Breast Cancer Detection Using CNN

Submitted in partial fulfillment of the requirements
for the degree of

PGDM Finance

by

A S Krishnapriya
(Roll No. xxxxxx)

Supervisors:
Dr.Rakesh Nigam

PGDM
MADRAS SCHOOL OF ECONOMICS
2020

Dedicated to my beloved parents.

Abstract

Early detection of breast cancer and it's correct identification can go a long way in not only saving a woman's life but also her quality of life. The identification of breast cancer depends on biomedical images such as mammography scans and histopathological slides which are obtained from biopsies. Analysing these images require specialists in the oncology domain, and there can be times of uncertainty and differences of opinion among experts. Computer aided diagnostic techniques step in to resolve such a problem. The use of Deep Learning in the field of radiology has shown great promise in improving early and accurate cancer detection.

Breast Cancer Detection using various Deep Learning algorithms by using mammographic scans and histopathological slides have been carried out by researchers in this domain. This study aims to compare the performance of ResNet of 3 different depths (a Convolutional Neural Network) trained from scratch for the task of classifying benign and malignant histopathological images. Kaiming initialization of weights was implemented and the final layer was replaced with a fully connected network with softmax activation.

Contents

List of Figures

List of Abbreviations

ANN	Artificial Neural Networks
CNN	Convolutional Neural Networks
ReLu	Rectified Linear Units
BreakHis	Breast Cancer Histopathological Database
MLP	Multi Layer Perceptron

List of Symbols

b	Bias
W	Weight
$\sum$	Summation
A	Vector of Activations
x_i	i^{th} input factor
y	actual output
$\hat{y}$	predicted output
α	learning rate
$g(.)$	the function of adding the product of weights and input to bias
Z	output of adding the product of weights and inputs to bias
G	Convolution function
m	number of rows of input matrix
n	number of columns of input matrix
j	number of rows of kernel
k	number of columns of kernel
f	kernel
v	input image
F	Kernel Size
V	Input image size
P	amount of zero padding
S	Stride

n_c	number of channels
n_f	number of filters
$\mathscr{F}$	Residual Function

Appendix A

Backpropagation in CNN

Before we dive into backpropagation, we need to revisit the chain rule of calculus, as it is vital (some would even say "integral"!) to understanding gradient descent. Let's take the function f(x,y,z) = (x+y)z.

