BS6207 Homework 4

Isaac Lin

All my input images are 128 * 128 pixels, RGB 3 channels.

Originally, my folder has:

artifacts	2380 images
cancer_regions	2878 images
normal_regions	1187 images
others	484 images

I split the images into train and test folders. I took the first 100 images from each classes and store them inside a new test folder.

Now my train folder has:

artifacts	2280 images
cancer_regions	2778 images
normal_regions	1087 images
others	384 images

My test folder has:

artifacts	100 images
cancer_regions	100 images
normal_regions	100 images
others	100 images

I split my train folder into 70% training data and 30% validation data. Images and labels are randomized and not picked in any order.

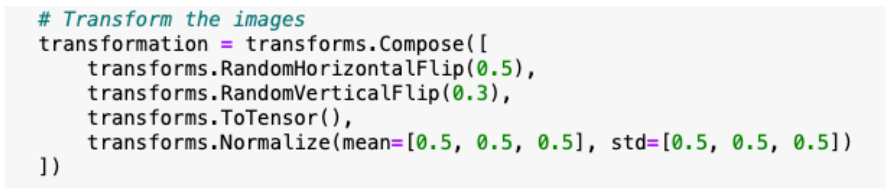
Here are my training and validation splits:

Training	4570 images
Validation	1959 images

Next, I need to prepare and augment the data.

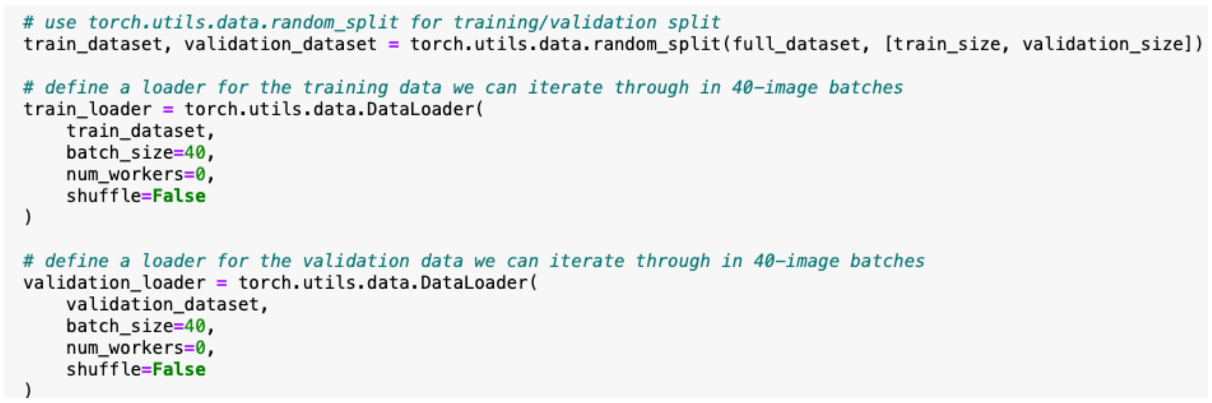
PyTorch includes functions for loading and transforming data. I will use these to create an iterative loader for training data, and a second iterative loader for test data. The loaders will transform the image data into tensors normalize them so that the pixel values are in a scale with a mean of 0.5 and a standard deviation of 0.5.

At this point I add transformations to randomly modify the images as they are added to a training batch. Images are flipped horizontally at random. This is done to prevent overfitting. The basic transformations are:



```
# Transform the images
transformation = transforms.Compose([
    transforms.RandomHorizontalFlip(0.5),
    transforms.RandomVerticalFlip(0.3),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
])
```

After training/validation split, the loaders will iterate images through 40-image batches.

