UNIVERSITY OF MICHIGAN

EECS 445 – Introduction to Machine Learning

Project 2 - Karl's Convolutional Kennel

Written by: Yigao Fang

Unique name: fgsepter

Mar.10th, 2021

1 Data Preprocessing

(a):

```
class ImageStandardizer(object):
    """Standardize a batch of images to mean 0 and variance 1.

    The standardization should be applied separately to each channel.
    The mean and standard deviation parameters are computed in 'fit(X)' and
    applied using 'transform(X)'.

    X has shape (N, image_height, image_width, color_channel)
    """
    def __init__(self):
        """Initialize mean and standard deviations to None."""
        super().__init__()
        self.image_mean = None
        self.image_std = None

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

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

i.

Mean: [123.084 117.488 93.312]
Std: [62.729 59.178 61.434]

ii.

We extract the per-channel image mean and standard deviation from the training set since we are using the training set to train the model, while other data partitions should not be changed for performance test. Once we normalize the data, the images input of CNN will have pixel color values within the same range and won't have different scales. This can help reduce the error when training. We should not extract the per-channel image mean and standard deviation from other data partitions since we do not want to change these images in order to test our model on images with different scales. We only want to normalize the training data in order to better training our model.

(b):

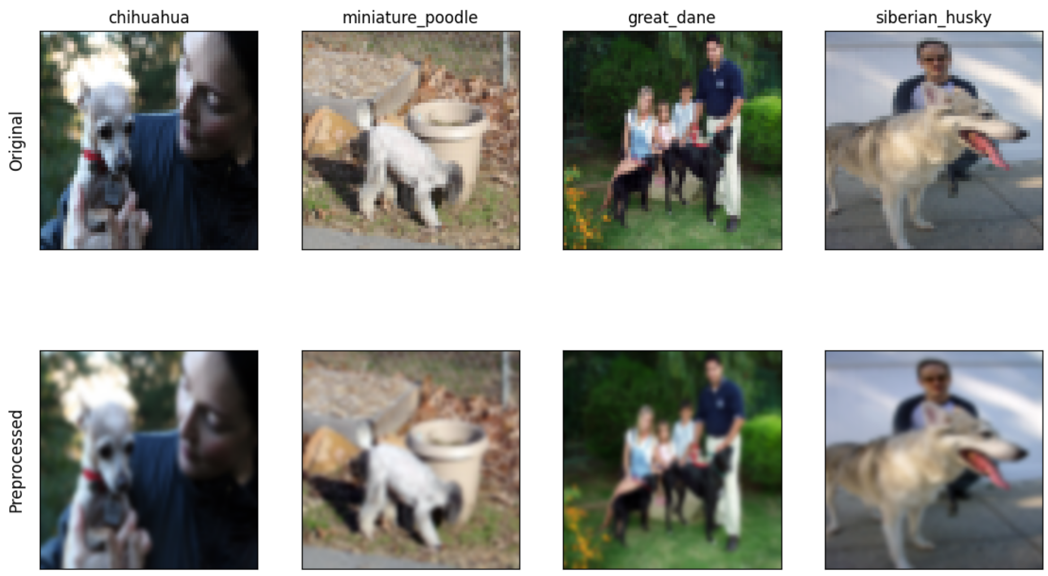


Figure 1.1:

The visible effects of our preprocessing is that the image has been blurred and become less pixelated, since the difference between the color of each pixel becomes smaller and less sharp. Some noise may be eliminated.

2 Convolutional Neural Networks

(a):

1. Layer 0: 0
2. Layer 1: (3*5*5+1)*16 = 1216
3. Layer 2: 0
4. Layer 3: (16*5*5+1)*64 = 25664
5. Layer 4: 0
6. Layer 5: (64*5*5+1)*8 = 12808
7. Layer 6: 32*2+2 = 66

Totally: 39754
According to the program output after implementing the model, I find that my result is correct.

(b):

```
class Source(nn.Module):
    def __init__(self):
        super().__init__()
        ## TODO: define each layer
        self.conv1 = nn.Conv2d(  in_channels = 3,
                                 out_channels= 16,
                                 kernel_size = (5,5),
                                 stride      = (2,2),
                                 padding     = 2  )
        self.pool = nn.MaxPool2d(kernel_size = (2,2),
                                 stride      = (2,2))
        self.conv2 = nn.Conv2d(  in_channels = 16,
                                 out_channels= 64,
                                 kernel_size = (5,5),
                                 stride      = (2,2),
                                 padding     = 2  )
        self.conv3 = nn.Conv2d(  in_channels = 64,
                                 out_channels= 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.fc1]
        conv = self.fc_1
        nn.init.normal_(conv.weight, 0.0, 1 / sqrt(32))
        nn.init.constant_(conv.bias, 0.0)
        ##

    def forward(self, x):
        N, C, H, W = x.shape
        ## TODO: forward pass
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = F.relu(self.conv3(x))
        x = x.view(-1, 32)
        x = self.fc_1(x)
        ##
        return x
```

(c):

```python
def predictions(logits):
    """Determine predicted class index given logits.
    Returns:
        the predicted class output as a PyTorch Tensor
    """
    # TODO implement predictions
    _, pred = torch.max(logits.data, axis = 1)
    return pred
```

(d):

```python
# TODO: define loss function, and optimizer
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr= 0.001, weight_decay=0.01)
#
```

(e):

