april9_sub

April 9, 2024

[ ]:
```
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import OneHotEncoder, StandardScaler
import pickle
from torch.utils.data import Dataset, DataLoader
from tqdm import tqdm
import os
from skimage import draw
import pandas as pd
```

1 Creating Dataset

1.1 Helper Functions

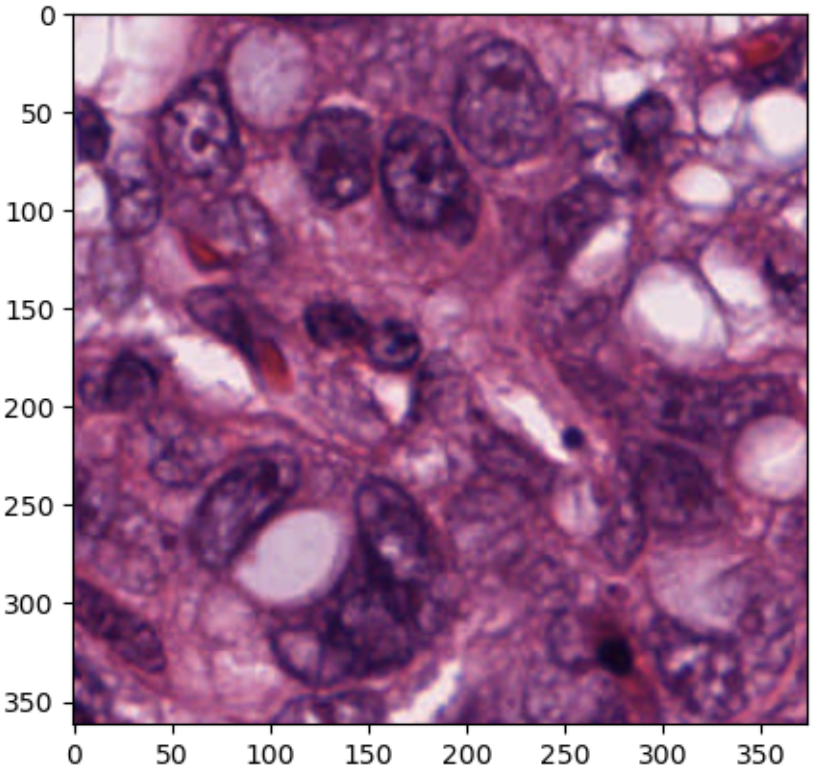
[ ]:
```
# Directories
csv_dir = "../../DATASET/csv"
im_dir = "../../DATASET/rgb"

# Get File Names
def get_files(dir):
    return [f for f in os.listdir(dir) if os.path.isfile(os.path.join(dir, f))]

# Get CSV Dataframes
def get_csv_df(dir, dir_paths, idx):
    path = os.path.join(dir, dir_paths[idx])
    return pd.read_csv(path)

# Get Image Data
def get_im_data(dir, dir_paths, idx):
    path = os.path.join(dir, dir_paths[idx])
    return plt.imread(path)

def crop_img(image, xmin, ymin, xmax, ymax):
    return image[ymin:ymax, xmin:xmax]
```

```
def get_xy(csv_df,idx):
    return csv_df[['coords_x', 'xmin', 'xmax', 'ymin', 'ymax', 'coords_y']].
 ↪iloc[idx].to_dict()

def cell_mask(img, x_cord, y_cord):
    if len(x_cord) != len(y_cord):
        raise ValueError("Length of x_cord and y_cord must be equal")

def get_nuc(img, csv, xshape, yshape, x_window = [70,100], y_window = [70,100]):
    num_nuc = len(csv)
    nucs_list = []
    for i in range(num_nuc):
        nuc = get_xy(csv, i)

        x_range =  nuc['xmax'] - nuc['xmin']
        y_range = nuc['ymax'] - nuc['ymin']
        if (x_range < x_window[0] or x_range > x_window[1]) and (y_range <␣
 ↪y_window[0] or y_range > y_window[1]):
            continue
        x =  np.array(nuc['coords_x'].split(",")).astype(int)
        y =  np.array(nuc['coords_y'].split(",")).astype(int)
        im_h, im_w, im_c = img.shape
        mask = np.zeros((im_h, im_w, im_c), dtype=np.uint8)
        rr, cc = draw.polygon(y, x)
        try:
            mask[rr, cc,:] = 1
        except IndexError:
            continue
        del_back = np.multiply(mask, img)
        cropped_nuc = crop_img(del_back, nuc['xmin'], nuc['ymin'], nuc['xmax'],␣
 ↪nuc['ymax'])
        blank_mask = np.zeros((xshape, yshape, 3))
        center_height = (blank_mask.shape[0] - cropped_nuc.shape[0]) // 2
        center_width = (blank_mask.shape[1] - cropped_nuc.shape[1]) // 2
        try:
            blank_mask[center_height:center_height + cropped_nuc.shape[0],␣
 ↪center_width:center_width + cropped_nuc.shape[1], :] = cropped_nuc
        except:
            continue
        nucs_list.append(blank_mask)
    return nucs_list
```

