EECS445 Project 2

uniqname: duanzq

1 Data Preprocessing

(a) i.

```
    def fit(self, X):
        """Calculate per-channel mean and standard deviation from dataset X."""
        # TODO: Complete this function
        X = np.array(X)
        self.image_mean = np.array([np.mean(X[:,:,:,0]), np.mean(X[:,:,:,1]),
np.mean(X[:,:,:,2])])
        self.image_std = np.array([np.std(X[:,:,:,0]), np.std(X[:,:,:,1]),
np.std(X[:,:,:,2])])

    def transform(self, X):
        """Return standardized dataset given dataset X."""
        # TODO: Complete this function
        return (X - self.image_mean) / self.image_std
```

	Channel R	Channel G	Channel B
Mean	123.084	117.488	93.312
STD	62.729	59.178	61.434

ii. The reason is that only the training set is used to train the model. It implies that images in other data partitions can not be accessed in the training process.

And the purpose of this standardization is to ensure that each input has a similar range, since inputs are multiplied by weights and added by biases and then gradients are calculated to backpropagate when training.

(b)

The visible effects of preprocessing on the image data:

1. the sharpness decreases so the preprocessed pictures become vaguer and smoother
2. the brightness is balanced so the preprocessed pictures can not be too dark

2 Convolutional Neural Networks

(a) Number of float-valued parameters: 39754

For the Convolutional Layer 1, #parameters = $(3 \times 5 \times 5 + 1) \times 16 = 1216$

For the Convolutional Layer 2, #parameters = $(16 \times 5 \times 5 + 1) \times 64 = 25664$

For the Convolutional Layer 3, #parameters = $(64 \times 5 \times 5 + 1) \times 8 = 12808$

For the Fully connected layer 1, #parameters = $(32 + 1) \times 2 = 66$

Thus, totally there are 39754 learnable parameters.

(b)

```
class Target(nn.Module):
    def __init__(self):
        super().__init__()

        ## TODO: define each layer

        self.conv1 = nn.Conv2d(3, 16, kernel_size=(5,5), stride=(2,2),
padding=2)
        self.pool = nn.MaxPool2d(kernel_size=(2,2), stride=(2,2), padding=0)
        self.conv2 = nn.Conv2d(16, 64, kernel_size=(5,5), stride=(2,2),
padding=2)
        self.conv3 = nn.Conv2d(64, 8, kernel_size=(5,5), stride=(2,2),
padding=2)
        self.fc_1 = nn.Linear(32, 2)

        ##

        self.init_weights()

    def init_weights(self):
        torch.manual_seed(42)
```

```
        for conv in [self.conv1, self.conv2, self.conv3]:
            C_in = conv.weight.size(1)
            nn.init.normal_(conv.weight, 0.0, 1 / sqrt(5 * 5 * C_in))
            nn.init.constant_(conv.bias, 0.0)

        ## TODO: initialize the parameters for [self.fc_1]
        nn.init.normal_(self.fc_1.weight, 0.0, 1 / sqrt(32))
        nn.init.constant_(self.fc_1.bias, 0.0)
        ##

    def forward(self, x):
        N, C, H, W = x.shape

        ## TODO: forward pass
        z = F.relu(self.conv1(x))
        z = self.pool(z)
        z = F.relu(self.conv2(z))
        z = self.pool(z)
        z = F.relu(self.conv3(z))
        z = torch.reshape(z, (N, 32))
        z = self.fc_1(z)
        ##
        return z
```

(c)

```
def predictions(logits):
    """Determine predicted class index given logits.

    Returns:
        the predicted class output as a PyTorch Tensor
    """
    # TODO implement predictions

    pred = torch.argmax(logits, dim=1)
    return pred
```

(d)

