CSC 4080: Artificial Intelligence in Medical Imaging and Health

Homework Set 2

Due date: Mar. 13, 2022

Note: The weight assigned to each problem is given below.

This HW set **counts 10% of your course grade**.

***In this assignment, you can use any external references as you like. But you are not allowed to copy any code directly from others, make sure any line of code is written by yourself. If you imitate someone else's code, please include your references near the code.

Short Answer questions:

1. Why do deep learning methods become popular in the field of medical image analysis? Answer this question from the perspective of data, algorithms, and industry development. (10%)

The data:

As times goes by, the training set for medical image analysis will be larger and larger, providing adequate resource for deep learning. And it's also easy to get the resource.

Now, there are good algorithm and technique to transform the image into digital matrix.

Some data set long-tailed and useless, which is efficient to dual with if we apply deep learning.

The algorithm:

Deep artificial neural networks began outperforming other established models on a number of important benchmarks and algorithm.

DLA is applied to medical images, Convolutional Neural Networks (CNN) are ideally suited for classification, segmentation, object detection, registration, and other tasks. CNN is an artificial visual neural network structure used for medical image pattern recognition based on convolution operation.

The industrial development:

DLA is generally applicable for detecting an abnormality and classify a specific type of disease. Sometimes the doctor can miss one disease of a patient. The frequency cut a lot with DLA. It can even substitute doctors and workers in hospital, because of the high efficiency.

2. List the most popular 5 evaluation metrics of deep learning algorithms on CAD (Computer-Aided Diagnosis) problems, explain their meanings and mathematic definitions. (10%)

They are: confusion matrix, specificity, sensitivity, AUC, ROC Curve.

Confusion matrix in deep learning means the matrix that illustrates the distribution of true labels and the predicted one. The definition is: (take binary classification as an example), where N(x,y) means the number of the examples that with ture label x and we predict it y.

True_labels	Predict_labels	0	1
0		N(0,0)	N(0,1)
1		N(1,0)	N(1,1)

Sensitivity, means the Proportion of patients with disease who are tested positive with a test, definition:

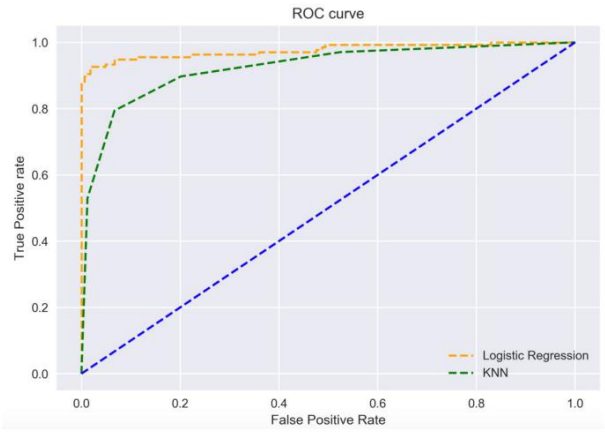
$$\text{sensitivity} = \frac{\text{True\_Positives}}{\text{True\_Positives} - \text{False\_Nrgatives}}$$

Specificity, Proportion of patients without disease who are tested negative with a test,

definition:

$$\text{specificity} = \frac{\text{True\_Negatives}}{\text{True\_Negatives+False\_Positives}}.$$

Roc curve means the model ability to distinguish between false & true positives. The curve definition is:

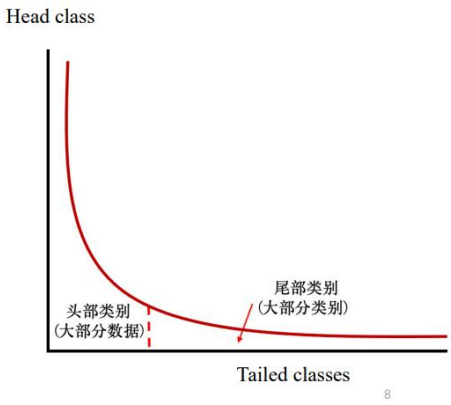


The x-ordinate is False Positive Rate, the y-ordinate is True Positive rate.

