# 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 [10]:
```
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 [11]:
```
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,
```

```python
                                            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 [13]:

```python
# 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 [14]:

```python
# Class counts and weights
class_counts = [
    313,  # Atelectasis
    141,  # Cardiomegaly
    341,  # Effusion
    580,  # Infiltration
    111,  # Mass
    151,  # Nodule
    45,   # Pneumonia
    141,  # Pneumothorax
    136,  # Consolidation
    62,   # Edema
    86,   # Emphysema
    117,  # Fibrosis
```

```
    114,  # Pleural_Thickening
    17    # 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 [15]:
```
# 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.Dropout(0.5),
    nn.Linear(num_ftrs, 1024),
    nn.BatchNorm1d(1024),
    nn.ReLU(),
    nn.Dropout(0.4),
    nn.Linear(1024, 512),
    nn.BatchNorm1d(512),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(512, 256),
    nn.BatchNorm1d(256),
    nn.ReLU(),
    nn.Linear(256, N_CLASSES),
    nn.Sigmoid()
)
print("Replaced classifier with a regularized version")
```

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

In [16]:
```
# 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

# 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: 4,397,326
```

In [ ]:

```
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 [ ]:

```
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:
```

```python
        model.load_state_dict(best_model_weights)
        print(f"Loaded best model with AUROC: {best_auroc:.4f}")

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

In [21]:

```python
model.to(device)
trainer(model, criterion, optimizer, trainloader, validloader, epochs=20)
```

```
Epoch 1/20 - Train Loss: 0.5966, Valid Loss: 0.5565
Accuracy: 0.8283, Mean AUROC: 0.7485
  New best model with AUROC: 0.7485
Epoch 2/20 - Train Loss: 0.4754, Valid Loss: 0.4628
Accuracy: 0.8998, Mean AUROC: 0.7735
  New best model with AUROC: 0.7735
Epoch 3/20 - Train Loss: 0.4166, Valid Loss: 0.3836
Accuracy: 0.9296, Mean AUROC: 0.7821
  New best model with AUROC: 0.7821
Epoch 4/20 - Train Loss: 0.3860, Valid Loss: 0.3253
Accuracy: 0.9321, Mean AUROC: 0.7853
  New best model with AUROC: 0.7853
Epoch 5/20 - Train Loss: 0.3663, Valid Loss: 0.2867
Accuracy: 0.9349, Mean AUROC: 0.7979
  Atelectasis: AUROC = 0.7845
  Cardiomegaly: AUROC = 0.9359
  Effusion: AUROC = 0.8918
  Infiltration: AUROC = 0.6144
  Mass: AUROC = 0.7792
  Nodule: AUROC = 0.5870
  Pneumonia: AUROC = 0.7316
  Pneumothorax: AUROC = 0.8771
  Consolidation: AUROC = 0.7660
  Edema: AUROC = 0.8860
  Emphysema: AUROC = 0.8929
  Fibrosis: AUROC = 0.7356
  Pleural_Thickening: AUROC = 0.7223
  Hernia: AUROC = 0.9664
  New best model with AUROC: 0.7979
Epoch 6/20 - Train Loss: 0.3563, Valid Loss: 0.2608
Accuracy: 0.9349, Mean AUROC: 0.8050
  New best model with AUROC: 0.8050
Epoch 7/20 - Train Loss: 0.3463, Valid Loss: 0.2464
Accuracy: 0.9357, Mean AUROC: 0.8032
Epoch 8/20 - Train Loss: 0.3379, Valid Loss: 0.2349
Accuracy: 0.9346, Mean AUROC: 0.8030
Epoch 9/20 - Train Loss: 0.3336, Valid Loss: 0.2272
Accuracy: 0.9330, Mean AUROC: 0.8035
Epoch 10/20 - Train Loss: 0.3275, Valid Loss: 0.2220
Accuracy: 0.9342, Mean AUROC: 0.8006
  Atelectasis: AUROC = 0.7756
  Cardiomegaly: AUROC = 0.9144
  Effusion: AUROC = 0.8953
  Infiltration: AUROC = 0.6231
  Mass: AUROC = 0.8531
  Nodule: AUROC = 0.6014
  Pneumonia: AUROC = 0.7590
  Pneumothorax: AUROC = 0.8651
  Consolidation: AUROC = 0.7486
  Edema: AUROC = 0.8553
  Emphysema: AUROC = 0.8983
  Fibrosis: AUROC = 0.7029
  Pleural_Thickening: AUROC = 0.7246
  Hernia: AUROC = 0.9921
Epoch 11/20 - Train Loss: 0.3320, Valid Loss: 0.2197
Accuracy: 0.9323, Mean AUROC: 0.8045
```

```
Early stopping triggered after 11 epochs
Loaded best model with AUROC: 0.8050
```

Out[21]:

```
{'train_loss': [0.5965555200204743,
  0.47537220895682175,
  0.4166333998278503,
  0.3859797182697063,
  0.3662604146103888,
  0.356332472909617,
  0.34631310171248336,
  0.3378532878004779,
  0.33355066670830436,
  0.32749856841090613,
  0.33195252366050987],
 'valid_loss': [0.5564775163332621,
  0.4627814706961314,
  0.38360670224825544,
  0.3252780273755391,
  0.2866893316109975,
  0.26078374139467875,
  0.24642280383904774,
  0.23487487602233886,
  0.22722331881523133,
  0.2219768660068512,
  0.2197488392194112],
 'valid_accuracy': [0.8282857142857143,
  0.8998095238095238,
  0.9296190476190476,
  0.9320952380952381,
  0.9348571428571428,
  0.9348571428571428,
  0.9357142857142857,
  0.9345714285714286,
  0.932952380952381,
  0.9341904761904762,
  0.9322857142857143],
 'best_auroc': np.float64(0.80495699487945)}
```

In [22]:

```python
# Evaluate the model on the test set
gt = torch.FloatTensor().to(device)
pred = torch.FloatTensor().to(device)

# Switch to evaluation mode
model.eval()

for i, (inp, target) in enumerate(test_loader):
    target = target.to(device)
    gt = torch.cat((gt, target), dim=0)

    bs, n_crops, c, h, w = inp.size()

    with torch.no_grad():
        input_var = inp.view(-1, c, h, w).to(device)
        output = model(input_var)

    output_mean = output.view(bs, n_crops, -1).mean(1)
    pred = torch.cat((pred, output_mean), dim=0)
```

```python
# Evaluate AUROC
AUROCs = compute_AUCs(gt, pred)
AUROC_avg = np.array(AUROCs).mean()

print(f'\n✅ Average AUROC: {AUROC_avg:.3f}')
for i in range(N_CLASSES):
    print(f'AUROC for {CLASS_NAMES[i]}: {AUROCs[i]:.3f}')
```

```
✅ Average AUROC: 0.803
AUROC for Atelectasis: 0.794
AUROC for Cardiomegaly: 0.876
AUROC for Effusion: 0.910
AUROC for Infiltration: 0.658
AUROC for Mass: 0.848
AUROC for Nodule: 0.558
AUROC for Pneumonia: 0.654
AUROC for Pneumothorax: 0.921
AUROC for Consolidation: 0.804
AUROC for Edema: 0.917
AUROC for Emphysema: 0.850
AUROC for Fibrosis: 0.796
AUROC for Pleural_Thickening: 0.653
AUROC for Hernia: 0.996
```