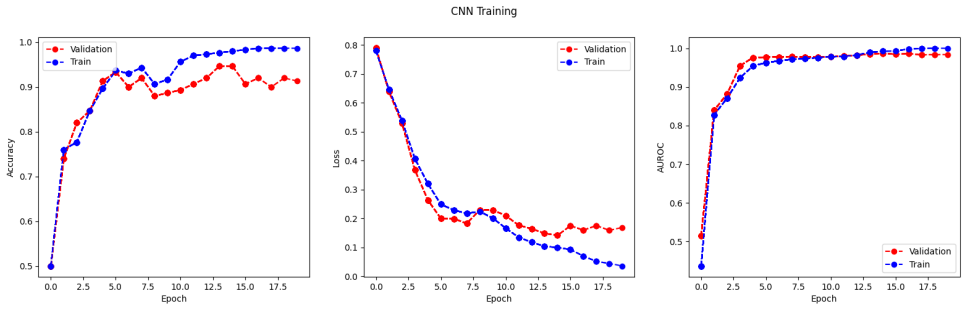
```
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
```

(e)

```
def early_stopping(stats, curr_patience, prev_val_loss):
    """Calculate new patience and validation loss.

    Increment curr_patience by one if new loss is not less than prev_val_loss
    Otherwise, update prev_val_loss with the current val loss

    Returns: new values of curr_patience and prev_val_loss
    """
    # TODO implement early stopping
    curr_val_loss = stats[-1][1]
```

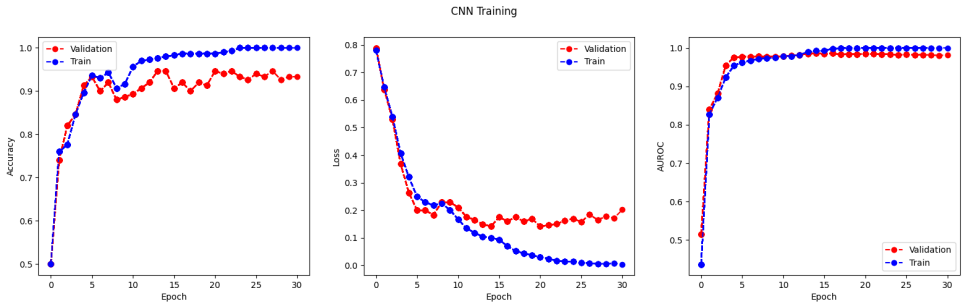
```
    if curr_val_loss >= prev_val_loss:
        curr_patience += 1
    else:
        curr_patience = 0
        prev_val_loss = curr_val_loss
    #
    return curr_patience, prev_val_loss

def train_epoch(data_loader, model, criterion, optimizer):
    """Train the `model` for one epoch of data from `data_loader`.

    Use `optimizer` to optimize the specified `criterion`
    """
    for i, (X, y) in enumerate(data_loader):
        # TODO implement training steps
        optimizer.zero_grad()

        y_pred = model(X)
        loss = criterion(y_pred, y)
        loss.backward()
        optimizer.step()
```

(f) i.



When preprocessing the data, the mean and std of RGB channels of training set are applied to both training and validation sets. And noise may occur in the algorithm for SGD when the step size is too large. And another noise source may come from overfitting: when epoch increases, train loss will keep decreasing and approach zero, while validation loss will stop decreasing at some point and may even increase as training goes on.

ii.

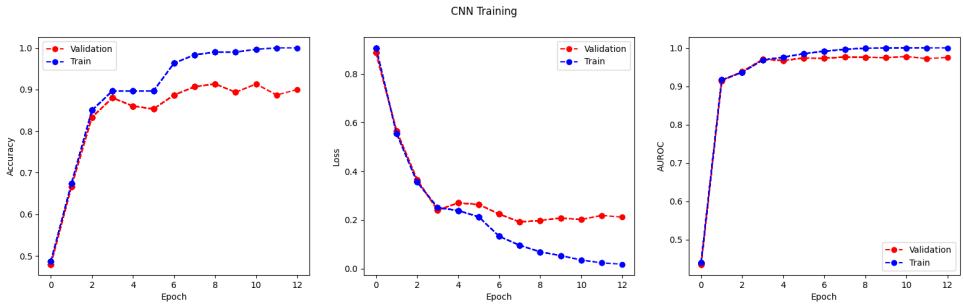


The model (patience:10) stops training at epoch 30.

