# HW3_CNN

October 3, 2025

# 1 HW3 Convolutional Neural Network

## 1.1 Overview

In this homework, you will get introduced to CNN. More specifically, you will try CNN on X-Ray images.

---

[1]:
```
import os
import random
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import time

# record start time
_START_RUNTIME = time.time()

# set seed
seed = 24
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
os.environ["PYTHONHASHSEED"] = str(seed)

# Define data and weight path
DATA_PATH = "../HW3_CNN-lib/data"
WEIGHT_PATH = "../HW3_CNN-lib/resnet18_weights_9.pth"
```

---

## 1.2 About Raw Data

Pneumonia is a lung disease characterized by inflammation of the airspaces in the lungs, most commonly due to an infection. In this section, you will train a CNN model to classify Pneumonia disease (Pneumonia/Normal) based on chest X-Ray images.

The chest X-ray images (anterior-posterior) were selected from retrospective cohorts of pediatric patients of one to five years old. All chest X-ray imaging was performed as part of patients' routine clinical care. You can refer to this link for more information.

### 1.3 1 Load and Visualize the Data [20 points]

The data is under `DATA_PATH`. In this part, you are required to load the data into the data loader, and calculate some statistics.

[2]:
```
#input
# folder: str, 'train', 'val', or 'test'
#output
# number_normal: number of normal samples in the given folder
# number_pneumonia: number of pneumonia samples in the given folder
def get_count_metrics(folder, data_path=DATA_PATH):

    '''
    TODO: Implement this function to return the number of normal and pneumonia␣
↪samples.
          Hint: !ls $DATA_PATH
    '''

    # your code here
    base = os.path.join(data_path, folder)
    n_norm = len(os.listdir(os.path.join(base, "NORMAL")))
    n_pneu = len(os.listdir(os.path.join(base, "PNEUMONIA")))
    return n_norm, n_pneu


#output
# train_loader: train data loader (type: torch.utils.data.DataLoader)
# val_loader: val data loader (type: torch.utils.data.DataLoader)
def load_data(data_path=DATA_PATH):

    '''
    TODO: Implement this function to return the data loader for
    train and validation dataset. Set batchsize to 32.

    You should add the following transforms (https://pytorch.org/docs/stable/
↪torchvision/transforms.html):
        1. transforms.RandomResizedCrop: the images should be cropped to 224 x␣
↪224
        2. transforms.ToTensor: just to convert data/labels to tensors
    You should set the *shuffle* flag for *train_loader* to be True, and False␣
↪for *val_loader*.

    HINT: Consider using `torchvision.datasets.ImageFolder`.
    '''
```

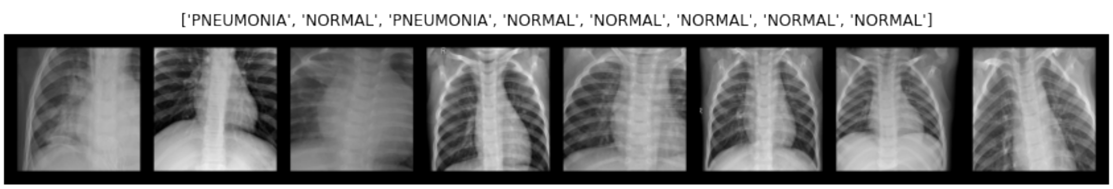
```
    import torchvision
    import torchvision.datasets as datasets
    import torchvision.transforms as transforms
    from torch.utils.data import DataLoader

    tfm = transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.ToTensor(),
    ])
    train_ds = datasets.ImageFolder(root=os.path.join(data_path, "train"),␣
↪transform=tfm)
    val_ds   = datasets.ImageFolder(root=os.path.join(data_path, "val"),  ␣
↪transform=tfm)

    train_loader = DataLoader(train_ds, batch_size=32, shuffle=True)
    val_loader   = DataLoader(val_ds,   batch_size=32, shuffle=False)


    return train_loader, val_loader
```

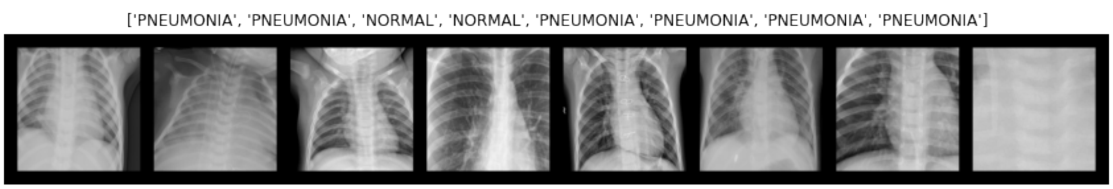
[3]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

assert type(get_count_metrics('train')) is tuple
assert type(get_count_metrics('val')) is tuple

assert get_count_metrics('train') == (335, 387)
assert get_count_metrics('val') == (64, 104)
```

[4]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

train_loader, val_loader = load_data()

assert type(train_loader) is torch.utils.data.dataloader.DataLoader

assert len(train_loader) == 23
```