1.2 Get image

[ ]:
```
csv_files = get_files(csv_dir)
im_files = get_files(im_dir)
im_files = get_files(im_dir)
img_array = get_im_data(im_dir, im_files, 0)
print(img_array.shape)
plt.imshow(img_array)
plt.show()
```

```
(362, 374, 3)
```



1.3 Boundary Regions from CSV files

[ ]:
```
csv_df = get_csv_df(csv_dir, csv_files, 0)
csv_df
```

[ ]:

	Unnamed: 0	raw_classification	main_classification	super_classification	\
0	0	lymphocyte	lymphocyte	sTIL	
1	1	tumor	tumor_nonMitotic	tumor_any	
2	2	tumor	tumor_nonMitotic	tumor_any	
3	3	tumor	tumor_nonMitotic	tumor_any	

```
4           4     apoptotic_body            AMBIGUOUS            AMBIGUOUS
5           5         fibroblast  nonTILnonMQ_stromal       nonTIL_stromal
6           6         lymphocyte           lymphocyte                 sTIL
7           7          unlabeled            AMBIGUOUS            AMBIGUOUS
8           8          unlabeled            AMBIGUOUS            AMBIGUOUS

       type  xmin  ymin  xmax  ymax  \
0  polyline    29   106    55   135
1  polyline   104   225   152   280
2  polyline     0   278    72   317
3  polyline    48   237    99   296
4  polyline   100   104   138   136
5  polyline   100    47   135    88
6  polyline   127   147   155   180
7  polyline   163   285   190   322
8  polyline   258   289   280   338

                                              coords_x  \
0                         47,29,35,45,53,55,53,51,47
1  106,104,106,114,116,120,132,135,138,141,143,15…
2  0,0,5,8,20,27,46,51,55,69,71,72,71,67,64,56,37…
3  97,92,73,56,53,50,48,48,50,55,61,73,80,89,90,9…
4  138,137,132,126,124,101,100,100,104,113,125,13…
5  135,134,127,125,113,105,101,100,100,101,113,12…
6  148,145,138,131,129,127,127,132,140,145,150,15…
7  190,190,187,183,182,174,171,168,167,163,168,18…
8        280,277,267,261,258,258,261,263,269,272,280

                                              coords_y
0                133,106,106,109,117,123,131,135,133
1  269,259,251,228,226,225,225,226,228,231,236,25…
2  301,295,286,284,279,278,278,279,280,289,293,29…
3  278,296,293,280,278,272,265,263,257,241,237,23…
4  121,131,136,136,135,121,116,112,107,104,104,10…
5          84,88,88,86,77,68,62,59,51,47,51,64,77,84
6  178,180,179,175,168,162,157,152,147,148,153,15…
7  293,297,310,321,322,322,321,320,314,291,285,28…
8        297,338,332,317,309,300,293,289,289,290,297
```

1.4 Match Files with Corresponding Images

[ ]:
```
matching_files = []
for csv in csv_files:
    for im in im_files:
        csv_d = os.path.splitext(csv)[0]
        im_d = os.path.splitext(im)[0]
        if csv_d == im_d:
```

```
            matching_files.append((csv,im))
print("Number of matching files =", len(matching_files))
matching_files[0]
```

```
Number of matching files = 1337
```