Patience value 5 works better. The reason is that when patience = 10 and epoch > 15, as the epoch increases, the train loss will decrease but the validation loss will not decrease, which implies overfitting. Thus, patience value 5 is enough for this problem.

Increased patience might be better if the model is underfitting with current patience. That is increased patience can lead to better performance in the validation set. To be more specific, consider the case that when patience is small, the algorithm may stop at local min validation loss. While the increased patience can skip the local min and find the global min.

iii.



The new size of the input to the fully connected layer: 128

Update:

```
class Target(nn.Module):
    def __init__(self):
        super().__init__()

        ## TODO: define each layer

        self.conv1 = nn.Conv2d(3, 16, kernel_size=(5,5), stride=(2,2),
padding=2)
        self.pool = nn.MaxPool2d(kernel_size=(2,2), stride=(2,2), padding=0)
        self.conv2 = nn.Conv2d(16, 64, kernel_size=(5,5), stride=(2,2),
padding=2)
        self.conv3 = nn.Conv2d(64, 32, kernel_size=(5,5), stride=(2,2),
padding=2)
        self.fc_1 = nn.Linear(128, 2)
        ##

        self.init_weights()

    def init_weights(self):
        torch.manual_seed(42)

        for conv in [self.conv1, self.conv2, self.conv3]:
            C_in = conv.weight.size(1)
            nn.init.normal_(conv.weight, 0.0, 1 / sqrt(5 * 5 * C_in))
            nn.init.constant_(conv.bias, 0.0)

        ## TODO: initialize the parameters for [self.fc_1]
        nn.init.normal_(self.fc_1.weight, 0.0, 1 / sqrt(32))
        nn.init.constant_(self.fc_1.bias, 0.0)
        ##

    def forward(self, x):
        N, C, H, W = x.shape

        ## TODO: forward pass
        z = F.relu(self.conv1(x))
```

```
        z = self.pool(z)
        z = F.relu(self.conv2(z))
        z = self.pool(z)
        z = F.relu(self.conv3(z))
        z = torch.reshape(z, (N, 128))
        z = self.fc_1(z)
        ##
        return z
```

	Epoch	Training AUROC	Validation AUROC
8 filters	14	0.9924	0.9867
32 filters	7	0.9964	0.976

When the number of filters is increased from 8 to 32, the Training AUROC performs better in fewer Epoch, but the Validation AUROC becomes worse. A possible reason is that when #filters is increased, more unrelated things are learned in the model, which leads to overfitting. Then it works better in the training set while worse in the validation set, since these unrelated things will only make classification worse.

(g)

	Training	Validation	Testing
Accuracy	0.98	0.9467	0.55
AUROC	0.9924	0.9867	0.6288

i. Since the training accuracy and auroc are just slightly larger than validation accuracy and auroc, there is no evidence of overfitting here.

ii. The trend is that the training and validation performance are always much better than the test performance.

A possible explanation is that the number of samples in the training and validation set is not large enough compared to the testing set. As a result, the model might overfit the training set and fail to correctly classify the testing set. Another explanation is that images in the testing set may be hugely different from images in the training and validation set in shooting angle, background, and so on. What's more, the model may use totally unrelated things to identify different dogs.