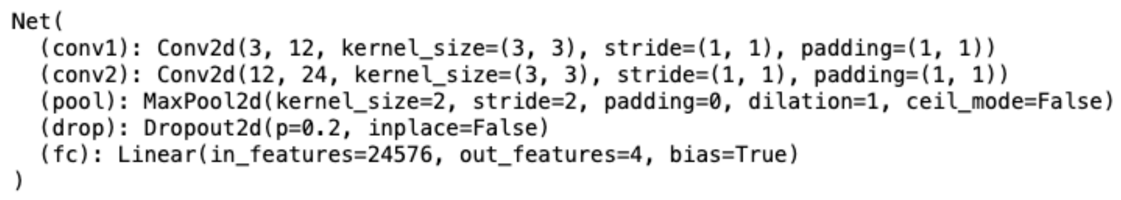


```
# use torch.utils.data.random_split for training/validation split
train_dataset, validation_dataset = torch.utils.data.random_split(full_dataset, [train_size, validation_size])

# define a loader for the training data we can iterate through in 40-image batches
train_loader = torch.utils.data.DataLoader(
    train_dataset,
    batch_size=40,
    num_workers=0,
    shuffle=False
)

# define a loader for the validation data we can iterate through in 40-image batches
validation_loader = torch.utils.data.DataLoader(
    validation_dataset,
    batch_size=40,
    num_workers=0,
    shuffle=False
)
```

Here is the network architecture for my CNN model.

1	`self.conv1 = nn.Conv2d(in_channels=3, out_channels=12, kernel_size=3, stride=1, padding=1)`
2	`self.pool = nn.MaxPool2d(kernel_size=2)`
3	`F.relu(self.pool(self.conv1(x)))`
4	`self.conv2 = nn.Conv2d(in_channels=12, out_channels=24, kernel_size=3, stride=1, padding=1)`
5	`self.pool = nn.MaxPool2d(kernel_size=2)`
6	`F.relu(self.pool(self.conv2(x)))`
7	`self.drop = nn.Dropout2d(p=0.2)`
8	`F.dropout(self.drop(x), training=self.training)`
9	`self.fc = nn.Linear(in_features=32 * 32 * 24, out_features=num_classes)`

Here is the initialization of my model.



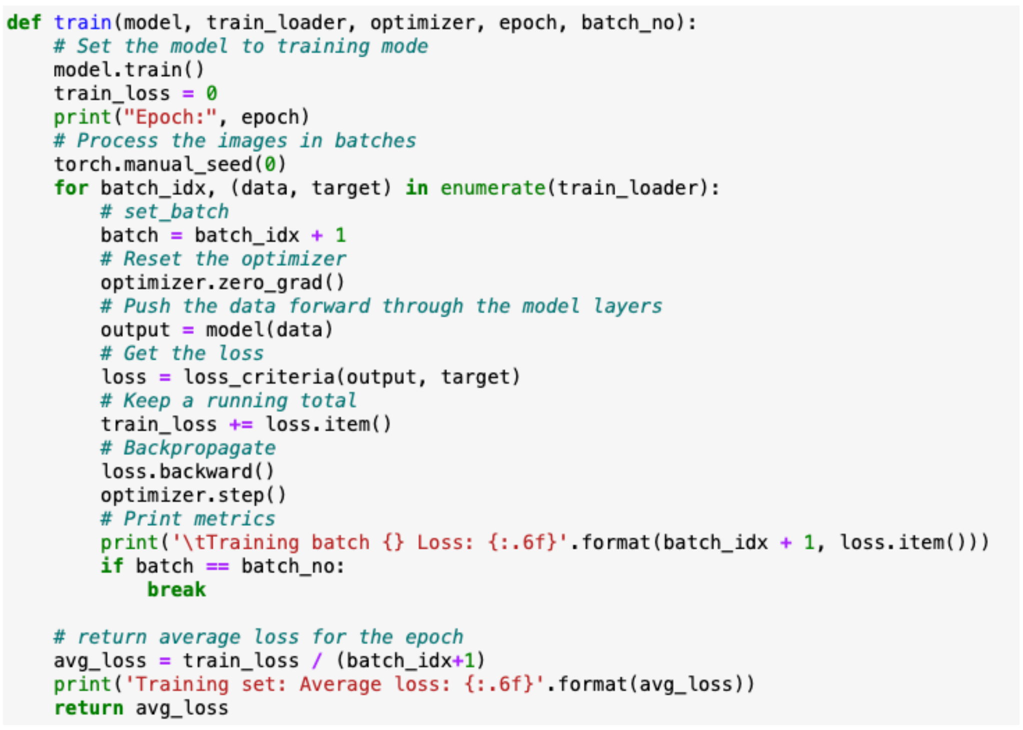
```
Net(
  (conv1): Conv2d(3, 12, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(12, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (drop): Dropout2d(p=0.2, inplace=False)
  (fc): Linear(in_features=24576, out_features=4, bias=True)
)
```

