# EECS 445 Project 2

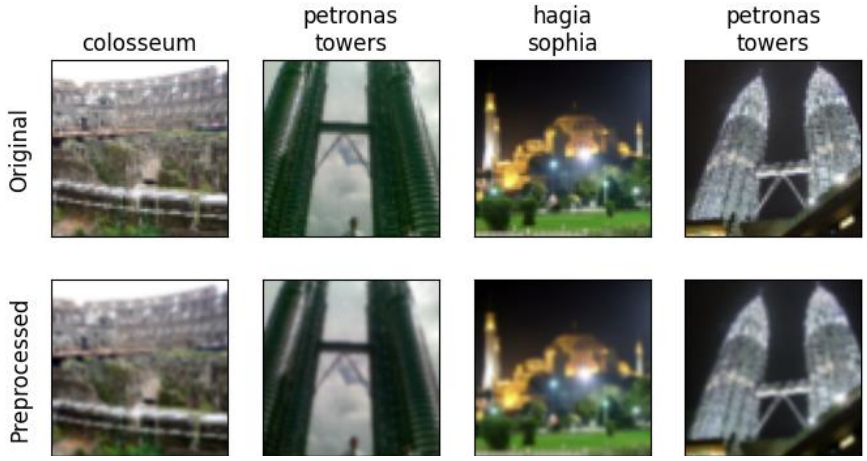
Shiyu Liu

## Question 1

(a) i. The mean of RGB color channels are [118.263 118.004 118.594] respectively and standard deviation of RGB color channels are [66.307 69.039 75.332] respectively.

ii. We extract the mean and standard deviation from the training set to ensure a consistency across all data partitions which makes it easier to compare the performances. It also helps the model to better generalize from the training data to unseen data.

(b)

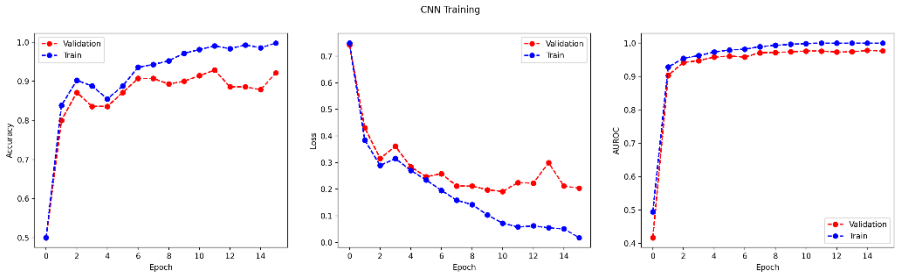



Question 2

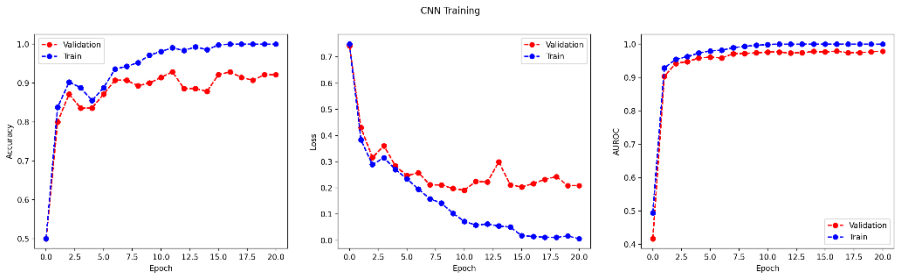
(a) Convolutional Layer 1: $(3 \times 5 \times 5 + 1) \times 16 = 1216$
Convolutional Layer 2: $(16 \times 5 \times 5 + 1) \times 64 = 25664$
Convolutional Layer 3: $(64 \times 5 \times 5 + 1) \times 8 = 12808$
Fully Connected Layer 1: $(32 + 1) \times 2 = 66$
Total learnable parameters: $1216 + 25664 + 12808 + 66 = 39754$

(f) i. One possible reason for this behavior is overfitting. As the model gets more complex, it might capture noise that reduces validation performance. Another possible reason might be the early stopping criteria. If the patience parameter for early stopping is too small, the training may terminate prematurely based on minor fluctuations in the validation loss.

ii.




