Project 2 - Nicole's Neighborhood Dog

1 Data PreProcessing

a) (i) Mean = [123.968, 117.887, 93.942]
Standard Deviation = [63.033, 59.667, 61.949]
(ii) We wish to eliminate confounding factors such as brightness, contrast, darkness etc. when training our model, and so we need to extract the mean and standard deviation and apply these values in our transform function to decrease variability in our model. In a real-world example, we wouldn't have the values from the validation and test set like we do here and instead would simply have to run the model on the images. Hence, we have to use our own test data's mean and standard deviation for our model to train on.

b)

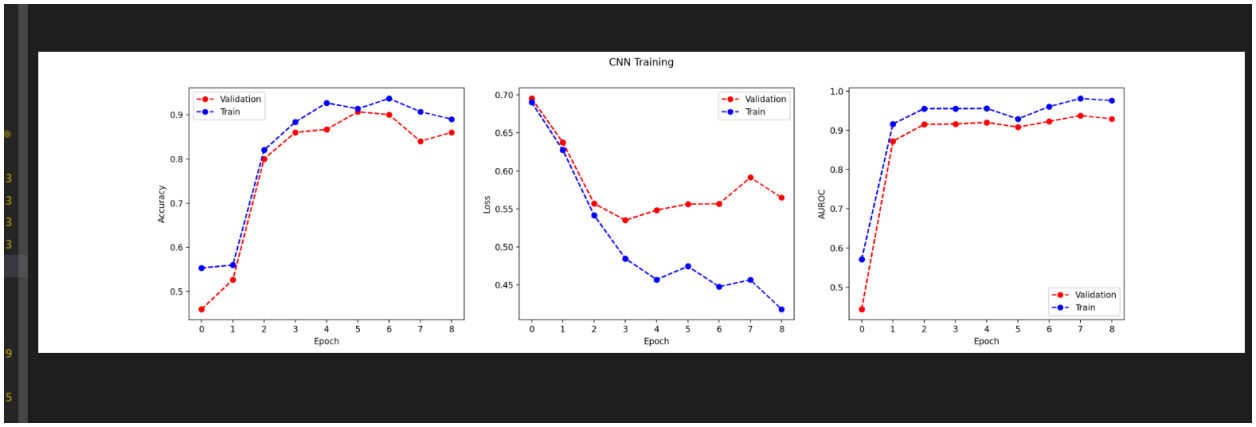
2 Convolutional Neural Networks

a) Input * output + bias
1st convolutional layer = 3 * 16 * 5 * 5 + 16 = 1216
2nd convolutional layer = 16 * 64 * 5 * 5 + 64 = 25664
3rd convolutional layer = 64 * 8 * 5 * 5 + 8 = 12808
Output layer = 32 * 2 = 64

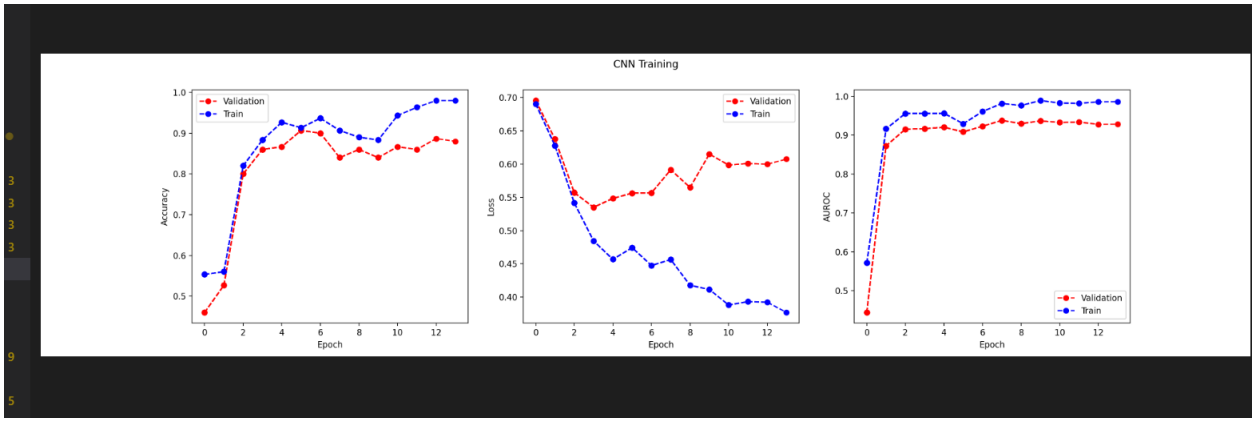
Total learnable parameters = 39752
From running code, number of float-valued parameters = 39754

b) Refer to Code Appendix
c) Refer to Code Appendix
d) Refer to Code Appendix
e) Refer to Code Appendix
f) (i) One reason is that the model is overfitting, thus becoming extremely good at classifying the training data but generalizing poorly and causing the classification of the validation data to become worse. Another reason that could be causing this is that the AUROC score for the validation data approaches 1 after the 2nd epoch, so the model might be thinking it is predicting everything perfectly when in fact it isn't.