128 * 128 RGB images will be convolved with a kernel size of 3, stride 1 and padding 1. It is then pooled with kernel size of 2 and then going through Relu activation function. For the first convolution, 3 channels in and 12 channels out. The second convolution is done with the same settings. 12 channels in and 24 channels out. Images are pooled twice, thus 128 / 2 / 2 = 32. This means that the resulting tensors are now 32 * 32, and we have generated 24 of them.

I implemented a dropout layer that randomly drops 20% of the features, to prevent overfitting. Finally, the tensors are passed to the flatten layer with the number of features as 32 * 32 * 24 = 24,576.

The training function consists of an iterative series of forward passes in which the training data is processed in batches by the layers in the network, and the optimizer goes back and adjusts the weights. I will also use a separate set of validation images to validate the model at the end of each epoch, so that I can track the performance improvement as the training process progresses. The training function is as follows:

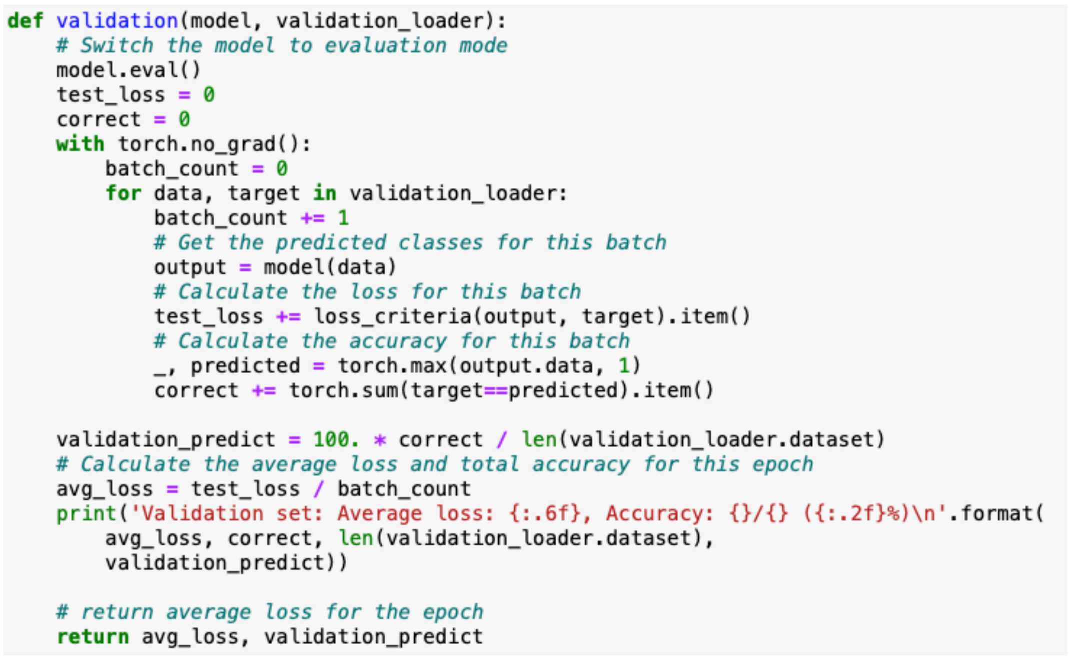
1. Set the model to training mode
2. Set the seed to 0
3. Process the images in batches. Inside each batch, I have to
   A. Import labels and features
   B. Reset the optimizer
   C. Push the data forward through the layers of the model
   D. Compute the loss
   E. Backpropagate
4. Compute the Average Loss of the Model during the Epoch



```python
def train(model, train_loader, optimizer, epoch, batch_no):
    # Set the model to training mode
    model.train()
    train_loss = 0
    print("Epoch:", epoch)
    # Process the images in batches
    torch.manual_seed(0)
    for batch_idx, (data, target) in enumerate(train_loader):
        # set_batch
        batch = batch_idx + 1
        # Reset the optimizer
        optimizer.zero_grad()
        # Push the data forward through the model layers
        output = model(data)
        # Get the loss
        loss = loss_criteria(output, target)
        # Keep a running total
        train_loss += loss.item()
        # Backpropagate
        loss.backward()
        optimizer.step()
        # Print metrics
        print('\tTraining batch {} Loss: {:.6f}'.format(batch_idx + 1, loss.item()))
        if batch == batch_no:
            break

    # return average loss for the epoch
    avg_loss = train_loss / (batch_idx+1)
    print('Training set: Average loss: {:.6f}'.format(avg_loss))
    return avg_loss
```