```python
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
    new_val_loss = stats[-1][1]
    if new_val_loss >= prev_val_loss:
        curr_patience += 1
    else:
        curr_patience = 0
        prev_val_loss = new_val_loss
    #
    return curr_patience, prev_val_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)
        loss = criterion(outputs, y)
        loss.backward()
        optimizer.step()
```

(f):

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

(i):

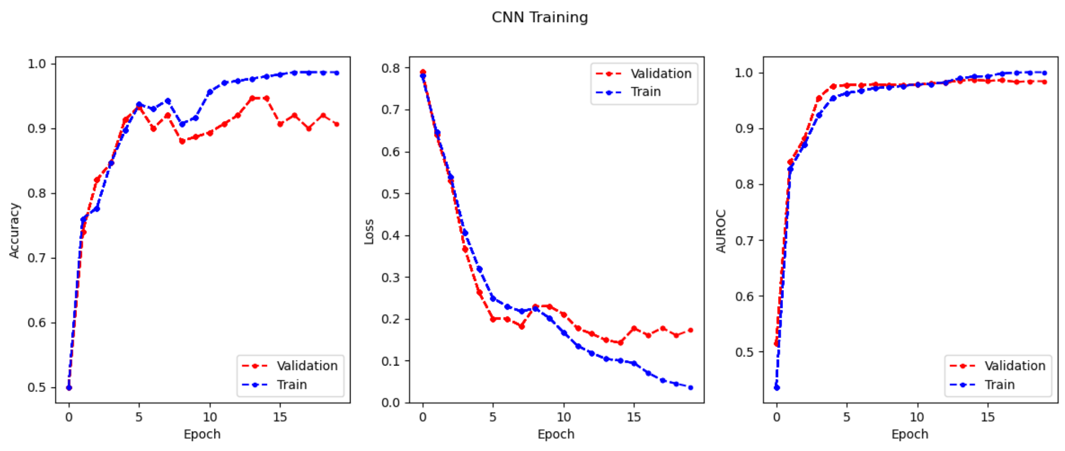


Figure 2.1: CNN Training with Patience 5

Sources of noise that contribute to this behavior:

1. The background of the graph.
2. Different illumination conditions.
3. The noise of camera imaging.

(ii):

When using patience of 5, my model stopped at epoch 19.
When increasing the patience to 10, my model stopped at epoch 24.
I think the patience value 5 works better. It can find the same best epoch as patience equals 10, while running less epochs, and hence it is more efficient.
In the scenario when the model is under-fitting, increased patience will be better. Increased patience will run more epochs to obtain a better model. Also, when we are facing locally optimal solutions, increasing patience will be better, since it can avoid the locally optimal solution and may find the actual best solution.

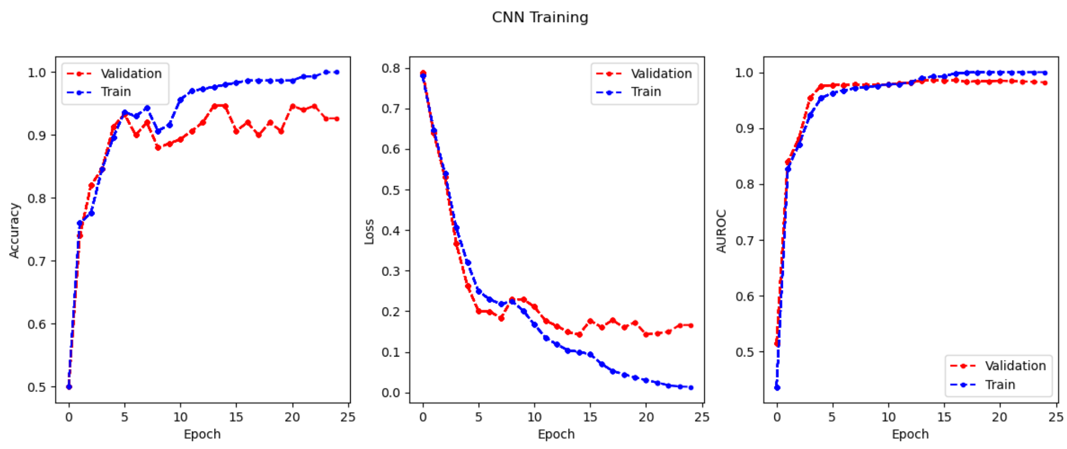


Figure 2.2: CNN Training with Patience 10

(iii):

The new size of the input of the fully connected layer should be $32 \times 4 = 128$.
The new size of the output of the fully connected layer should still be 2.

Table 1:

	Epoch	Training AUROC	Validation AUROC
8 filters	14	0.9923	0.9867
32 filters	7	0.9964	0.976

According to the table, as the number of filters from 8 to 32, number of epochs decreases, amd the training AUROC of the model increases, while the validation AUROC of the model decreases.

When we increase the number of filters, the output of the third convolution layer and the input channel of the fully connected layer increase accordingly, which lead to more complex model. This will consider more features and may lead to the increasing of validation error, and hence the validation AUROC will be worse.

On the contrary, when we increase the model complexity, more features are considered and the bias will decrease, leading to the increasing of the training AUROC.

(g):

Table 2:

	Training	**Validation**	**Testing**
Accuracy	0.98	0.9467	0.55
AUROC	0.9923	0.9867	0.6284

(i):

From the Table above, I notice that the training performance does not exceeds validation performance too much. Hence there is no obvious evidence for over-fitting.

(ii):

I notice the trend that the performance for test data decreases a lot comparing to the validation performance.

My explanation is that, comparing with the test data set, the training data set may be biased. When we are training our model on the training data set, we may consider background or other useless pixels as features. We can get a good performance on the training set, but if we change to the testing set, and our model still consider these useless features, we may have worse results.

3 Visualizing what the CNN has learned

(a):

$$\alpha_1^1 = \frac{1}{Z}\sum_i\sum_j \frac{\partial y^1}{\partial A_{ij}^1} = \frac{3}{16}$$

$$\alpha_2^1 = \frac{1}{Z}\sum_i\sum_j \frac{\partial y^1}{\partial A_{ij}^2} = \frac{7}{16}$$

$$\begin{aligned}
L^1 &= ReLU(\sum_k \alpha_k^1 A^k) \\
&= ReLU(\frac{3}{16}\begin{bmatrix}1 & 1 & 2 & 1\\ 1 & 2 & 1 & 0\\ 0 & 1 & 0 & -1\\ 0 & 1 & -2 & -2\end{bmatrix} + \frac{7}{16}\begin{bmatrix}1 & 1 & 1 & 1\\ 2 & 2 & 2 & 2\\ 2 & 2 & 1 & 0\\ -1 & -1 & -1 & 0\end{bmatrix}) \\
&= \begin{bmatrix}0.625 & 0.625 & 0.8125 & 0.625\\ 1.0625 & 1.25 & 1.0625 & 0.875\\ 0.875 & 1.0625 & 0.4375 & 0\\ 0 & 0.25 & 0 & 0\end{bmatrix})
\end{aligned}$$

(b):

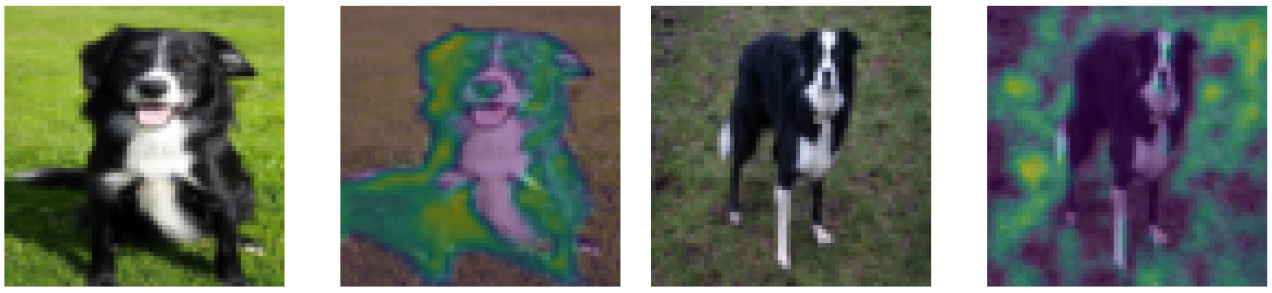


Figure 3.1: Visualizing of Grad-CAM

The dark colors of the original figures become light, which has higher values. It appears to use the black fur of the Collies as features to identify the Collie class. Since Collies has black fur while Golden Retrievers does not.

However, some of the grass are also highlighted, which is the noise that this method gives.

(c):

The Grad-Cam visualizations confirm your hypothesis. According to the discussion in (b), CNN appear to not only using the black fur as the features, but also using the grass as features. Grass is the background of the image, and it is harmful to be considered as part of the features. If we change the background of the image, our model may give us different results.

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(  in_channels = 3,
                                 out_channels= 16,
                                 kernel_size = (5,5),
                                 stride      = (2,2),
                                 padding     = 2   )
        self.pool  = nn.MaxPool2d(kernel_size = (2,2),
                                 stride      = (2,2))
        self.conv2 = nn.Conv2d(  in_channels = 16,
                                 out_channels= 64,
                                 kernel_size = (5,5),
                                 stride      = (2,2),
                                 padding     = 2   )
        self.conv3 = nn.Conv2d(  in_channels = 64,
                                 out_channels= 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.fc1]
        conv = self.fc_1
        nn.init.normal_(conv.weight, 0.0, 1 / sqrt(32))
        nn.init.constant_(conv.bias, 0.0)
        ##
    def forward(self, x):
        N, C, H, W = x.shape
        ## TODO: forward pass
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = F.relu(self.conv3(x))
        x = x.view(-1, 32)
        x = self.fc_1(x)
        ##
        return x
```

(b):

```
# TODO: define loss function, and optimizer
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr= 0.001, weight_decay=0.01)
#
```

(c):

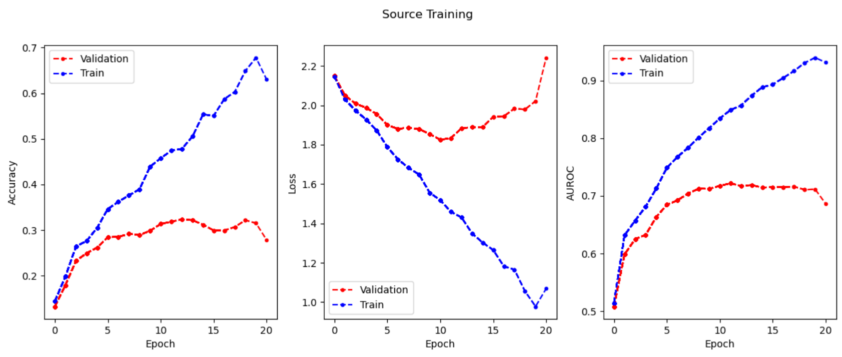
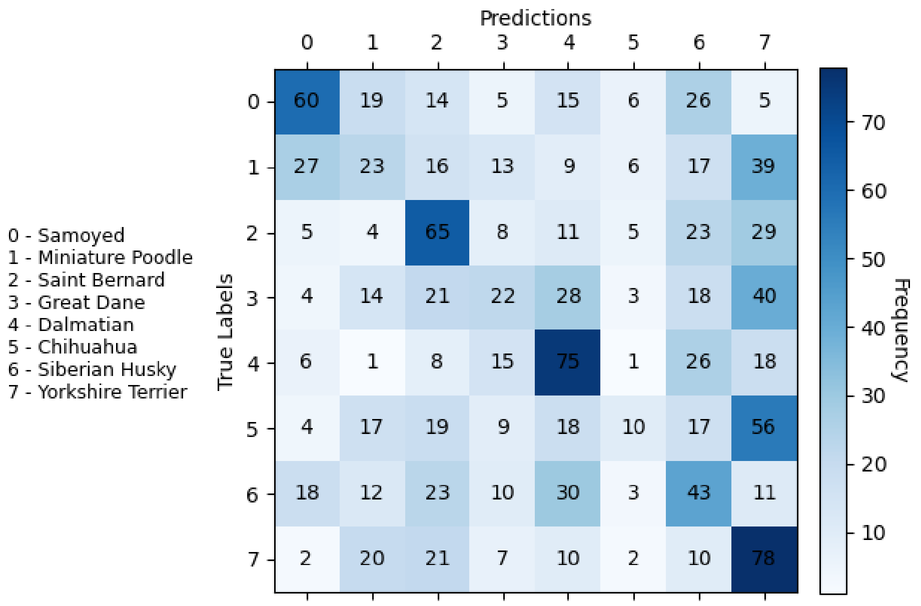


Figure 4.1:

The epoch number with the lowest validation loss is 10.

