Applying Augmentation, Fine-Tuning Last Dense Block, Transition Layer, Classifier

> In this notebook, we will only apply **augmentation**, **fine-tune the last dense block**, **transition layer**, and **classifier**. We'll use this approach as a comparison with other notebooks that use different techniques.

In [1]:
```
import os
import copy
import pandas as pd
import torch
import torchvision
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, WeightedRandomSampler
import torchvision.transforms as transforms
import torchvision.models as models
from PIL import Image
from sklearn.preprocessing import StandardScaler, LabelEncoder
import numpy as np
from sklearn.metrics import roc_auc_score
import torch.nn.functional as F
```

In [2]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device.type}")
```

```
Using device: cuda
```

In [3]:
```
CKPT_PATH = '../model.pth.tar'
N_CLASSES = 14
CLASS_NAMES = [ 'Atelectasis', 'Cardiomegaly', 'Effusion', 'Infiltration', 'Mass',
                'Pneumothorax', 'Consolidation', 'Edema', 'Emphysema', 'Fibrosis',
DATA_DIR = '../images'
TEST_IMAGE_LIST = '../test_list.txt'
BATCH_SIZE = 64
```

In [4]:
```
class ChestXrayDataSet(Dataset):
    def __init__(self, data_dir, image_list_file, transform=None):
        """
        Args:
            data_dir: path to image directory.
            image_list_file: path to the file containing images
                with corresponding labels.
            transform: optional transform to be applied on a sample.
        """
        image_names = []
        labels = []
        with open(image_list_file, "r") as f:
            for line in f:
                items = line.split()
                image_name= items[0]
                label = items[1:]
```

```python
                label = [int(i) for i in label]
                image_name = os.path.join(data_dir, image_name)
                image_names.append(image_name)
                labels.append(label)

        self.image_names = image_names
        self.labels = labels
        self.transform = transform

    def __getitem__(self, index):
        """
        Args:
            index: the index of item

        Returns:
            image and its labels
        """
        image_name = self.image_names[index]
        image = Image.open(image_name).convert('RGB')
        label = self.labels[index]
        if self.transform is not None:
            image = self.transform(image)
        return image, torch.FloatTensor(label)

    def __len__(self):
        return len(self.image_names)
```

In [5]:
```python
class DenseNet121(nn.Module):
    """Model modified.

    The architecture of our model is the same as standard DenseNet121
    except the classifier layer which has an additional sigmoid function.

    """
    def __init__(self, out_size):
        super(DenseNet121, self).__init__()
        self.densenet121 = torchvision.models.densenet121(pretrained=True)
        num_ftrs = self.densenet121.classifier.in_features
        self.densenet121.classifier = nn.Sequential(
            nn.Linear(num_ftrs, out_size),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.densenet121(x)
        return x
```

In [6]:
```python
def compute_AUCs(gt, pred):
    """Computes Area Under the Curve (AUC) from prediction scores.

    Args:
        gt: Pytorch tensor on GPU, shape = [n_samples, n_classes]
          true binary labels.
        pred: Pytorch tensor on GPU, shape = [n_samples, n_classes]
          can either be probability estimates of the positive class,
```

```
            confidence values, or binary decisions.

    Returns:
        List of AUROCs of all classes.
    """
    AUROCs = []
    gt_np = gt.cpu().numpy()
    pred_np = pred.cpu().numpy()
    for i in range(N_CLASSES):
        AUROCs.append(roc_auc_score(gt_np[:, i], pred_np[:, i]))
    return AUROCs
```

In [7]:

```
import torchvision.transforms as transforms

TRAIN_LIST = "../train_list.txt"
VALID_LIST = "../val_list.txt"
IMAGE_DIR = "../images"

data_transforms = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

train_dataset = ChestXrayDataSet(IMAGE_DIR, TRAIN_LIST, transform=data_transforms)
trainloader = torch.utils.data.DataLoader(train_dataset, batch_size=64, shuffle=Fal
```

In [8]:

```
images, labels = next(iter(trainloader))
print("Images shape:", images.shape)
print("Labels shape:", labels.shape)
```

```
Images shape: torch.Size([64, 3, 224, 224])
Labels shape: torch.Size([64, 14])
```

In [9]:

```
model = DenseNet121(N_CLASSES).to(device)
model = torch.nn.DataParallel(model).to(device)

if os.path.isfile(CKPT_PATH):
    print("=> loading checkpoint")
    modelCheckpoint = torch.load(CKPT_PATH)['state_dict']
    for k in list(modelCheckpoint.keys()):
        index = k.rindex('.')
        if (k[index - 1] == '1' or k[index - 1] == '2'):
            modelCheckpoint[k[:index - 2] + k[index - 1:]] = modelCheckpoint[k]
            del modelCheckpoint[k]
    model.load_state_dict(modelCheckpoint)
    print("=> loaded checkpoint")
else:
    print("=> no checkpoint found")

normalize = transforms.Normalize([0.485, 0.456, 0.406],
                                 [0.229, 0.224, 0.225])

test_dataset = ChestXrayDataSet(data_dir=DATA_DIR,
```

```
                                    image_list_file=TEST_IMAGE_LIST,
                                    transform=transforms.Compose([
                                        transforms.Resize(256),
                                        transforms.TenCrop(224),
                                        transforms.Lambda
                                        (lambda crops: torch.stack([transforms.ToTe
                                        transforms.Lambda
                                        (lambda crops: torch.stack([normalize(crop)
                                    ]))
test_loader = DataLoader(dataset=test_dataset, batch_size=BATCH_SIZE,
                             shuffle=False, num_workers=0, pin_memory=True)
```

```
c:\Users\aroce\miniforge3\envs\572\Lib\site-packages\torchvision\models\_utils.py:20
8: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be remov
ed in the future, please use 'weights' instead.
  warnings.warn(
c:\Users\aroce\miniforge3\envs\572\Lib\site-packages\torchvision\models\_utils.py:22
3: UserWarning: Arguments other than a weight enum or `None` for 'weights' are depre
cated since 0.13 and may be removed in the future. The current behavior is equivalen
t to passing `weights=DenseNet121_Weights.IMAGENET1K_V1`. You can also use `weights=
DenseNet121_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
=> loading checkpoint
=> loaded checkpoint
```

In [10]:

```
# Data transformations
train_transforms = transforms.Compose([
    transforms.Resize(256),
    transforms.RandomResizedCrop(224),
    transforms.RandomRotation(10),
    transforms.ColorJitter(brightness=0.2, contrast=0.2),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

valid_transforms = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
```

In [11]:

```
# Class counts and weights
class_counts = [
    959,  # Atelectasis
    276,  # Cardiomegaly
    972,  # Effusion
    1591, # Infiltration
    313,  # Mass
    461,  # Nodule
    140,  # Pneumonia
    469,  # Pneumothorax
    397,  # Consolidation
    140,  # Edema
    208,  # Emphysema
    244,  # Fibrosis
```

```
    349,  # Pleural_Thickening
    35    # Hernia
]

total_count = sum(class_counts)
class_weights = [total_count / count for count in class_counts]
class_weights_cpu = torch.tensor(class_weights, dtype=torch.float)

# Compute sample weights
sample_weights = []
for _, label in train_dataset:
    label = label.float()
    weight = torch.sum(class_weights_cpu * label).item()
    sample_weights.append(weight)

sample_weights = torch.tensor(sample_weights, dtype=torch.float)
sampler = WeightedRandomSampler(weights=sample_weights, num_samples=len(sample_weig
class_weights_gpu = torch.tensor(class_weights, dtype=torch.float).to(device)
```

In [12]:

```
# Replace the classifier with a more regularized version
old_classifier = model.module.densenet121.classifier
num_ftrs = None
if isinstance(old_classifier, nn.Sequential):
    # Get input features from the first Linear layer
    for layer in old_classifier:
        if isinstance(layer, nn.Linear):
            num_ftrs = layer.in_features
            break
else:
    # Direct linear layer
    num_ftrs = old_classifier.in_features

model.module.densenet121.classifier = nn.Sequential(
    nn.Linear(num_ftrs, 1024),
    nn.ReLU(),
    nn.Linear(1024, 512),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(512, 256),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(256, N_CLASSES),
    nn.Sigmoid()
)

print("Replaced classifier with a regularized version")
```

```
Replaced classifier with a regularized version
```

In [13]:

```
# Freeze all parameters
for param in model.module.densenet121.parameters():
    param.requires_grad = False

# Unfreeze the classifier
for param in model.module.densenet121.classifier.parameters():
    param.requires_grad = True
```

```python
# Also unfreeze the last dense block (denseblock4)
for param in model.module.densenet121.features.denseblock4.parameters():
    param.requires_grad = True

# Also unfreeze the transition layer before denseblock4
for param in model.module.densenet121.features.transition3.parameters():
    param.requires_grad = True

# Count trainable parameters
trainable_params = [p for p in model.parameters() if p.requires_grad]
print(f"Number of trainable parameters: {sum(p.numel() for p in trainable_params):,
```

```
Number of trainable parameters: 4,393,742
```

In [14]:
```python
criterion = nn.BCELoss()

# Add weight decay to combat overfitting
optimizer = optim.Adam(
    filter(lambda p: p.requires_grad, model.parameters()),
    lr=1e-4,
    weight_decay=1e-5
)

# Prepare datasets
train_dataset = ChestXrayDataSet(IMAGE_DIR, TRAIN_LIST, transform=train_transforms)
valid_dataset = ChestXrayDataSet(IMAGE_DIR, VALID_LIST, transform=valid_transforms)

# Create data loaders
trainloader = DataLoader(train_dataset, sampler=sampler, batch_size=64, pin_memory=
validloader = DataLoader(valid_dataset, batch_size=64, shuffle=False, pin_memory=Tr
```

In [15]:
```python
def trainer(model, criterion, optimizer, trainloader, validloader, epochs=20, patie
    train_loss, valid_loss, valid_accuracy = [], [], []

    # To track best model
    best_auroc = 0.0
    best_model_weights = None
    counter = 0  # For early stopping

    for epoch in range(epochs):
        # Training Phase
        model.train()
        epoch_train_loss = 0.0
        for inputs, labels in trainloader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()

            outputs = model(inputs)
            loss = criterion(outputs, labels)

            loss.backward()
            optimizer.step()
            epoch_train_loss += loss.item() * inputs.size(0)

        train_loss.append(epoch_train_loss / len(trainloader.dataset))
```

```
        # Validation Phase
        model.eval()
        epoch_valid_loss = 0.0
        all_labels = torch.FloatTensor().to(device)
        all_outputs = torch.FloatTensor().to(device)

        with torch.no_grad():
            for inputs, labels in validloader:
                inputs, labels = inputs.to(device), labels.to(device)

                outputs = model(inputs)
                loss = criterion(outputs, labels)

                epoch_valid_loss += loss.item() * inputs.size(0)

                all_labels = torch.cat((all_labels, labels), 0)
                all_outputs = torch.cat((all_outputs, outputs), 0)

        # Calculate metrics
        predictions = (all_outputs > 0.5).float()
        correct = (predictions == all_labels).sum().item()
        total = all_labels.numel()
        accuracy = correct / total

        aurocs = compute_AUCs(all_labels, all_outputs)
        mean_auroc = np.mean(aurocs)

        valid_loss.append(epoch_valid_loss / len(validloader.dataset))
        valid_accuracy.append(accuracy)

        if verbose:
            print(f"Epoch {epoch+1}/{epochs} - Train Loss: {train_loss[-1]:.4f}, Va
            print(f"Accuracy: {accuracy:.4f}, Mean AUROC: {mean_auroc:.4f}")

            if (epoch + 1) % 5 == 0 or epoch == epochs - 1:
                for i, auroc in enumerate(aurocs):
                    print(f"  {CLASS_NAMES[i]}: AUROC = {auroc:.4f}")

        # Early stopping and model saving
        if mean_auroc > best_auroc:
            best_auroc = mean_auroc
            best_model_weights = copy.deepcopy(model.state_dict())
            print(f"  New best model with AUROC: {best_auroc:.4f}")
            counter = 0  # Reset counter
        else:
            counter += 1
            if counter >= patience:
                print(f"  Early stopping triggered after {epoch+1} epochs")
                break

    # Load the best model weights
    if best_model_weights is not None:
        model.load_state_dict(best_model_weights)
        print(f"Loaded best model with AUROC: {best_auroc:.4f}")
```