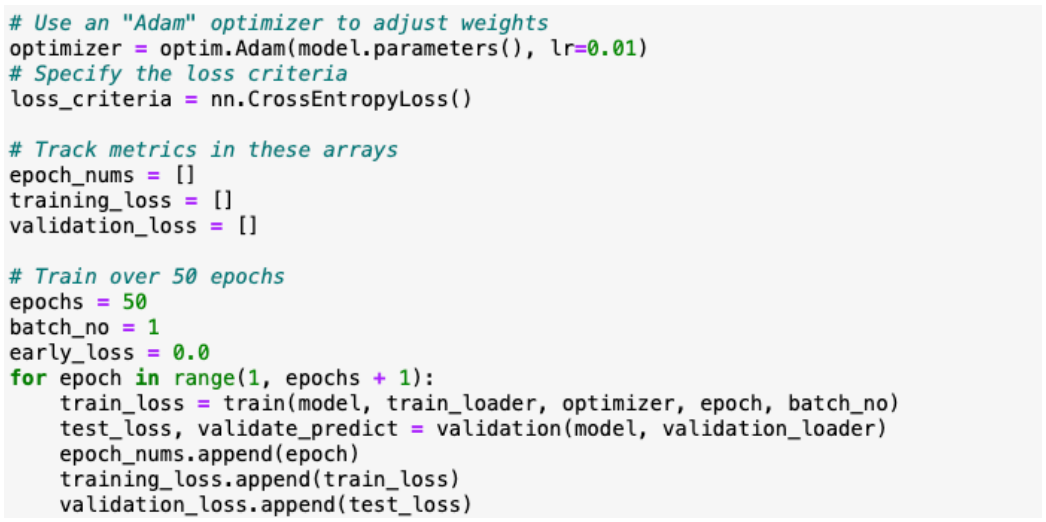
For the validation function, I set the model in evaluation mode to get the accuracy with respect to the labels. The validation function is as follows:

1. Set the model to evaluation mode
2. Process the images in batches. Inside each batch, I have to
    A. Get the prediction for each image in the batch
    B. Calculate the loss for the batch
    C. Calculate the accuracy for this batch
3. Calculate the average accuracy and loss for the epoch