3 Visualizing what the CNN has learned

(a)

$$\alpha_k^c = \frac{1}{Z}\sum_i\sum_j \frac{\partial y^c}{\partial A_{ij}^k}$$

$$\Rightarrow \alpha_1^1 = \frac{1}{16}(-1+1-2-1-1+1+1+1+2+2) = \frac{3}{16}$$

$$\text{and } \alpha_2^1 = \frac{1}{16}(1+2+2+2+2+1+1-1-2-1) = \frac{7}{16}$$

$$\text{Then } L^c_{\text{Grad-CAM}} = \text{ReLU}\left(\sum_k \alpha_k^c A^k\right)$$

$$= \text{ReLU}(\frac{3}{16}\begin{bmatrix}1 & 1 & 2 & 1\\ 1 & 2 & 1 & 0\\ 0 & 1 & 0 & -1\\ 0 & 1 & -2 & -2\end{bmatrix} + \frac{7}{16}\begin{bmatrix}1 & 1 & 1 & 1\\ 2 & 2 & 2 & 2\\ 2 & 2 & 1 & 0\\ -1 & -1 & -1 & 0\end{bmatrix}) = \begin{bmatrix}\frac{5}{8} & \frac{5}{8} & \frac{13}{16} & \frac{5}{8}\\ \frac{17}{16} & \frac{5}{4} & \frac{17}{16} & \frac{7}{8}\\ \frac{7}{8} & \frac{17}{16} & \frac{7}{16} & 0\\ 0 & 0 & 0 & 0\end{bmatrix}$$

(b) CNN appears to use ears, eyes, legs, body pattern, shadows and even background grass or trees to identify the Collie class.

(c) Yes, it confirms my hypothesis. The CNN is overfitting and uses unrelated things to identify different dog breeds.

4 Transfer Learning & Data Augmentation

4.1 Transfer Learning

(a)

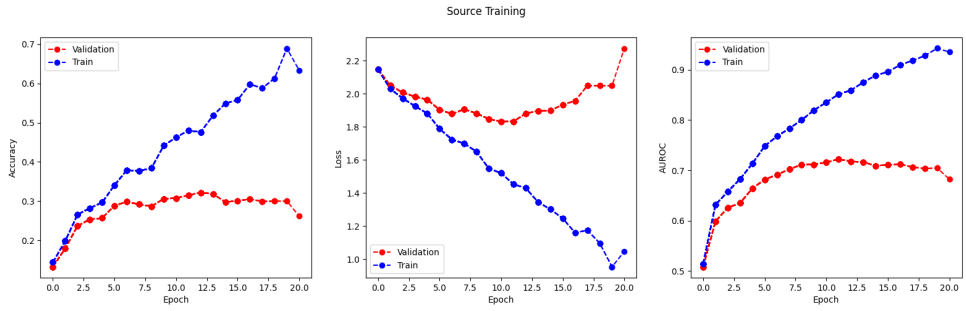
```
class Source(nn.Module):
    def __init__(self):
        super().__init__()

        ## TODO: define each layer
        self.conv1 = nn.Conv2d(3, 16, kernel_size=(5,5), stride=(2,2),
padding=2)
        self.pool = nn.MaxPool2d(kernel_size=(2,2), stride=(2,2), padding=0)
        self.conv2 = nn.Conv2d(16, 64, kernel_size=(5,5), stride=(2,2),
padding=2)
        self.conv3 = nn.Conv2d(64, 8, kernel_size=(5,5), stride=(2,2),
padding=2)
        self.fc1 = nn.Linear(32, 8)
        ##

        self.init_weights()

    def init_weights(self):
        torch.manual_seed(42)
        for conv in [self.conv1, self.conv2, self.conv3]:
            C_in = conv.weight.size(1)
            nn.init.normal_(conv.weight, 0.0, 1 / sqrt(5 * 5 * C_in))
            nn.init.constant_(conv.bias, 0.0)

        ## TODO: initialize the parameters for [self.fc1]
        nn.init.normal_(self.fc1.weight, 0.0, 1 / sqrt(32))
        nn.init.constant_(self.fc1.bias, 0.0)
        ##

    def forward(self, x):
        N, C, H, W = x.shape

        ## TODO: forward pass
        z = F.relu(self.conv1(x))
        z = self.pool(z)
```

```
        z = F.relu(self.conv2(z))
        z = self.pool(z)
        z = F.relu(self.conv3(z))
        z = torch.reshape(z, (N, 32))
        z = self.fc1(z)
        ##

        return z
```