[ ]:
```
('SP.3_#_E_#_TCGA-E2-A158-01Z-00-DX1_id-5e83b15bddda5f83987d58dd_left-57092_top-27151_bottom-27425_right-57362.csv',
 'SP.3_#_E_#_TCGA-E2-A158-01Z-00-DX1_id-5e83b15bddda5f83987d58dd_left-57092_top-27151_bottom-27425_right-57362.png')
```

1.5 Create Pickle Array

[ ]:
```
csv_dir = "../../DATASET/csv"
im_dir = "../../DATASET/rgb"
csv_files = get_files(csv_dir)
im_files = get_files(im_dir)
im_save_dir = "../../DATASET/nuc"
class_map = {'raw_classification':0, "main_classification":1,␣
 ↪'super_classification':2}
org_class = class_map['super_classification']
pickle_array = []

final_df = pd.DataFrame(columns=["file_name", 'raw_classification',␣
 ↪"main_classification",'super_classification'])
iters = 0
stop = 100
for i, files in enumerate(matching_files):
    csv_path = os.path.join(csv_dir, files[0])
    csv_df = pd.read_csv(csv_path)

    im_path = os.path.join(im_dir, files[1])
    img = plt.imread(im_path)

    try:
        nucs = get_nuc(img, csv_df, xshape = 80, yshape = 80, x_window =␣
 ↪[25,75], y_window = [25,75])
    except IndexError:
        print(files)
        continue

    for j, nuc in enumerate(nucs):
        label_df = csv_df[['raw_classification', "main_classification",␣
 ↪'super_classification']].iloc[j]
        label = label_df.to_numpy()
```

```
        if 'unlabeled' not in label and 'AMBIGUOUS' not in label and np.
 ↪mean(nuc) > 0:
            im_name = f"im_{iters}.png"
            pickle_lst = [nuc.transpose(2,0,1), label]
            pickle_array.append(pickle_lst)

len(pickle_array)
```

[ ]:
```
28130
```

1.6 Create Dataset

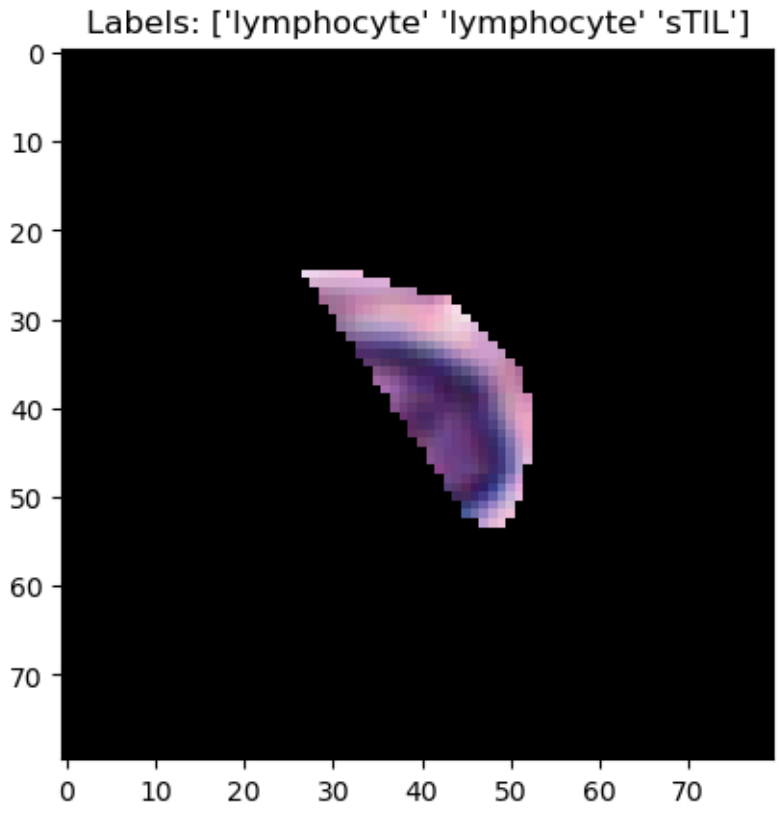
[ ]:
```
pickle_path = os.path.join(im_save_dir, 'nuc_data.pkl')
with open(pickle_path, 'wb') as f:
    pickle.dump(pickle_array, f)
```