Patience = 5




Patience = 10

The model stops at epoch 20. Patience value of 10 works better because it results in a more stabilized validation loss compared to a patience value of 5. In a dataset with relatively large noise and more outliers, it might be better to increase patience value because it allows the model to recover from sudden changes and allow the model to continuously improve.

iii. The new size of the input to the fully connected layer is $64 \times 2 \times 2 = 256$.

| | Epoch | Training AUROC | Validation AUROC |
|---|---|---|---|
| 8 filters | 10 | 0.9987 | 0.9761 |
| 64 filters | 7 | 0.9993 | 0.9698 |

The Validation AUROC decreases as we increase the number of filters from 8 to 64. The mismatch between dataset and model complexity might account for this change: when dataset is relatively simple and can be well represented with a smaller number of filters, increasing the number of filters may lead to an overfitted model that struggles to capture relevant features which causes a decrease in the Validation AUROC.

(g) i.

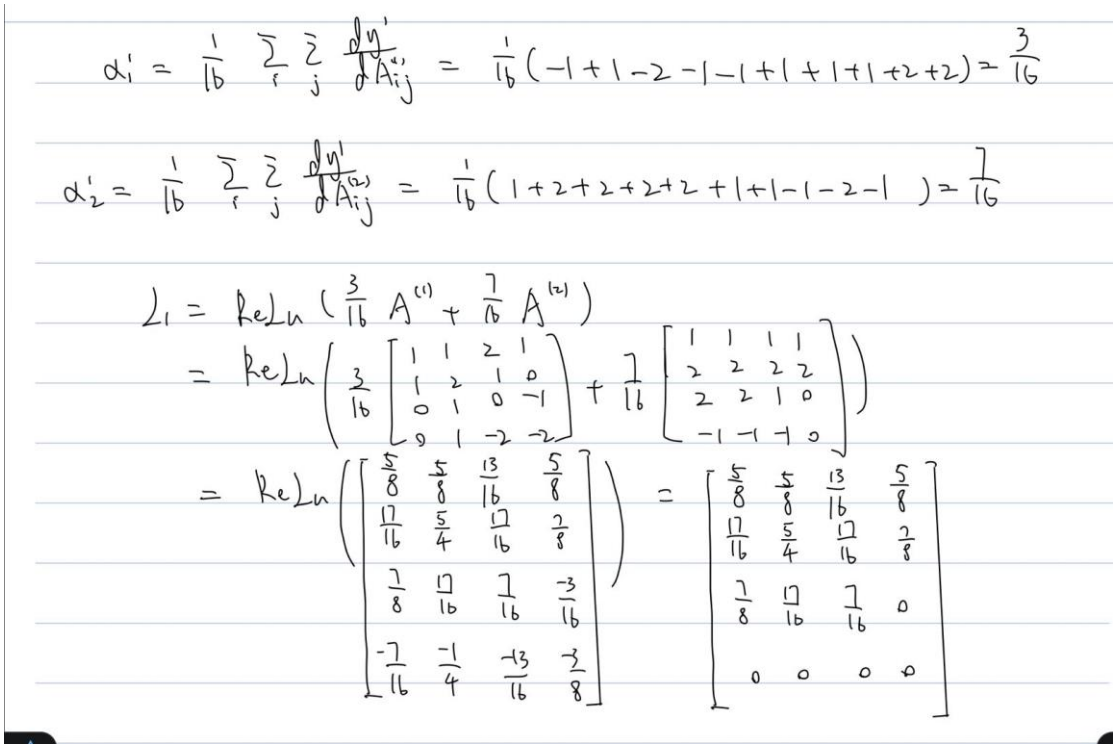
| | Training | Validation | Testing |
|---|---|---|---|
| Accuracy | 0.981 | 0.9143 | 0.6161 |
| AUROC | 0.9987 | 0.9761 | 0.684 |

ii. There is some evidence of overfitting: the training accuracy is very close to 1 and much higher than validation accuracy.

iii. The performance of validation is much better than test. One possible hypothesis for this issue is lack of interpretability of CNN: the model might not emphasize the pixels that represent the Pantheon and Hofburg Imperial Palace but on other pixels that are irrelevant for the subject.

