Slide 1

CS 8395 Assignment 2

Daniel Yan

Slide 2

Introduction

- Task: Classify image of skin deformation into one of seven categories

Slide 3

Rationale for Method

- Classification using pretrained densenet121: transfer learning
- Alternative architectures attempted:
  - VGG
  - Resnet
- Unsuccessful alternative ideas (similar validation accuracy)
  - Binary prediction of if image is class 1, then classify remaining 6 classes
  - Ensemble methods

Slide 4

Figure of Network Structure

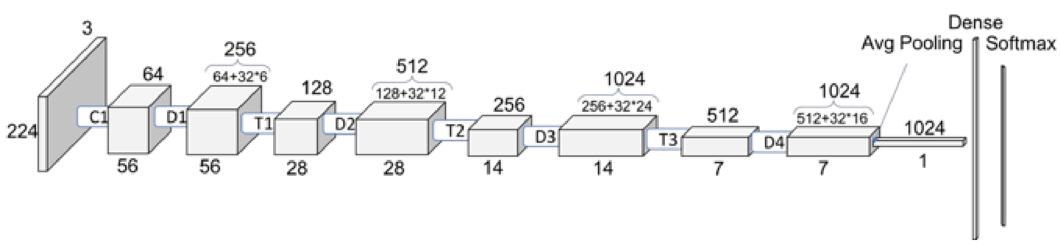


Figure: https://towardsdatascience.com/understanding-and-visualizing-densenets-7f688092391a

Original Paper: Huang, Gao, et al. "Densely connected convolutional networks." *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2017.

Slide 5

Input/Output formatting

- Training/Validation Split: 10% of training data (~900 images) used for validation
- Input: Resized to 224x224 to use pretrained ImageNet model for transfer learning (also I have a laptop GPU)
- Output: Seven values for probabilities of each class
- Postprocessing: Take maximum probability as prediction

Slide 6

Tricks

- Early Stopping: Save and use the model with the lowest validation loss
- Useful due to spikes in validation loss during training

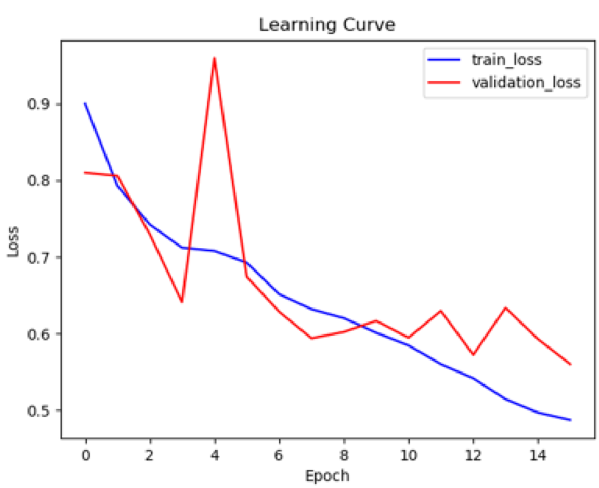
Slide 7

Hyperparameters

Epochs	50 with early stopping
Batch Size	8
Learning Rate	0.001 (Default for Adam)
Optimizer	Adam
Loss Function	Cross Entropy
OS	Windows
GPU	GTX 970M
Parameters of Layers	Pretrained DenseNet-121

Slide 8

Learning Curve



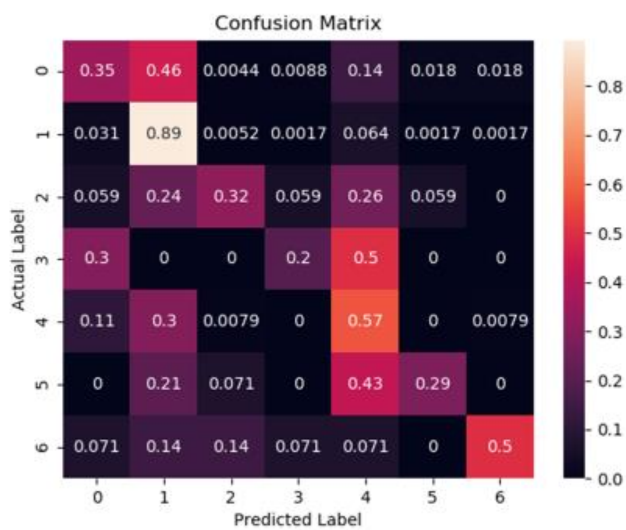
Slide 9

Test Metrics

- Accuracy: 0.691
- Precision: 0.518 (sklearn macro weighted)
- Recall: 0. 447 (sklearn macro weighted)

Slide 10

Confusion Matrix



Slide 11

Conclusions

- Key Challenge: Different distribution of training/validation and testing sets
  - Must avoid overfitting testing set
- Future Improvements
  - Explore statistical modeling to make model more robust to different distributions

Slide 12

Acknowledgements

- Huang, Gao, et al. "Densely connected convolutional networks." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017.

Code

generate_labels.py:

```python
# 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
```

```python
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.
```

```python
# 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

```python
# 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()
```