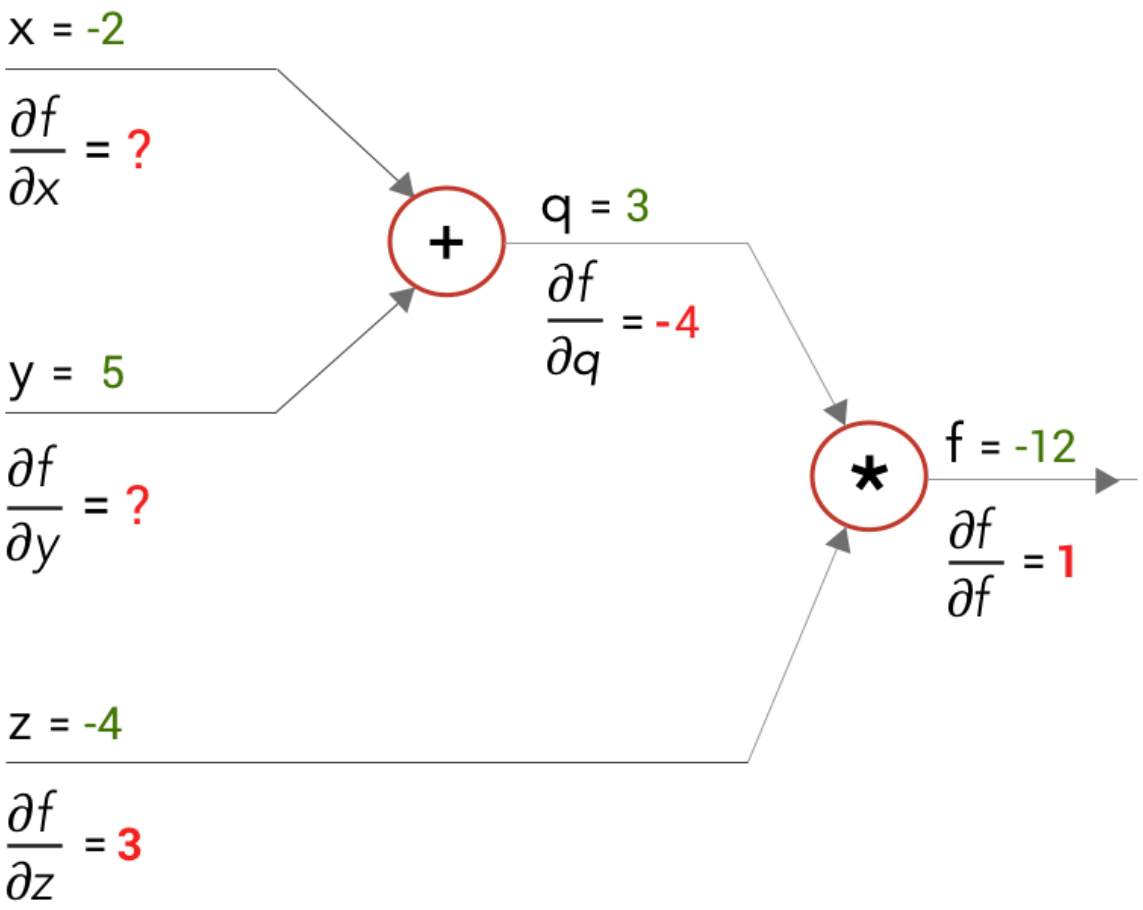
$$f(x,y,z) = (x+y)z \tag{A.1}$$

$$Let, q = (x+y) \tag{A.2}$$

$$then, f = q*z \tag{A.3}$$

Let's take a dummy example of x =-2 , y=5 and z =4, by the equation by we would get f = -12

This here is the forward pass. Now, let's get to the backward pass. We first compute the gradient of the output at every step. We need to find $\frac{\partial f}{\partial y}$ and $\frac{\partial f}{\partial x}$ This can be done using the chain rule as follows :



$f = q * z$

$\frac{\partial f}{\partial q} = z \mid z = -4$

$\frac{\partial f}{\partial z} = q \mid q = 3$

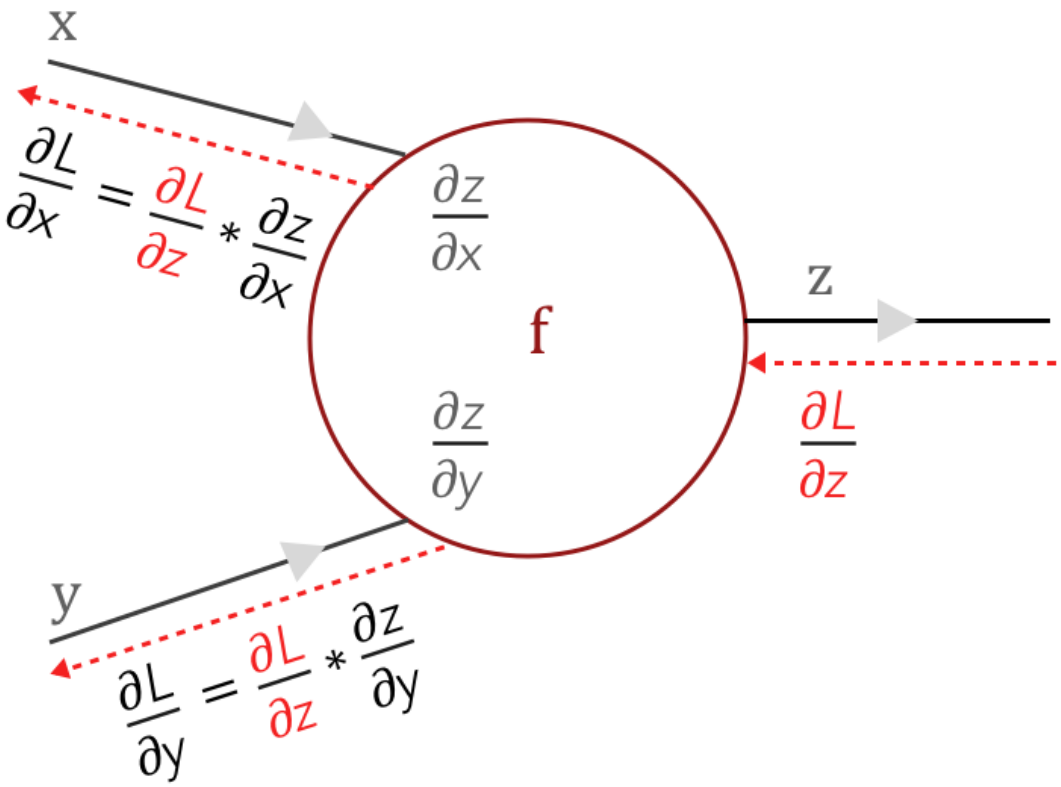
$q = x + y$

$\frac{\partial q}{\partial x} = 1 \qquad \frac{\partial q}{\partial y} = 1$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} * \frac{\partial q}{\partial x} = -4 * 1 = -4 \tag{A.4}$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} * \frac{\partial q}{\partial y} = -4 * 1 = -4 \tag{A.5}$$

