Biostatistics (STA5195) Final Lab Report

Classifying Skin Cancer

Christopher Rutherford

December 4, 2020

Abstract

In this report, we will demonstrate how a pre-trained ResNet18 model [1] can be used in the training and classification of new, unseen images. In particular, we want to classify two types of images - some benign skin moles or other growths, and malignant skin cancers. Like previous labs, we will perform this work in Google Colab, along with downloading images to and importing from Google Drive.

1 Introduction

Convolutional neural networks (CNNs) are very effective and powerful deep learning algorithms that can be used for the classification of images [2]. In particular, for images, they are typically used to detect objects that either take up the majority of the image/are the subject, such as telling apart dogs and cats, or the presence/absence of the subject of interest. For this report, we'll demonstrate how the pre-trained ResNet18 model can assist us in training and detecting images of skin growths to differentiate between benign growths/moles and malignant cancers.

2 Theory

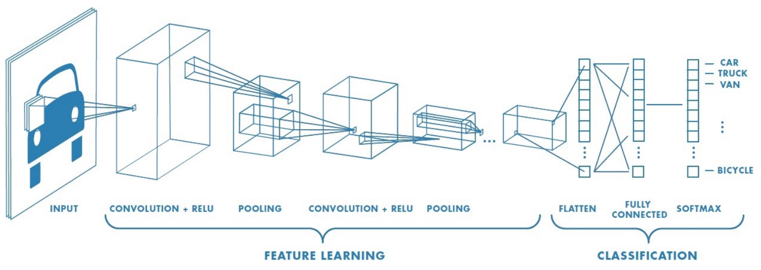

Figure 1: CNN architecture [3]

CNNs have a few main components in their inner workings: convolutional layers, pooling layers, and fully connected layers. The convolutional layers are responsible for extracting features from the images through the use of filters (also called kernels). These kernels "slide" over the image to extract these features. Typically, these features include some sort of edge detection, or some other characteristic detected by these kernels after transforming the images. Pooling layers are responsible for reducing the dimensionality of the data. This

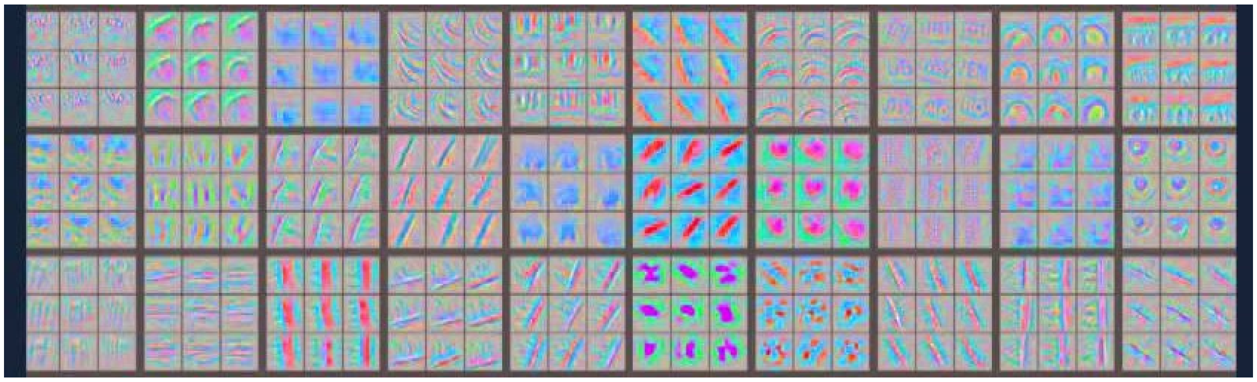

Figure 2: Feature map example of a CNN [4]

helps reduce processing time and power by storing only the most critical information needed to classify the image. The two most common types of pooling include max pooling and average pooling. Like convolution, pooling slides over the image to extract features, but can incorporate stride as well, which decides how many pixels to shift over at a time.