Question 3

(a)



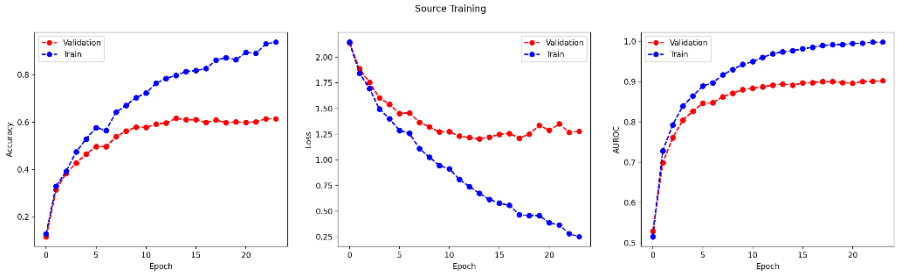
(b) The CNN appears to use features with light colors to identify the Hofburg Imperial Palace class. Many white and light blue objects such as the sky and clothes have high values in the Viridis color scale.

(c) The Grad-Cam visualizations is inline with my predictions since the pixels that have high intensity is irrelevant to the prediction of the Hofburg Imperial Palace class. The pixels that are relevant to the architectures doesn't have a high weight so the model doesn't perform well on the test set.

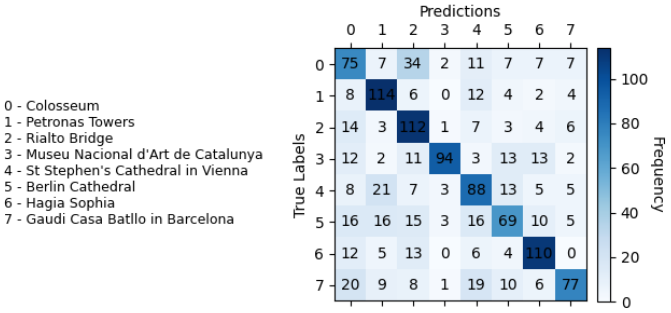
Question 4

4.1 Transfer Learning

(c) The epoch number with the lowest validation loss is epoch 13.



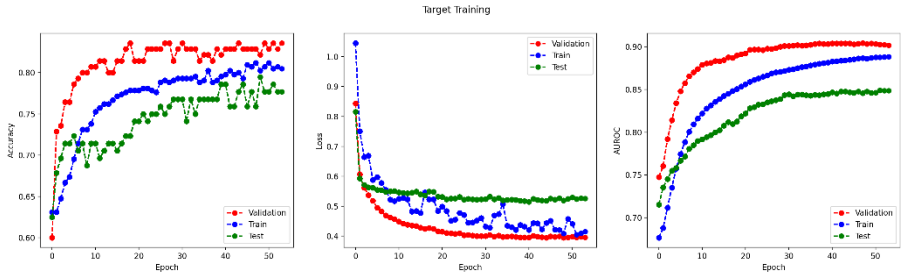

(d)



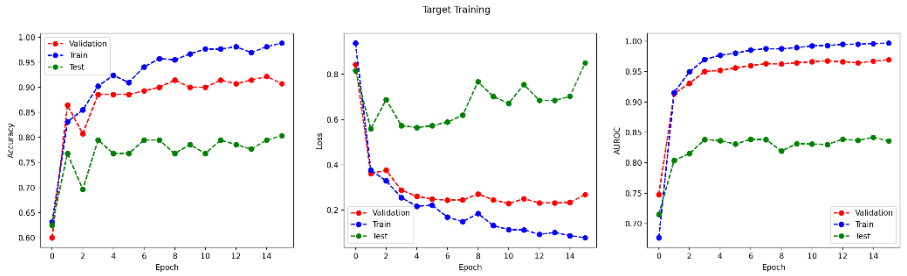

The most accurate landmark is 1 – Petronas Towers and the least accurate landmark is 5 - Berlin Cathedral. A possible reason for this result is because Petronas Towers has the lightest color compared with its surroundings whereas the Berlin Cathedral has the darkest color compared with its surroundings. If the landmark has light colors, the model can capture its shape and make more accurate predictions.