AUC, means the area under the Roc curve. AUC=$S_{roc\ curve}$

3. What is the main challenge of CAD problems nowadays? Answer this question with a real-world example like the instructor introduced in the class. (10%)

Unbalanced data, Long tailed data distribution; Weakly labelled data, Noisy label Useless data, noisy data, simple samples and hard samples. Example: The unbalanced classes in our training set of this assignment.



Overfitting and underfitting can cause large error, which will be shown when the training time is too long or too short. Like the example in this homework:

```
20it [00:06,  3.23it/s]
Training:Epoch[003/050] Iteration[020/020] folder: 01 Loss: 2.155 Acc:67.95%
5it [00:01,  3.29it/s]
Valid:   Epoch[003/050] Iteration[005/005] folder: 01 Loss: 1.388 Acc:62.82%
city of malignant:0.88
```

This is underfitting, the immature of the model causes the low accuracy.

```
20it [00:06,  3.10it/s]
Training:Epoch[038/050] Iteration[020/020] folder: 01 Loss: 0.06056 Acc:96.79%
5it [00:01,  3.30it/s]
Valid:   Epoch[038/050] Iteration[005/005] folder: 01 Loss: 1.615 Acc:74.36% AU
city of malignant:0.94
```

This is overfitting, the accuracy is very high on the training set, but the performance on the test set doesn't match it.

Programming problems:

1. data processing

```python
class MyDataset(Dataset):
    def __init__(self, data_dir, indexes, transform=None):
        """
        My Dataset for CAD with BUSI dataset
            param data_dir: str, path for the dataset
            param train: whether this is defined for training or testing
            param transform: torch.transform, data pre-processing pipeline
        """
        ### Begin your code ###
        self.data_info = self.get_data(data_dir,indexes)
        self.transform = transform

        ### End your code ###
```

The dataloader contains 3 parameters. Data_dir means the path of the folder, indexes means the indexes of image that is inputted. For example, if indexes =[31,11,140,30,90], that means the data set contain the 31^{st} , 12^{th} , 91^{st} , 141^{st} , 32^{th} images in the folder. Transform means the method that the images should be transformed.

```python
@staticmethod
def get_data(data_dir,indexes):
    '''
    Load the dataset and store it in your own data structure(s)
    '''
    ### Begin your code ###
    data_info=[os.path.join(data_dir, i ) for i in os.listdir(data_dir)]

    return [data_info[i] for i in indexes]
    ### End your code ###
```

In the get_data method, we get the particular image that we need for the dataset.

```python
def __getitem__(self, index):
    '''
    Get sample-label pair according to index
    '''
    ### Begin your code ###
    labels_dict={'benign':0,'malignant':1,'normal':2}
    path_img= self.data_info[index]

    img = Image.open(path_img).convert('RGB')

    if self.transform is not None:
        img = self.transform(img)
    for i in labels_dict:
        if i in path_img:
            return img,labels_dict[i]

    ### End your code ###
```

In the getitem method, we transform the image and return the label. If 'benign' is in the name of the image, we put it with label '0'. Others are the similar way.

2. Model(CNN)

```
def __init__(self, n_out):
    '''
    Define the layers that you need
    '''
    super(CNN, self).__init__()
    ### Begin your code ###
    self.features = nn.Sequential(
        nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=3, stride=2),
        nn.Conv2d(64, 64, kernel_size=5, padding=2),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=3, stride=2),

    )
    self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
    self.classifier = nn.Sequential(
        # nn.Dropout(),
        nn.Linear(64 * 6 * 6, n_out),
        # nn.Softmax(dim=1)
    )
    ### End your code ###
```

```
def forward(self, x):
    '''
    Define the forward propagation for data sample x
    '''
    ### Begin your code ###
    x = self.features(x)
    x = self.avgpool(x)
    x = torch.flatten(x, 1)
    x = self.classifier(x)
    return x
    ### End your code ###
```

We apply the convolution layer, next ReLU mathod, then Maxpooling method, and repeat it again. Then we use the average pooling to reduce parameters. We make it flatten and finally the linear method and outputs the results.