[5]:
```
# DO NOT MODIFY THIS PART


import torchvision
import matplotlib.pyplot as plt

def imshow(img, title):
```

```python
    npimg = img.numpy()
    plt.figure(figsize=(15, 7))
    plt.axis('off')
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.title(title)
    plt.show()

def show_batch_images(dataloader, k=8):
    images, labels = next(iter(dataloader))
    images = images[:k]
    labels = labels[:k]
    img = torchvision.utils.make_grid(images, padding=25)
    imshow(img, title=["NORMAL" if x==0  else "PNEUMONIA" for x in labels])

train_loader, val_loader = load_data()
for i in range(2):
    show_batch_images(train_loader)
```




### 1.4 2 Build the Model [35 points]

This time, you will define a CNN architecture. Instead of an MLP, which used linear, fully-connected layers, you will use the following: - Convolutional layers, which can be thought of as stack of filtered images. - Maxpooling layers, which reduce the x-y size of an input, keeping only the most active pixels from the previous layer. - The usual Linear + Dropout layers to avoid overfitting and produce a 2-dim output.

Below is a typical CNN architicture which consists of [INPUT - CONV - RELU - POOL - FC] layers.

### 1.4.1 2.1 Convolutional Layer Output Volume [10 points]

Before we get started, let us do a warm-up question.

Calculate the output volume for a convolutional layer: given the input volume size $W$, the kernel/filter size $F$, the stride $S$, and the amount of zero padding $P$ used on the border, calculate the output volume size.

[6]:
```
def conv_output_volume(W, F, S, P):

    """
    TODO: Given the input volume size $W$, the kernel/filter size $F$,
    the stride $S$, and the amount of zero padding $P$ used on the border,
    calculate the output volume size.
    Note the output should a integer.
    """

    # your code here
    return int((W - F + 2*P)//S + 1)
```

[7]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

assert conv_output_volume(W=7, F=3, S=1, P=0) == 5
assert conv_output_volume(W=7, F=3, S=2, P=0) == 3
assert conv_output_volume(W=8, F=3, S=2, P=0) == 3
```

### 1.4.2 2.2 Define CNN [15 points]

Now, define your own CNN model below. Note that, the more convolutional layers you include, the more complex patterns the model can detect. For now, it is suggested that your final model include 2 or 3 convolutional layers as well as linear layers + dropout in between to avoid overfitting.

It is also a good practice to look at existing research and implementations of related models as a starting point for defining your own models. You may find it useful to look at this PyTorch classification example.

Please do not use the same model structure as in Section 2.3. Specifically, let's define a small model with less than 10 layers/modules (must be fewer than 20).

[8]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F

class SimpleCNN(nn.Module):
    def __init__(self):
        super().__init__()
        # 3x224x224 -> 32x112x112
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1, stride=1)
```

```python
        self.pool1 = nn.MaxPool2d(2,2)
        # 32x112x112 -> 64x56x56
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1, stride=1)
        self.pool2 = nn.MaxPool2d(2,2)
        # 64x56x56 -> 128x28x28
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1, stride=1)
        self.pool3 = nn.MaxPool2d(2,2)

        # flatten: 128*28*28 = 100352
        self.dropout = nn.Dropout(p=0.5)
        self.fc1 = nn.Linear(128*28*28, 256)
        self.fc2 = nn.Linear(256, 2)

    def forward(self, x):
        x = F.relu(self.conv1(x)); x = self.pool1(x)
        x = F.relu(self.conv2(x)); x = self.pool2(x)
        x = F.relu(self.conv3(x)); x = self.pool3(x)
        x = torch.flatten(x, 1)
        x = self.dropout(F.relu(self.fc1(x)))
        x = self.fc2(x)
        return x
```

[9]:
```python
simple_model = SimpleCNN()
simple_model_size = sum([param.nelement() * param.element_size() for param in␣
 ↪simple_model.parameters()]) / 1e9
print('SimpleCNN size in GB:', simple_model_size)
assert simple_model_size <= 1, 'SimpleCNN is too large! Please minimize the␣
 ↪number of parameters.'
```

```
SimpleCNN size in GB: 0.10313652
```

[10]:
```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
simple_model = SimpleCNN()

assert issubclass(type(simple_model), nn.Module), "Your CNN model should be a␣
 ↪torch.nn.Module instance."
assert len(list(simple_model.modules())) < 20, "Your CNN model is too big.␣
 ↪Please re-design."

test_input = torch.zeros(32, 3, 224, 224)
test_output = simple_model(test_input)
assert test_output.shape == torch.Size([32,2]), "Your CNN model has a wrong␣
 ↪output size."
```

