# Final_project_cancerdata

May 31, 2022

## 1 Assignment 4

### 1.1 Handin

**1. A short abstract describing the project in your own words. This should be 200 words or less.**

I decided to tackle the "Cancer Identification" task, which entails classifying tiles from large pathology slide scans. Really, the task should be to classify the *slide*, not the tiles, but that would require a little more coding than I think is really useful for this particular exercise. This is largely because of the way the training and testing data are split (more on this below). Anyway, I decided to train a classification network, taking the tiles as input and outputting a classification tensor of length 4.

**2. A summary of your work. Include your methods, experiments, and results.**

My approach was to first perform an H&E normalization on the data (more on the reasoning below), then do a channel normalization as per the tutorials from the previous assignment, then train the network. I first tried resnet18, but I was only really able to get around 66% accuracy with that network even after several rounds of training. So I tried a transformer-based network next (ViT_b_16), but this only got up to about 63% accuracy before the model appeared to start overtraining (training data performance was outperforming test data performance by ~10%).

Scoring both of my models on the held-out validation data provided scores of ~65% accuracy for resnet18 and ~60% for ViT_b_16.

**3. A brief reflection on what you learned.**

I think the choice of some hyperparameters can really affect the success of training a network model. For example, I found that even small tweaks to learning rate can affect how well a particular set of training epochs will perform. Also I think some domain knowledge can significantly improve ones ability to frame a particular ML task (see discussion of the big problems with this task below). I found adapting pre-built networks to be straightforward, and end-to-end training on my GPU at home was not terribly painful. The full training of my final resnet18 model only took around 5 hours (broken up).

I also initially misunderstood how StepLR worked, and was probably slowing down the training way too early for end-to-end training.

**4. Any acknowledgements or citations of material you found helpful.**

I specifically call out references in the code below. Much of the code has been adapted from the tutorials covered in the last assignment. Code for calculating a running mean and std came from Googling (reference is in comments). H&E normalization code came from https://github.com/schaugf/HEnorm_python, which was implementing an algorithm described in Macenko, et al., 2009.

## 1.2 Problems with the Cancer Identification task

The primary underlying goal of the cancer identification task is to be able to identify cancer types from images. However the typical medical pathology image is a very large, high resolution slide scan that is going to be too large to be practically trained using off-the-shelf network models. Hence for this task, the large slides have been broken up into small tiles, which were used for training. Presumably, one could could apply a model trained on tiles to tiles from a whole slide, then use a voting scheme to determine the cancer type on the slide (e.g., plurality of tiles classified). However, breaking these slides up into tiles creates a few problems with respect to the training itself.

**Problem 1**: Classifying single tiles eliminates the possibility of considering larger context in the classification. This can be problematic for pathology slides, since cancerous tissue may only represent a fraction of the total tissue on the slide, and thus many tiles will not actually contain information relevant to the cancer type. Moreover, even cancerous tissue itself can be rather heterogeneous, and differences between cancer types may not be manifested at the scale that can be captured with small tiles. This means training will be very *noisy*, and relevant biological information will be missed.

**Problem 2**: The bigger problem with this task as set up for the class is that there is significant test-set leakage in the data. Tiles within a cancer type were randomly assigned to the train, test, and validation datasets, which means tiles originating from a given slide could be found across the three datasets. This means that biases that can be attributed to the slide or sample *but not the cancer* could be learned by the model and exploited to give correct classifications. For example, imagine that you have a slide overstained with hematoxylin (the "H" in H&E), and that slide contains data for cancer X. Even after RGB normalization, the ratio of H/E staining will still be skewed with respect to other slides. The network could learn this skew, and assign tiles to the correct cancer (X), simply because the network knows that cancers Y and Z do not have any overstained slides.

I also noted that there appears to be different types of tissues in different tiles. This suggests to me that there may be biases in terms of tissue representation across the datasets. If cancer X has significantly more e.g., muscle tissue that other slides, the network would only need to learn what muscle tissue looks like to improve accuract of assignment to cancer X.