3. Loss function

Since the distribution of each classes is unbalanced, we need to get the wight to the traditional $-\log(p_t)$ method. We should add a weight. Here the focal loss is applied. That is $-(1-p_t)^{\gamma}\log(p_t)$.

```python
def __init__(self, alpha=0.25, gamma=2):
    '''
    Initialize some essential hyperparameters for your loss function
    '''
    super(MyLoss, self).__init__()
    ### Begin your code ###
    self.gamma = gamma
    self.alpha=alpha
    ### End your code ###

def forward(self, outputs, labels):
    '''
    Define the calculation of the loss
    '''
    ### Begin your code ###

    logpt = -f.cross_entropy(outputs,labels)


    loss = - (1 - torch.exp(logpt)) ** self.gamma * logpt

    return loss
```

In initial method, we input gamma(γ), and in the forward method, we return the loss.

4. Train

```python
train_transform = transforms.Compose([
    transforms.Resize((64, 64)),
    transforms.RandomResizedCrop(size=64, scale=(0.8, 1.0), ratio=(0.8, 1.25)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.18,0.18,0.18], std=[0.24,0.24,0.24])
])

valid_transform = transforms.Compose([
    transforms.Resize((64, 64)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.18,0.18,0.18], std=[0.24,0.24,0.24])
])
```

Here is the basic transformation process. The random resize in the train_transform is to cut the parameters randomly, to reduce the probability of overfitting. The normalized method is to make the learning rate applicable for the parameters, also better scale invariance and smoother optimization landscape.

```python
k=5
seed = 1
MAX_EPOCH = 50
BATCH_SIZE = 32
LR = 0.001
weight_decay = 1e-3
log_interval = 2
val_interval = 1
data_dir = "Dataset_BUSI"
indexes=np.arange(len([os.path.join(data_dir, i ) for i in os.listdir(data_dir)]))
set_seed(seed)
print('random seed:', seed)
main()
```

Here K means the K-Folder, indexes contain the indexes for each images.

```python
start = time.time()

kfold=KFold(n_splits=k,  shuffle=True, random_state=1)
folder_acc=[]

inf=[[[] for i in range(MAX_EPOCH)],[[] for i in range(MAX_EPOCH)]]

for x,data in enumerate(kfold.split(indexes)):
```

We use the KFold package from sklearn.model_selection to realize K-Folder. Then the indexes is cut into training indexes and testing indexes. The inf list is used to save the accuracy of each validation and epoch. The folder_acc saves the maximum accuracy in epochs for each fold of validation. The 5 cross validation starts with the for loop.

```
net = CNN(3)
net.to(device)
net.train()

# ============================ step 2/5 define the loss function ===================
criterion = MyLoss()

# ============================ step 3/5 define the optimizer =======================
optimizer = optim.Adam(net.parameters(), weight_decay=weight_decay)
```

Here we define the neutral net, applying the loss and optimizer.

```
train_data = MyDataset(data_dir, indexes[data[0]], transform=train_transform)
valid_data = MyDataset(data_dir, indexes[data[1]], transform=valid_transform)
train_loader = DataLoader(dataset=train_data, batch_size=BATCH_SIZE, shuffle=True)
valid_loader = DataLoader(dataset=valid_data, batch_size=BATCH_SIZE)
max_acc = 0.
reached = 0
```

Here we distribute the indexes for the training set and test set, and each transformation.