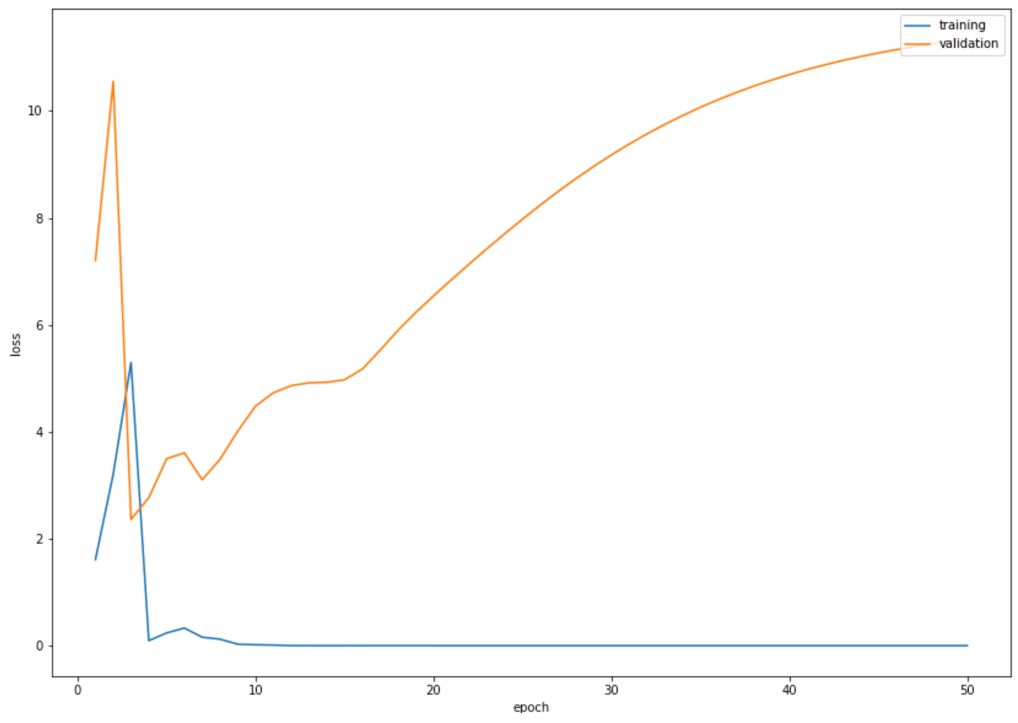


```python
def validation(model, validation_loader):
    # Switch the model to evaluation mode
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        batch_count = 0
        for data, target in validation_loader:
            batch_count += 1
            # Get the predicted classes for this batch
            output = model(data)
            # Calculate the loss for this batch
            test_loss += loss_criteria(output, target).item()
            # Calculate the accuracy for this batch
            _, predicted = torch.max(output.data, 1)
            correct += torch.sum(target==predicted).item()

    validation_predict = 100. * correct / len(validation_loader.dataset)
    # Calculate the average loss and total accuracy for this epoch
    avg_loss = test_loss / batch_count
    print('Validation set: Average loss: {:.6f}, Accuracy: {}/{} ({:.2f}%)\n'.format(
        avg_loss, correct, len(validation_loader.dataset),
        validation_predict))

    # return average loss for the epoch
    return avg_loss, validation_predict
```

I test the model using learning rate = 0.01, number of epochs = 50. I did a test run training over 1 batch (40 samples), here is my training and validation results. I use Adam as the optimizer, and Cross Entropy Loss as my loss criteria.