(ii) When patience = 5, epoch = 8
Training Plot:



When patience = 10, epoch = 13
Training Plot:



(iii) new input size of FC layer = 64 * 2 * 2 = 256

Updated architecture: input features in FC change from 32 to 256
And output features in conv3 change from 8 to 64

```
## TODO: define each layer
#change padding using formula
self.conv1 = torch.nn.Conv2d(in_channels = 3, out_channels = 16, kernel_size = 5, stride = 2, padding = 2)
self.pool = torch.nn.MaxPool2d(kernel_size = 2, stride = 2, padding = 0)
self.conv2 = torch.nn.Conv2d(in_channels = 16, out_channels = 64, kernel_size = 5, stride = 2, padding = 2)
self.conv3 = torch.nn.Conv2d(in_channels = 64, out_channels = 64, kernel_size = 5, stride = 2, padding = 2)
self.fc_1 = torch.nn.Linear(in_features = 256, out_features = 2)
##
```

	Epoch	Training AUROC	Validation AUROC
8 filters	3	0.9556	0.9164
64 filters	2	0.975	0.9362

g) (i)

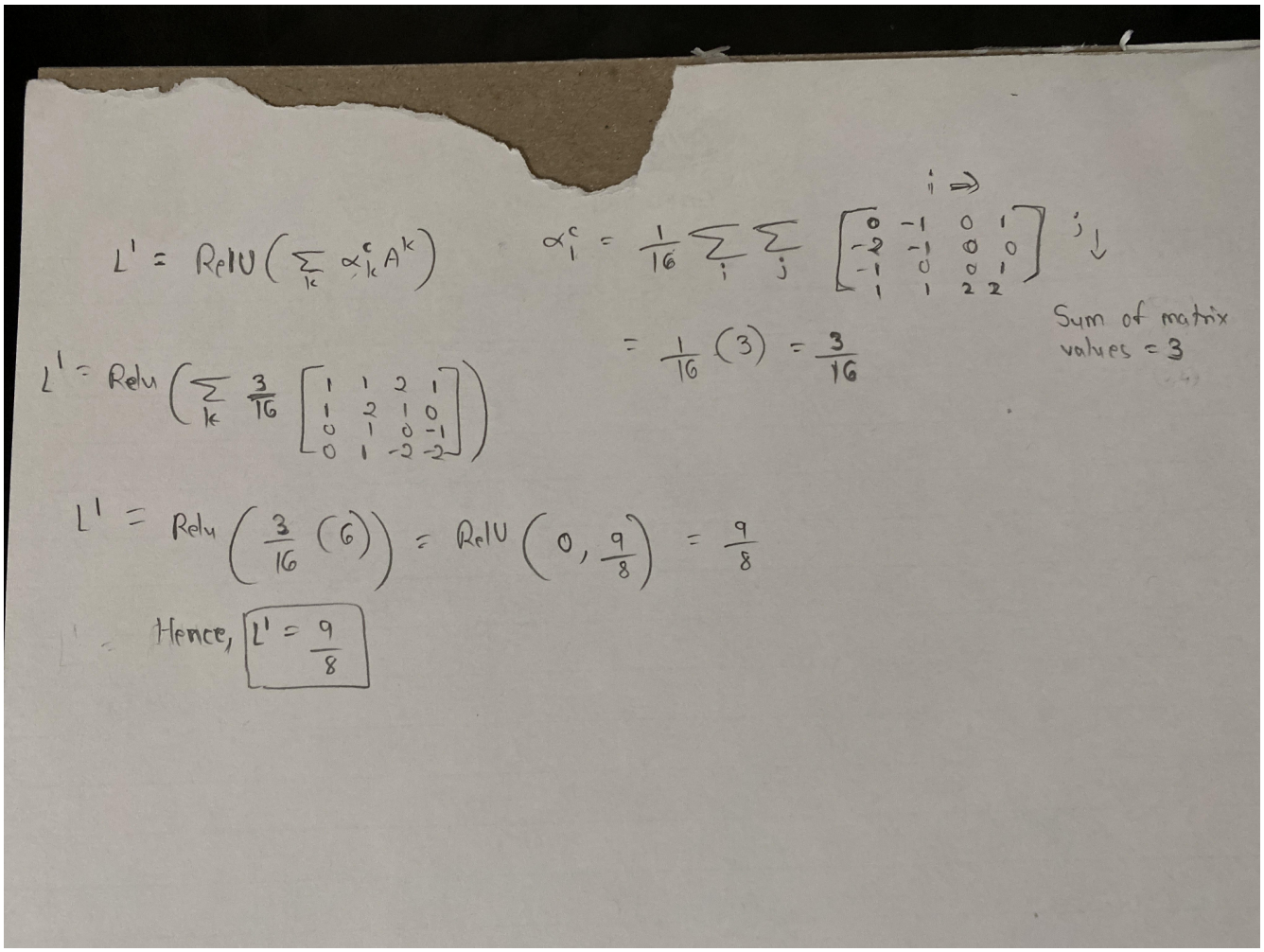
	Training	Validation	Testing
Accuracy	0.8833	0.86	0.7
AUROC	0.9556	0.9164	0.69

(ii) There isn't a huge difference in the training and auroc score in the validation and training data sets, hence the model isn't overfitting by much.