**How I would overcome these problems**: Several approaches could be taken to overcome the challenges I listed above.

1. First, it is very important to not include data from one slide in both your training and testing/validation datasets. Slides should be assigned to a dataset and split within that dataset. This will eliminate some test-set leakage.
2. Second, H&E data should be normalized to mitigate staining differences between slides. The approach described in Macenko et al., 2009 seems to work well and is computationally efficient.
3. Third, tissue/cell types should be well balanced across your training and test/validation sets so that the model doesn't just learn that e.g., muscle tissue will tend to be from cancer X vs. cancer y.
4. Finally, it would probably be a good idea to include healthy tissue as a 5th category, which would further reduce the possibilty of learning the wrong information based on slide biases.

Ultimately, a network architecture that can more efficiently analyze a larger context than just a single tile would also help with some of the problems in this task. For example, each tile could be concatenated with scaled-down representations of surrounding tissue, and this fed to either a CNN or transformer-based network so that multiscale information could be included in the embedding.

## 1.3 Code

[1]:
```
import copy
import PIL
import time
import torch

import matplotlib.pyplot as plt
import numpy as np
import torch.optim as optim
import torchvision.models as models

from pathlib import Path
from skimage.color import rgb2gray
from skimage.util import invert
from torch import FloatTensor
from torch import nn
from torch.nn import Conv2d, Linear
from torch.optim import lr_scheduler
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision.datasets import ImageFolder
from torchvision.utils import make_grid

# normalizeStaining comes from https://github.com/schaugf/HEnorm_python
# This uses a H&E normalization strategy described in:
#  M Macenko, M Niethammer, JS Marron, D Borland, JT Woosley, G Xiaojun,
#   C Schmitt, NE Thomas, IEEE ISBI, 2009.
from normalizeStaining import normalizeStaining

# data directory
root = Path("E:\data512_2k")

# setup device
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
```