The dataloader method cut the whole sets into batches. The max_acc is the max_accuracy in validations of each folder. Reached is the epoch that reach that accuracy.

```
for epoch in range(1, MAX_EPOCH + 1):

    sensitivity = []
    specificity = []
    Pre_list=[]
    labels_list=[]

    loss_mean = 0.

    for i, data in tqdm(enumerate(train_loader)):

        # forward
        inputs, labels = data
        inputs = inputs.to(device)
        labels = labels.to(device)
        outputs = net(inputs)
```

The sensitivity list is used to save the sensitivities of the 3 classes. The specificity list is used to store the specificity of the 3 classes. The Pre_list saves all the prediction in the epoch. The labels_list saves all the true labels. Proba_list contain the probability of each result (outputs through softmax function)

Then in the next loop, we duel with the images in matches. Inputs is the image and labels is the label. Then the net shows the output.

```
Proba=list(nn.Softmax(dim=1)(outputs).detach().numpy())
Pre_list=Pre_list+list(predicted)
labels_list=labels_list+list(labels)
loss_mean += loss.item()

train_acc=accuracy_score(labels,predicted)
```

First we get the outputs that get through the softmax function. The prediction and true labels in the batch is added into the Pre_list and Labels_list. The loss_mean is defined as the total loss in the epoch. The accuracy_score from sklearn is used to calculate the accuracy.

```
        try:
            Auc= roc_auc_score(labels, Proba,multi_class='ovo')

            con_mat = confusion_matrix(labels,predicted)##here we define the confusion_matrix
            for j in range(3):
                number = np.sum(con_mat[:,:])
                tp = con_mat[j][j]
                fn = np.sum(con_mat[j,:]) - tp
                fp = np.sum(con_mat[:,j]) - tp
                tn = number - tp - fn - fp
                sen = tp / (tp + fn)
                spe = tn / (tn + fp)
                sensitivity.append(sen)
                specificity.append(spe)
    ### End your code ###
    # calculate the accuracy of this training iteration
    # print log

            if (i+1) % log_interval == 0:
                loss_mean = loss_mean / log_interval
                print("Training:Epoch[{:0>3}/{:0>3}] Iteration[{:0>3}/{:0>3}] Loss: {:.4f} Acc:{:.2%} AUC:{:.2}\n Sensitivity of benign:{:.2} Sensitivity
                    epoch, MAX_EPOCH, i+1, len(train_loader), loss_mean, train_acc,Auc,sensitivity[0],sensitivity[1],specificity[0],specificity[1]))
                loss_mean = 0.
```

AUC is calculated by roc_auc_score in sklearn. Confusion matrix is calculated by the package confusion_matrix in sklearn. Then according to the definition of the confusion matrix that is raised above, we compute the sensitivity of each classes and specificity of each classes and add them to the corresponding list.

Notice that here is the try method. The reason is that in some batches there may lose one kinds of labels, for example, the batch only contain label 0 or 1, then the ROC is not well-defined, and the sensitivity and specificity can also lead to 0/0. So here we use try method to avoid the problem. Then for each two iterations, we print information about the training set once.

```
        except:
            if (i+1) % log_interval == 0:
                loss_mean = loss_mean / log_interval
                print("Training:Epoch[{:0>3}/{:0>3}] Iteration[{:0>3}/{:0>3}] Loss: {:.4f} Acc:{:.2%} \n".format(
                    epoch, MAX_EPOCH, i+1, len(train_loader), loss_mean, train_acc))
                loss_mean = 0.
    epoch_acc=accuracy_score(labels_list,Pre_list)
    inf[0][epoch-1].append(epoch_acc)
```

If the rare situation above happen, we execute the except method. We only outputs the accuracy at this moment. Finally, we use the Pre_list and labels_list to get the accuracy in this epoch. The training accuracy of the epoch is added to the list inf[0][epoch-1].(inf[0] is for training accuracy, while inf[1] is for the test accuracy.

```
if epoch % val_interval == 0:
    Pre_list=[]
    labels_list=[]

    loss_val = 0.
    net.eval()
    with torch.no_grad():
        for j, data in tqdm(enumerate(valid_loader)):
            inputs, labels = data
            inputs = inputs.to(device)
            labels = labels.to(device)
            outputs = net(inputs)

            loss = criterion(outputs, labels)

            _, predicted = torch.max(outputs.data, 1)

            loss_val += loss.item()
```