(iii) The test accuracy and auroc scores are far worse than the validation accuracy and auroc scores, which implies significant overfitting. The trend observed here is that the model does worse for the validation data than the training data, and far worse for the test data than the training data and validation data. It could mean that the model is bad at generalizing out-of-sample data or that our data isn't varied enough and that the model is failing when it comes to certain types of images with higher than usual contrast or brightness which were factors we tried to eliminate in our pre-processing section.

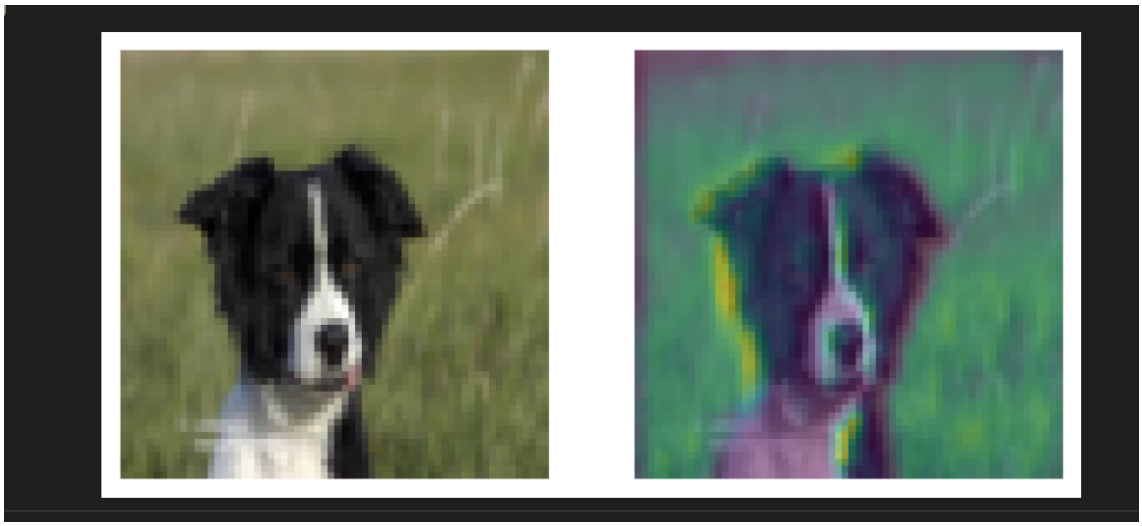
3 Visualizing What The CNN Has Learned

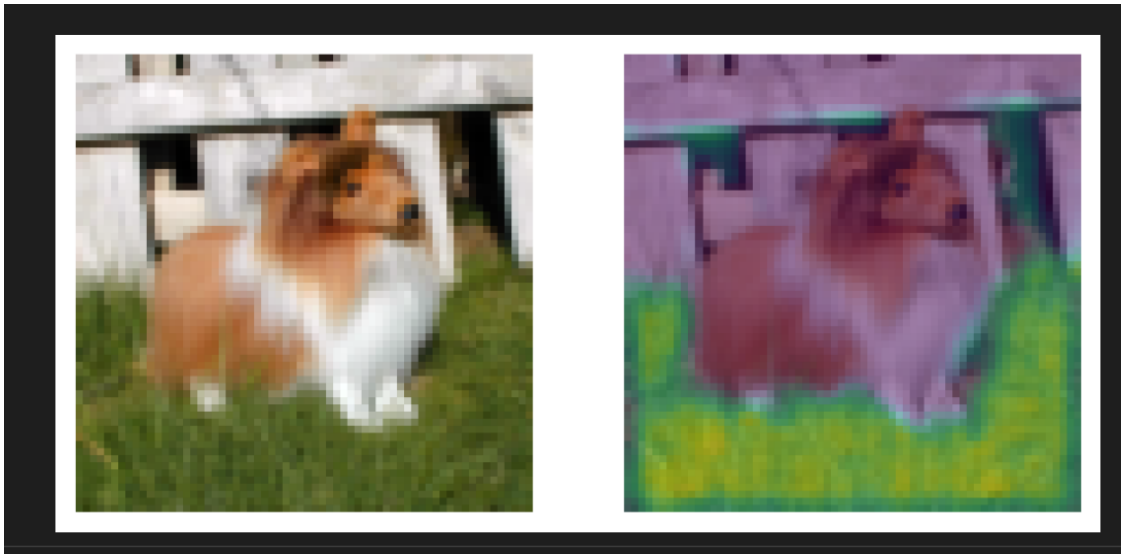
a) L1 = 9/8



b) Some images indicate that the CNN is tracing the outline of the image of a god whereas some images suggest it is analyzing the background. There doesn't seem to be much facial recognition of the dog, but rather the outline and the background in how it's distinguishing the different dog breeds. Doesn't seem like a very effective method.

c) The surroundings seem to play a significant role in the model's ability to distinguish the dog which seems odd. Also, Brightness and contrast do seem to play a certain role in the classification process, because the model seems to be doing better on images that have lower brightness/ that are darker.
For example, image 1,2 and image 2 show how the model seems to identify the grass around the dogs even more than the outline of the dogs.