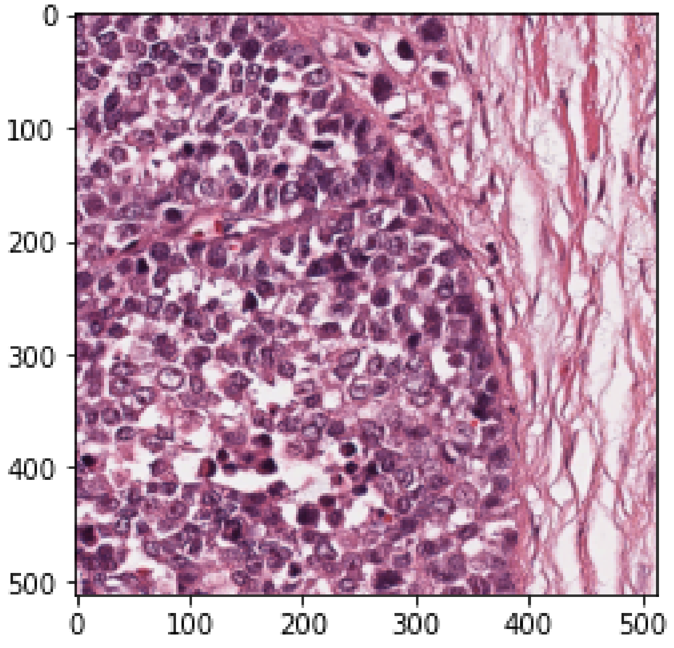
```
cuda:0
```

### 1.3.1 Stain normalization

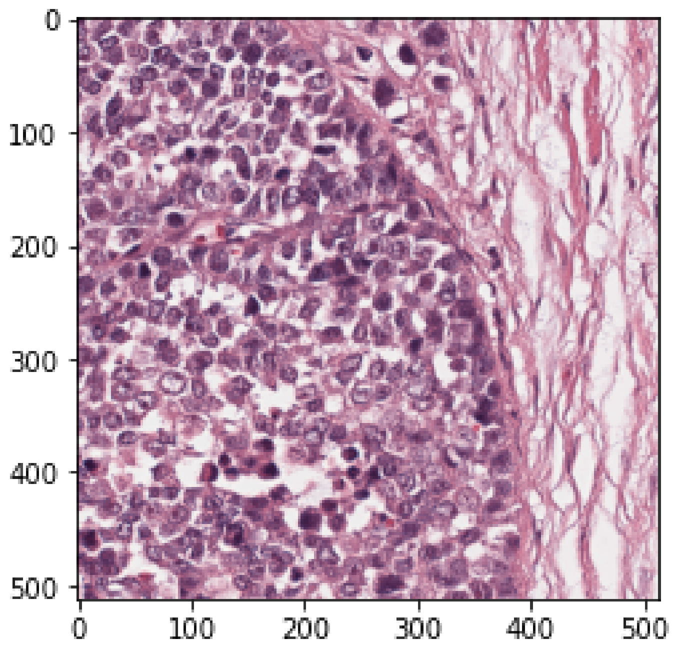
The next few cells show how I was exploring using the stain normalization tool from https://github.com/schaugf/HEnorm_python. One of the neat advantages of this tool is that it reshapes the RGB data into two channels, one each for the hematoxylin and eosin stains. I figured using two channels to represent the data would be a little more efficient and possibly provide better performance for the models.

```
[2]: # Test out normalizeStaining
     impath = list(root.glob('*/*/*.jpg'))[24]
     with PIL.Image.open(impath) as im:
         print(im.mode)
         im = np.asarray(im)
     plt.imshow(im);
```

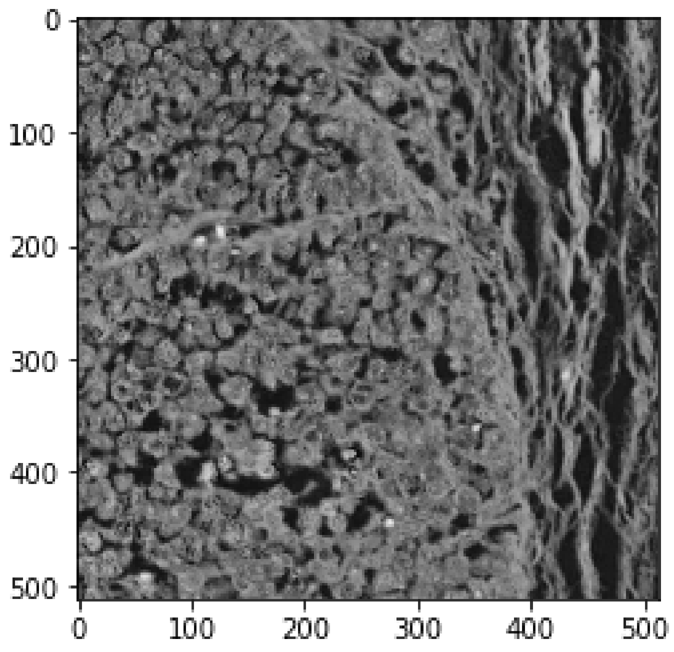
```
RGB
```



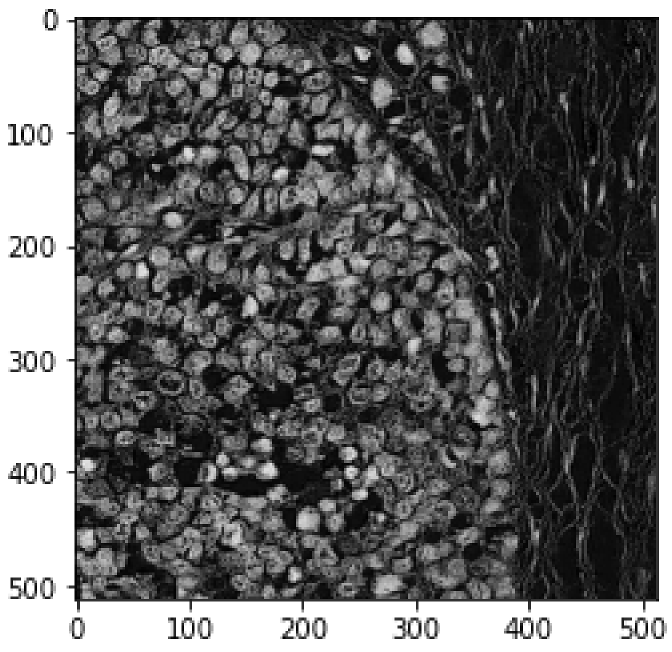
```
[3]: im_norm, h, e = normalizeStaining(im)
     plt.imshow(im_norm);
```



[4]:
```
plt.imshow(invert(rgb2gray(e)), cmap='gray');
```



[5]:
```
plt.imshow(invert(rgb2gray(h)), cmap='gray');
```



I was pleased with the performance of the stain normalization tool, so I created a function and a pytorch Transform class to use. The transform class uses a cheap hack to circumvent a problem I had where some crops of some of the tiles didn't have enough information to solve the stain separation (e.g., were blank, horribly out of focus, etc.). I figured feeding random data to the network in a few cases would not have a huge impact on training, but would save me a lot of troubleshooting.

