and the number of correct predictions  
 # with the counters from above.* **for** batch\_idx, batch\_sample **in** enumerate(test\_loader):  
 *# Send data and the labels to GPU/CPU* data, target = batch\_sample[**"image"**].to(device, dtype=torch.float32), batch\_sample[**"label"**].to(device,  
 dtype=torch.long)  
 *# Get the label with one less dimension* target = target[:, 0]  
 *# Obtain the output from the model* output = model(data)  
 *# Calculate the loss using cross entropy.* loss = F.cross\_entropy(output, target)  
 *# Increment the total test loss* test\_loss += loss.item()  
 *# Get the prediction by getting the index with the maximum probability* pred = output.argmax(dim=1, keepdim=**True**)  
 *# Get the number of correct predictions* correct += pred.eq(target.view\_as(pred)).sum().item()  
  
 *# Append test loss to total losses* test\_losses.append(test\_loss / len(test\_loader))  
  
 *# Print out the statistics for the testing set.* print(**'\nTest set: Average loss: {:.6f}'**.format(  
 test\_loss / len(test\_loader)))  
 *# Print out the number of correct predictions* print(**'\nTest set: Correct Predictions: {}/{}'**.format(  
 correct, len(test\_loader.dataset)))  
 *# Print out testing accuracy* print(**"\nTest set: Accuracy: {}"**.format(float(correct/len(test\_loader.dataset))))  
  
  
  
**def** main():  
 *# Command line arguments for hyperparameters of model/training.* parser = argparse.ArgumentParser(description=**'PyTorch Object Detection'**)  
 parser.add\_argument(**'--batch-size'**, type=int, default=8, metavar=**'N'**,  
 help=**'input batch size for training (default: 8)'**)  
 parser.add\_argument(**'--test-batch-size'**, type=int, default=64, metavar=**'N'**,  
 help=**'input batch size for testing (default: 64)'**)  
 parser.add\_argument(**'--epochs'**, type=int, default=50, metavar=**'N'**,  
 help=**'number of epochs to train (default: 50)'**)  
 parser.add\_argument(**'--gamma'**, type=float, default=1, metavar=**'N'**,  
 help=**'gamma value for learning rate decay (default: 1)'**)  
 parser.add\_argument(**'--no-cuda'**, action=**'store\_true'**, default=**False**,  
 help=**'disables CUDA training'**)  
 parser.add\_argument(**'--seed'**, type=int, default=1, metavar=**'S'**,  
 help=**'random seed (default: 1)'**)  
 args = parser.parse\_args()  
 *# Command to use gpu depending on command line arguments and if there is a cuda device* use\_cuda = **not** args.no\_cuda **and** torch.cuda.is\_available()  
  
 *# Random seed to use* torch.manual\_seed(args.seed)  
  
 *# Set to either use gpu or cpu* device = torch.device(**"cuda" if** use\_cuda **else "cpu"**)  
  
 *# GPU keywords.* kwargs = {**'num\_workers'**: 1, **'pin\_memory'**: **True**} **if** use\_cuda **else** {}  
  
 *# Load in the dataset and split into training and validation* data = ImagesDataset(csv\_file=**"../data/labels/formatted\_train\_labels.csv"**, root\_dir=**"../data/resized224/train/"**, transform=ToTensor())  
 train\_size = int(0.9 \* len(data))  
 test\_size = len(data) - train\_size  
 train\_data, val\_data = torch.utils.data.random\_split(data, [train\_size, test\_size])  
 *# Create data loader for training and validation* train\_loader = DataLoader(train\_data, batch\_size=args.batch\_size, shuffle=**True**, num\_workers=0)  
 val\_loader = DataLoader(val\_data, batch\_size=args.test\_batch\_size, shuffle=**False**, num\_workers=0)  
  
 *# Use densenet* model = models.densenet121(pretrained=**True**)  
 *# Number of classes is 7* num\_classes = 7  
 *# Reshape the output for densenet for this problem* model.classifier = nn.Linear(1024, num\_classes)  
 *# Send model to gpu* model = model.to(device)  
 *# Specify Adam optimizer* optimizer = optim.Adam(model.parameters())  
  
 *# Store training and validation losses over time* train\_losses = []  
 val\_losses = []  
  
 *# Create scheduler.* scheduler = StepLR(optimizer, step\_size=1, gamma=args.gamma)  
  