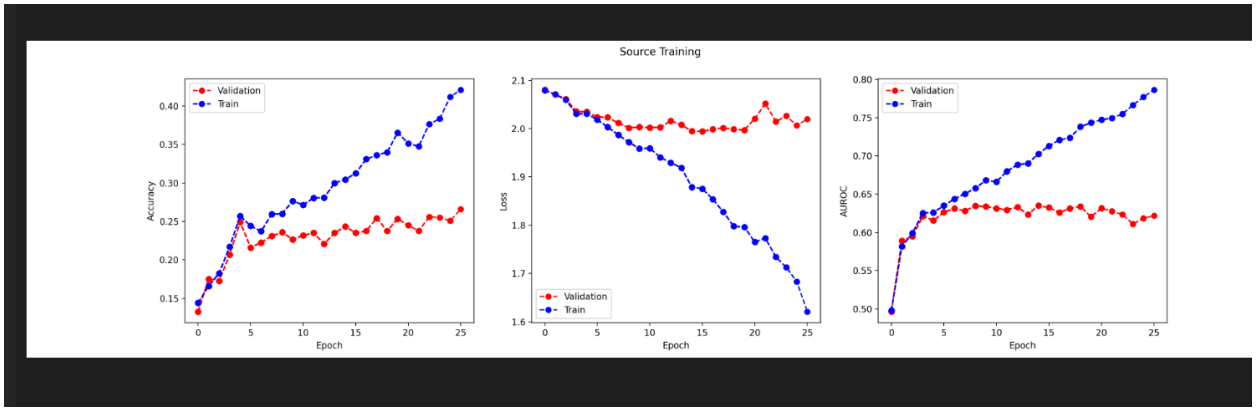




4 Transfer Learning And Data Augmentation

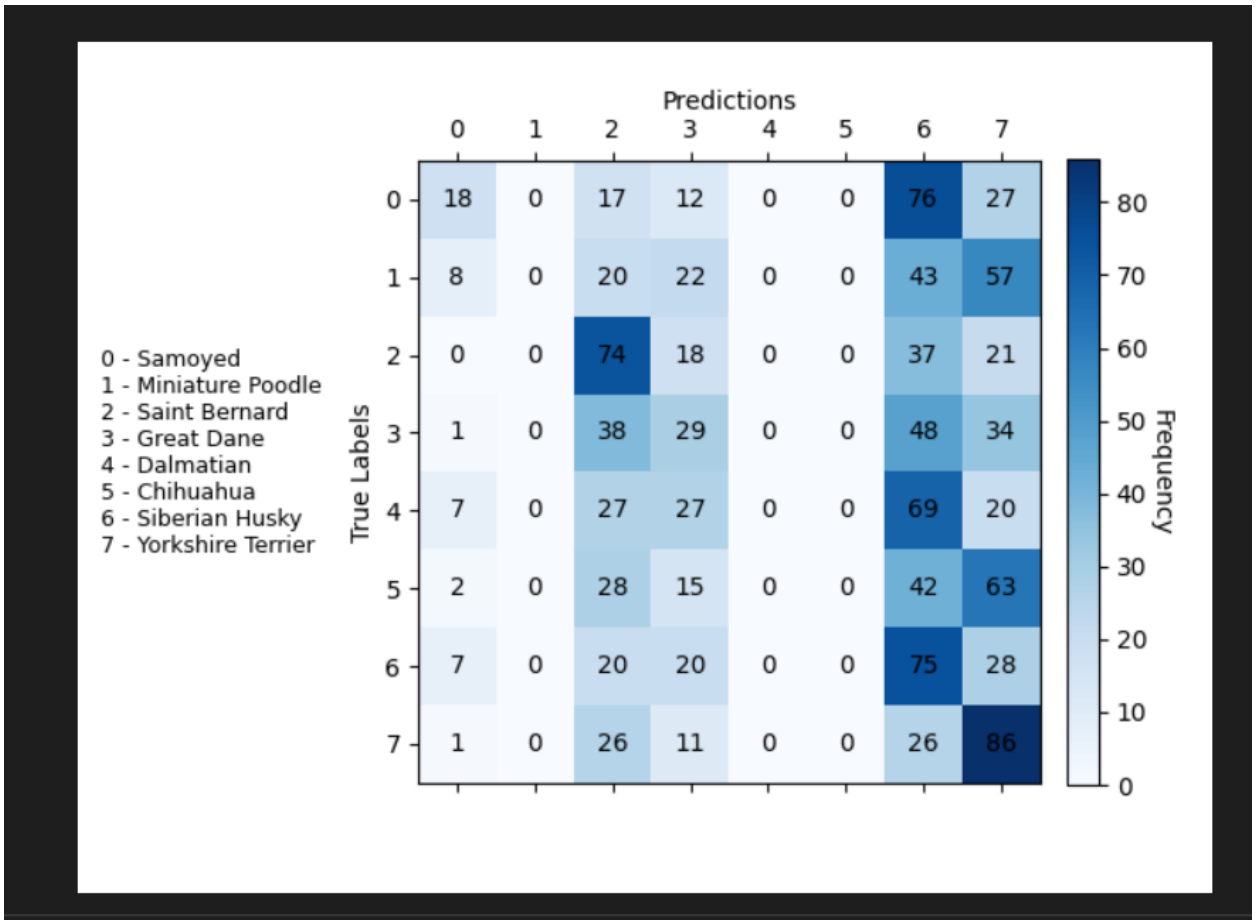
4.1 Transfer Learning

a) Refer to Code Appendix
b) Refer to Code Appendix
c) Epoch = 15 where validation loss (lowest) = 1.9941



d) The breed Yorkshire Terrier is the one the model performs the most accurately on with a true label and prediction score of 86, meaning it is performing extremely well on this particular breed. It is the least accurate with scores of 0 on Miniature Poodle, Dalmatian, and Chihuahua. Maybe because miniature poodles and chihuahuas are smaller and our model is more prone to analyzing the background of the dog, it isn't able to accurately distinguish between smaller dogs. For a large

dog breed like the dalmatian, it is interesting why it fails. Perhaps because many images of dalmatians involve mostly the face and body of the dalmatian and not much background and our model doesn't seem to be too keen