[6]:
```
class ExtractStains:
    '''Normalize H&E staining and assign H&E staining to channels'''

    def __call__(self, x):
        '''x is a RGB PIL image, returns a Tensor'''
        outarr = np.zeros((2, *x.size))
        im = np.asarray(x)
        try:
            _, h, e = normalizeStaining(im)
            outarr[0, ...] = invert(rgb2gray(h))
            outarr[1, ...] = invert(rgb2gray(e))
        except:
            # This hack saved me a ton of time
            print(f'Failed to normalize image, using random data')
            outarr = np.random.rand(2, *x.size)

        return FloatTensor(outarr)

def extract_stains(pil_im):
    outarr = np.zeros((*pil_im.size, 2))
    im = np.asarray(pil_im)
    _, h, e = normalizeStaining(im)
    outarr[..., 0] = invert(rgb2gray(h))
    outarr[..., 1] = invert(rgb2gray(e))
    return outarr
```

The next cell was run once to calculate average H&E values for normalization purposes. It also caught some tiles that threw exceptions with the stain normalization tool, which I manually eliminated from the datasets.

I used the following code to calculate average H&E values, and identify "blank" tiles that remained in the dataset blank tiles were subsequently eliminated from the dataset adapted from: https://www.geeksforgeeks.org/expression-for-mean-and-variance-in-a-running-stream/

```
n = 0
running_sum = np.zeros(2)
running_sumsq = np.zeros(2)
for impath in root.glob('*/*/*.jpg'):
    n += 1
    with PIL.Image.open(impath) as im:
        try:
            arr = extract_stains(im)
        except:
            print(f'Check image at {impath}')
    x = arr.mean(axis=(0,1))
    running_sum += x
    running_sumsq += (x**2)
    running_mean = running_sum/n
    running_var = (running_sumsq/n) - (running_mean**2)
stds = np.sqrt(running_var)
print(running_mean)
print(stds)
```

**means = 0.21757233, 0.25600197**

**stds = 0.06741126, 0.05294553**

## 1.4 Dataloaders

In the next few cells, I composed my transforms and set up my dataloaders

[7]:
```
means = np.array([0.21757233, 0.25600197])
stds = np.array([0.06741126, 0.05294553])

data_transform_train = transforms.Compose([
        transforms.Resize(256),
        transforms.RandomResizedCrop(192),
        transforms.RandomHorizontalFlip(),
        ExtractStains(),
        transforms.Normalize(mean=means, std=stds)
    ])

data_transform_test = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(192),
        ExtractStains(),
        transforms.Normalize(mean=means, std=stds)
    ])
```