(d):



	0	1	2	3	4	5	6	7
0	60	19	14	5	15	6	26	5
1	27	23	16	13	9	6	17	39
2	5	4	65	8	11	5	23	29
3	4	14	21	22	28	3	18	40
4	6	1	8	15	75	1	26	18
5	4	17	19	9	18	10	17	56
6	18	12	23	10	30	3	43	11
7	2	20	21	7	10	2	10	78

Figure 4.2: Confusion Matrix

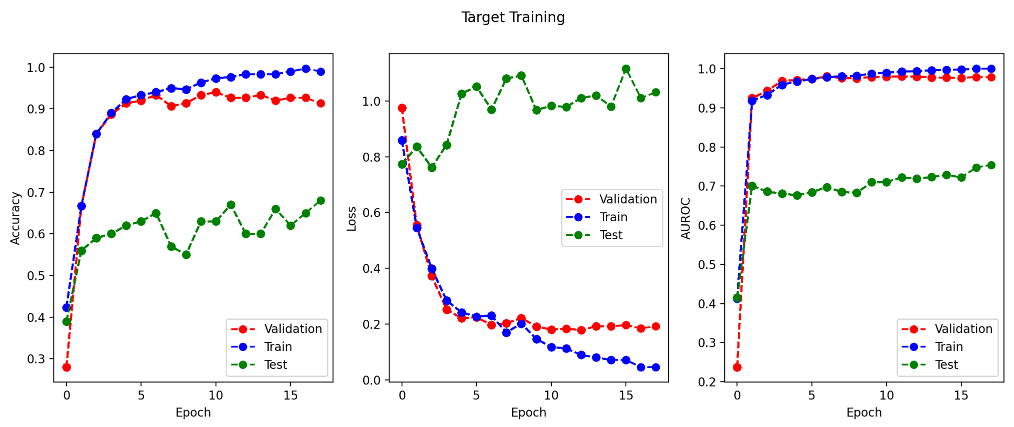
Most accurate: Yorkshire Terrier
Least accurate: Chihuahua
Reason: The numbers on the diagonal of the confusion matrix is the numbers of the correct prediction. Notice that each breed has 150 test data points. Hence the breed which has the largest number on the diagonal pixel is the most accurate, and the breed which has the lowest number on the diagonal pixel is the least accurate. **Yorkshire Terrier** has 78 on the diagonal pixel, and hence it is the most accurate. **Chihuahua** has 10 on the diagonal pixel, and hence it is the least accurate. This may due to the fact that **Chihuahua** is small and is hard to catch in the figure, and our model tend to not predict **Chihuahua** as the results. While the fur of **Yorkshire Terrier** is long, and it is a feature that easy to catch, and our model tend to predict **Yorkshire Terrier** as the results. We can see from the matrix that the column 5 has very little predictions, while column 7 has many predictions.

(e):

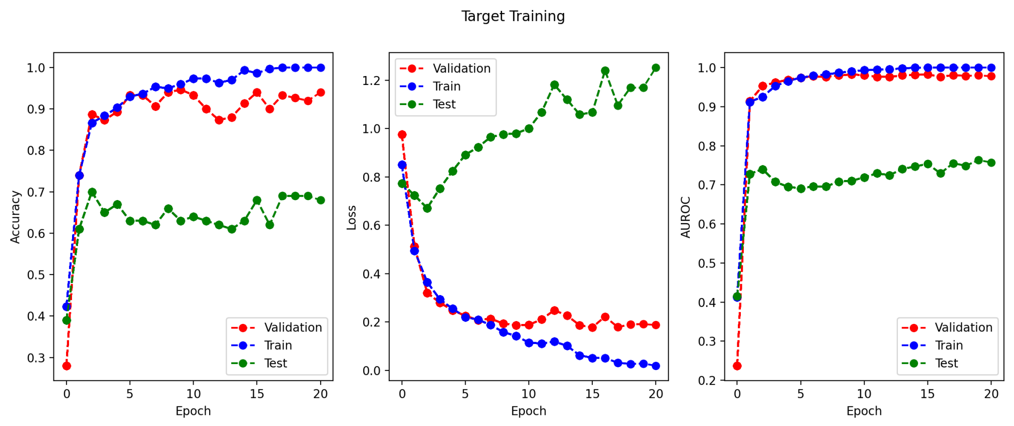
```
#TODO: define loss function, and optimizer
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(),lr = 0.001)
#
```

```
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
    i = 0
    # print(num_layers)
    for name, param in model.named_parameters():
        if i < 2 * num_layers:
            # print(name)
            param.requires_grad = False
        i = i + 1
    # print("done")
```

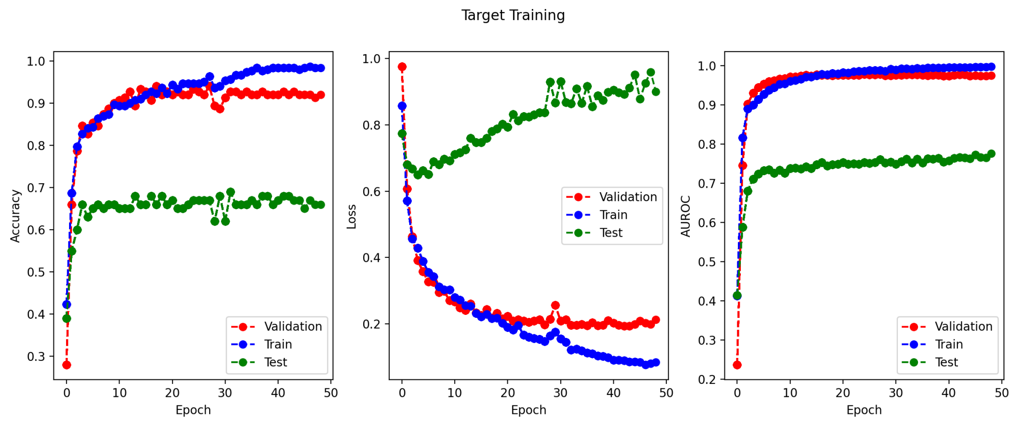
(f):



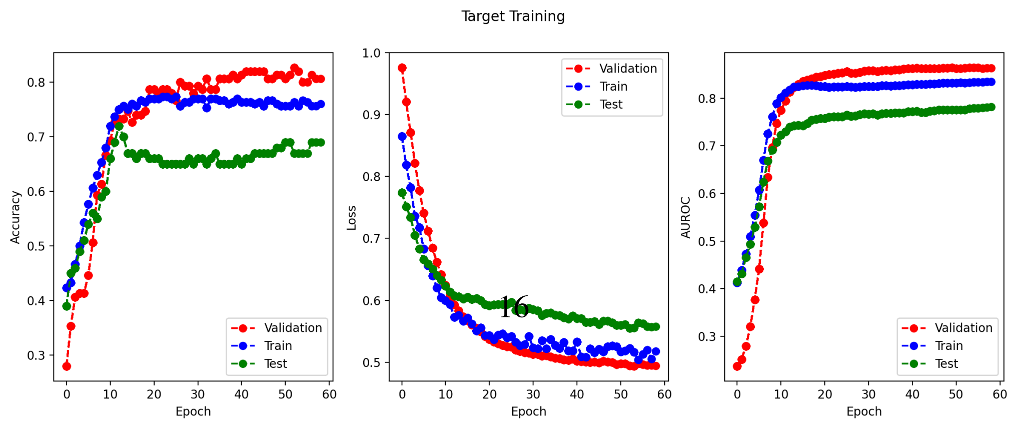
(a) Freeze no layers



(b) Freeze one layer



(c) Freeze two layers



(d) Freeze three layers

Table 3:

AUROC	**TRAIN**	**VAL**	**TEST**
Freeze all CONV layers	0.8332	0.8642	0.778
Freeze first two CONV laters	0.9963	0.9771	0.7656
Freeze first CONV layer	1.0	0.9826	0.7536
Freeze no layers	0.9937	0.9797	0.7188
No pretaining or Transfer Learning	0.9923	0.9867	0.6284

1. Transfer learning does help.

2. The source test is helpful.

3. Comparing to the CNN in section 2, which only has AUROC of 0.6284 for test sets, the AUROC score for test sets with transfer learning method is over 0.7, which is much higher. When freeze three layers, the test AUROC score even becomes 0.778, which is much higher than previous score. Transfer learning preprocessed and trained the data with more labels, and use the parameters of the model to classify binary labels. Using other labels can help us identify more useful features, and can help us classify the current two labels better. Hence I hypothesize that it was helpful.

4. Compare to freezing just a subset of layers or freezing none layers, freezing all convolutional layers experienced more epochs before stopping, while having a much better test AUROC score. However, its training and validation AUROC are much lower.

5. I think the reason is that, transfer learning can utilize pre-trained source parameters and adapt it to the target model. Freeze all layers can adapt the target model very similar to the source model, and hence the training and validation AUROC will be lower due to the difference between source data and target data, while the test AUROC can be closer to the training AUROC. In this situation, freezing all layers gives us best performance. However, when the data set changes, number of freeze layers giving the best performance will change accordingly.