(b)

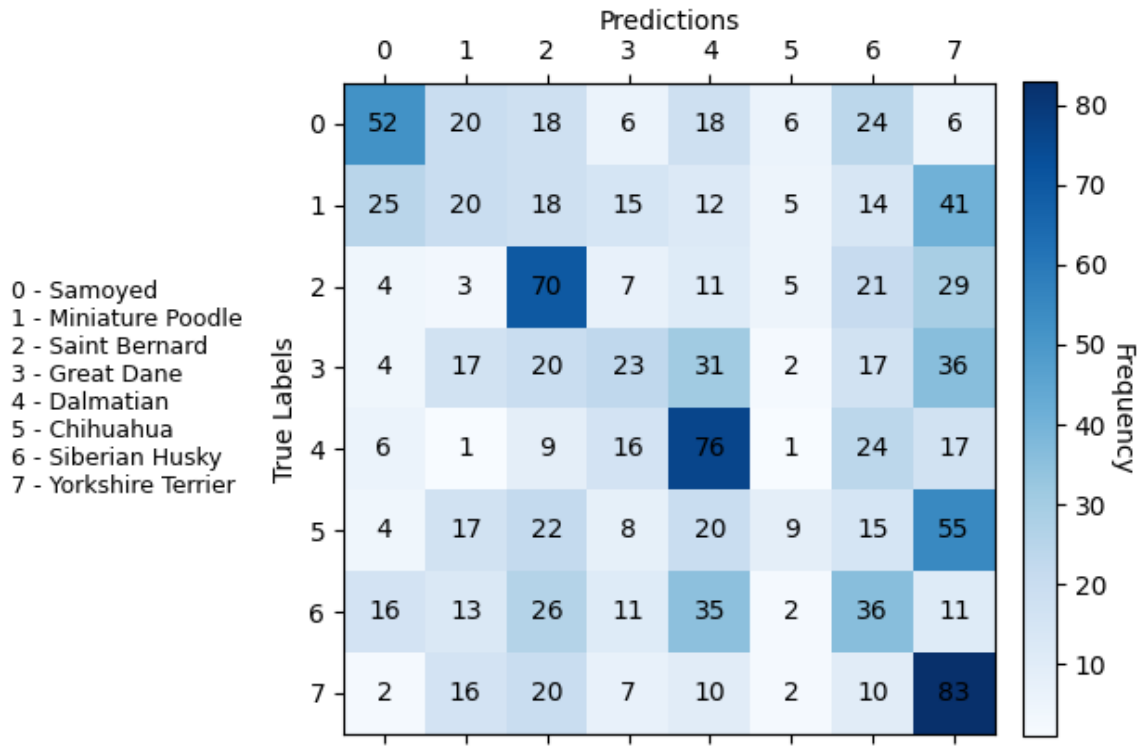
```
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3, weight_decay=0.01)
```

(c)



The epoch number with the lowest validation loss: 10

(d)



The classifier is the most accurate for Yorkshire Terrier and it is the least accurate for Chihuahua.

The possible reason is that Yorkshire has some special features that other dogs do not have, while Chihuahua does not have any special features and is likely to be classified as other dog breeds.