A CNN is one such massive computational graph. Imagine a node with a gate (activation function) f that receives inputs x and y. Let's assume this gives an output z. The gradients of the output z with respect to x and y are : $\frac{\partial z}{\partial x}$ and $\frac{\partial f}{\partial y}$. These are called local gradients. During the forward pass, the input is passed through the CNN and the loss is calculated using the loss function. During the backward pass, we work this loss backwards by calculating the gradient of the loss from the previous layer. For this we find $\frac{\partial L}{\partial z}$. To propagate this loss across the layer, we need to find the partial derivative with respect to the input : $\frac{\partial L}{\partial x}$ and $\frac{\partial L}{\partial y}$. This is where the chain rule comes in. This process can be diagramatically represented as follows :

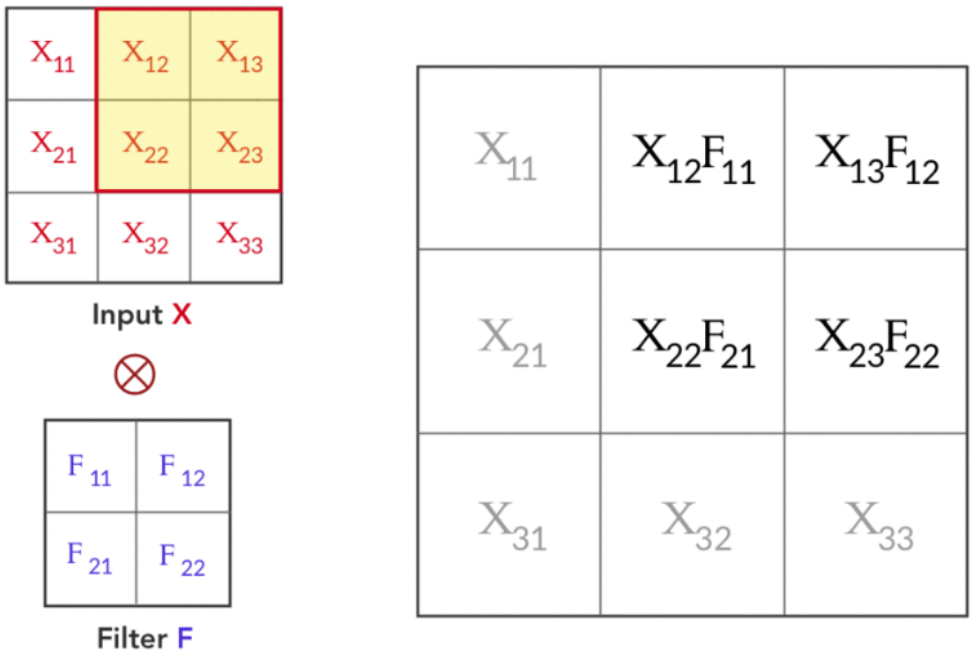


$\frac{\partial z}{\partial x}$ & $\frac{\partial z}{\partial y}$ *are local gradients*

$\frac{\partial L}{\partial z}$ *is the loss from the previous layer which has to be backpropagated to other layers*

Let's see how this happens for convolutions. Let's assume a convolution is carried out by

a filter **F** of size 2x2 and an input **X** of size 3x3. The output **O** the output will be of size 2x2. This is represented below :



$$O_{11} = X_{11}F_{11} + X_{12}F_{12} + X_{21}F_{21} + X_{22}F_{22}$$

$$O_{12} = X_{12}F_{11} + X_{13}F_{12} + X_{22}F_{21} + X_{23}F_{22}$$

So, during the backward pass $\frac{\partial L}{\partial O}$ is computed. Similarly as mentioned before $\frac{\partial L}{\partial X}$, this becomes the loss gradient of the previous layer, and $\frac{\partial L}{\partial F}$ are also computed.

$$F_{updated} = F - \alpha * \frac{\partial L}{\partial F} \tag{A.6}$$

(A.7)

This is how the filter gets As we are dealing with a matrix of inputs, the gradient for each element is to be computed. $\frac{\partial L}{\partial F}$ can be computed by using the chain rule ement wise, in the following manner :

$$\frac{\partial L}{\partial F_i} = \sum_{k=1}^{M} \frac{\partial L}{\partial O_k} * \frac{\partial O_k}{\partial F_i} \tag{A.8}$$