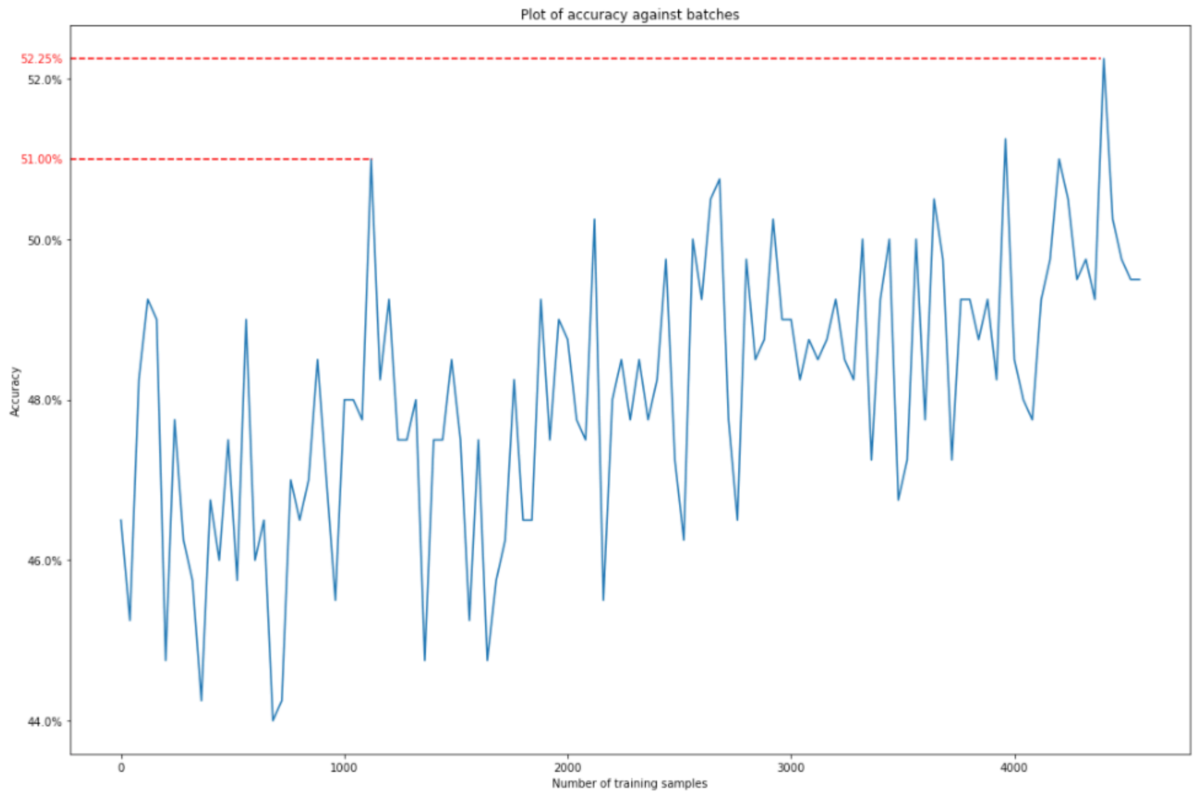


```python
# Use an "Adam" optimizer to adjust weights
optimizer = optim.Adam(model.parameters(), lr=0.01)
# Specify the loss criteria
loss_criteria = nn.CrossEntropyLoss()

# Track metrics in these arrays
epoch_nums = []
training_loss = []
validation_loss = []

# Train over 50 epochs
epochs = 50
batch_no = 1
early_loss = 0.0
for epoch in range(1, epochs + 1):
    train_loss = train(model, train_loader, optimizer, epoch, batch_no)
    test_loss, validate_predict = validation(model, validation_loader)
    epoch_nums.append(epoch)
    training_loss.append(train_loss)
    validation_loss.append(test_loss)
```