4.2 Data Augmentation

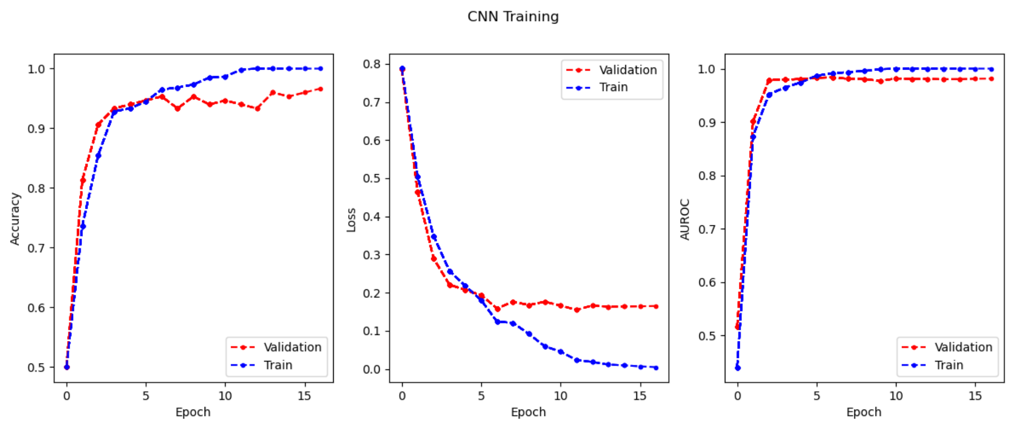
(a):

```python
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
        theta = deg * (2  * np.random.rand() - 1)
        return rotate(img, theta, reshape=False)

    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
        gray = np.mean(img, axis= 2, keepdims = True)
        return np.dstack((gray, gray, gray))
    return _grayscale
```

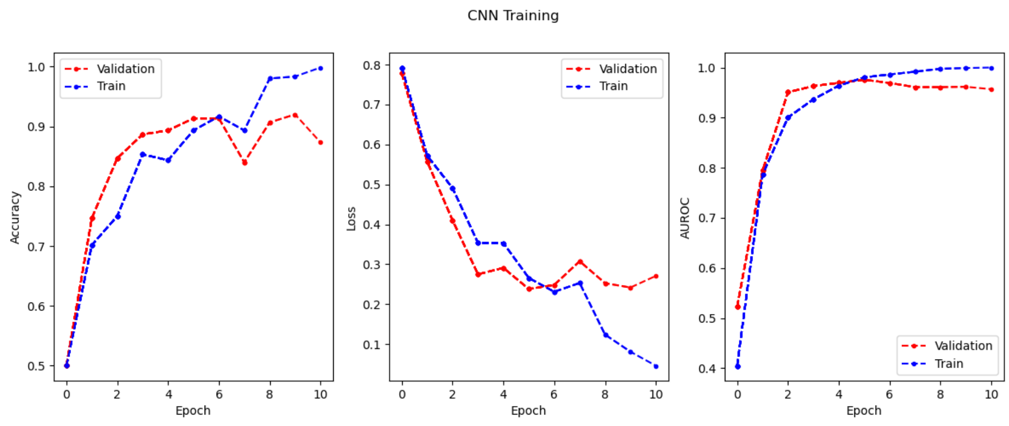
(b):

The epoch with lowest validation loss:

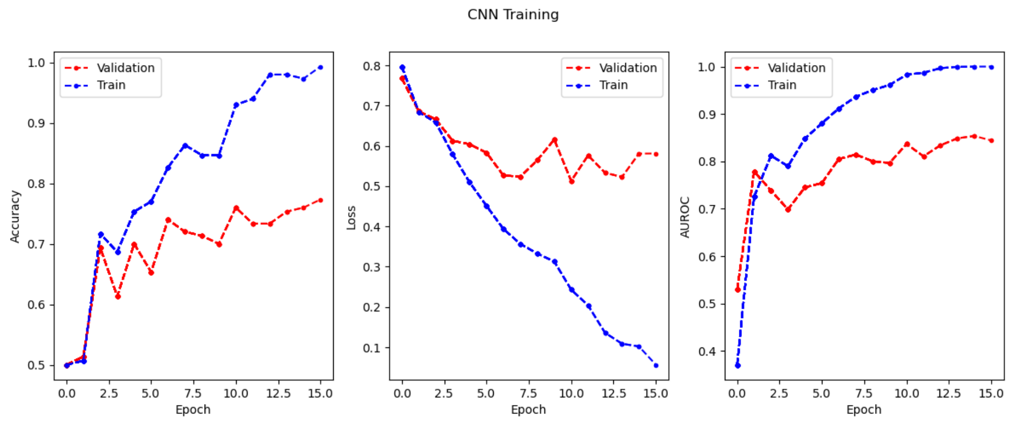
1. Rotation(keep original): 11
2. Grayscale(keep original): 5
3. Grayscale(discard original): 10



(a) Rotation(keep original)



(b) Grayscale(keep original)



(c) Grayscale(discard original)

Figure 4.4:

Table 4:

AUROC	**TRAIN**	**VAL**	**TEST**
Rotation(keep original)	1.0	0.981	0.6488
Grayscale(keep original)	0.9886	0.9744	0.7244
Grayscale(discard original)	0.9472	0.843	0.7676
No augmentation	0.9923	0.9867	0.6284

(c):

1. After applying the data augmentation of rotation, the loss function reaches the minimum faster, and the plot becomes smooth and has less oscillation. I think the reason for this change is that the image has less noise with this data augmentation. Rotation can produce more training images from different angles, and this can increase the training data set and provide us with more training example, which can help us train the model faster and better.

2. After applying the data augmentation of grayscale and keeping the original data, the loss function reaches the minimum faster, and the test AUROC increases. I think the reason for this change is that, we change the color to gray, and this can help reduce the noise. The green grass will less effect the training model, and our model can catch the useful features more accurately.

3. After applying the data augmentation of grayscale and discarding the original data, the plot has more oscillation, and the validation loss has a relatively high value. Also, the loss function reaches the minimum after more epochs. As the training AUROC and accuracy increase, the validation AUROC and accuracy do not increase a lot. However, the test AUROC increases a lot. I think the reason for this change is that we discard the original images, which decrease the size of the training data set. With less images, our model will take more epochs in order to get a better performance. The reason for the huge improvement of the performance is that, once we discard the original images, the influence of the original harmful features(i.e. the green grass) won't influence our model any more. Hence we may obtain a better model with high performance.

5 Challenge

5.1 Regularization

Previously, we normalized the input pictures to have mean equals 0 and standard deviation equals 1. This can reduce the difference among pictures and improve the accuracy.

However, another thing that we need to consider is the noise points in the pictures. Recall that **Gaussian Filter** is a good method to reduce the noise. Hence to better improve my model, I also apply a 3 $\times$ 3 Gaussian Filter to the normalized pictures to reduce noise. I rewrote the `transform` function in `dataset_challenge.py` and examined whether this method will help.

```
def transform(self, X):
    """Return standardized dataset given dataset X."""
    # TODO: Complete this function
    N = X.shape[0]
    normalized = (X - self.image_mean) / self.image_std
    output = normalized.copy()

    kernel_size = 3  # Modify this hyperparameter here
    sigma = 2        # Modify this hyperparameter here
    kernel = np.zeros([kernel_size,kernel_size])
    for i in range(kernel_size):
        for j in range(kernel_size):
            kernel[i,j] = np.exp(  (-(i-kernel_size//2)**2-(j-kernel_size//2)**2)/(2*
                                                          sigma**2)) /(2*np.pi*sigma**2)

    for i in range(N):
        for j in range(3):
            image = normalized[i,:,:,j]
            filtered = scipy.ndimage.convolve(image, kernel, mode = 'constant')
            output[i,:,:,j] = filtered

    return output
```

I first choose $\sigma = 0.572$, which will give a Gaussian Filter with sum 1. This gave me a performance of AUROC for **Grayscale(discard original)** at the lowest validation loss equals 0.8048, which is better than the original performance 0.7676 without Gaussian Filter. Then I choose to have $\sigma = 2$, which gave me a performance equals 0.7948, which is also better. This indicates that applying Gaussian Filter to eliminate noise may be a good approach.

Gaussian Filter contains two hyperparameters, the `kernel_size`, and the `sigma`. I will discuss the choice of these two hyperparameters in Section 5.4.

5.2 Feature selection

Previously we continue to use Convolutional Neural Networks to select the features from pictures. This is a deep learning method that apply filters to extract features. We are given three convolutional layers, two max pooling layers, and one fully connected layers to obtain the features. Less layers may result in under-fitting, while more layers may result in over-fitting. Hence finding a suitable model architecture may be beneficial to our results. In the next subsection, I will discuss how did I improve the current feature selection architecture.

5.3 Model architecture

Previously, we apply the model with three convolutional layers, two max pooling layers, and one fully connected layer. Model architecture will definitely influence the model accuracy. Hence I explored which architecture will gives us the highest accuracy.

The original architecture(Appendix B) has 8 filters in Convolutional Layer 3, and a fully connected layer 1 with 32 inputs. More filters may increase the accuracy of the model, so I attempted to modify the number of filters in Convolutional Layer 3.

Table 5:

AUROC	**TRAIN**	**VAL**	**TEST**
4 filters	0.9939	0.8123	0.792
8 filters	0.996	0.8512	0.8048
16 filters	0.9984	0.8377	0.8288
32 filters	0.9976	0.8352	0.8156

According to the above table, I find that 16 filters for the third convolutional layer gave me the best test performance. Similarly, I attempted to modify the number of filters in other layers, my final architecture will be specified later.

Another architecture I attempted to modify is the number of layers. Currently we have 3 convolutional layers and one fully connected layer. I tried to add one more fully connected layer to see the performance:

```
    self.fc_1 = nn.Linear(64,16)
    self.fc_2 = nn.Linear(16,2)
    # 16 can be replaced by 32 and 8 to see the performance
```

Table 6:

AUROC	**TRAIN**	**VAL**	**TEST**
16 filters with one fc layer	0.9984	0.8377	0.8288
16 filters with two fc layers(64-32, 32-2)	0.9825	0.7797	0.7812
16 filters with two fc layers(64-16, 16-2)	0.9874	0.829	0.8172
16 filters with two fc layers(64-8, 8-2)	0.9922	0.8318	0.7928

From the table above, I find that one fully connected layer gave me better performance.