We can subsitute the value for the local gradients and obtain $\frac{\partial L}{\partial F}$ for each element. The local gradients can be obtained by

$$O_{11} = X_{11}F_{11} + X_{12}F_{12} + X_{21}F_{21} + X_{22}F_{22} \tag{A.9}$$

$$\frac{\partial O_1 1}{\partial X_1 1} = F_1 1 ...andsoon. \tag{A.10}$$

$$\tag{A.11}$$

Then we use this to find :

$$\frac{\partial L}{\partial X_i} = \sum_{k=1}^{M} \frac{\partial L}{\partial O_k} * \frac{\partial O_k}{\partial X_i} \tag{A.12}$$

A.1 Stochastic Gradient Descent

Let us begin the derivation of the learning algorithm by first looking at a loss function **J** for parameters θ

$$J(\theta) = \theta^2 \tag{A.13}$$

The objective of the learning algorithm is to find the value of the parameter that minimises the cost We can compute the different values of the function by changing theta and updtating it

based on the updation rule.α is the learning rate. To start,

$$\theta = 3 \tag{1.13}$$

$$\frac{\partial J(\theta)}{\theta} = 2\theta = 6 \tag{1.14}$$

$$\alpha\frac{\partial J(\theta)}{\theta} = 0.6 \tag{1.15}$$

$$\theta := \theta - \alpha\frac{\partial J(\theta)}{\theta} \tag{1.16}$$

$$\theta = 3 - 0.6 = 2.4 \tag{1.17}$$

The value of theta is updated as shown above until the minimum is reached i.e if on the next iteration, if the value of the function increases, we know we have reached the minimum. This gives the direction in which to move in. α is the learning rate, this tells us how fast to move. Gradient Descent for multiple variables uses the chain rule, quotient rule, sum rule and scalar multiple rule of calculus.

Function:

$$J(\theta_1, \theta_2) = {\theta_1}^2 + {\theta_2}^2$$

Objective:

$$\min_{\theta_1, \theta_2} J(\theta_1, \theta_2)$$

Update rules:

$$\theta_1 := \theta_1 - \alpha\frac{\partial}{\partial\theta_1}J(\theta_1, \theta_2)$$

$$\theta_2 := \theta_2 - \alpha\frac{\partial}{\partial\theta_2}J(\theta_1, \theta_2)$$

Derivatives:

$$\frac{\partial}{\partial\theta_1}J(\theta_1, \theta_2) = \frac{\partial}{\partial\theta_1}{\theta_1}^2 + \frac{\partial}{\partial\theta_1}{\theta_2}^2 = 2\theta_1$$

$$\frac{\partial}{\partial\theta_2}J(\theta_1, \theta_2) = \frac{\partial}{\partial\theta_2}{\theta_1}^2 + \frac{\partial}{\partial\theta_2}{\theta_2}^2 = 2\theta_2$$

For a function $h(x) = \theta_0 + \theta_1 x$ let us calculate the derivatives. The derivation is as follows.

We multiply the cost function $J(\theta)$ by half so that the 2 cancels out when we take the

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

$$\frac{\partial}{\partial\theta_0}J(\theta_0, \theta_1) = \frac{\partial}{\partial\theta_0}\left(\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2\right)$$

$$= \frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial\theta_0}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

$$= \frac{1}{m}\sum_{i=1}^{m}2\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)\frac{\partial}{\partial\theta_0}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)$$

$$= \frac{2}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)$$

derivative. This can be summed up below :

Cost Function – "One Half Mean Squared Error":

$$J(\theta_0, \theta_1) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

Objective:

$$\min_{\theta_0, \theta_1} J(\theta_0, \theta_1)$$

Update rules:

$$\theta_0 := \theta_0 - \alpha\frac{\partial}{\partial\theta_0}J(\theta_0, \theta_1)$$
$$\theta_1 := \theta_1 - \alpha\frac{\partial}{\partial\theta_1}J(\theta_0, \theta_1)$$

Derivatives:

$$\frac{\partial}{\partial\theta_0}J(\theta_0, \theta_1) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)$$

$$\frac{\partial}{\partial\theta_1}J(\theta_0, \theta_1) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)\cdot x^{(i)}$$

Here, we can leave out the 1/m term, but we want it show that the same learning rate

is applied to all batches used in training of size m. Hence α = 1/m in the derivative. if we were to slightly modify this, we would have the algorithm for stochastic gradient descent.If we were to repeatedly run through the training set and update the parameters every time an example was encountered according to the gradient error, we have the Stochastic Gradient Descent Algorithm:

Loop {

for i=1 to m, {

$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)} \qquad \text{(for every } j\text{)}.$$

}

}

Chapter 2

Pytorch Implementation

We pick up the pytorch implementation after describing the data transformations in chapter 3. For making this implementation the github repositories of the studies mentioned in chapter 2 [Sha19] as well the pytorch documentation [Doc17] was referred to. Pytorch offers a tutorial on how to implement CNNs and train them from scratch. There is extensive documentation in the website about how to use the functions used in this study. We create a dataloader which facilitates loading of the images into our model in an iterative manner, then we create the loaders for training, test and validation set.

```
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),data_transforms[x])
                                          for x in ['train','test','val']}

#here, we make the dataloaders, these load the images into the model
dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=batch_size,
                                              shuffle=True, num_workers=4)
               for x in ['train', 'test','val']}
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'test','val']}
```

Figure 2.1

After that we specify the architecture of the model we are about to use, and specify the

```
#making the training, test and val sets and their dataloaders
train_set = datasets.ImageFolder(root="/content/drive/My Drive/40mag/train",transform=train_transforms)

train_loader = DataLoader(train_set,batch_size=batch_size,shuffle=True,num_workers=4)



test_set = datasets.ImageFolder(root="/content/drive/My Drive/40mag/test",transform=test_transforms)

test_loader = DataLoader(test_set,batch_size=batch_size,shuffle=False,num_workers=4)

val_set = datasets.ImageFolder(root="/content/drive/My Drive/40mag/val",transform=val_transforms)

val_loader = DataLoader(val_set,batch_size=batch_size,shuffle=True,num_workers=4)
```

Figure 2.2

kind of weights initialization. Here Kaiming initialization of weights was implemented. Then a function to train the model is written. An example of the implementation of the train function is also shown below, note that the parametrs can be changed to get better results. After training a function to test the model is written and the results such as the ROC curve is obtained. We can save the trained model to load it later for sanity checks ,deployment.

```
import torch
import torch.nn as nn
from torch.hub import load_state_dict_from_url


__all__ = ['ResNet', 'resnet18', 'resnet34', 'resnet50', 'resnet101',
           'resnet152', 'resnext50_32x4d', 'resnext101_32x8d',
           'wide_resnet50_2', 'wide_resnet101_2']


model_urls = {
    'resnet18': 'https://download.pytorch.org/models/resnet18-5c106cde.pth',
    'resnet34': 'https://download.pytorch.org/models/resnet34-333f7ec4.pth',
    'resnet50': 'https://download.pytorch.org/models/resnet50-19c8e357.pth',
    'resnet101': 'https://download.pytorch.org/models/resnet101-5d3b4d8f.pth',
    'resnet152': 'https://download.pytorch.org/models/resnet152-b121ed2d.pth',
    'resnext50_32x4d': 'https://download.pytorch.org/models/resnext50_32x4d-7cdf4587.pth',
    'resnext101_32x8d': 'https://download.pytorch.org/models/resnext101_32x8d-8ba56ff5.pth',
    'wide_resnet50_2': 'https://download.pytorch.org/models/wide_resnet50_2-95faca4d.pth',
    'wide_resnet101_2': 'https://download.pytorch.org/models/wide_resnet101_2-32ee1156.pth',
}
```

Figure 2.3

```python
def conv3x3(in_planes, out_planes, stride=1, groups=1, dilation=1):
    """3x3 convolution with padding"""
    return nn.Conv2d(in_planes, out_planes, kernel_size=3, stride=stride,
                     padding=dilation, groups=groups, bias=False, dilation=dilation)


def conv1x1(in_planes, out_planes, stride=1):
    """1x1 convolution"""
    return nn.Conv2d(in_planes, out_planes, kernel_size=1, stride=stride, bias=False)


class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, inplanes, planes, stride=1, downsample=None, groups=1,
                 base_width=64, dilation=1, norm_layer=None):
        super(BasicBlock, self).__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        if groups != 1 or base_width != 64:
            raise ValueError('BasicBlock only supports groups=1 and base_width=64')
        if dilation > 1:
            raise NotImplementedError("Dilation > 1 not supported in BasicBlock")
        # Both self.conv1 and self.downsample layers downsample the input when stride != 1
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = norm_layer(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = norm_layer(planes)
        self.downsample = downsample
        self.stride = stride
```