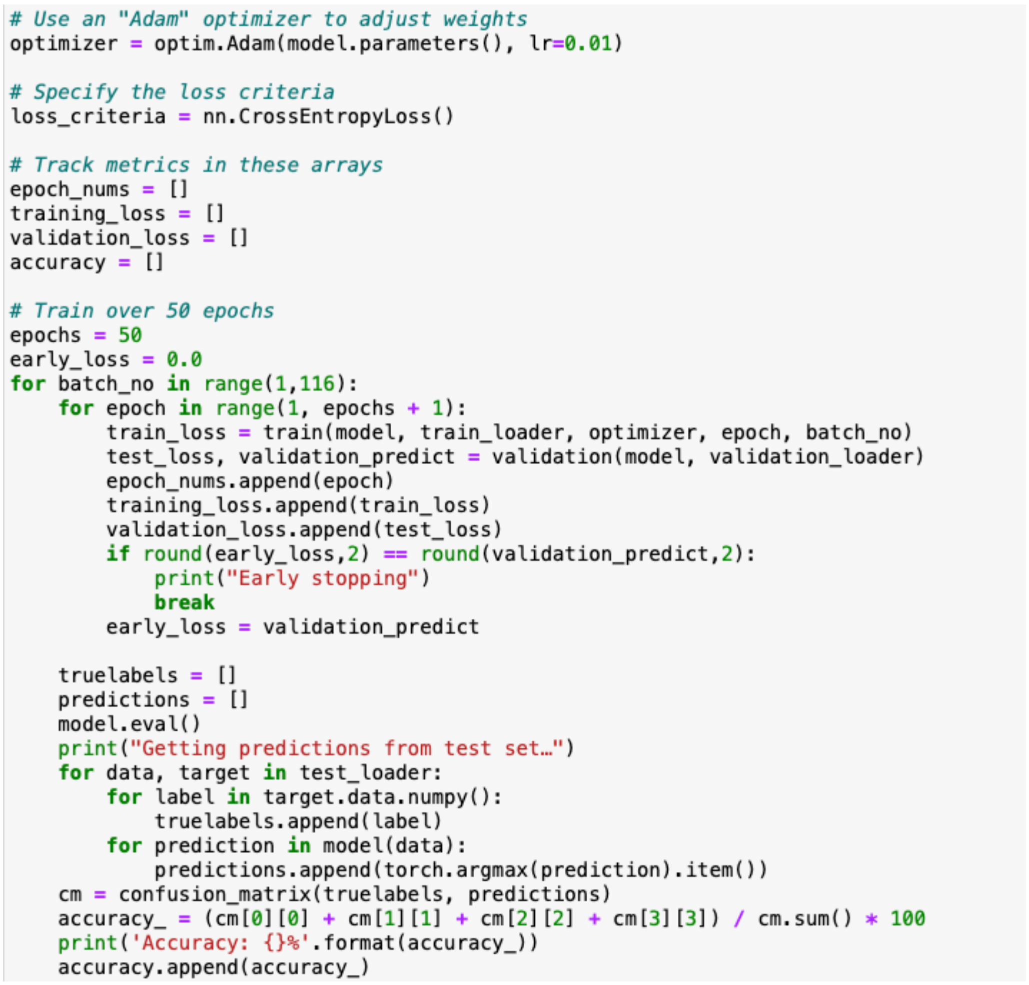


As you observe, the training loss converges to 0 at about epochs < 10, therefore I can implement early stopping here. As the training data overfits, the validation loss will be high, which is not desirable.

I prepare and augment the test dataset, and did a prediction over the test dataset, at sample intervals starting from 40, 80, 120, 160, … , 4570. Batch 1 starts from 40, and the last batch 115 ends at 4570. Here are my results.