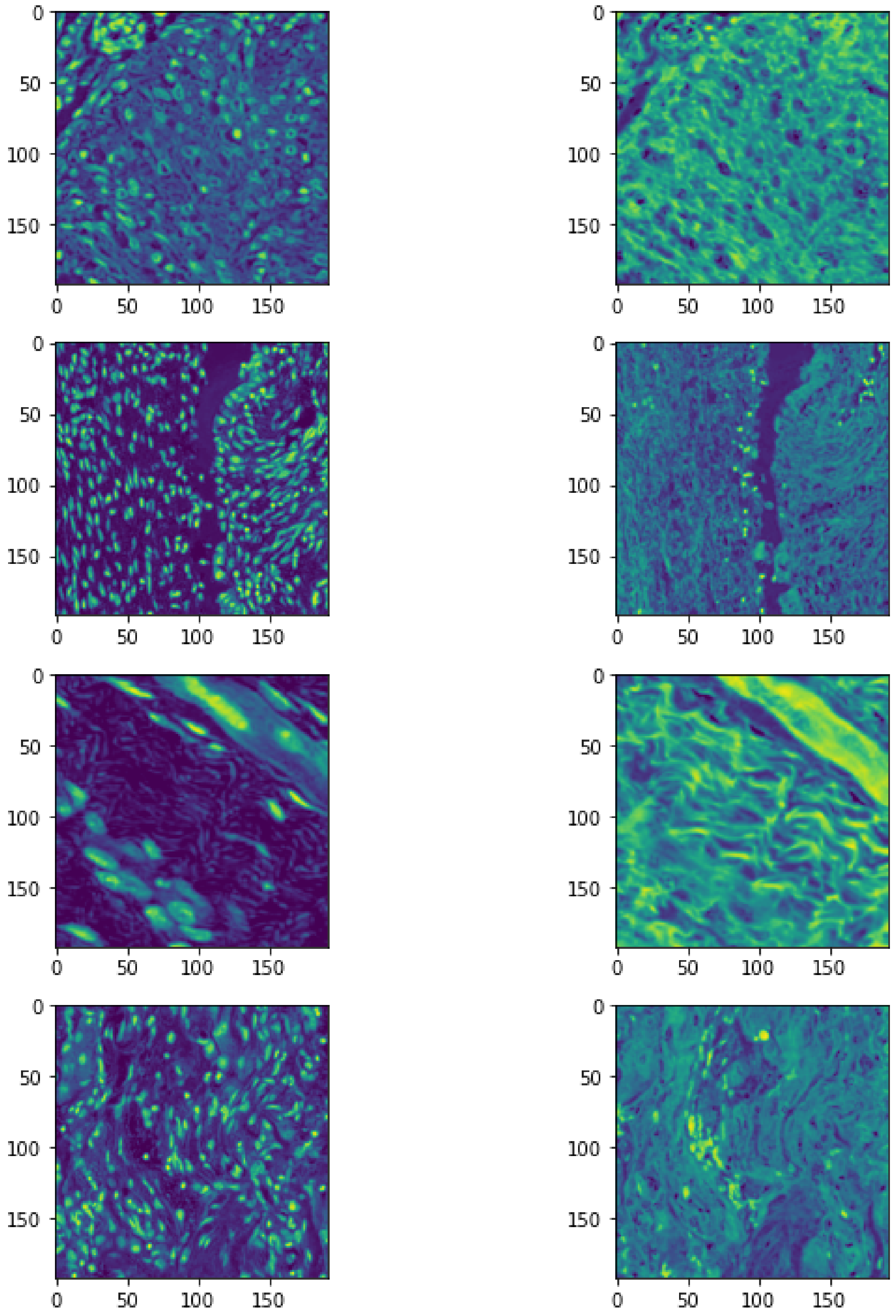
[8]:
```
#datasets
data_test = ImageFolder(root / "test", transform=data_transform_test)
data_train = ImageFolder(root / "train", transform=data_transform_train)
data_valid = ImageFolder(root / "valid", transform=data_transform_test)

#dataloaders
dl_test = DataLoader(data_test, batch_size=4, shuffle=True)
dl_train = DataLoader(data_train, batch_size=4, shuffle=True)
dl_valid = DataLoader(data_valid, batch_size=4, shuffle=True)
```

Here, I examine the performance of my dataloaders and see that they are working as intended:

[9]:
```
ims, _ = next(iter(dl_train))
print(ims.shape)
fig, axs = plt.subplots(ims.shape[0],2, figsize=(10,10), layout='tight')
for i, ax in enumerate(axs):
    for j, axx in enumerate(ax):
        axx.imshow(ims[i, j, :, :])
```

```
torch.Size([4, 2, 192, 192])
```



### 1.5 resnet18

Here I am modifying resnet18 to take in a two channel image, and output only 4 possible classes:

[10]:
```
# load resnet with random weights and change first and last layers to␣
 ↪accomodate these data

model = models.resnet18()
model.conv1 = Conv2d(2, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),␣
 ↪bias=False)
model.fc = Linear(in_features=512, out_features=4, bias=True)
model.to(device);
```

In the next couple of cells I set up training as per the previous assignment, using code modified from the tutorials:

[11]:
```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
exp_lr_scheduler = lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)
```