Figure 2.4

```
    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)

        return out


class Bottleneck(nn.Module):
    # Bottleneck in torchvision places the stride for downsampling at 3x3 convolution(self.conv2)
    # while original implementation places the stride at the first 1x1 convolution(self.conv1)
    # according to "Deep residual learning for image recognition"https://arxiv.org/abs/1512.03385.
    # This variant is also known as ResNet V1.5 and improves accuracy according to
    # https://ngc.nvidia.com/catalog/model-scripts/nvidia:resnet_50_v1_5_for_pytorch.

    expansion = 4
```

Figure 2.5

```
def __init__(self, inplanes, planes, stride=1, downsample=None, groups=1,
             base_width=64, dilation=1, norm_layer=None):
    super(Bottleneck, self).__init__()
    if norm_layer is None:
        norm_layer = nn.BatchNorm2d
    width = int(planes * (base_width / 64.)) * groups
    # Both self.conv2 and self.downsample layers downsample the input when stride != 1
    self.conv1 = conv1x1(inplanes, width)
    self.bn1 = norm_layer(width)
    self.conv2 = conv3x3(width, width, stride, groups, dilation)
    self.bn2 = norm_layer(width)
    self.conv3 = conv1x1(width, planes * self.expansion)
    self.bn3 = norm_layer(planes * self.expansion)
    self.relu = nn.ReLU(inplace=True)
    self.downsample = downsample
    self.stride = stride

def forward(self, x):
    identity = x

    out = self.conv1(x)
    out = self.bn1(out)
    out = self.relu(out)

    out = self.conv2(out)
    out = self.bn2(out)
    out = self.relu(out)

    out = self.conv3(out)
    out = self.bn3(out)

    if self.downsample is not None:
        identity = self.downsample(x)

    out += identity
    out = self.relu(out)

    return out
```

Figure 2.6

```python
class ResNet(nn.Module):

    def __init__(self, block, layers, num_classes=1000, zero_init_residual=False,
                 groups=1, width_per_group=64, replace_stride_with_dilation=None,
                 norm_layer=None):
        super(ResNet, self).__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        self._norm_layer = norm_layer

        self.inplanes = 64
        self.dilation = 1
        if replace_stride_with_dilation is None:
            # each element in the tuple indicates if we should replace
            # the 2x2 stride with a dilated convolution instead
            replace_stride_with_dilation = [False, False, False]
        if len(replace_stride_with_dilation) != 3:
            raise ValueError("replace_stride_with_dilation should be None "
                             "or a 3-element tuple, got {}".format(replace_stride_with_dilation))
        self.groups = groups
        self.base_width = width_per_group
        self.conv1 = nn.Conv2d(3, self.inplanes, kernel_size=7, stride=2, padding=3,
                               bias=False)
        self.bn1 = norm_layer(self.inplanes)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self._make_layer(block, 64, layers[0])
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2,
                                       dilate=replace_stride_with_dilation[0])
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2,
                                       dilate=replace_stride_with_dilation[1])
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2,
                                       dilate=replace_stride_with_dilation[2])
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512 * block.expansion, num_classes)

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
            elif isinstance(m, (nn.BatchNorm2d, nn.GroupNorm)):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
```

Figure 2.7

```
    if zero_init_residual:
        for m in self.modules():
            if isinstance(m, Bottleneck):
                nn.init.constant_(m.bn3.weight, 0)
            elif isinstance(m, BasicBlock):
                nn.init.constant_(m.bn2.weight, 0)

def _make_layer(self, block, planes, blocks, stride=1, dilate=False):
    norm_layer = self._norm_layer
    downsample = None
    previous_dilation = self.dilation
    if dilate:
        self.dilation *= stride
        stride = 1
    if stride != 1 or self.inplanes != planes * block.expansion:
        downsample = nn.Sequential(
            conv1x1(self.inplanes, planes * block.expansion, stride),
            norm_layer(planes * block.expansion),
        )

    layers = []
    layers.append(block(self.inplanes, planes, stride, downsample, self.groups,
                        self.base_width, previous_dilation, norm_layer))
    self.inplanes = planes * block.expansion
    for _ in range(1, blocks):
        layers.append(block(self.inplanes, planes, groups=self.groups,
                            base_width=self.base_width, dilation=self.dilation,
                            norm_layer=norm_layer))

    return nn.Sequential(*layers)
```

Figure 2.8

```
    def _forward_impl(self, x):
        # See note [TorchScript super()]
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)

        return x

    def forward(self, x):
        return self._forward_impl(x)


def _resnet(arch, block, layers, pretrained, progress, **kwargs):
    model = ResNet(block, layers, **kwargs)
    if pretrained:
        state_dict = load_state_dict_from_url(model_urls[arch],
                                              progress=progress)
        model.load_state_dict(state_dict)
    return model


def resnet18(pretrained=False, progress=True, **kwargs):
    r"""ResNet-18 model from
    `"Deep Residual Learning for Image Recognition" <https://arxiv.org/pdf/1512.03385.pdf>`_
    Args:
        pretrained (bool): If True, returns a model pre-trained on ImageNet
        progress (bool): If True, displays a progress bar of the download to stderr
    """
    return _resnet('resnet18', BasicBlock, [2, 2, 2, 2], pretrained, progress,
                   **kwargs)
```