(e)

```
    #TODO: define loss function, and optimizer
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

    #TODO: patience for early stopping
    patience = 5
    curr_patience = 0
```

```
def freeze_layers(model, num_layers=0):
    """Stop tracking gradients on selected layers."""
    #TODO: modify model with the given layers frozen
    #      e.g. if num_layers=2, freeze CONV1 and CON2
    #      Hint: https://pytorch.org/docs/master/notes/autograd.html

    count = 0
    for param in model.parameters():
        if count == 2*num_layers:
            break
        param.requires_grad = False
        count += 1
```

(f)

	AUROC	AUROC	AUROC
	TRAIN	VAL	TEST
Freeze all CONV layers (Fine-tune FC layer)	0.827	0.8542	0.7832
Freeze first two CONV layers (Fine-tune last CONV and FC layers)	0.9892	0.976	0.7532
Freeze first CONV layer (Fine-tune last 2 conv. and fc layers)	0.9998	0.9828	0.7588
Freeze no layers (Fine-tune all layers)	0.9912	0.9808	0.7252
No Pretraining or Transfer Learning (Section 2 performance)	0.9924	0.9867	0.6288

The transfer learning improves the TEST AUROC by at least ≈0.1 when no layers are frozen and at most ≈0.15 when all CONV layers are frozen. When more CONV layers are frozen, both TRAIN and VAL AUROC decrease, while TEST AUROC increases. Thus, the source task was helpful. It can provide learned and "meaningful" parameters to initialize conv.weight and conv.bias.

Compared to freezing just a subset, the training curve generated by freezing all convolutional layers contains more epochs, TEST loss vibrates less violently and its values become smaller, and both values of TEST accuracy and TEST auroc increase. And that changes are more obvious when comparing freezing all convolutional layers versus freezing none. The reason is that when more

CONV layers are frozen, fewer parameters are trainable, leading to more epochs to terminate the algorithm and less possibility of overfitting to target training set.

4.2 Data Augmentation

(a)

```
def Rotate(deg=20):
    """Return function to rotate image."""

    def _rotate(img):
        """Rotate a random amount in the range (-deg, deg).

        Keep the dimensions the same and fill any missing pixels with black.

        :img: H x W x C numpy array
        :returns: H x W x C numpy array
        """
        # TODO
        angle = np.random.uniform(-deg, deg)
        rotated_img = rotate(img, angle, reshape=False)
        return rotated_img.astype('uint8')

    return _rotate


def Grayscale():
    """Return function to grayscale image."""

    def _grayscale(img):
        """Return 3-channel grayscale of image.

        Compute grayscale values by taking average across the three channels.

        :img: H x W x C numpy array
        :returns: H x W x C numpy array

        """
        # TODO
        H, W, C = img.shape
        grayscale_img = np.zeros(img.shape)
        avg = np.average(img, axis=2)
        for i in range(C):
            grayscale_img[:,:,i] = avg
        return grayscale_img.astype('uint8')

    return _grayscale
```

(b) i.

For Rotation (keep original):

```
    augmentations = [Rotate()]
    imgs = augment(
            f"{args.datadir}/images/{row['filename']}",
            augmentations,
            n=1,
            original=True,  # TODO: change to False to exclude original image.
     )
```

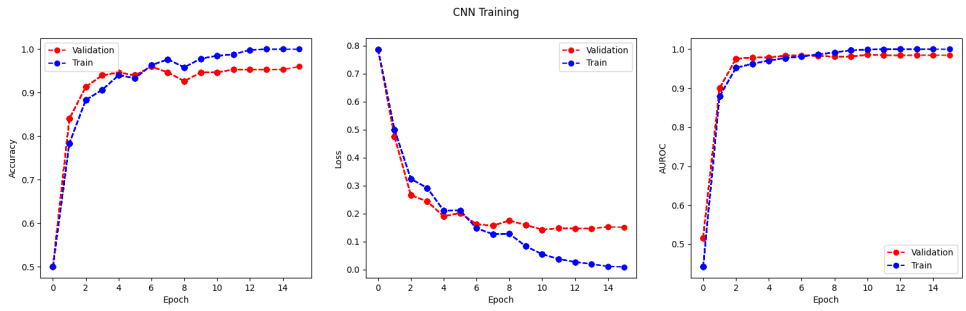
For Grayscale (keep original):

```
    augmentations = [Grayscale()]
    imgs = augment(
            f"{args.datadir}/images/{row['filename']}",
            augmentations,
            n=1,
            original=True,  # TODO: change to False to exclude original image.
     )
```