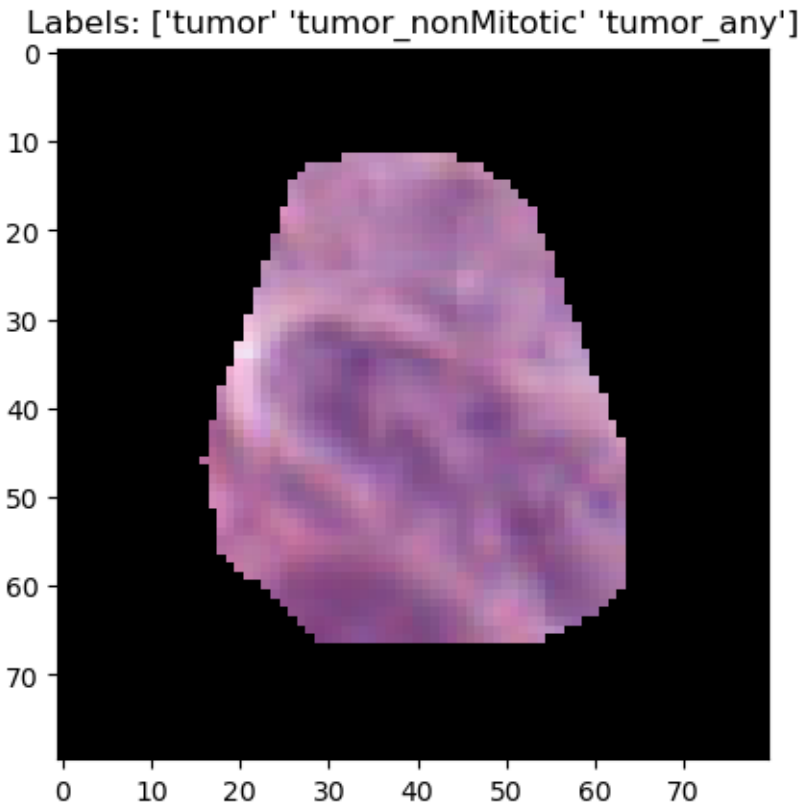
1.7 Pickle Loader

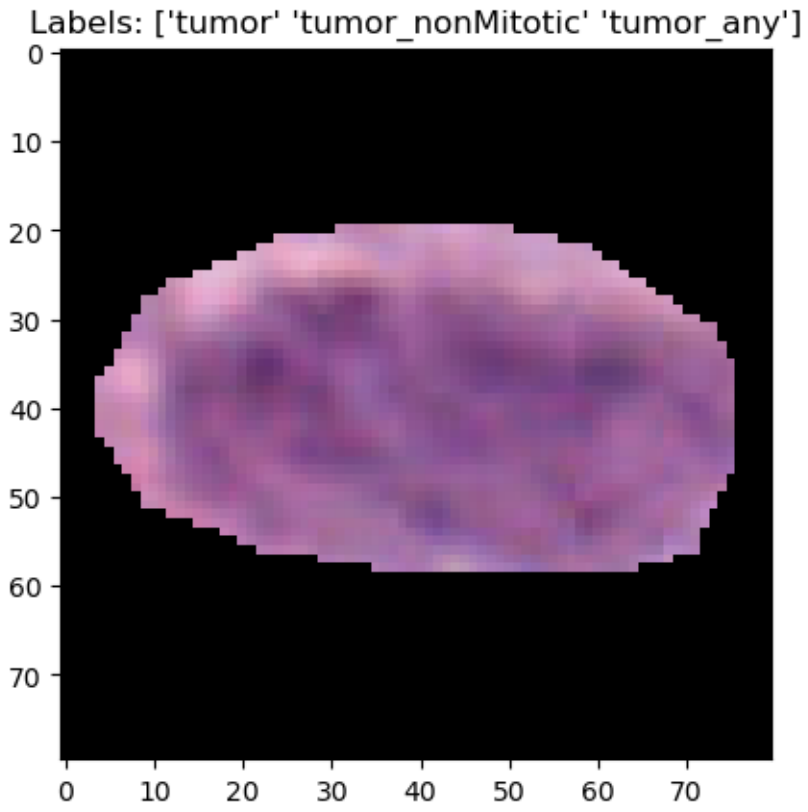
[ ]:
```
def load_dataset(path):
    with open(path, 'rb') as f:
        data = pickle.load(f)
    return data
im_save_dir = "../../DATASET/nuc"
pickle_path = os.path.join(im_save_dir, 'nuc_data.pkl')
data = load_dataset(pickle_path)
```