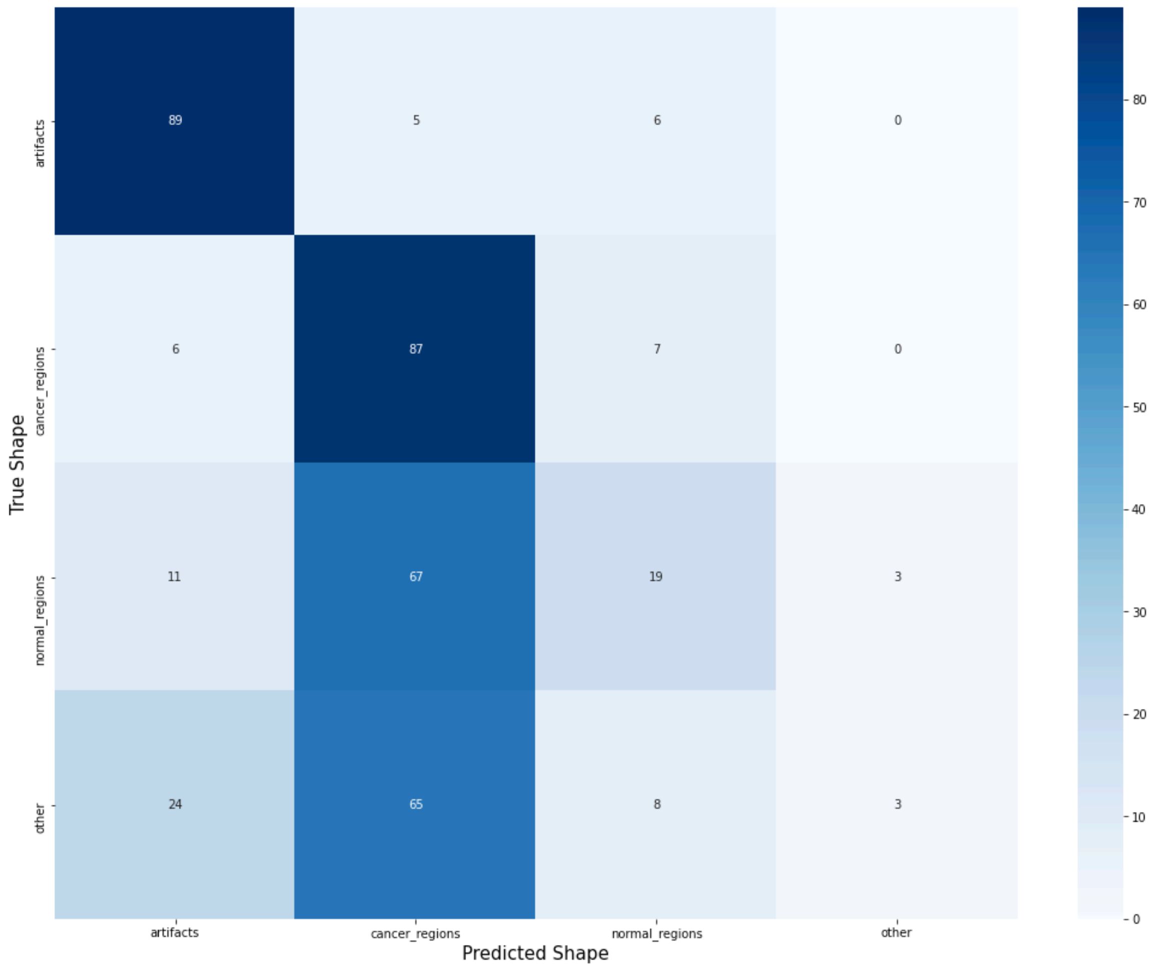


The least number of training samples to hit a 51.0% accuracy is 1120. Increasing the number of training samples to 4400, the accuracy only improves marginally to 52.25%. Thus 1120 is the optimal least number of training samples to get a good accuracy for my model. Below shows the code to plot my accuracy against number of training samples graph. Here, I implemented early stopping. If validation accuracy is consistent and the same across two consecutive epochs, then the epoch breaks, and the batch training ends.



```python
# Use an "Adam" optimizer to adjust weights
optimizer = optim.Adam(model.parameters(), lr=0.01)

# Specify the loss criteria
loss_criteria = nn.CrossEntropyLoss()

# Track metrics in these arrays
epoch_nums = []
training_loss = []
validation_loss = []
accuracy = []

# Train over 50 epochs
epochs = 50
early_loss = 0.0
for batch_no in range(1,116):
    for epoch in range(1, epochs + 1):
        train_loss = train(model, train_loader, optimizer, epoch, batch_no)
        test_loss, validation_predict = validation(model, validation_loader)
        epoch_nums.append(epoch)
        training_loss.append(train_loss)
        validation_loss.append(test_loss)
        if round(early_loss,2) == round(validation_predict,2):
            print("Early stopping")
            break
        early_loss = validation_predict

    truelabels = []
    predictions = []
    model.eval()
    print("Getting predictions from test set…")
    for data, target in test_loader:
        for label in target.data.numpy():
            truelabels.append(label)
        for prediction in model(data):
            predictions.append(torch.argmax(prediction).item())
    cm = confusion_matrix(truelabels, predictions)
    accuracy_ = (cm[0][0] + cm[1][1] + cm[2][2] + cm[3][3]) / cm.sum() * 100
    print('Accuracy: {}%'.format(accuracy_))
    accuracy.append(accuracy_)
```

Here shows the confusion matrix after running all 115 batches on 50 epochs with early stopping.



artifacts and cancer_regions classes have very high accuracy of prediction. A lot of the samples in normal_regions and other are misclassified as cancer_regions. This is probably due to class imbalance. The model trains over many artifacts and cancer_regions samples, thus the overall prediction accuracy is not as high as expected.