on facial recognition, hence it also performs worse on dalmatians. Yorkshire terriers probably have the right balance of background features, brightness, and contrast that our model is able to easily distinguish.



e) Refer to Code Appendix
f) Run train target.py

	AUROC		
	Train	Val	Test
Freeze all CONV layers (Fine-tune FC layer)			
Freeze first two CONV layers (Fine-tune last CONV and FC layers)			

Freeze first CONV layer (Fine-tune last 2 conv. and fc layers)			
Freeze no layers (Fine-tune all layers)			
No Pre Training or Transfer Learning (Section 2(g) performance)			

4.2 Data Augmentation

a) Refer to Code Appendix
b) Experiment on augmented data
(i)
(ii) When only rotate:

```
augmentations = [Rotate()] #Grayscale(), Rotate()
```

```
original=True,  #
```

When only grayscale:

```
augmentations = [Grayscale()] #Grayscal
```

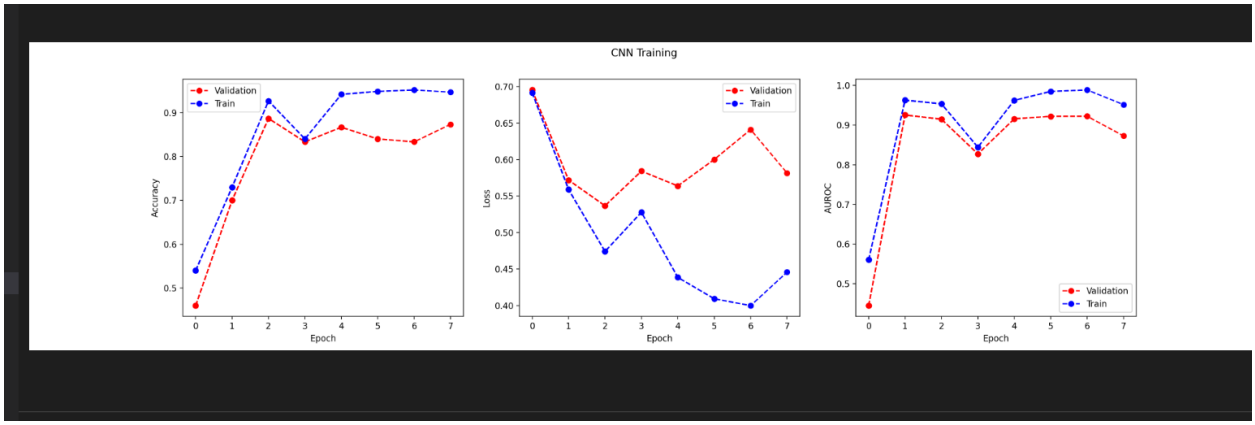
```
original=True,  #
```

When only grayscale and original set to false:

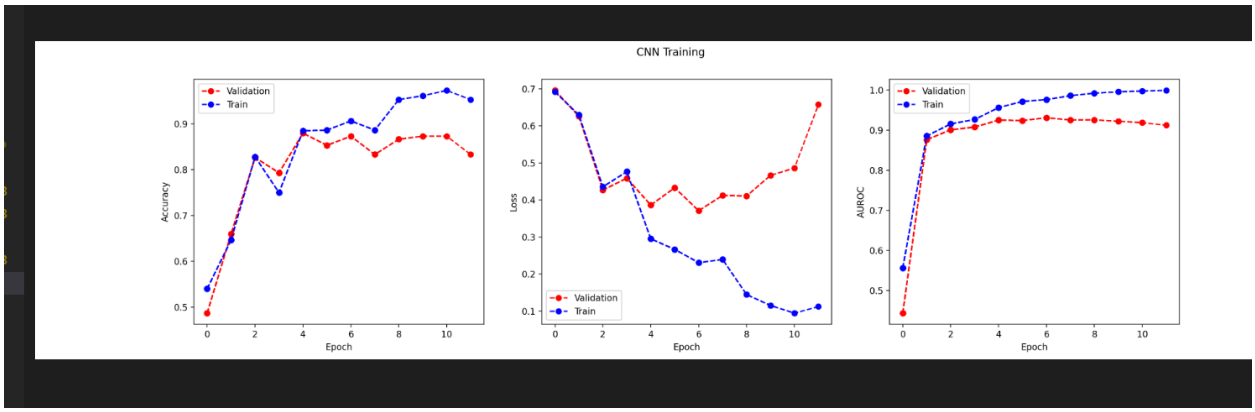
```
augmentations = [Grayscale()] #Grayscal
```

```
    n=1,
    original=False,  # TODO:
)
```

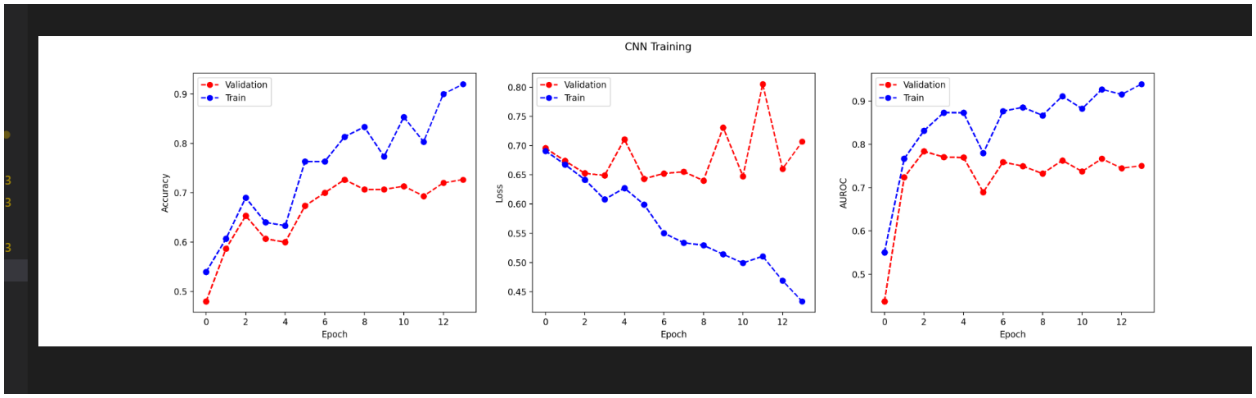
(iii)
Rotate CNN training plot:



Grayscale (with original) training plot:



Grayscale (without original) training plot:



	AUROC		
	Train	Val	Test
Rotation (keep original)	0.953	0.9205	0.6396
Grayscale (keep original)	0.9925	0.9285	0.72
Grayscale (discard original)	0.8696	0.7379	0.722
No augmentation (Section 2(g) performance)	0.9556	0.9164	0.69

c) Grayscale with original set to false does the worst whereas grayscale with original set to true does the best in the train and validation auroc scores. However, grayscale with discarded original has the highest auroc score. From the graphs, we can see how the loss for grayscale images seems to fluctuate a lot more, but it also has the best accuracy. Overall, grayscale with original set to true does the best. Grayscale probably does better than rotate because making all the colors the same results in less features/ feature variance for the CNN to have to figure out and hence it performs better. Rotate performs about the same or slightly worse than with no augmentation, and it makes sense since rotating the image and filling the missing pixels with black isn't changing many features for the CNN model. Grayscale , however, produces a noticeable difference.

5 Challenge

Setting a weight decay corresponds to setting this parameter. If you set it to a high value, the network does not care so much about correct predictions on the training set and rather

keeps the weights low, hoping for good generalization performance on the unseen data. Hence, I set the weight decay to be 0.09, the same as Architecture C, because I didn't want my model to overfit, but at the same time, it didn't want it to underfit by setting too high of a weight. I also added an additional convolutional layer because I feel that the problem the model was facing when running on the test data is that it wasn't able to identify enough features and failed to distinguish between different breeds. Adding another convolutional layer might help, though there is always the possibility of overfitting. Furthermore, I set the augmented data to grayscale and original to true because that's what seemed to have performed the best on the data in section 4.