Figure 2.9

```
#here we load the model we wrote the code for previously
model = resnet50(pretrained=False)

for param in model.parameters():
    param.requires_grad = False  #this is for whether we want to freeze our parameters or not

#this prints the model architecture
    print(model)




from collections import OrderedDict


#here we describe the structure of our classifier

classifier = nn.Sequential(OrderedDict([
                          ('fc1', nn.Linear(2048, 512)),
                           ('relu', nn.ReLU()),
                           ('dropout1', nn.Dropout(p=0.05)),
                           ('fc2', nn.Linear(512, 2)),
                            # ('relu', nn.ReLU()),
                           ('output', nn.LogSoftmax(dim=1))
                           ]))

#loading our classifier
model.fc = classifier
num_classes =2
valloss = []
trainloss =[]
valacc= []
trainacc = []
patience = 30 #threshold for early stopping

early_stopping = EarlyStopping(patience=patience, verbose=True) # if 30 times conescutively the loss does not decrease, the model stops training
```

Figure 2.10

```python
#Function to train the model
def train_model(model, criterion, optimizer, scheduler, num_epochs,patience):
    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    for epoch in range(1, num_epochs+1):
        print('Epoch {}/{}'.format(epoch, num_epochs))
        print('-' * 10)


        for phase in ['train', 'val']:
            if phase == 'train':
                scheduler.step()
                model.train()  #training mode
            else:
                model.eval()   # eval mode

            running_loss = 0.0
            running_corrects = 0

            # loading data with dataloaders
            for inputs, labels in dataloaders[phase]:
                inputs, labels = inputs.to(device), labels.to(device)

                # initialize the parameter gradients
                optimizer.zero_grad()

                # forward prop

                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    loss = criterion(outputs, labels)
                    _, preds = torch.max(outputs, 1)

                    # backward prop
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                # calculating accuracy, loss
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)
```

Figure 2.11

```
            epoch_loss = running_loss / dataset_sizes[phase]
            epoch_acc = running_corrects.double() / dataset_sizes[phase]
            if phase == "train" :
                trainloss.append (epoch_loss) #storing each loss and accuracy of epoch in a list
                trainacc.append(epoch_acc)
            else :
                valloss.append (epoch_loss)
                valacc.append(epoch_acc)


            print('{} Loss: {:.4f} Acc: {:.4f}'.format(
                phase, epoch_loss, epoch_acc)) #printing each loss and accuracy for both phases

            # this stores the model weights for the best model
            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                en = epoch
                best_model_wts = copy.deepcopy(model.state_dict())

        early_stopping(valloss[epoch-1], model) #early stopping

        if early_stopping.early_stop:
              print("Early stopping")
              break
        model.load_state_dict(torch.load('checkpoint.pt')) #stores the checkpoint if early stopping is implemented


        print()

    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(
        time_elapsed // 60, time_elapsed % 60))
    print('Best valid accuracy: {:4f}'.format(best_acc))
    #print ("Best Model weights : ",best_model_wts)  #Optional code to print best model weights
    print (trainloss)
    print (valloss)
```

Figure 2.12

```
#as accuracy gets stored as a tensor, we need to convert it to numpy for plotting a graph
def tensortonumpy (acc) :
      x = acc
      nparray = []

      for i in range (en):
           a = x[i]
           a = a.cpu()
           b = a.numpy()
           c = float(b)
           nparray.append(c)
      return nparray
val_acc = tensortonumpy(valacc)
train_acc =tensortonumpy(trainacc)
print(train_acc)
print(val_acc)

model.load_state_dict(best_model_wts) #loads weights of the best model

#in this section we plot the graphs
def mean(li): return sum(li)/len(li)
plt.figure(figsize=(14, 4))
plt.xlabel('training batch')
plt.ylabel('loss')
plt.plot([mean(trainloss[i:i+num_epochs]) for i in range(len(trainloss))])

def mean(li): return sum(li)/len(li)
plt.figure(figsize=(14, 4))
plt.xlabel('validation batch')
plt.ylabel('loss')
plt.plot([mean(valloss[i:i+num_epochs]) for i in range(len(valloss))])

def mean(li): return sum(li)/len(li)
plt.figure(figsize=(14, 4))
plt.xlabel('training batch')
plt.ylabel('epoch')
plt.plot(train_acc)

def mean(li): return sum(li)/len(li)
plt.figure(figsize=(14, 4))
plt.xlabel('validation batch')
plt.ylabel('epoch')
plt.plot(val_acc)
```