Previously we use `ReLU` as activation function. As introduced in class, we have other activation functions such as `Sigmoid`, `ELU`, etc. Here I tried to modify the activation function and see the performance:

Table 7:

AUROC	TRAIN	VAL	TEST
ReLU	0.9984	0.8377	0.8288
ELU	0.9822	0.7804	0.8144
Sigmoid	0.8693	0.7129	0.7716
tanh	0.9318	0.781	0.7736

According to the above table, I finally choose `ReLU` as the activation function.

I also attempted to change the max pooling layer to average pooling layer. However, it lower the performance of our model. After some searching, I find that adding a BN layer may improve the model performance. It can normalize the output channels from each convolutional layer and using the normalized images as the input of next layer. However, since we already normalized our images at the beginning of the project, it also did not improve my model much. Considering the complexity of our model, I decide to not include BN layer in our model.

Finally, my model architecture is given by:

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

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

        self.init_weights()
    def init_weights(self):
        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]
        conv = self.fc_c
        nn.init.normal_(conv.weight, 0.0, 1 / sqrt(64))
        nn.init.constant_(conv.bias, 0.0)
        ##
```

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

        ## TODO: forward pass
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = F.relu(self.conv3(x))
        x = x.view(-1, 64)
        x = self.fc_c(x)
        ##
        return x
```

5.4 Hyperparameters

Learning rate is the hyperparameter that may influence the accuracy of our model. During previous works, we always choose `lr = 0.001`. Changing this value may influence the result of our model a lot. Hence I want to find which value of `learning rate` gives best model. Choosing different value of `lr`, I obtained the table below. According to the table, `lr = 0.001` gives us best performance. Hence I will choose `lr` = 0.001.

Table 8:

AUROC	TRAIN	VAL	TEST
`lr` = 0.0001	0.9482	0.8633	0.7368
`lr` = 0.001	0.996	0.8512	0.8048
`lr` = 0.01	0.7639	0.7216	0.683

Note that our learning rate is not necessary to be a constant during the training processing. In each epoch, we can have a different learning rate. After some searching, I learned two generally used learning rate functions, `CosineAnnealing` and `warm_up`. `CosineAnnealing` changing the learning rate similar to a `cos` function. This is very helpful to avoid locally optimal solution and find the actual best solution. `Warm_up` first apply small learning rate, and then large learning rate, and then small learning rate. This may help our model reaches the lowest loss faster.

Another hyperparameter that may influence the accuracy is the `sigma` value of Gaussian Filter. The AUROC performances with lowest validation loss for different `sigma` value are recorded below:

Table 9:

AUROC	TRAIN	VAL	TEST
`sigma` = 0.572	0.996	0.8512	0.8048
`sigma` = 1	0.9939	0.8489	0.8024
`sigma` = 1.5	0.9684	0.8258	0.7608
`sigma` = 2	0.991	0.8372	0.7948
`sigma` = 2.5	0.9627	0.8197	0.7772
`sigma` = 3	0.9592	0.8194	0.782

Also I test the performance when hyperparameter `kernel_size` equals 4 and 5, which will give a 4×4 and 5×5 Gaussian Kernel. However, these two did not have a better performance than `kernel_size` equals 3. Finally, I choose to use

$$\texttt{kernel\_size} = 3$$
$$\texttt{sigma} = 0.572$$

5.5 Transfer learning

As shown in Table 3, transfer learning indeed increases the AUROC score for test data points a lot. Comparing with **Freeze all layers**, **Freeze first two layers**, **Freeze first layer**, and **Freeze no layers**, I find that using **Freeze no layers** gives me the best AUROC score for test points. Hence I applied transfer learning with **Freeze no layers** when predicting the challenge figures.

5.6 Data augmentation

As shown in Table 4, data augmentation indeed increases the AUROC score for test data points a lot. Comparing with **Rotation(keep original)**, **Grayscale(keep original)**, and **Grayscale(discard original)**, I find that using the **grayscale** and **discard original** gives me the best AUROC score for test points.

Apart from these three data augmentation methods, I also think of other two augmentations. The first one is flip the images and keep the original images. The second is translation the images randomly within a range and keep the original images. Also, any two of the above methods can be done in the mean time, and I test the performance for all of them. I add two functions in `augment_data.py` to flip and translate the images:

```python
def Flip():
    """Return function to flip image."""

    def _flip(img):
        """Return flip of image.
        :img: H x W x C numpy array
        :returns: H x W x C numpy array
        """
        # TODO
        return cv2.flip(img,1)
    return _flip

def Translate(dis = 5):
    """Return function to translate image in the range (-dis,dis)."""

    def _translate(img):
        # TODO
        out = np.float32([[1, 0, random.uniform(-dis,dis)],
                          [0, 1, random.uniform(-dis,dis)]])
        return cv2.warpAffine(img, out, (64,64) )
    return _translate
```

After some testing, I find that when apply grayscale and translation simultaneously, the test AUROC gives me best performance.

Hence I applied the grayscale augmentation with translation when predicting the challenge figures.

5.7 Model evaluation

Previously, I implement the function `early_stopping` to set the stopping point as the epoch when the validation loss does not decrease for certain number of epochs which defined as `patience`. I set the value of `patience` equals 5 and 10 and compare the performance. When the value of patience is too small, the model is easily under-fitting, while when the value of patience is too large, it is possible that we will run too many epochs. Setting this number of patience aims for finding the epoch with the lowest validation loss, since the criteria of lowest validation loss may gives us the best model.

Another two possible criteria are the validation accuracy and the validation AUROC. Our goal is to find the model that gives the highest accuracy when classifying **collie** and **golden_retriever**. This is a binary classifying problem, and accuracy can give us the direct results that which model best classified the validation sets.

AUROC score is also a good criteria to find the best model. It is especially useful when the data set is unbalanced, and can responses to the prediction results of unbalanced data sets more accurately.

5.8 Whether GPU

I did not use GPU hardware to train my model.

A dataset_challenge.py

```python
"""
EECS 445 - Introduction to Machine Learning
Winter 2021 - Project 2
Dogs Dataset
    Class wrapper for interfacing with the dataset of dog images
    Usage: python dataset.py
"""

import os
import random
import numpy as np
import pandas as pd
import torch
from matplotlib import pyplot as plt
from imageio import imread
from PIL import Image
from torch.utils.data import Dataset, DataLoader
from utils import config
import scipy.ndimage


def get_train_val_test_loaders(task, batch_size, **kwargs):
    """Return DataLoaders for train, val and test splits.

    Any keyword arguments are forwarded to the DogsDataset constructor.
    """
    tr, va, te, _ = get_train_val_test_datasets(task, **kwargs)

    tr_loader = DataLoader(tr, batch_size=batch_size, shuffle=True)
    va_loader = DataLoader(va, batch_size=batch_size, shuffle=False)
    te_loader = DataLoader(te, batch_size=batch_size, shuffle=False)

    return tr_loader, va_loader, te_loader, tr.get_semantic_label


def get_challenge(task, batch_size, **kwargs):
    """Return DataLoader for challenge dataset.

    Any keyword arguments are forwarded to the DogsDataset constructor.
    """
    tr = DogsDataset("train", task, **kwargs)
    ch = DogsDataset("challenge", task, **kwargs)

    standardizer = ImageStandardizer()
    standardizer.fit(tr.X)
    tr.X = standardizer.transform(tr.X)
    ch.X = standardizer.transform(ch.X)

    tr.X = tr.X.transpose(0, 3, 1, 2)
    ch.X = ch.X.transpose(0, 3, 1, 2)

    ch_loader = DataLoader(ch, batch_size=batch_size, shuffle=False)
    return ch_loader, tr.get_semantic_label
```

```
def get_train_val_test_datasets(task="default", **kwargs):
    """Return DogsDatasets and image standardizer.

    Image standardizer should be fit to train data and applied to all splits.
    """
    tr = DogsDataset("train", task, **kwargs)
    va = DogsDataset("val", task, **kwargs)
    te = DogsDataset("test", task, **kwargs)

    # Resize
    # We don't resize images, but you may want to experiment with resizing
    # images to be smaller for the challenge portion. How might this affect
    # your training?
    # tr.X = resize(tr.X)
    # va.X = resize(va.X)
    # te.X = resize(te.X)

    # Standardize
    standardizer = ImageStandardizer()
    standardizer.fit(tr.X)
    tr.X = standardizer.transform(tr.X)
    va.X = standardizer.transform(va.X)
    te.X = standardizer.transform(te.X)

    # Transpose the dimensions from (N,H,W,C) to (N,C,H,W)
    tr.X = tr.X.transpose(0, 3, 1, 2)
    va.X = va.X.transpose(0, 3, 1, 2)
    te.X = te.X.transpose(0, 3, 1, 2)

    return tr, va, te, standardizer


def resize(X):
    """Resize the data partition X to the size specified in the config file.

    Use bicubic interpolation for resizing.

    Returns:
        the resized images as a numpy array.
    """
    image_dim = config("image_dim")
    image_size = (image_dim, image_dim)
    resized = []
    for i in range(X.shape[0]):
        xi = Image.fromarray(X[i]).resize(image_size, resample=2)
        resized.append(xi)
    resized = [np.asarray(im) for im in resized]

    return resized


class ImageStandardizer(object):
    """Standardize a batch of images to mean 0 and variance 1.
    The standardization should be applied separately to each channel.
    The mean and standard deviation parameters are computed in `fit(X)` and
    applied using `transform(X)`.

    X has shape (N, image_height, image_width, color_channel)
    """

    def __init__(self):
        """Initialize mean and standard deviations to None."""
        super().__init__()
        self.image_mean = None
        self.image_std = None

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

    def transform(self, X):
        """Return standardized dataset given dataset X."""
        # TODO: Complete this function
        N = X.shape[0]
        normalized = (X - self.image_mean) / self.image_std
        output = normalized.copy()

        kernel_size = 3
        sigma = 0.572
        kernel = np.zeros([kernel_size,kernel_size])
        for i in range(kernel_size):
            for j in range(kernel_size):
                kernel[i,j] = np.exp(  (-(i-kernel_size//2)**2-(j-kernel_size//2)
                                                                **2)/(2*sigma**2)) /(2
                                                                *np.pi*sigma**2)


        for i in range(N):
            for j in range(3):
                image = normalized[i,:,:,j]
                filtered = scipy.ndimage.convolve(image, kernel, mode = 'constant
                                                                ')
                output[i,:,:,j] = filtered

        return output



class DogsDataset(Dataset):
    """Dataset class for dog images."""
    def __init__(self, partition, task="target", augment=False):
        """Read in the necessary data from disk.

        For parts 2, 3 and data augmentation, `task` should be "target".
        For source task of part 4, `task` should be "source".

        For data augmentation, `augment` should be True.
        """
        super().__init__()

        if partition not in ["train", "val", "test", "challenge"]:
            raise ValueError("Partition {} does not exist".format(partition))

        np.random.seed(42)
        torch.manual_seed(42)
        random.seed(42)
        self.partition = partition
        self.task = task
        self.augment = augment
        # Load in all the data we need from disk
        if task == "target" or task == "source":
            self.metadata = pd.read_csv(config("csv_file"))
        if self.augment:
            print("Augmented")
            self.metadata = pd.read_csv(config("augmented_csv_file"))
        self.X, self.y = self._load_data()

        self.semantic_labels = dict(
            zip(
                self.metadata[self.metadata.task == self.task]["numeric_label"],
                self.metadata[self.metadata.task == self.task]["semantic_label"],
            )
        )

    def __len__(self):
        """Return size of dataset."""
        return len(self.X)

    def __getitem__(self, idx):
        """Return (image, label) pair at index `idx` of dataset."""
        return torch.from_numpy(self.X[idx]).float(), torch.tensor(self.y[idx]).
                                                      long()

    def _load_data(self):
        """Load a single data partition from file."""
        print("loading %s..." % self.partition)

        df = self.metadata[
            (self.metadata.task == self.task)
            & (self.metadata.partition == self.partition)
        ]

        if self.augment:
            path = config("augmented_image_path")
        else:
            path = config("image_path")

        X, y = [], []
        for i, row in df.iterrows():
            label = row["numeric_label"]
            image = imread(os.path.join(path, row["filename"]))
            X.append(image)
            y.append(row["numeric_label"])
        return np.array(X), np.array(y)

    def get_semantic_label(self, numeric_label):
        """Return the string representation of the numeric class label.

        (e.g., the numberic label 1 maps to the semantic label 'miniature_poodle
                                                                ').
        """
        return self.semantic_labels[numeric_label]


if __name__ == "__main__":
    np.set_printoptions(precision=3)
    tr, va, te, standardizer = get_train_val_test_datasets(task="target", augment
                                                            =False)
    print("Train:\t", len(tr.X))
    print("Val:\t", len(va.X))
    print("Test:\t", len(te.X))
    print("Mean:", standardizer.image_mean)
    print("Std: ", standardizer.image_std)
```