[12]:
```
# From the last project:
def train_model(device, dataloaders, dataset_sizes, model, criterion,␣
 ↪optimizer, scheduler, num_epochs=25):
    since = time.time()

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

    for epoch in range(num_epochs):
        print(f'Epoch {epoch}/{num_epochs - 1}')
        print('-' * 10)

        # Each epoch has a training and validation phase
        for phase in ['train', 'test']:
            if phase == 'train':
                model.train()  # Set model to training mode
            else:
                model.eval()   # Set model to evaluate mode

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data.
            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device)
                # zero the parameter gradients
                optimizer.zero_grad()

                # forward
                # track history if only in train
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    # backward + optimize only if in training phase
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                # statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)
            if phase == 'train':
                scheduler.step()

            epoch_loss = running_loss / dataset_sizes[phase]
            epoch_acc = running_corrects.double() / dataset_sizes[phase]

            print(f'{phase} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}')

            # deep copy the model
            if phase == 'test' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())

        print()

    time_elapsed = time.time() - since
    print(f'Training complete in {time_elapsed // 60:.0f}m {time_elapsed % 60:.
↪0f}s')
    print(f'Best val Acc: {best_acc:4f}')

    # load best model weights
    model.load_state_dict(best_model_wts)
    return model
```

First attempt at training:

[ ]:
```
dataloaders = {}
dataloaders['train'] = dl_train
```

```
dataloaders['test'] = dl_test

dataset_sizes = {}
dataset_sizes['train'] = len(data_train)
dataset_sizes['test'] = len(data_test)

trained_model = train_model(device, dataloaders, dataset_sizes, model,␣
 ↪criterion, optimizer, exp_lr_scheduler)
```

```
[ ]: torch.save(trained_model, 'attempt1_resnet18')
```

Second attempt at training, but using a somewhat more aggressive LR scheduler (larger step size but faster decay). I didn't have a good reason for this, it was just something I wanted to experiment with. I also increased batch size in the dataloaders.

```
[ ]: #dataloaders
dl_test = DataLoader(data_test, batch_size=12, shuffle=True)
dl_train = DataLoader(data_train, batch_size=12, shuffle=True)
dl_valid = DataLoader(data_valid, batch_size=12, shuffle=True)
dataloaders = {}
dataloaders['train'] = dl_train
dataloaders['test'] = dl_test

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
exp_lr_scheduler = lr_scheduler.StepLR(optimizer, step_size=15, gamma=0.05)

# Run model again to see if there is any improvement
trained_model2 = train_model(device, dataloaders, dataset_sizes, model,␣
 ↪criterion, optimizer, exp_lr_scheduler, num_epochs=50)
torch.save(trained_model2, 'attempt2_resnet18')
```

I tried one more time with resnet18, this time using best model from last run

```
[ ]: optimizer = optim.SGD(trained_model2.parameters(), lr=0.001, momentum=0.9)
exp_lr_scheduler = lr_scheduler.StepLR(optimizer, step_size=15, gamma=0.05)
trained_model3 = train_model(device, dataloaders, dataset_sizes,␣
 ↪trained_model2, criterion, optimizer, exp_lr_scheduler, num_epochs=50)
torch.save(trained_model3, 'attempt3_resnet18')
```

### 1.5.1 resnet18 final performance

Best accuracy on test data: 0.667500

## 1.6 VisionTransformer

I next tried a transformer-based network, choosing mostly default hyperparameters for optimization. Again, this was modified to take a two-channel image and output four different classes.

I saw a few examples of using Adam instead of SGD for ViT, so I tried that first. But that seemed to cause convergence problems, so I switched back to SGD.

```
[14]: model = models.vit_b_16()
model.conv_proj = Conv2d(2, 768, kernel_size=(16, 16), stride=(16, 16))
model.heads.head = Linear(in_features=768, out_features=4, bias=True)
model.to(device);
```

```
[ ]: data_transform_train = transforms.Compose([
        transforms.Resize(256),
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        ExtractStains(),
        transforms.Normalize(mean=means, std=stds)
    ])

data_transform_test = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        ExtractStains(),
        transforms.Normalize(mean=means, std=stds)
    ])

#datasets
data_test = ImageFolder(root / "test", transform=data_transform_test)
data_train = ImageFolder(root / "train", transform=data_transform_train)
data_valid = ImageFolder(root / "valid", transform=data_transform_test)

#dataloaders
dl_test = DataLoader(data_test, batch_size=10, shuffle=True)
dl_train = DataLoader(data_train, batch_size=10, shuffle=True)
dl_valid = DataLoader(data_valid, batch_size=10, shuffle=True)
dataloaders = {}
dataloaders['train'] = dl_train
dataloaders['test'] = dl_test

criterion = nn.CrossEntropyLoss()

optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
exp_lr_scheduler = lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.25)

# Run model again to see if there is any improvement
vit_model = train_model(device, dataloaders, dataset_sizes, model, criterion,␣
 ↪optimizer, exp_lr_scheduler, num_epochs=10)
torch.save(vit_model, 'attempt1_vit')
```