6 Code Appendix

1) Fit and transform

```
def fit(self, X):
    """Calculate per-channel mean and standard deviation from dataset X."""
    # TODO: Complete this function
    color_mean = np.mean(X, axis = (0,1,2), dtype = np.float64)
    color_std = np.std(X, axis = (0,1,2), dtype = np.float64)

    self.image_mean = color_mean
    self.image_std = color_std

def transform(self, X):
    """Return standardized dataset given dataset X."""
    # TODO: Complete this function
    """
    for i in X:
        for j in i:
                j[0] = (j[0] - self.image_mean[0])/self.image_std[0]
                j[1] = (j[1] - self.image_mean[1])/self.image_std[1]
                j[2] = (j[2] - self.image_mean[2])/self.image_std[2]
    """
    X = (X - self.image_mean)/self.image_std

    X_transformed = X
    return X_transformed
```

2) Convolutional Neural Network

b) init_weights and forward

```python
class Target(nn.Module):
    def __init__(self):
        """Define the architecture, i.e. what layers our network contains.
        At the end of __init__() we call init_weights() to initialize all model parameters (weights and biases)
        in all layers to desired distributions."""
        super().__init__()

        ## TODO: define each layer
        #change padding using formula
        self.conv1 = torch.nn.Conv2d(in_channels = 3, out_channels = 16, kernel_size = 5, stride = 2, padding = 2)
        self.pool = torch.nn.MaxPool2d(kernel_size = 2, stride = 2, padding = 0)
        self.conv2 = torch.nn.Conv2d(in_channels = 16, out_channels = 64, kernel_size = 5, stride = 2, padding = 2)
        self.conv3 = torch.nn.Conv2d(in_channels = 64, out_channels = 8, kernel_size = 5, stride = 2, padding = 2)
        self.fc_1 = torch.nn.Linear(in_features = 32, out_features = 2)
        ##

        self.init_weights()
```

```python
    def init_weights(self):
        """Initialize all model parameters (weights and biases) in all layers to desired distributions"""
        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]
        FC_in = 32
        nn.init.normal_(self.fc_1.weight, 0.0, 1/FC_in)
        nn.init.constant_(self.fc_1.bias, 0.0)
        ##
```

```python
    def forward(self, x):
        """This function defines the forward propagation for a batch of input examples, by
           successively passing output of the previous layer as the input into the next layer (after applying
           activation functions), and returning the final output as a torch.Tensor object

           You may optionally use the x.shape variables below to resize/view the size of
           the input matrix at different points of the forward pass"""
        N, C, H, W = x.shape

        ## TODO: forward pass
        #import torch.nn.functional as F -->
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = F.relu(self.conv3(x)) #no pooling after 3rd conv layer
        x = torch.flatten(x, start_dim = 1) # flatten all dimensions except the batch dimension
        x = F.relu(self.fc_1(x))
        z = x
        ##
```

c) predictions

```
227
228   def predictions(logits):
229       """Determine predicted class index given logits.
230
231       Returns:
232           the predicted class output as a PyTorch Tensor
233       """
234       # TODO implement predictions
235       #logits = array of either 2 for binary classes or 8 for multiclass. Want to return index
236       #Whichever value is higher = more likely to be correct
237       pred = torch.argmax(logits, 1)
238       return pred
239
```

d) criterion and optimizer

```
    )
# Model
model = Target()

# TODO: define loss function, and optimizer. Replace "None" with your code.
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr = 1e-3)
#

print("Number of float-valued parameters:", count_parameters(model))

# Attempts to restore the latest checkpoint if exists
```

e) early_stopping and train_epoch

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

    Increment curr_count_to_patience by one if new loss is not less than global_min_loss
    Otherwise, update global_min_loss with the current val loss, and reset curr_count_to_patience to 0

    Returns: new values of curr_count_to_patience and global_min_loss
    """
    # TODO implement early stopping
    new_loss = round(stats[-1][1], 4)
    if new_loss >= global_min_loss:
        curr_count_to_patience += 1
    else:
        global_min_loss = new_loss
        curr_count_to_patience = 0
    #
    return curr_count_to_patience, global_min_loss
```

```python
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()
        outputs = model(X) #make predictions for this batch
        loss = criterion(y, outputs) #actual, output
        loss.backward() #updates for each parameter
        optimizer.step()
```

4) Transfer Learning and Data Augmentation

4.1 Transfer Learning

a) Source model architecture

```python
class Source(nn.Module):
    def __init__(self):
        """Define the architecture, i.e. what layers our network contains.
        At the end of __init__() we call init_weights() to initialize all model parameters (weights and biases)
        in all layers to desired distributions."""
        super().__init__()