Here we apply the criteria that has been used on training set. And get the loss and accuracy for one epoch.

```
            # calculate the accuracy of the validation predictions
            ### Begin your code ###
                Pre_list=Pre_list+list(predicted)
                labels_list=labels_list+list(labels)
                loss_mean += loss.item()
            val_acc=accuracy_score(labels_list,Pre_list)
            if val_acc > max_acc:
                max_acc = val_acc
                reached = epoch
            inf[1][epoch-1].append(val_acc)
            print("Valid:\t Epoch[{:0>3}/{:0>3}] Iteration[{:0>3}/{:0>3}] folder: {:0>2} Loss: {:.4} Acc:{:.2%} \n".format(
                epoch, MAX_EPOCH, j+1, len(valid_loader),x+1,loss_val,val_acc))
    folder_acc.append(max_acc)
```

We calculate the accuracy and put the test accuracy for each epoch into the inf list. Folder_acc will append the max_accuracy among 30 epochs.

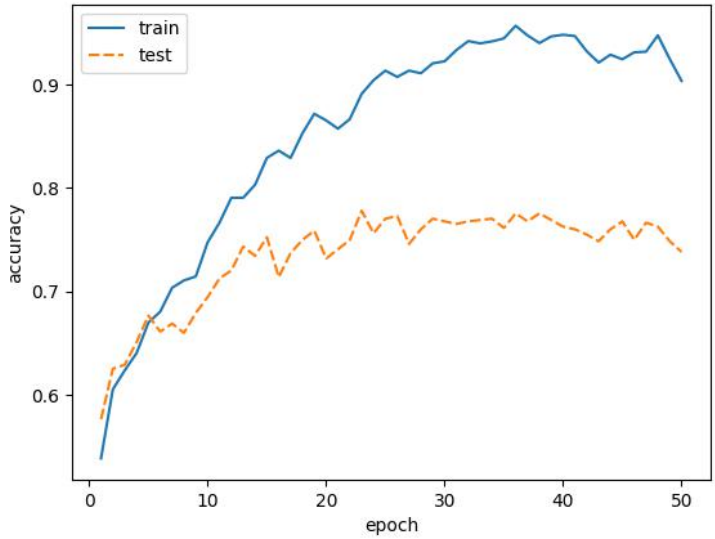
```
    print('\nTraining for folder {:0>2} finish, the time consumption of {} epochs is {}s\n'.format(x+1,MAX_EPOCH, round(time.time() - start)))
    print('The max validation accuracy is: {:.2%}, reached at epoch {}.\n'.format(max_acc, reached))
print(11111111111111111111111111111111111111111111111111111111111111111111111111111111)
print('\nTraining for all {} folders finish, the time consumption of {} epochs is {}s\n'.format(x+1,MAX_EPOCH, round(time.time() - start)))
print('The max validation accuracy is: {:.2%}.\n'.format(sum(folder_acc)/k))
```

Then we print the maximum accuracy in 50 epochs and where the maximum reaches. The final cross validation result is the mean value of the folder_acc, which contain 5 numbers representing the maximum accuracy in 5 cross validations.

```
for i in range(MAX_EPOCH):
    inf[0][i]=sum(inf[0][i])/k
    inf[1][i]=sum(inf[1][i])/k

inf=pd.DataFrame(inf,columns=np.arange(1,MAX_EPOCH+1),index=['train','test']).T
ax=sns.lineplot(data=inf)
ax.set(xlabel='epoch',ylabel='accuracy')
plt.show()
```

Then we can get the average cross validation accuracy in each epochs with the data in inf list. The graph is below:



This graph shows that the accuracy on the training set has an tendency to increase as the epochs increase. However, in the test set, it first increases with that in the training set, and then tend to be smooth. The reason may be overfitting.

The result is that the average 5-fold validation accuracy is 79.94%, with 3 of numbers in folder_acc larger than 80%.

```
The max validation accuracy is: 79.74%.
```