```python
    return {
        'train_loss': train_loss,
        'valid_loss': valid_loss,
        'valid_accuracy': valid_accuracy,
        'best_auroc': best_auroc
    }
```

In [16]:
```python
model.to(device)
trainer(model, criterion, optimizer, trainloader, validloader, epochs=20)
```

```
Epoch 1/20 - Train Loss: 0.5128, Valid Loss: 0.3110
Accuracy: 0.8932, Mean AUROC: 0.5464
  New best model with AUROC: 0.5464
Epoch 2/20 - Train Loss: 0.3846, Valid Loss: 0.3070
Accuracy: 0.8932, Mean AUROC: 0.6059
  New best model with AUROC: 0.6059
Epoch 3/20 - Train Loss: 0.3779, Valid Loss: 0.2933
Accuracy: 0.8932, Mean AUROC: 0.6986
  New best model with AUROC: 0.6986
Epoch 4/20 - Train Loss: 0.3703, Valid Loss: 0.2840
Accuracy: 0.8960, Mean AUROC: 0.7465
  New best model with AUROC: 0.7465
Epoch 5/20 - Train Loss: 0.3605, Valid Loss: 0.2764
Accuracy: 0.8963, Mean AUROC: 0.7712
  Atelectasis: AUROC = 0.7912
  Cardiomegaly: AUROC = 0.9431
  Effusion: AUROC = 0.8185
  Infiltration: AUROC = 0.6108
  Mass: AUROC = 0.8355
  Nodule: AUROC = 0.7722
  Pneumonia: AUROC = 0.5734
  Pneumothorax: AUROC = 0.8203
  Consolidation: AUROC = 0.7534
  Edema: AUROC = 0.8917
  Emphysema: AUROC = 0.7355
  Fibrosis: AUROC = 0.6895
  Pleural_Thickening: AUROC = 0.7631
  Hernia: AUROC = 0.7987
  New best model with AUROC: 0.7712
Epoch 6/20 - Train Loss: 0.3551, Valid Loss: 0.2689
Accuracy: 0.8980, Mean AUROC: 0.7917
  New best model with AUROC: 0.7917
Epoch 7/20 - Train Loss: 0.3526, Valid Loss: 0.2626
Accuracy: 0.8962, Mean AUROC: 0.8019
  New best model with AUROC: 0.8019
Epoch 8/20 - Train Loss: 0.3487, Valid Loss: 0.2585
Accuracy: 0.8996, Mean AUROC: 0.8058
  New best model with AUROC: 0.8058
Epoch 9/20 - Train Loss: 0.3423, Valid Loss: 0.2542
Accuracy: 0.8988, Mean AUROC: 0.8124
  New best model with AUROC: 0.8124
Epoch 10/20 - Train Loss: 0.3407, Valid Loss: 0.2562
Accuracy: 0.8969, Mean AUROC: 0.8142
  Atelectasis: AUROC = 0.8301
  Cardiomegaly: AUROC = 0.9437
  Effusion: AUROC = 0.8356
  Infiltration: AUROC = 0.6672
  Mass: AUROC = 0.8776
  Nodule: AUROC = 0.8332
  Pneumonia: AUROC = 0.5787
  Pneumothorax: AUROC = 0.8509
  Consolidation: AUROC = 0.7767
  Edema: AUROC = 0.8879
  Emphysema: AUROC = 0.8263
  Fibrosis: AUROC = 0.7351
  Pleural_Thickening: AUROC = 0.7992
```