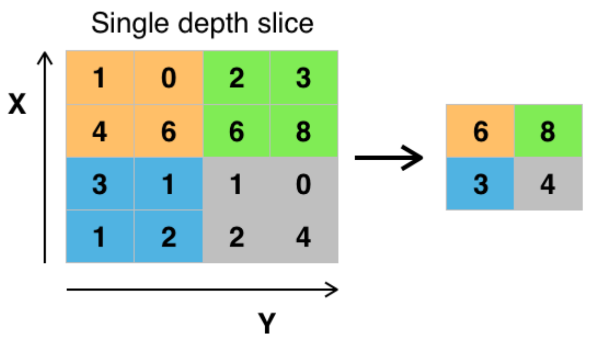

Figure 3: Max pooling with a 2x2 filter and stride of 2 [5]

At this point, we get to the fully connected layers, named so because each node/neuron in these layers is connected to each neuron in the preceding and succeeding layers. These layers are responsible for receiving the transformed data and deciding which class the image belongs to. When training the network, this also includes a loss layer, such as softmax or cross-entropy, which penalizes misclassifications.

3 Procedures

We begin by installing and importing a couple of important libraries:

```
!pip install flashtorch
!pip install barbar
!pip install git+https://github.com/williamedwardhahn/mpcr
from mpcr import *
from flashtorch.utils import apply_transforms
from flashtorch.saliency import Backprop
import itertools
from sklearn.metrics import confusion_matrix
```

`flashtorch` is a library that allows us to visualize the features that are learned by a neural network [6]. `barbar` shows us the progress of our model training with a fancy, more detailed progress bar. Now we mount our Google Drive:

```
drive.mount('/content/drive')
```

We'll direct Python to the correct directory containing our images to use for our model. I personally used the `bing-image-downloader` library to grab the images, but this can easily cause irrelevant images to be downloaded. As such, manually saving the images is likely to yield a more useful set of images.

```
data_dir = '/content/drive/My Drive/Data/skin/'
```

Since we're dealing with the classification of skin cancer, we'll create a list for the following labels:

```
dataset_labels = ["benign", "malignant"]
```

Now that we have our images ready to access, we can get started with the actual model.

4 Analysis

We begin preparing the data for the model by applying the required transformations:

```
mean = np.array([0.485, 0.456, 0.406])
std = np.array([0.229, 0.224, 0.225])

data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224),  #Data Augmentation
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
    ]),
}
```

Now we apply the transformations and assign the class/label name to each image:

```
batch_size = 16
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
    data_transforms[x]) for x in ['train', 'val']}
dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x],
    batch_size=batch_size, shuffle=True, num_workers=4) for x in ['
    train', 'val']}
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'val'
    ]}
class_names = image_datasets['train'].classes
device = torch.device("cuda:0" if torch.cuda.is_available() else "
    cpu")
```

A batch size of 16 tells us to grab randomly grab 16 of the images images at a time and use that as a set of inputs. The last line tells our algorithm to perform the training on the GPU, which we configured to use in Colab.

We can verify the images were loaded in properly by checking the dataset sizes:

```
dataset_sizes
```

Output: `{'train': 89, 'val': 22}`

Looks good. We now create a helper function that plots some of the images from our dataset, along with their corresponding label:

```
def imshow(inp, title = " "):
    fig, ax = plt.subplots()
    inp = inp.numpy().transpose((1, 2, 0))
    inp = std * inp + mean
    inp = np.clip(inp, 0, 1)
    ax.imshow(inp)
    plt.title(title, loc='center')
    # fig.set_size_inches(5, 5)
    plt.show()
```

```
inputs, classes = next(iter(dataloaders['train']))
inputs = inputs[:4]
classes = classes[:4]
out = torchvision.utils.make_grid(inputs)
imshow(out, title=[dataset_labels[x] for x in classes])
```