 *# Store the lowest loss found so far for early stopping* lowest\_loss = 1000  
  
 *# Train the model for the set number of epochs* **for** epoch **in** range(1, args.epochs + 1):  
 *# Train and validate for this epoch* train(args, model, device, train\_loader, optimizer, epoch, train\_losses)  
 test(args, model, device, val\_loader, val\_losses)  
 scheduler.step()  
  
 *# If we find the lowest loss so far, store the model and learning curve* **if** lowest\_loss > val\_losses[epoch - 1]:  
 *# Update the lowest loss* lowest\_loss = val\_losses[epoch - 1]  
 print(**"New lowest validation loss: "**, lowest\_loss)  
  
 *# Create learning curve* figure, axes = plt.subplots()  
 *# Set axes labels and title* axes.set(xlabel=**"Epoch"**, ylabel=**"Loss"**, title=**"Learning Curve"**)  
 *# Plot the learning curves for training and validation loss* axes.plot(np.array(train\_losses), label=**"train\_loss"**, c=**"b"**)  
 axes.plot(np.array(val\_losses), label=**"validation\_loss"**, c=**"r"**)  
 plt.legend()  
 *# Save the figure* plt.savefig(MODEL\_NAME + **".png"**)  
 plt.close()  
  
 *# Save the model* torch.save(model.state\_dict(), MODEL\_NAME + **".pt"**)  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 main()

test.py

*# Name: Daniel Yan  
# Email: daniel.yan@vanderbilt.edu  
# Description: Predict label for a single image.  
  
# Imports***import** argparse  
**import** numpy **as** np  
**from** PIL **import** Image  
**import** torch  
**import** torch.nn **as** nn  
**from** torchvision **import** transforms, models  
  
*# Constants*MODEL\_NAME = **"densenet\_pretrained.pt"  
  
  
def** main():  
 *# Command line arguments for the image path and x and y coordinates* parser = argparse.ArgumentParser(description=**'Predict Class for Single Image'**)  
 parser.add\_argument(**'image\_path'**, help=**'path to the image to display'**)  
 args = parser.parse\_args()  
  
 *# Open the image passed by the command line argument* image = Image.open(args.image\_path)  
 *# Convert to numpy array and transpose to get right dimensions* image = np.array(image)  
 image = image.transpose((2, 0, 1))  
 *# Convert to torch image* image = torch.from\_numpy(image).float()  
 *# Normalize image* in\_transform = transforms.Compose([transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])  
 image = in\_transform(image)  
 *# Create tensor of 64 images with all being 0s except first image, since we need  
 # test batch size fo 64 for the model* tensor = torch.tensor((), dtype=torch.float32)  
 tensor = tensor.new\_zeros((64, 3, 224, 224))  
 tensor[0, :, :, :] = image  
  
 *# Specify cuda device* device = torch.device(**"cuda"**)  
 *# Send image to cuda device* tensor = tensor.to(device, dtype=torch.float32)  
  
 *# Use densenet* model = models.densenet121()  
 *# Number of classes is 7* num\_classes = 7  
 *# Reshape the output for densenet for this problem* model.classifier = nn.Linear(1024, num\_classes)  
 *# Send model to gpu and load in saved parameters for prediction* model = model.to(device)  
 model.load\_state\_dict(torch.load(MODEL\_NAME))  
 *# Specify that we are in evaluation phase* model.eval()  
  
 *# No gradient calculation because we are in testing phase.* **with** torch.no\_grad():  
 *# Get the prediction and print* output = model(tensor)  
 print(output.argmax(dim=1, keepdim=**True**)[0].item())  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 main()

test\_metrics.py

*# Name: Daniel Yan  
# Email: daniel.yan@vanderbilt.edu  
# Description: Calculate accuracy, precision, and recall for the testing set.  
  
# Imports***import** matplotlib.pyplot **as** plt  
**import** os  
**import** numpy **as** np  
**import** pandas **as** pd  
**import** torch  
**import** torch.nn **as** nn  
**from** torch.utils.data **import** Dataset, DataLoader  
**from** torchvision **import** transforms, models  
**from** sklearn **import** metrics  
**from** skimage **import** io  
**import** seaborn  
  
*# Constants*MODEL\_NAME = **"densenet\_pretrained.pt"***# Class for the dataset***class** ImagesDataset(Dataset):  
 **def** \_\_init\_\_(self, csv\_file, root\_dir, transform=**None**):  
 *"""  
 Args:  
 csv\_file (string): Path to the csv file with annotations.  
 root\_dir (string): Directory with all the images.  
 transform (callable, optional): Optional transform to be applied  
 on a sample.  
 """* self.labels\_df = pd.read\_csv(csv\_file, sep=**"\t"**, header=**None**)  
 self.root\_dir = root\_dir  
 self.transform = transform  
  