```
  Hernia: AUROC = 0.9563
  New best model with AUROC: 0.8142
Epoch 11/20 - Train Loss: 0.3384, Valid Loss: 0.2555
Accuracy: 0.8982, Mean AUROC: 0.8179
  New best model with AUROC: 0.8179
Epoch 12/20 - Train Loss: 0.3403, Valid Loss: 0.2540
Accuracy: 0.8975, Mean AUROC: 0.8178
Epoch 13/20 - Train Loss: 0.3341, Valid Loss: 0.2547
Accuracy: 0.8990, Mean AUROC: 0.8180
  New best model with AUROC: 0.8180
Epoch 14/20 - Train Loss: 0.3320, Valid Loss: 0.2475
Accuracy: 0.9021, Mean AUROC: 0.8159
Epoch 15/20 - Train Loss: 0.3322, Valid Loss: 0.2519
Accuracy: 0.8995, Mean AUROC: 0.8132
  Atelectasis: AUROC = 0.8331
  Cardiomegaly: AUROC = 0.9363
  Effusion: AUROC = 0.8172
  Infiltration: AUROC = 0.6968
  Mass: AUROC = 0.8806
  Nodule: AUROC = 0.8314
  Pneumonia: AUROC = 0.5578
  Pneumothorax: AUROC = 0.8446
  Consolidation: AUROC = 0.7488
  Edema: AUROC = 0.8829
  Emphysema: AUROC = 0.8531
  Fibrosis: AUROC = 0.7378
  Pleural_Thickening: AUROC = 0.7905
  Hernia: AUROC = 0.9738
Epoch 16/20 - Train Loss: 0.3263, Valid Loss: 0.2501
Accuracy: 0.8999, Mean AUROC: 0.8196
  New best model with AUROC: 0.8196
Epoch 17/20 - Train Loss: 0.3270, Valid Loss: 0.2522
Accuracy: 0.8982, Mean AUROC: 0.8208
  New best model with AUROC: 0.8208
Epoch 18/20 - Train Loss: 0.3247, Valid Loss: 0.2527
Accuracy: 0.8973, Mean AUROC: 0.8229
  New best model with AUROC: 0.8229
Epoch 19/20 - Train Loss: 0.3204, Valid Loss: 0.2511
Accuracy: 0.8992, Mean AUROC: 0.8203
Epoch 20/20 - Train Loss: 0.3200, Valid Loss: 0.2528
Accuracy: 0.8985, Mean AUROC: 0.8109
  Atelectasis: AUROC = 0.8250
  Cardiomegaly: AUROC = 0.8983
  Effusion: AUROC = 0.8097
  Infiltration: AUROC = 0.6734
  Mass: AUROC = 0.8837
  Nodule: AUROC = 0.8368
  Pneumonia: AUROC = 0.5605
  Pneumothorax: AUROC = 0.8420
  Consolidation: AUROC = 0.7524
  Edema: AUROC = 0.8813
  Emphysema: AUROC = 0.8693
  Fibrosis: AUROC = 0.7317
  Pleural_Thickening: AUROC = 0.8127
  Hernia: AUROC = 0.9755
Loaded best model with AUROC: 0.8229
```

Out[16]:

```
{'train_loss': [0.512840187379292,
  0.3846053685460772,
  0.3778936295849936,
  0.3703442132472992,
  0.36047095111438204,
  0.355126211302621,
  0.3526099014282227,
  0.3487398329802922,
  0.34231844799859185,
  0.34071678485189166,
  0.33843190312385557,
  0.34026975512504576,
  0.33408795050212314,
  0.33195429291043965,
  0.33215225066457477,
  0.32628397720200675,
  0.3270384621620178,
  0.3247446628979274,
  0.3204167122500283,
  0.3200161421298981],
 'valid_loss': [0.3110121432940165,
  0.3070401926835378,
  0.2932629346847534,
  0.284041223526001,
  0.27637522260348,
  0.2689167575041453,
  0.2626188385486603,
  0.2585227874914805,
  0.2542443307240804,
  0.25621569991111753,
  0.25548014442125955,
  0.2539857657750448,
  0.254738495349884,
  0.24748066345850628,
  0.25194777647654215,
  0.2500927368799845,
  0.25216431975364684,
  0.2527491796016693,
  0.25113846600055695,
  0.2527521518866221],
 'valid_accuracy': [0.8932142857142857,
  0.8932142857142857,
  0.8932142857142857,
  0.895952380952381,
  0.8963095238095238,
  0.8979761904761905,
  0.8961904761904762,
  0.8996428571428572,
  0.8988095238095238,
  0.8969047619047619,
  0.8982142857142857,
  0.8975,
  0.8990476190476191,
  0.9021428571428571,
  0.8995238095238095,
  0.8998809523809523,
```

```
 0.8982142857142857,
 0.8972619047619048,
 0.8991666666666667,
 0.898452380952381],
'best_auroc': np.float64(0.8229393220565882)}
```