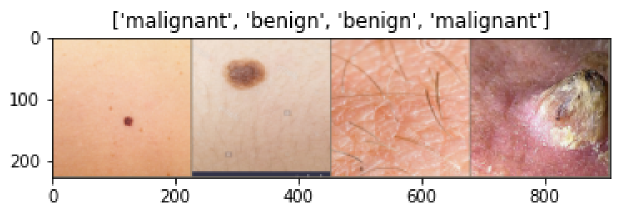

Figure 4: Sample of images

Now that we know our images have been imported and labeled, we can go ahead and prepare the model:

```
def train_model(model, num_epochs=25):

    model = model.to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=0.001, momentum
=0.9)
    scheduler = lr_scheduler.StepLR(optimizer, step_size=7, gamma
=0.1)

    for epoch in range(num_epochs):

        print('Epoch: ',epoch+1,'/',num_epochs)

        ###Train
        model.train()
        running_corrects = 0
        for inputs, labels in dataloaders['train']:
            inputs = inputs.to(device)
            labels = labels.to(device)

            optimizer.zero_grad()

            outputs = model(inputs)

            preds = torch.max(outputs, 1)[1]
            running_corrects += torch.sum(preds == labels.data)

            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

        print("Train ", 'Acc: {:.2f}'.format(running_corrects.double
()/dataset_sizes['train']))

        scheduler.step()

        ###Val
        model.eval()
        running_corrects = 0
        for inputs, labels in dataloaders['val']:
            inputs = inputs.to(device)
            labels = labels.to(device)

            outputs = model(inputs)
            preds = torch.max(outputs, 1)[1]
```

```python
            running_corrects += torch.sum(preds == labels.data)

        print("Valid ", 'Acc: {:.2f}'.format(running_corrects.double
    ()/dataset_sizes['val']))
        print("######################")
    return model
```

```python
model = models.resnet18(pretrained=True)
num_ftrs = model.fc.in_features
model.fc = nn.Linear(num_ftrs, 102)
```

This block of code downloads and loads in the pre-trained ResNet18 model and uses it as our base model.

The next block actually trains our model, borrowing some features from ResNet18.

```python
model = train_model(model, num_epochs=30)
```

For brevity, we will only show the last epoch's output:

```
Epoch:  30 / 30
Train  Acc: 0.96
Valid  Acc: 0.82
######################
```

Certainly not bad, especially considering how the less than desirable images are likely giving the model trouble in accurately classifying the images. We can print out some of the model's predictions:

```python
def visualize_model(model, num_images=16):
    model.eval()
    index = 0
    for i, (inputs, labels) in enumerate(dataloaders['val']):
        inputs = inputs.to(device)
        labels = labels.to(device)

        outputs = model(inputs)

        preds = torch.max(outputs, 1)[1]

        for j in range(inputs.size()[0]):
            index += 1
            title1 = 'predicted: ' + dataset_labels[preds[j]] + '
     class: ' + dataset_labels[labels[j]]
            imshow(inputs.cpu().data[j],title1)

            if index == num_images:
                return
```

```python
visualize_model(model)
```

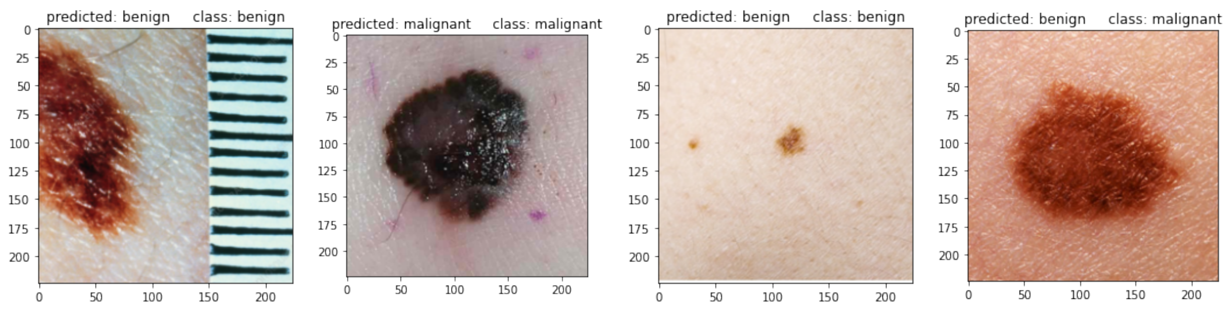

Figure 5: Predictions and true classes for four images

A few mistakes, but still manages to pick up on more obviously benign/malignant images. We can also test the model by grabbing a new image, applying the proper transformations, and performing a prediction on it.

```
image = io.imread('https://www.merckmanuals.com/-/media/manual/home/
    images/basal_cell_carcinoma_b_high.jpg')
plt.imshow(image);
```

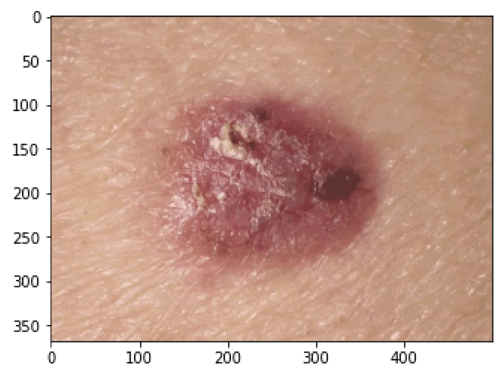

Figure 6: New image to create prediction on

To feed this image into the model, we have to transform it first and attach it to the GPU.

```
img = apply_transforms(image).clone().detach().requires_grad_(True).
    to(device)
```

Now we can make our prediction:

```
outputs = model(img)
preds = torch.max(outputs, 1)[1]
```

```
print('predicted: ' + dataset_labels[preds])
```

```
predicted: malignant
```

And this is correct! Our image is a picture of basal cell carcinoma, a type of skin cancer.