2 Show Images and Labels

[ ]:
```
for i in range(3):
    plt.title(f"Labels: {data[i][1]}")
    plt.imshow(data[i][0].transpose(1,2,0))
    plt.show()
```


Labels: ['lymphocyte' 'lymphocyte' 'sTIL']
0
10
20
30
40
50
60
70
0
10
20
30
40
50
60
70


Labels: ['tumor' 'tumor_nonMitotic' 'tumor_any']
0
10
20
30
40
50
60
70
0
10
20
30
40
50
60
70



3 Dataset Class

[ ]:
```
class NuCLSDataset(Dataset):
    def __init__(self, X, y, mode='CNN', bkgd='black'):
        super().__init__()
        self.bkgd = bkgd
        padded_X = X
        if self.bkgd == 'black':
            self.X = padded_X # just pad the input X's with black to the right␣
↪dimension
            # print(f'Shape of padded X: {self.X.shape}')
        else: # change black pixels to average value fo pixels in the cropped␣
↪image
            for i in range(padded_X.shape[0]):
                im = padded_X[i].transpose(1,2,0)
                non_black_pixels = im[im.sum(axis=2) > 0]
                average_color = np.mean(non_black_pixels, axis=0)
                mask = np.all(im == [0, 0, 0], axis=-1)
                im[mask] = average_color.astype(float)
                padded_X[i] = im.transpose(2,0,1)
```

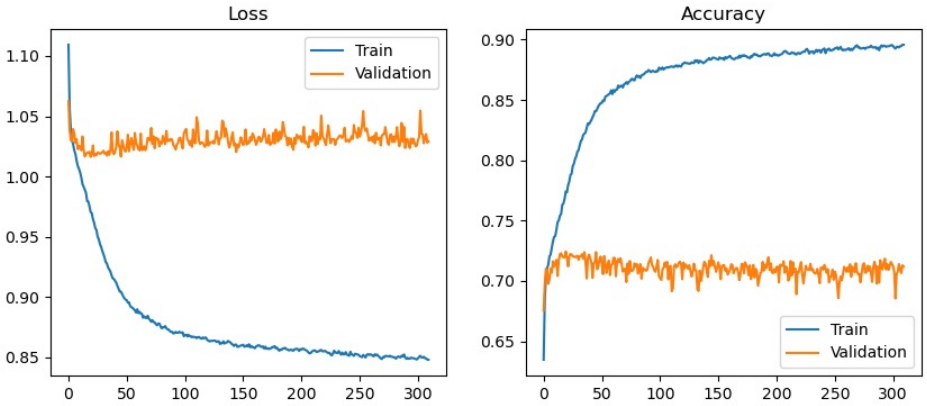
```
            self.X = padded_X

        self.X = torch.tensor(self.X, dtype=torch.float)
        self.y = torch.tensor(y, dtype=torch.float)
        self.mode = mode


    def __len__(self):
        return len(self.y)

    def __getitem__(self, idx):
        if self.mode == 'CNN':
            return self.X[idx], self.y[idx] # CNN returns index, for logreg␣
↪returns flttened image
        else:
            return self.X[idx].reshape(-1), self.y[idx]
```