B train_challenge.py

```
"""
EECS 445 - Introduction to Machine Learning
Winter 2021 - Project 2
Train Challenge
    Train a convolutional neural network to classify the heldout images
    Periodically output training information, and saves model checkpoints
    Usage: python train_challenge.py
"""
import torch
import numpy as np
import random
from dataset_challenge import get_train_val_test_loaders
from model.challenge import Challenge
from train_common import *
from utils import config
import utils
from sklearn import metrics
from torch.nn.functional import softmax

def main():
    # Data loaders
    if check_for_augmented_data("./data"):
        tr_loader, va_loader, te_loader, _ = get_train_val_test_loaders(
            task="target",
            batch_size=config("challenge.batch_size"), augment = True
        )
    else:
        tr_loader, va_loader, te_loader, _ = get_train_val_test_loaders(
            task="target",
            batch_size=config("challenge.batch_size"),
        )
    # Model
    model = Challenge()

    # TODO: define loss function, and optimizer
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr= 0.001)
    #

    # Attempts to restore the latest checkpoint if exists
    print("Loading challenge...")
    model, start_epoch, stats = restore_checkpoint(model, config("challenge.
                                            checkpoint"))

    axes = utils.make_training_plot()

    # Evaluate the randomly initialized model
    evaluate_epoch(
        axes, tr_loader, va_loader, te_loader, model, criterion, start_epoch,
                                                stats
    )

    # initial val loss for early stopping
```

```
        prev_val_loss = stats[0][1]

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

        # Loop over the entire dataset multiple times
        epoch = start_epoch
        while curr_patience < patience:
            # Train model
            train_epoch(tr_loader, model, criterion, optimizer)

            # Evaluate model
            evaluate_epoch(
                axes, tr_loader, va_loader, te_loader, model, criterion, epoch + 1,
                                                                    stats
            )

            # Save model parameters
            save_checkpoint(model, epoch + 1, config("challenge.checkpoint"), stats)

            #TODO: Implement early stopping
            curr_patience, prev_val_loss = early_stopping(
                stats, curr_patience, prev_val_loss
            )
            #
            epoch += 1
        print("Finished Training")
        # Save figure and keep plot open
        utils.save_challenge_training_plot()
        utils.hold_training_plot()

    if __name__ == "__main__":
        main()
```

C challenge.py

```
"""
EECS 445 - Introduction to Machine Learning
Winter 2021 - Project 2
Challenge
    Constructs a pytorch model for a convolutional neural network
    Usage: from model.challenge import Challenge
"""
import torch
import torch.nn as nn
import torch.nn.functional as F
from math import sqrt


class Challenge(nn.Module):
    def __init__(self):
        super().__init__()

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

        self.init_weights()

    def init_weights(self):
        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]
        conv = self.fc_c
        nn.init.normal_(conv.weight, 0.0, 1 / sqrt(64))
        nn.init.constant_(conv.bias, 0.0)
        ##

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

        ## TODO: forward pass
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = F.relu(self.conv3(x))
        x = x.view(-1, 64)
        x = self.fc_c(x)
        ##
        return x
```

D target.py

```
"""
EECS 445 - Introduction to Machine Learning
Winter 2021 - Project 2
Target CNN
    Constructs a pytorch model for a convolutional neural network
    Usage: from model.target import target
"""

import torch
import torch.nn as nn
import torch.nn.functional as F
from math import sqrt
from utils import config


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

        ## TODO: define each layer
        self.conv1 = nn.Conv2d(  in_channels = 3,
                                 out_channels= 16,
                                 kernel_size = (5,5),
                                 stride      = (2,2),
                                 padding     = 2  )
        self.pool = nn.MaxPool2d(kernel_size = (2,2),
                                 stride      = (2,2))
        self.conv2 = nn.Conv2d(  in_channels = 16,
                                 out_channels= 64,
                                 kernel_size = (5,5),
                                 stride      = (2,2),
                                 padding     = 2  )
        self.conv3 = nn.Conv2d(  in_channels = 64,
                                 out_channels= 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]

        conv = self.fc_1
        nn.init.normal_(conv.weight, 0.0, 1 / sqrt(32))
        nn.init.constant_(conv.bias, 0.0)
        ##

    def forward(self, x):
        N, C, H, W = x.shape
        ## TODO: forward pass
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = F.relu(self.conv3(x))
        x = x.view(-1, 32)
        x = self.fc_1(x)

        ##
        return x
```

E source.py

```
"""
EECS 445 - Introduction to Machine Learning
Winter 2021 - Project 2
Source CNN
    Constructs a pytorch model for a convolutional neural network
    Usage: from model.source import Source
"""
import torch
import torch.nn as nn
import torch.nn.functional as F
from math import sqrt
from utils import config


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

        ## TODO: define each layer
        self.conv1 = nn.Conv2d(  in_channels = 3,
                                 out_channels= 16,
                                 kernel_size = (5,5),
                                 stride      = (2,2),
                                 padding     = 2  )
        self.pool = nn.MaxPool2d(kernel_size = (2,2),
                                 stride      = (2,2))
        self.conv2 = nn.Conv2d(  in_channels = 16,
                                 out_channels= 64,
                                 kernel_size = (5,5),
                                 stride      = (2,2),
                                 padding     = 2  )
        self.conv3 = nn.Conv2d(  in_channels = 64,
                                 out_channels= 8,
                                 kernel_size = (5,5),
                                 stride      = (2,2),
                                 padding     = 2  )
        self.fc_2 = 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]
        conv = self.fc_2
        nn.init.normal_(conv.weight, 0.0, 1 / sqrt(32))
        nn.init.constant_(conv.bias, 0.0)

        ##
    def forward(self, x):
        N, C, H, W = x.shape
        ## TODO: forward pass
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = F.relu(self.conv3(x))
        x = x.view(-1, 32)
        x = self.fc_2(x)
        ##
        return x
```