### 1.4.3 2.3 Using Predefined CNN Model [10 points]

In this section, we will import a predefined CNN, the ResNet18 model, which is pretty successful in many image classification tasks. We will modify the last layer to use it on our binary classification problem, but keep the rest of the structure the same

[11]:
```
from torchvision import models
import torch.nn as nn

def get_cnn_model():
    model = models.resnet18(pretrained=False)
    in_feats = model.fc.in_features
    model.fc = nn.Linear(in_feats, 2)

    # freeze everything except the new head
    for name, p in model.named_parameters():
        if not name.startswith("fc."):
            p.requires_grad = False
    return model
```

[12]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

assert issubclass(type(get_cnn_model()), nn.Module), "Your CNN model should be␣
↪a torch.nn.Module instance"
model = get_cnn_model()
assert len(list(model.modules())) == 68, "# of modules mismtach - Please use␣
↪ResNet18"
assert len(list(model.parameters())) == 62, "# of parameter tensors mismtach -␣
↪different model. Please use ResNet18"
```

## 1.5 3 Training the Network [25 points]

Due to the computation environment constraint, we will load some pre-trained weights instead of training everything from scratch.

[13]:
```
model = get_cnn_model()
#Load the pretrained weights
#If it fails, it probably means you did not define the model correctly
model.load_state_dict(torch.load(WEIGHT_PATH, map_location='cpu'))
```

[13]:
```
<All keys matched successfully>
```

### 1.5.1 3.1 Criterion and Opimizer [10 points]

In this part, you will define the loss and optimizer for the model and then perform model training.

[14]:
```python
"""
TODO: Specify loss function (CrossEntropyLoss) and assign it to `criterion`.
Spcify optimizer (SGD) and assign it to `optimizer`.
Hint: the learning rate is usually a small number on the scale of 1e-4 ~ 1e-2
"""

import torch
import torch.nn as nn

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-2)
```

[15]:
```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

assert isinstance(criterion, torch.nn.modules.loss.CrossEntropyLoss)
assert isinstance(optimizer, torch.optim.SGD)
```

### 1.5.2 3.2 Training [15 points]

Now let us train the CNN model we previously created.

Remember that from the previous HW, to train the model, you should follow the following step: - Clear the gradients of all optimized variables - Forward pass: compute predicted outputs by passing inputs to the model - Calculate the loss - Backward pass: compute gradient of the loss with respect to model parameters - Perform a single optimization step (parameter update) - Update average training loss

[16]:
```python
# number of epochs to train the model
# make sure your model finish training within 4 minutes on a CPU machine
# You can experiment different numbers for n_epochs, but even 1 epoch should be␣
↪good enough.
n_epochs = 1

def train_model(model, train_dataloader, n_epoch=1, optimizer=optimizer,␣
↪criterion=criterion):
    model.train()
    for epoch in range(n_epoch):
        losses = []
        for x, y in train_dataloader:
            optimizer.zero_grad()
            logits = model(x)             # (B,2)
            loss = criterion(logits, y)  # y long tensor (B,)
            loss.backward()
            optimizer.step()
            losses.append(loss.item())
        print(f"Epoch {epoch}: curr_epoch_loss={sum(losses)/len(losses):.6f}")
```

```
    return model
```

[17]:
```
# get train and val data loader
train_loader, val_loader = load_data()

seed = 24
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)

model = train_model(model, train_loader)
```

```
Epoch 0: curr_epoch_loss=0.182608
```

[ ]:
```

```

### 1.6 4 Test the Trained Network [20 points]

In this step, you will test your model on the validation data and evaluate its performance.

[18]:
```
import numpy as np
import torch

def eval_model(model, dataloader):
    model.eval()
    y_pred, y_true = [], []
    with torch.no_grad():
        for x, y in dataloader:
            logits = model(x)                        # (B,2)
            pred = torch.argmax(logits, dim=1) # (B,)
            y_pred.append(pred.cpu().numpy())
            y_true.append(y.cpu().numpy())
    y_pred = np.concatenate(y_pred, axis=0)
    y_true = np.concatenate(y_true, axis=0)
    return y_pred, y_true
```

[19]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
from sklearn.metrics import accuracy_score

y_pred, y_true = eval_model(model, val_loader)
acc = accuracy_score(y_true, y_pred)
print(("Validation Accuracy: " + str(acc)))
assert acc > 0.7, "Validation Accuracy below 0.7 for validation data!"
assert len(y_true) == len(y_pred) == 168, "Output size is wrong"
```

```
Validation Accuracy: 0.8690476190476191
```

[20]:
```
#As noted before, please make sure the whole notebook does not exceed 4 mins on␣
↪a CPU
print("Total running time = {:.2f} seconds".format(time.time() -␣
↪_START_RUNTIME))
```

```
Total running time = 426.78 seconds
```

[ ]:
```

```