4 Basic CNN network

[ ]:
```
class Cell_CNN(nn.Module):
    def __init__(self, num_classes=4):
        super().__init__()

        # conv
        self.conv1 = nn.Conv2d(3,100, kernel_size=5, stride=1)
        self.conv2 = nn.Conv2d(100,200, kernel_size=5, stride = 2, padding=1)
        self.conv3 = nn.Conv2d(200,300, kernel_size=5, stride = 2, padding=1)

        # pool
        self.pool1 = nn.MaxPool2d(kernel_size=3, stride=2)
        self.pool2 = nn.MaxPool2d(kernel_size=3, stride=2)
        self.pool3 = nn.MaxPool2d(kernel_size=3, stride=2)

        # activation function
        self.relu = nn.ReLU()
        self.softmax = nn.Softmax(dim=-1)

        # dropout layer
        self.dropout = nn.Dropout(0.8)

        # batch normalization
        self.bn1 = nn.BatchNorm2d(100)
        self.bn2 = nn.BatchNorm2d(200)
        self.bn3 = nn.BatchNorm2d(300)

        # flatten
```

```
        self.flatten = nn.Flatten()

        # fully connected layer
        fcconst = 300
        self.fc1 = nn.Linear(fcconst, fcconst)
        self.fc2 = nn.Linear(fcconst, num_classes)

    def forward(self, x):

        x = self.relu(self.bn1(self.conv1(x)))
        x = self.pool1(x)
        x = self.relu(self.bn2(self.conv2(x)))
        x = self.pool2(x)
        x = self.relu(self.bn3(self.conv3(x)))
        x = self.pool3(x)
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)
        x = self.softmax(x)
        return x    # do not apply softmax

# model = Cell_CNN()
```

5 First Results from Training a Basic CNN structure

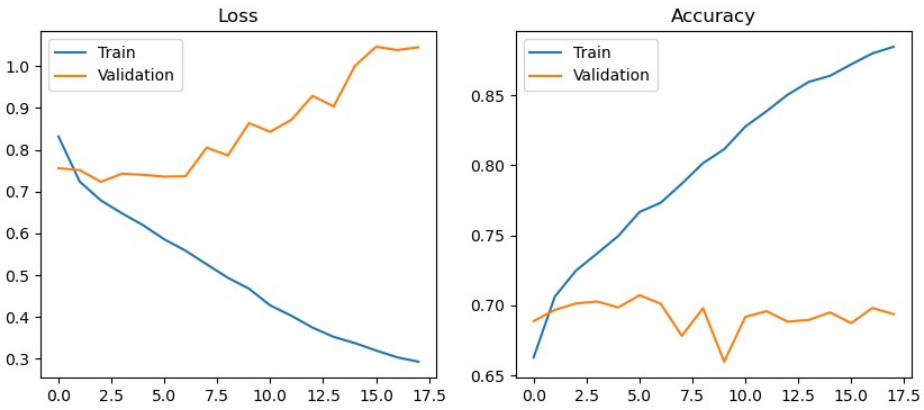


6 Densenet

[ ]:
```
from torchvision import models
class DenseNet(nn.Module):
    def __init__(self, num_classes=4):
```

```
        super(DenseNet, self).__init__()
        self.densenet = models.densenet121(pretrained=False)

        num_features = self.densenet.classifier.in_features
        self.densenet.classifier = nn.Linear(num_features, num_classes)

    def forward(self, x):
        x = self.densenet(x)
        return x

# densenet = DenseNet()
```