        ## TODO: define each layer
        self.conv1 = torch.nn.conv2d(in_channels = 3, out_channels = 16, kernel_size = 5, stride_size = 2, padding = 2)
        self.pool = torch.nn.MaxPool2d(kernel_size = 2, stride = 2, padding = 0)
        self.conv2 = torch.nn.conv2d(in_channels = 16, out_channels = 64, kernel_size = 5, stride_size = 2, padding = 2)
        self.conv3 = torch.nn.conv2d(in_channels = 64, out_channels = 8, kernel_size = 5, stride_size = 2, padding = 2)
        self.fc1 = torch.nn.Linear(in_features = 32, out_features = 8)
        ##
```

```python
def init_weights(self):
    """Initialize all model parameters (weights and biases) in all layers to desired distributions"""

    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]
    FC_in = 32
    #self.fc1.weight = 1/FC_in
    nn.init.normal_(self.fc1.weight, 0.0, 1/FC_in)
    nn.init.constant_(self.fc1.bias, 0.0)
    ##
```

```python
def forward(self, x):
    """This function defines the forward propagation for a batch of input examples, by
       successively passing output of the previous layer as the input into the next layer (after applying
       activation functions), and returning the final output as a torch.Tensor object

       You may optionally use the x.shape variables below to resize/view the size of
       the input matrix at different points of the forward pass"""
    N, C, H, W = x.shape

    ## TODO: forward pass
    x = self.pool(F.relu(self.conv1(x)))
    x = self.pool(F.relu(self.conv2(x)))
    x = F.relu(self.conv3(x))
    x = torch.flatten(x, start_dim = 1)
    x = F.relu(self.fc1(x))
    z = x
    ##

    return z
```

b) Criterion and optimizer

```python
model = Source()

# TODO: define loss function, and optimizer
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr = 1e-3, weight_decay = 0.01)
#

print("Number of float-valued parameters:", count_parameters(model))
```

e) freeze layers

```python
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 CONV2
    #      Hint: https://pytorch.org/docs/master/notes/autograd.html
    for index in range(num_layers):
        for param in model.features[index].parameters():
            param.require_grad = False
    for param in model.named_parameters():
        param.requires_grad = False
```

4.2 Data Augmentation

a) Rotate and Grayscale

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

    def _rotate(img):
        """Rotate a random integer 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
        degree = np.random.randint(low = -deg, high = deg)
        return rotate(img, angle = degree, reshape = False)

    return _rotate
```

```python
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.

        Round to the nearest integer.

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

        """
        # TODO
        m = np.repeat(np.expand_dims(np.mean(img, axis = 2), axis = 2), 3, axis = 2)
        m = m.astype(np.uint8)
        return m

    return _grayscale
```

5 Challenge

```python
class Challenge(nn.Module):
    def __init__(self):
        """Define the architecture, i.e. what layers our network contains.
        At the end of __init__() we call init_weights() to initialize all model parameters (weights and biases)
        in all layers to desired distributions."""
        super().__init__()

        ## TODO: define each layer of your network
        self.conv1 = torch.nn.Conv2d(in_channels = 3, out_channels = 16, kernel_size = 5, stride = 2, padding = 2)
        self.pool = torch.nn.MaxPool2d(kernel_size = 2, stride = 2, padding = 0)
        self.conv2 = torch.nn.Conv2d(in_channels = 16, out_channels = 64, kernel_size = 5, stride = 2, padding = 2)
        self.conv3 = torch.nn.Conv2d(in_channels = 64, out_channels = 8, kernel_size = 5, stride = 2, padding = 2)
        self.conv4 = torch.nn.Conv2d(in_channels = 8, out_channels = 2, kernel_size = 2, stride = 2, padding = 0)
        self.fc1 = torch.nn.Linear(in_features = 8, out_features = 8)
        ##

        self.init_weights()
```

```python
def init_weights(self):
    """Initialize all model parameters (weights and biases) in all layers to desired distributions"""
    ## TODO: initialize the parameters for your network

    torch.manual_seed(42)
    for conv in [self.conv1, self.conv2, self.conv3, self.conv4]:
        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)

    FC_in = 32
    nn.init.normal_(self.fc1.weight, 0.0, 1/FC_in)
    nn.init.constant_(self.fc1.bias, 0.0)
    ##
```

```python
def forward(self, x):
    """This function defines the forward propagation for a batch of input examples, by
       successively passing output of the previous layer as the input into the next layer (after applying
       activation functions), and returning the final output as a torch.Tensor object

       You may optionally use the x.shape variables below to resize/view the size of
       the input matrix at different points of the forward pass"""
    N, C, H, W = x.shape

    ## TODO: forward pass
    x = self.pool(F.relu(self.conv1(x)))
    x = self.pool(F.relu(self.conv2(x)))
    x = self.pool(F.relu(self.conv3(x)))
    x = F.relu(self.conv4(x))
    x = torch.flatten(x, start_dim = 1)
    x = F.relu(self.fc1(x))
    z = x
    ##

    return z
```