 **def** \_\_len\_\_(self):  
 **return** len(self.labels\_df)  
  
 **def** \_\_getitem\_\_(self, idx):  
 **if** torch.is\_tensor(idx):  
 idx = idx.tolist()  
  
 img\_name = os.path.join(self.root\_dir,  
 self.labels\_df.iloc[idx, 0])  
 image = io.imread(img\_name)  
 label = self.labels\_df.iloc[idx, 1:]  
 sample = {**'image'**: image, **'label'**: label}  
  
 **if** self.transform:  
 sample = self.transform(sample)  
  
 **return** sample  
  
**class** ToTensor(object):  
 *"""Convert ndarrays in sample to Tensors."""* **def** \_\_call\_\_(self, sample):  
 image, label = sample[**'image'**], sample[**'label'**]  
 *# Normalize images with mean and standard deviation for pretrained models* normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],  
 std=[0.229, 0.224, 0.225])  
 in\_transform = transforms.Compose([normalize])  
 *# swap color axis because  
 # numpy image: H x W x C  
 # torch image: C X H X W* image = image.transpose((2, 0, 1))  
 image = torch.from\_numpy(image).float()  
 image = in\_transform(image)  
 *# Format label as torch tensor* label = torch.from\_numpy(np.array(label).astype(int))  
 **return** {**'image'**: image,  
 **'label'**: label}  
  
  
**def** main():  
 *# Load in the test dataset* data = ImagesDataset(csv\_file=**"../data/labels/formatted\_test\_labels.csv"**, root\_dir=**"../data/resized224/test/"**,  
 transform=ToTensor())  
 *# Create data loader for batch testing* test\_loader = DataLoader(data, batch\_size=64, shuffle=**False**, num\_workers=0)  
  
 *# Specify cuda device* device = torch.device(**"cuda"**)  
  
 *# Use densenet* model = models.densenet121()  
 *# Number of classes is 7* num\_classes = 7  
 *# Reshape the output for densenet for this problem* model.classifier = nn.Linear(1024, num\_classes)  
 *# Send model to gpu and load in saved parameters for prediction* model = model.to(device)  
 model.load\_state\_dict(torch.load(MODEL\_NAME))  
 *# Specify that we are in evaluation phase* model.eval()  
  
 *# No gradient calculation because we are in testing phase.* **with** torch.no\_grad():  
 *# Accumulate the predictions and actual labels* predictions = torch.tensor((), dtype=torch.long).to(device)  
 actual = torch.tensor((), dtype=torch.long).to(device)  
  
 *# Iterate through all batches.* **for** batch\_idx, batch\_sample **in** enumerate(test\_loader):  
 *# Send data and the labels to GPU/CPU* data, target = batch\_sample[**"image"**].to(device, dtype=torch.float32), batch\_sample[**"label"**].to(device,  
 dtype=torch.long)  
 *# Get the label with one less dimension* target = target[:, 0]  
 *# Predict the current batch* output = model(data)  
 *# Get the maximum probability from softmax, and slice to get rid of unneeded dimension.* output = output.argmax(dim=1, keepdim=**True**)[:, 0]  
 *# Append prediction and actual values to cumulative predictions.* predictions = torch.cat((predictions, output), 0)  
 actual = torch.cat((actual, target), 0)  
  
 *# Convert to numpy array* predictions = predictions.cpu().numpy()  
 actual = actual.cpu().numpy()  
  
 *# Use scikit-learn to print out accuracy, precision, and recall* print(**"Test set accuracy: "**, metrics.accuracy\_score(actual, predictions))  
 print(**"Test set precision: "**, metrics.precision\_score(actual, predictions, average=**"macro"**))  
 print(**"Test set recall: "**, metrics.recall\_score(actual, predictions, average=**"macro"**))  
  
 *# Use scikit-learn to calculate confusion matrix* confusion\_matrix = metrics.confusion\_matrix(actual, predictions, normalize=**"true"**)  
 *# Use seaborn to plot heatmap* axes = seaborn.heatmap(confusion\_matrix, annot=**True**)  
 axes.set(xlabel=**"Predicted Label"**, ylabel=**"Actual Label"**, title=**"Confusion Matrix"**)  
 *# Save as image and show plot.* plt.savefig(**"confusion\_matrix.png"**)  
 plt.show()  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 main()