7 First Results form PyTorch Default Class of DenseNet before our implementation



8 Trainer Class

[ ]:
```
class Trainer:

    def __init__(self, model, opt_method, learning_rate, batch_size, epoch, l2):
        self.device = (
            "cuda"
            if torch.cuda.is_available()
            else "mps"
            if torch.backends.mps.is_available()
            else "cpu")
        self.model = model.to(self.device)

        if opt_method == "adam":
            self.optimizer = torch.optim.Adam(model.parameters(),␣
 ↪learning_rate, weight_decay=l2)
        else:
```

```
            raise NotImplementedError("This optimization is not supported")

        self.epoch = epoch
        self.batch_size = batch_size

    def train(self, train_data, val_data, early_stop=True, verbose=True,␣
↪draw_curve=True):
        train_loader = DataLoader(train_data, batch_size=self.batch_size,␣
↪shuffle=True)

        train_loss_list, train_acc_list = [], []
        val_loss_list, val_acc_list = [], []
        weights = self.model.state_dict()
        lowest_val_loss = np.inf
        loss_func = nn.CrossEntropyLoss()
        for n in tqdm(range(self.epoch), leave=False):
            self.model.train()
            epoch_loss, epoch_acc = 0.0, 0.0
            for X_batch, y_batch in train_loader:
                X_batch, y_batch = X_batch.to(self.device), y_batch.to(self.
↪device)
                batch_importance = y_batch.shape[0] / len(train_data)
                y_pred = self.model(X_batch)
                batch_loss = loss_func(y_pred, y_batch)

                self.optimizer.zero_grad()
                batch_loss.backward()
                self.optimizer.step()

                epoch_loss += batch_loss.detach().cpu().item() *␣
↪batch_importance
                batch_acc = torch.sum(torch.argmax(y_pred, axis=-1) == torch.
↪argmax(y_batch, axis=-1)) / y_batch.shape[0]
                epoch_acc += batch_acc.detach().cpu().item() * batch_importance
            train_loss_list.append(epoch_loss)
            train_acc_list.append(epoch_acc)
            val_loss, val_acc = self.evaluate(val_data)
            val_loss_list.append(val_loss)
            val_acc_list.append(val_acc)

            if early_stop:
                if val_loss < lowest_val_loss:
                    lowest_val_loss = val_loss
                    weights = self.model.state_dict()

        if draw_curve:
            x_axis = np.arange(self.epoch)
```

```
            fig, axes = plt.subplots(1, 2, figsize=(10, 4))
            axes[0].plot(x_axis, train_loss_list, label="Train")
            axes[0].plot(x_axis, val_loss_list, label="Validation")
            axes[0].set_title("Loss")
            axes[0].legend()
            axes[1].plot(x_axis, train_acc_list, label='Train')
            axes[1].plot(x_axis, val_acc_list, label='Validation')
            axes[1].set_title("Accuracy")
            axes[1].legend()

        if early_stop:
            self.model.load_state_dict(weights)

        return {
            "train_loss_list": train_loss_list,
            "train_acc_list": train_acc_list,
            "val_loss_list": val_loss_list,
            "val_acc_list": val_acc_list,
        }

    def evaluate(self, data, print_acc=False):
        self.model.eval()
        loader = DataLoader(data, batch_size=self.batch_size, shuffle=True)
        loss_func = nn.CrossEntropyLoss()
        acc, loss = 0.0, 0.0
        for X_batch, y_batch in loader:
            with torch.no_grad():
                X_batch, y_batch = X_batch.to(self.device), y_batch.to(self.
↪device)
                batch_importance = y_batch.shape[0] / len(data)
                y_pred = self.model(X_batch)
                batch_loss = loss_func(y_pred, y_batch)
                batch_acc = torch.sum(torch.argmax(y_pred, axis=-1) == torch.
↪argmax(y_batch, axis=-1)) / y_batch.shape[0]
                acc += batch_acc.detach().cpu().item() * batch_importance
                loss += batch_loss.detach().cpu().item() * batch_importance
        if print_acc:
            print(f"Accuracy: {acc:.3f}")
        return loss, acc
```

9 Load Dataset

[ ]:
```
from sklearn.model_selection import train_test_split
X = np.array([data[i][0] for i in range(len(data))])
y = np.array([data[i][1][class_map['super_classification']] for i in␣
↪range(len(data))])
```

```
encoder = OneHotEncoder()
y = encoder.fit_transform(y.reshape(-1,1)).toarray()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪shuffle=True)

train_data = NuCLSDataset(X=X_train, y=y_train, mode='CNN', bkgd='avg')
test_data = NuCLSDataset(X=X_test, y=y_test, mode='CNN', bkgd='avg')
X.shape, y.shape
```

[ ]: ((28130, 3, 80, 80), (28130, 4))

10 Train Networks

[ ]:
```
# dense_raw = DenseNet(num_classes=4)
# basic_cnn = Cell_CNN(num_classes=4)
# trainer = Trainer(dense_raw, "adam", 1e-4, 128, 2, 1e-5)
# trainer.train(train_data, test_data)
```