F Augment_data.py

```
"""
EECS 445 - Introduction to Machine Learning
Winter 2021  - Project 2

Script to create an augmented dataset.
"""

import argparse
import csv
import glob
import os
import sys
import numpy as np
from scipy.ndimage import rotate
from imageio import imread, imwrite
import random
import cv2


def Flip():
    """Return function to flip image."""

    def _flip(img):
        """Return flip of image.
        :img: H x W x C numpy array
        :returns: H x W x C numpy array
        """
        # TODO
        return cv2.flip(img,1)
    return _flip

def Translate(dis = 5):
    """Return function to translate image in the range (-dis,dis)."""

    def _translate(img):
        # TODO
        out = np.float32([[1, 0, random.uniform(-dis,dis)],
                          [0, 1, random.uniform(-dis,dis)]])
        return cv2.warpAffine(img, out, (64,64) )
    return _translate


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
        theta = deg * (2  * np.random.rand() - 1)
        return rotate(img, theta, reshape=False)
    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
        gray = np.mean(img, axis= 2, keepdims = True)
        return np.dstack((gray,gray,gray))
    return _grayscale


def augment(filename, transforms, n=1, original=True):
    """Augment image at filename.

    :filename: TODO
    :transforms: List of image transformations
    :n: number of augmented images to save
    :returns: a list of augmented images, where the first image is the original

    """
    print(f"Augmenting {filename}")
    img = imread(filename)
    '''
    G = [Grayscale()]
    res = []
    for i in range(n):
        new = img
        for g in G:
            new = g(new)
        res.append(new)
    for i in range(n):
        new = img
        for transform in transforms:
            new = transform(new)
        res.append(new)
    return res
    '''
    res = [img] if original else []
    for i in range(n):
        new = img
        for transform in transforms:
            new = transform(new)
        res.append(new)
    return res

def main(args):
    """Create augmented dataset."""
    reader = csv.DictReader(open(args.input, "r"), delimiter=",")
    writer = csv.DictWriter(
        open(f"{args.datadir}/augmented_dogs.csv", "w"),
        fieldnames=["filename", "semantic_label", "partition", "numeric_label", "
                                                            task"],
    )
    augment_partitions = set(args.partitions)

    # TODO: change 'augmentations' to specify which augmentations to apply
    augmentations = [Grayscale()]

    writer.writeheader()
    os.makedirs(f"{args.datadir}/augmented/", exist_ok=True)
    for f in glob.glob(f"{args.datadir}/augmented/*"):
        print(f"Deleting {f}")
        os.remove(f)
    for row in reader:
        if row["partition"] not in augment_partitions:
            imwrite(
                f"{args.datadir}/augmented/{row['filename']}",
                imread(f"{args.datadir}/images/{row['filename']}"),
            )
            writer.writerow(row)
            continue
        imgs = augment(
            f"{args.datadir}/images/{row['filename']}",
            augmentations,
            n=1,
            original=False,  # TODO: change to False to exclude original image.
        )
        for i, img in enumerate(imgs):
            fname = f"{row['filename'][:-4]}_aug_{i}.png"
            imwrite(f"{args.datadir}/augmented/{fname}", img)
            writer.writerow(
                {
                    "filename": fname,
                    "semantic_label": row["semantic_label"],
                    "partition": row["partition"],
                    "numeric_label": row["numeric_label"],
                    "task": row["task"],
                }
            )


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("input", help="Path to input CSV file")
    parser.add_argument("datadir", help="Data directory", default="./data/")
    parser.add_argument(
        "-p",
        "--partitions",
        nargs="+",
        help="Partitions (train|val|test|challenge|none)+ to apply augmentations
                                                            to. Defaults to train",
        default=["train"],
    )
    main(parser.parse_args(sys.argv[1:]))
```

G train_cnn.py

```
"""
EECS 445 - Introduction to Machine Learning
Winter 2021  - Project 2
Train CNN
    Train a convolutional neural network to classify images
    Periodically output training information, and saves model checkpoints
    Usage: python train_cnn.py
"""
import torch
import numpy as np
import random
from dataset import get_train_val_test_loaders
from model.target import Target
from train_common import *
from utils import config
import utils

torch.manual_seed(42)
np.random.seed(42)
random.seed(42)

def main():
    """Train CNN and show training plots."""
    # Data loaders
    if check_for_augmented_data("./data"):
        tr_loader, va_loader, te_loader, _ = get_train_val_test_loaders(
            task="target", batch_size=config("cnn.batch_size"), augment=True
        )
    else:
        tr_loader, va_loader, te_loader, _ = get_train_val_test_loaders(
            task="target",
            batch_size=config("cnn.batch_size"),
        )

    # Model
    model = Target()

    # TODO: define loss function, and optimizer
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr= 0.001)
    #



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

    # Attempts to restore the latest checkpoint if exists
    print("Loading cnn...")
    model, start_epoch, stats = restore_checkpoint(model, config("cnn.checkpoint"
                                                 ))

    axes = utils.make_training_plot()
    # Evaluate the randomly initialized model
    evaluate_epoch(
        axes, tr_loader, va_loader, te_loader, model, criterion, start_epoch,
                                                     stats
    )

    # initial val loss for early stopping
    prev_val_loss = stats[0][1]

    # TODO: define patience for early stopping
    patience = 10
    curr_patience = 0
    #

    # Loop over the entire dataset multiple times
    # for epoch in range(start_epoch, config('cnn.num_epochs')):
    epoch = start_epoch


    while curr_patience < patience:
        # Train model
        train_epoch(tr_loader, model, criterion, optimizer)

        # Evaluate model
        evaluate_epoch(
            axes, tr_loader, va_loader, te_loader, model, criterion, epoch + 1,
                                                     stats
        )

        # Save model parameters
        save_checkpoint(model, epoch + 1, config("cnn.checkpoint"), stats)

        # update early stopping parameters
        curr_patience, prev_val_loss = early_stopping(
            stats, curr_patience, prev_val_loss
        )
        epoch += 1

    print("Finished Training")
    # Save figure and keep plot open
    utils.save_cnn_training_plot()
    utils.hold_training_plot()


if __name__ == "__main__":
    main()
```

H test_cnn.py

```
"""
EECS 445 - Introduction to Machine Learning
Winter 2021  - Project 2
Test CNN
    Test our trained CNN from train_cnn.py on the heldout test data.
    Load the trained CNN model from a saved checkpoint and evaulates using
    accuracy and AUROC metrics.
    Usage: python test_cnn.py
"""

import torch
import numpy as np
import random
from dataset import get_train_val_test_loaders
from model.target import Target
from train_common import *
from utils import config
import utils
from sklearn import metrics
from torch.nn.functional import softmax

torch.manual_seed(42)
np.random.seed(42)
random.seed(42)


def main():
    """Print performance metrics for model at specified epoch."""
    # Data loaders
    tr_loader, va_loader, te_loader, _ = get_train_val_test_loaders(
        task="target",
        batch_size=config("cnn.batch_size"),
    )

    # Model
    model = Target()

    # define loss function
    criterion = torch.nn.CrossEntropyLoss()

    # Attempts to restore the latest checkpoint if exists
    print("Loading cnn...")
    model, start_epoch, stats = restore_checkpoint(model, config("cnn.checkpoint"
                                                   ))

    axes = utils.make_training_plot()

    # Evaluate the model
    evaluate_epoch(
        axes,
        tr_loader,
        va_loader,
        te_loader,
        model,
        criterion,
        start_epoch,
        stats,
        include_test=True,
        update_plot=False,
    )


if __name__ == "__main__":
    main()
```