10 epochs took about an hour, so I will try 35 more (with less decay in the LR).

```
[ ]: optimizer = optim.SGD(vit_model.parameters(), lr=0.001, momentum=0.9)
     exp_lr_scheduler = lr_scheduler.StepLR(optimizer, step_size=25, gamma=0.25)

     # Run model again to see if there is any improvement
     vit_model2 = train_model(device, dataloaders, dataset_sizes, vit_model,␣
      ↪criterion, optimizer, exp_lr_scheduler, num_epochs=35)
     torch.save(vit_model2, 'attempt2_vit')
```

And one more training session:

```
[ ]: optimizer = optim.SGD(vit_model2.parameters(), lr=0.001, momentum=0.9)
     exp_lr_scheduler = lr_scheduler.StepLR(optimizer, step_size=25, gamma=0.25)

     # Run model again to see if there is any improvement
     vit_model3 = train_model(device, dataloaders, dataset_sizes, vit_model2,␣
      ↪criterion, optimizer, exp_lr_scheduler, num_epochs=35)
     torch.save(vit_model3, 'attempt3_vit')
```

One last training session for the ViT model:

```
[ ]: optimizer = optim.SGD(vit_model3.parameters(), lr=0.001, momentum=0.9)
     exp_lr_scheduler = lr_scheduler.StepLR(optimizer, step_size=25, gamma=0.25)

     # Run model again to see if there is any improvement
     vit_model4 = train_model(device, dataloaders, dataset_sizes, vit_model3,␣
      ↪criterion, optimizer, exp_lr_scheduler, num_epochs=35)
     torch.save(vit_model4, 'attempt4_vit')
```

At this point, it looked like the model was starting to overtrain, so:

### 1.6.1 ViT_b_16 Final performance

Best accuracy on test data: 0.6275

## 1.7 Scoring on validation data

```
[15]: resnet = torch.load('attempt3_resnet18')
      vit = torch.load('attempt3_vit')

      resnet.to(device);
      vit.to(device);
```

```
[16]: def test(device, dataloader, model, loss_fn):
          size = len(dataloader.dataset)
          num_batches = len(dataloader)
          model.eval()
          test_loss, correct = 0, 0
          with torch.no_grad():
```

```
        for X, y in dataloader:
            X, y = X.to(device), y.to(device)
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss:␣
↪{test_loss:>8f} \n")
```

### 1.7.1 Resnet18 validation

64.6% accuracy

[19]:
```
data_transform_test = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(192),
        ExtractStains(),
        transforms.Normalize(mean=means, std=stds)
    ])

data_valid = ImageFolder(root / "valid", transform=data_transform_test)
dl_valid = DataLoader(data_valid, batch_size=12, shuffle=True)

test(device, dl_valid, resnet, criterion);
```

```
Failed to normalize image, using random data
Test Error:
 Accuracy: 64.6%, Avg loss: 0.838670
```

### 1.7.2 ViT_b_16 validation

60.4% accuracy

[20]:
```
data_transform_test = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        ExtractStains(),
        transforms.Normalize(mean=means, std=stds)
    ])

data_valid = ImageFolder(root / "valid", transform=data_transform_test)
dl_valid = DataLoader(data_valid, batch_size=12, shuffle=True)

test(device, dl_valid, vit, criterion);
```

```
Failed to normalize image, using random data
Test Error:
 Accuracy: 60.4%, Avg loss: 0.943952
```

```
[ ]:
```