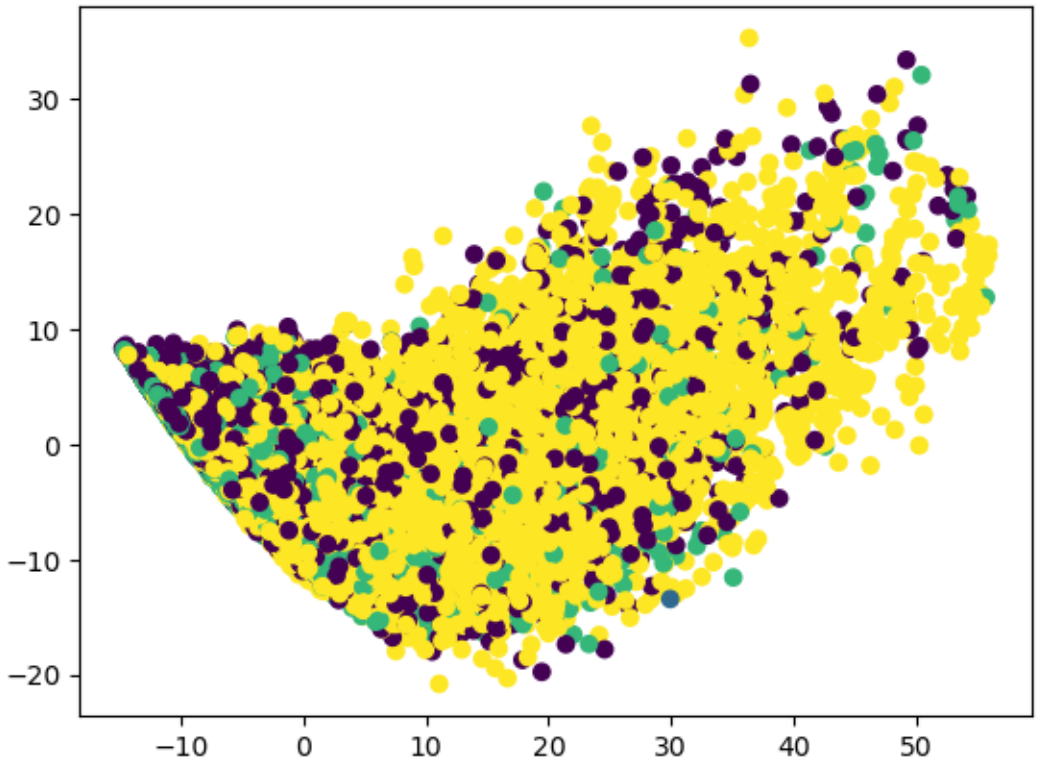
11 PCA and K Means Clustering

[ ]:
```
X = np.array([data[i][0] for i in range(len(data))])
y = np.array([data[i][1][2] for i in range(len(data))])
print(X)
print(y)
```

[ ]:
```
from sklearn.decomposition import PCA
pca = PCA(n_components=0.95)
X_pca = pca.fit_transform(X.reshape(-1, 80*80*3))
print(X_pca)
```

11.1 PCA True Labels

[ ]:
```
pc1 = X_pca[:,0]
pc2 = X_pca[:,1]
y_encoded = encoder.fit_transform(y.reshape(-1,1)).toarray()
y_labels = np.argmax(y_encoded, axis=-1)
plt.scatter(pc1, pc2, c=y_labels)
plt.show()
```



[ ]:
```
from sklearn.cluster import KMeans
num_clusters = 4

kmeans = KMeans(n_clusters=num_clusters)
kmeans.fit(X_pca)

labels = kmeans.labels_
```

[ ]:
```
y_train_true_labs = y

sclass2idx = {'nonTIL_stromal': 0, 'other_nucleus': 1, 'sTIL': 2, 'tumor_any':␣
↪3}
y_train_labs_list = y_train_true_labs.squeeze().tolist()
y_train_num = [sclass2idx[y_val] for y_val in y_train_labs_list]

acc = (sum(labels==y_train_num))/len(labels)
print(acc)
```

```
0.3680412371134021
```

11.2 K-Means Clustered

[ ]:
```
plt.scatter(pc1, pc2, c=labels)
```

[ ]:
```
<matplotlib.collections.PathCollection at 0x17708de10>
```