Figure 2.13

```
num_epochs =300
if use_gpu:
    print ("Using GPU: "+ str(use_gpu))
    model = model.cuda()

# NLLLoss because our output is LogSoftmax
criterion = nn.NLLLoss()

# Adam optimizer with a learning rate
#optimizer = optim.Adam(model.fc.parameters(), lr=0.0001,weight_decay = 0.001)
optimizer = optim.SGD(model.fc.parameters(), lr = .0001, momentum=0.09, weight_decay = 0.01)
# Decay LR by a factor--  every -- epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer, step_size=50, gamma=0.1)


model_ft = train_model(model, criterion, optimizer, exp_lr_scheduler, num_epochs,patience)
torch.save(model_ft.state_dict(), "resnet50trial.pth")
```

Figure 2.14

```
[20] # Do validation on the test set
     def test(model, dataloaders, device):
       model.eval()
       actuals = []
       probabilities = []
       predictions = []
       accuracy = 0

       model.to(device)

       for images, labels in dataloaders['test']:
         images = Variable(images)
         labels = Variable(labels)
         images, labels = images.to(device), labels.to(device)

         output = model.forward(images)
         ps = torch.exp(output)
         equality = (labels.data == ps.max(1)[1])
         accuracy += equality.type_as(torch.FloatTensor()).mean()
         ##




         with torch.no_grad():
             for data, target in test_loader:
                 data, target = data.to(device), target.to(device)
                 output = model(data)
                 out    = output.cpu()
                 prediction = out.argmax(dim=1, keepdim=True)
                 actuals.extend(target.view_as(prediction) == 1)
                 probabilities.extend(np.exp((out[:, 1])))

         return [i.item() for i in actuals], [i.item() for i in probabilities]

         print("Testing Accuracy: {:.3f}".format(accuracy/len(dataloaders['test'])))

     actuals, class_probabilities = test(model, dataloaders,device)
```

Figure 2.15

```
actuals, class_probabilities = test(model, dataloaders,device)

fpr, tpr, _ = sklearn.metrics.roc_curve(actuals, class_probabilities)
roc_auc = auc(fpr, tpr)
plt.figure()
lw = 2
plt.plot(fpr, tpr, color='darkorange',
         lw=lw, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC for digit=1 class Malignant')
plt.legend(loc="lower right")
plt.show()

def test_label_predictions(model, device, test_loader):
    model.eval()
    actuals = []
    predictions = []
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            prediction = output.argmax(dim=1, keepdim=True)
            actuals.extend(target.view_as(prediction))
            predictions.extend(prediction)
    return [i.item() for i in actuals], [i.item() for i in predictions]

actuals, predictions = test_label_predictions(model, device, test_loader)
print('Confusion matrix:')
print(confusion_matrix(actuals, predictions))
print('F1 score: %f' % f1_score(actuals, predictions, average='micro'))
print('Accuracy score: %f' % accuracy_score(actuals, predictions))

#test(model, dataloaders, device)
```

Figure 2.16

```
checkpoint = {'input_size': [3, 224, 224],
                  'batch_size': dataloaders['train'].batch_size,
                   'output_size': 2,
                   'state_dict': model.state_dict(),
                   'data_transforms': data_transforms,
                   'optimizer_dict':optimizer.state_dict(),
                   'class_to_idx': model.class_to_idx,
                   'epoch': model.epochs}
torch.save(checkpoint, 'checkpoint.pth')
```

Figure 2.17

This page was intentionally left blank.

Bibliography

[Doc17] Pytorch Documentation. *Pytorch Documentation*. Web resource. The code was built used on the documentation available from here. 2017. URL: Pytorch.org.

[Sha19] Pulkit Sharma. *Build an Image Classification Model using Convolutional Neural Networks in PyTorch*. Web Artcicle. 2019. URL: https://www.analyticsvidhya.com/blog/2019/10/building-image-classification-models-cnn-pytorch/.

This page was intentionally left blank.

Acknowledgments

I wish to record a deep sense of gratitude to **Prof. Rakesh Nigam**, my supervisor for his valuable guidance and constant support at all stages of my study. I wish to thank my parents for their constant support, and my friends and classmates who though miles apart helped a lot. None of this would be possible without the almighty, I thank god for giving me faith and strength to move forward in the times I nedded it most.