Next, we'll max out our batch size to 1024 to use all of the data for testing.

```
batch_size = 1024 #large batch size so we test all the data
```

```python
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
    data_transforms[x]) for x in ['train', 'val']}
dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x],
    batch_size=batch_size, shuffle=True, num_workers=4) for x in ['
    train', 'val']}
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'val'
    ]}
class_names = image_datasets['train'].classes
device = torch.device("cuda:0" if torch.cuda.is_available() else "
    cpu")
```

Here we'll label each input image accordingly:

```python
inputs, labels = next(iter(dataloaders['val']))
inputs = inputs.to(device)
labels = labels.detach().cpu().numpy()
```

Next, we'll create our predictions from these images.

```python
outputs = model(inputs)
preds = torch.max(outputs, 1)[1].detach().cpu().numpy()
```

The confusion matrix shows us how the images were (mis)classified.

```python
cm = confusion_matrix(preds.reshape(-1),labels)
```

```python
print("Confusion Matrix:")
cm
```

```
Confusion Matrix:
array([[11,  4],
      [ 0,  7]])
```

We'll provide a little more detail as to exactly what these numbers represent:

```python
print(str(cm[0,0])+" benign cases correctly classified (TN)")
print(str(cm[1,1])+" malignant cases correctly classified (TP)")
print(str(cm[0,1])+" malignant cases incorrectly classified as
    benign (FN)")
print(str(cm[1,0])+" benign cases incorrectly classified as
    malignant (FP)")
```

```
11 benign cases correctly classified (TN)
7 malignant cases correctly classified (TP)
4 malignant cases incorrectly classified as benign (FN)
0 benign cases incorrectly classified as malignant (FP)
```

Using these numbers, we can calculate some important metrics to help us further evaluate the model's performance.

The simplest one, accuracy, is the number of correct classifications divided by the total number of data points.

```python
(cm[0, 0]+cm[1,1])/sum(sum(cm))
```

```
0.8181818181818182
```

The true positive (TP) rate, also called the recall, tells us how many cases of cancer were detected out of all of the true cancer cases.

```
recall = cm[1,1]/(cm[1,1]+cm[0,1])
recall
```

```
0.6363636363636364
```

The true negative (TN) rate, also called specificity, tells us how many cases of benign skin growths were correctly identified as such.

```
cm[0,0]/(cm[0,0]+cm[1,0])
```

```
1.0
```

The false positive (FP) rate tells us how many benign cases were incorrectly classified as maligant.

```
cm[1,0]/(cm[0,0]+cm[1,0])
```

```
0.0
```

The precision tells us how many of the samples classified as positive by the model are actually positive.

```
prec = cm[1,1]/(cm[1,1]+cm[1,0])
prec
```

```
1.0
```

The f1 score is the harmonic mean of precision recall, which is especially useful for when classes are imbalanced.

```
f1 = 2*(prec*recall)/(prec+recall)
f1
```

```
0.7777777777777778
```

5 Conclusions

Overall, despite the fact some less than useful images were found by Bing, the model performed quite well on our images. The ones that were obviously malignant (rough texture, uneven, multicolored etc.) were correctly identified as such, while benign ones (single color, flat, symmetric) were all correctly identified. Some images that may have been benign ended up in the malignant data set, so some misclassifications due to this did occur. Ideally, our dataset would consist of images that are exclusively closeups of the growths, as these would be the most likely way to photograph them for classification. For instance, a few of the pictures include the entirety of people's backs, possibly giving the model a much harder time to determine how to classify the growth.

References

[1] ResNet | Pytorch: https://pytorch.org/hub/pytorch_vision_resnet/.

[2] *Convolutional neural network*, available at https://en.wikipedia.org/wiki/Convolutional_neural_network.

[3] Understanding of Convolutional Neural Network (CNN) — Deep Learning: https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148

[4] *Visualizing and Understanding Deep Neural Networks by Matt Zeiler*, available at https://www.youtube.com/watch?v=ghEmQSxT6tw

[5] By Aphex34 - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=45673581

[6] *Flashtorch*, available at https://github.com/MisaOgura/flashtorch.