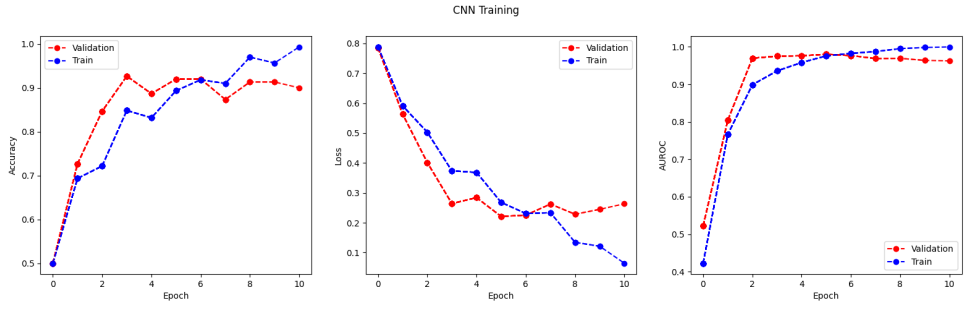
For Grayscale (discard original)

```
    augmentations = [Grayscale()]
    imgs = augment(
            f"{args.datadir}/images/{row['filename']}",
            augmentations,
            n=1,
            original=False,  # TODO: change to False to exclude original image.
     )
```

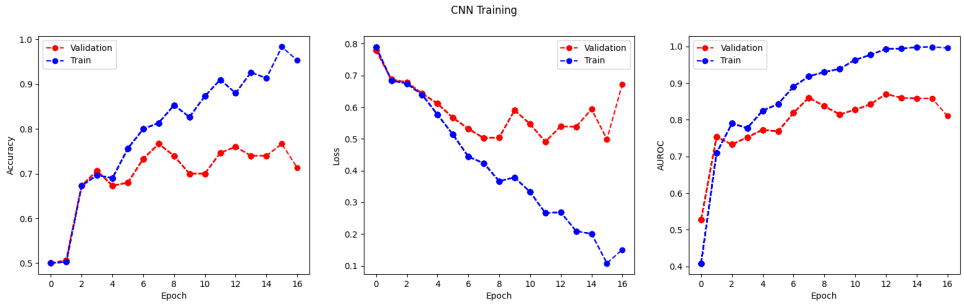
iii.



Rotation (keep original)



Grayscale (keep original)



Grayscale (discard original)

	AUROC	AUROC	AUROC
	TRAIN	VAL	TEST
Rotation (keep original)	0.9978	0.9851	0.6356
Grayscale (keep original)	0.9862	0.9787	0.7208
Grayscale (discard original)	0.9606	0.8457	0.762
No augmentation (Section 2 performance)	0.9924	0.9867	0.6288

(c)

When Rotation (keep original) is applied, the validation and training plots do not change much from the original one. The possible reason is that this augmentation technique just provides basic random rotation but the combination of pixels in the images does not change much. Applying affine transformations may be better since it can simulate taking photos from different shooting angles.

When Grayscale (keep original) is applied, the algorithm terminated in fewer epochs, and the validation loss stops decreasing earlier than the original. It implies this augmentation technique helps to prevent overfitting. The possible reason is that grayscale images discard the effect of color and just focus on edges and shapes. It can help to prevent the CNN from learning some unrelated things such as background grass (green) and so on.

When Grayscale (discard original) is applied, the validation performance becomes much worse than the training performance after just several epochs. And the validation loss oscillates viciously at the end. The possible reason is that this augmentation technique totally prevents the CNN to learn anything from color.