I train]_common.py

```
"""
EECS 445 - Introduction to Machine Learning
Winter 2021  - Project 2

Helper file for common training functions.
"""

from utils import config
import numpy as np
import itertools
import os
import torch
from torch.nn.functional import softmax
from sklearn import metrics
import utils


def count_parameters(model):
    """Count number of learnable parameters."""
    return sum(p.numel() for p in model.parameters() if p.requires_grad)


def save_checkpoint(model, epoch, checkpoint_dir, stats):
    """Save a checkpoint file to `checkpoint_dir`."""
    state = {
        "epoch": epoch,
        "state_dict": model.state_dict(),
        "stats": stats,
    }

    filename = os.path.join(checkpoint_dir, "epoch={}.checkpoint.pth.tar".format(
                                                    epoch))
    torch.save(state, filename)


def check_for_augmented_data(data_dir):
    """Ask to use augmented data if `augmented_dogs.csv` exists in the data
                                                    directory."""
    if "augmented_dogs.csv" in os.listdir(data_dir):
        print("Augmented data found, would you like to use it? y/n")
        print(">> ", end="")
        rep = str(input())
        return rep == "y"
    return False


def restore_checkpoint(model, checkpoint_dir, cuda=False, force=False, pretrain=
                                                    False):
    """Restore model from checkpoint if it exists.

    Returns the model and the current epoch.
    """
    try:
        cp_files = [
            file_
            for file_ in os.listdir(checkpoint_dir)
            if file_.startswith("epoch=") and file_.endswith(".checkpoint.pth.tar
                                                            ")
        ]
    except FileNotFoundError:
        cp_files = None
        os.makedirs(checkpoint_dir)
    if not cp_files:
        print("No saved model parameters found")
        if force:
            raise Exception("Checkpoint not found")
        else:
            return model, 0, []

    # Find latest epoch
    for i in itertools.count(1):
        if "epoch={}.checkpoint.pth.tar".format(i) in cp_files:
            epoch = i
        else:
            break

    if not force:
        print(
            "Which epoch to load from? Choose in range [0, {}].".format(epoch),
            "Enter 0 to train from scratch.",
        )
        print(">> ", end="")
        inp_epoch = int(input())
        if inp_epoch not in range(epoch + 1):
            raise Exception("Invalid epoch number")
        if inp_epoch == 0:
            print("Checkpoint not loaded")
            clear_checkpoint(checkpoint_dir)
            return model, 0, []
    else:
        print("Which epoch to load from? Choose in range [1, {}].".format(epoch))
        inp_epoch = int(input())
        if inp_epoch not in range(1, epoch + 1):
            raise Exception("Invalid epoch number")

    filename = os.path.join(
        checkpoint_dir, "epoch={}.checkpoint.pth.tar".format(inp_epoch)
    )

    print("Loading from checkpoint {}?".format(filename))

    if cuda:
        checkpoint = torch.load(filename)
    else:
        # Load GPU model on CPU
        checkpoint = torch.load(filename, map_location=lambda storage, loc:
                                                    storage)
    try:
        start_epoch = checkpoint["epoch"]
        stats = checkpoint["stats"]
        if pretrain:
            model.load_state_dict(checkpoint["state_dict"], strict=False)
        else:
            model.load_state_dict(checkpoint["state_dict"])
        print(
            "=> Successfully restored checkpoint (trained for {} epochs)".format(
                checkpoint["epoch"]
            )
        )
    except:
        print("=> Checkpoint not successfully restored")
        raise

    return model, inp_epoch, stats


def clear_checkpoint(checkpoint_dir):
    """Remove checkpoints in `checkpoint_dir`."""
    filelist = [f for f in os.listdir(checkpoint_dir) if f.endswith(".pth.tar")]
    for f in filelist:
        os.remove(os.path.join(checkpoint_dir, f))

    print("Checkpoint successfully removed")


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
    new_val_loss = stats[-1][1]
    if new_val_loss >= prev_val_loss:
        curr_patience += 1
    else:
        curr_patience = 0
        prev_val_loss = new_val_loss
    #
    return curr_patience, prev_val_loss


def evaluate_epoch(
    axes,
    tr_loader,
    val_loader,
    te_loader,
    model,
    criterion,
        epoch,
        stats,
        include_test=False,
        update_plot=True,
        multiclass=False,
):
    """Evaluate the `model` on the train and validation set."""

    def _get_metrics(loader):
        y_true, y_pred, y_score = [], [], []
        correct, total = 0, 0
        running_loss = []
        for X, y in loader:
            with torch.no_grad():
                output = model(X)
                predicted = predictions(output.data)
                y_true.append(y)
                y_pred.append(predicted)
                if not multiclass:
                    y_score.append(softmax(output.data, dim=1)[:, 1])
                else:
                    y_score.append(softmax(output.data, dim=1))
                total += y.size(0)
                correct += (predicted == y).sum().item()
                running_loss.append(criterion(output, y).item())
        y_true = torch.cat(y_true)
        y_pred = torch.cat(y_pred)
        y_score = torch.cat(y_score)
        loss = np.mean(running_loss)
        acc = correct / total
        if not multiclass:
            auroc = metrics.roc_auc_score(y_true, y_score)
        else:
            auroc = metrics.roc_auc_score(y_true, y_score, multi_class="ovo")
        return acc, loss, auroc

    train_acc, train_loss, train_auc = _get_metrics(tr_loader)
    val_acc, val_loss, val_auc = _get_metrics(val_loader)

    stats_at_epoch = [
        val_acc,
        val_loss,
        val_auc,
        train_acc,
        train_loss,
        train_auc,
    ]
    if include_test:
        stats_at_epoch += list(_get_metrics(te_loader))

    stats.append(stats_at_epoch)
    utils.log_training(epoch, stats)
    if update_plot:
        utils.update_training_plot(axes, epoch, stats)
```

```
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)
        loss = criterion(outputs, y)
        loss.backward()
        optimizer.step()






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

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

    # print(logits.shape) [32,2]
    _, pred = torch.max(logits.data, axis = 1)
    return pred
```

J train_source.py

```
"""
EECS 445 - Introduction to Machine Learning
Winter 2021 - Project 2
Train Source CNN
    Train a convolutional neural network to classify images.
    Periodically output training information, and saves model checkpoints
    Usage: python3 train_source.py
"""

import torch
import numpy as np
import random
from dataset import get_train_val_test_loaders
from model.source import Source
from train_common import *
from utils import config
import utils
from sklearn import metrics
import torch.nn.functional as F

torch.manual_seed(42)
np.random.seed(42)
random.seed(42)


def main():
    """Train source model on multiclass data."""
    # Data loaders
    tr_loader, va_loader, te_loader, _ = get_train_val_test_loaders(
        task="source",
        batch_size=config("source.batch_size"),
    )

    # Model
    model = Source()

    # TODO: define loss function, and optimizer
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr= 0.001, weight_decay=0.01
                                            )
    #

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

    # Attempts to restore the latest checkpoint if exists
    print("Loading source...")
    model, start_epoch, stats = restore_checkpoint(model, config("source.
                                            checkpoint"))

    axes = utils.make_training_plot("Source Training")

    # Evaluate the randomly initialized model
    evaluate_epoch(
        axes,
        tr_loader,
        va_loader,
        te_loader,
        model,
        criterion,
        start_epoch,
        stats,
        multiclass=True,
    )

    # initial val loss for early stopping
    prev_val_loss = stats[0][1]

    # TODO: patience for early stopping
    patience = 10
    curr_patience = 0
    #

    # Loop over the entire dataset multiple times
    epoch = start_epoch
    while curr_patience < patience:
        # Train model
        train_epoch(tr_loader, model, criterion, optimizer)

        # Evaluate model
        evaluate_epoch(
            axes,
            tr_loader,
            va_loader,
            te_loader,
            model,
            criterion,
            epoch + 1,
            stats,
            multiclass=True,
        )

        # Save model parameters
        save_checkpoint(model, epoch + 1, config("source.checkpoint"), stats)

        curr_patience, prev_val_loss = early_stopping(
            stats, curr_patience, prev_val_loss
        )
        epoch += 1

    # Save figure and keep plot open
    print("Finished Training")
    utils.save_source_training_plot()
    utils.hold_training_plot()


if __name__ == "__main__":
    main()
```

K train_target.py

```python
"""
EECS 445 - Introduction to Machine Learning
Winter 2021 - Project 2
Train Target
    Train a convolutional neural network to classify images.
    Periodically output training information, and saves model checkpoints
    Usage: python train_target.py
"""

import torch
import numpy as np
import random
from dataset import get_train_val_test_loaders
from model.target import Target
from train_common import *
from utils import config
import utils
from sklearn import metrics
import torch.nn.functional as F
import copy

torch.manual_seed(42)
np.random.seed(42)
random.seed(42)


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

    i = 0
    # print(num_layers)
    for name, param in model.named_parameters():
        if i < 2 * num_layers:
            # print(name)
            param.requires_grad = False
        i = i + 1
    # print("done")




def train(tr_loader, va_loader, te_loader, model, model_name, num_layers=0):
    """Train transfer learning model."""
    #TODO: define loss function, and optimizer
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(),lr = 0.001)
    #

    print("Loading target model with", num_layers, "layers frozen")
    model, start_epoch, stats = restore_checkpoint(model, model_name)
    axes = utils.make_training_plot("Target Training")

    evaluate_epoch(
        axes,
        tr_loader,
        va_loader,
        te_loader,
        model,
        criterion,
        start_epoch,
        stats,
        include_test=True,
    )

    # initial val loss for early stopping
    prev_val_loss = stats[0][1]

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

    # Loop over the entire dataset multiple times
    epoch = start_epoch
    while curr_patience < patience:
        # Train model
        train_epoch(tr_loader, model, criterion, optimizer)

        # Evaluate model
        evaluate_epoch(
            axes,
            tr_loader,
            va_loader,
            te_loader,
            model,
            criterion,
            epoch + 1,
            stats,
            include_test=True,
        )

        # Save model parameters
        save_checkpoint(model, epoch + 1, model_name, stats)

        curr_patience, prev_val_loss = early_stopping(
            stats, curr_patience, prev_val_loss
        )
        epoch += 1

    print("Finished Training")

    # Keep plot open
    utils.save_tl_training_plot(num_layers)
    utils.hold_training_plot()
```

```
def main():
    """Train transfer learning model and display training plots.

    Train four different models with {0, 1, 2, 3} layers frozen.
    """
    # data loaders
    tr_loader, va_loader, te_loader, _ = get_train_val_test_loaders(
        task="target",
        batch_size=config("target.batch_size"),
    )

    freeze_none = Target()
    print("Loading source...")
    freeze_none, _, _ = restore_checkpoint(
        freeze_none, config("source.checkpoint"), force=True, pretrain=True
    )

    freeze_one = copy.deepcopy(freeze_none)
    freeze_two = copy.deepcopy(freeze_none)
    freeze_three = copy.deepcopy(freeze_none)

    freeze_layers(freeze_one, 1)
    freeze_layers(freeze_two, 2)
    freeze_layers(freeze_three, 3)

    train(tr_loader, va_loader, te_loader, freeze_none, "./checkpoints/target0/",
                                               0)
    train(tr_loader, va_loader, te_loader, freeze_one, "./checkpoints/target1/",
                                              1)
    train(tr_loader, va_loader, te_loader, freeze_two, "./checkpoints/target2/",
                                              2)
    train(tr_loader, va_loader, te_loader, freeze_three, "./checkpoints/target3/"
                                              , 3)


if __name__ == "__main__":
    main()
```