(f)

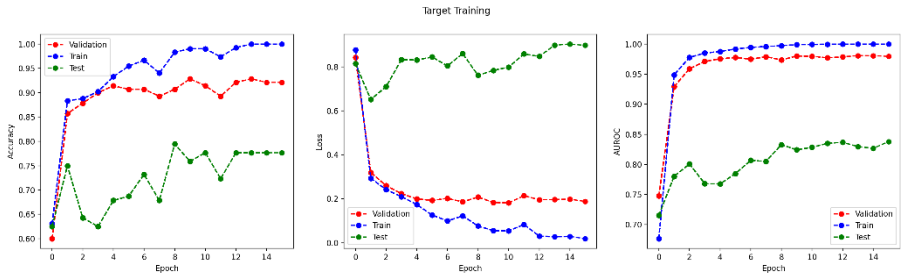
| | AUROC | | |
|---|---|---|---|
| | TRAIN | VAL | TEST |
| Freeze all CONV layers (Fine-tune FC layer) | 0.8878 | 0.9027 | 0.8489 |
| Freeze first two CONV layers (Fine-tune last CONV and FC layers) | 0.9922 | 0.9663 | 0.8307 |
| Freeze first CONV layer (Fine-tune last 2 conv. and fc layers) | 0.9994 | 0.9798 | 0.8284 |
| Freeze no layers (Fine-tune all layers) | 1.0 | 0.9837 | 0.8138 |
| No Pretraining or Transfer Learning (Section 2(g) performance) | 0.9987 | 0.9761 | 0.684 |



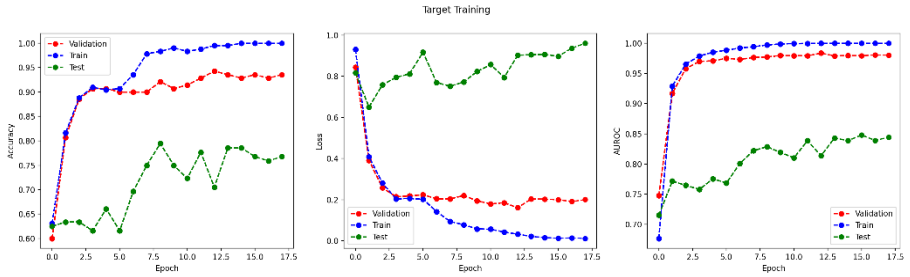

Freeze all CONV layers (Fine-tune FC layer)




Freeze first two CONV layers (Fine-tune last CONV and FC layers)




Freeze first CONV layer (Fine-tune last 2 conv. and fc layers)



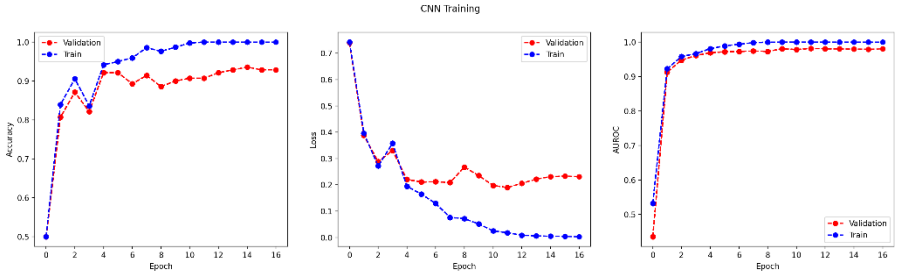

Freeze no layers (Fine-tune all layers)

From the table above, we can see that while some models have lower training and validation AUROC than the model from 2g, all of them has higher testing AUROC than the original model (the model that freezes all layers have the best performance).
Comparing with CNN in section 2, we can see that transfer learning is definitely helpful, since all of the test AUROCs from transfer learning models are better than the original model. The source task is also helpful because it helps us determine the epoch with the lowest validation loss in the source training data so we can better generalize the target model and reduces potential bias. I hypothesize it is helpful because the source task is like bootstrapping: it finds out the best result using the population and then use it to generalize one portion of it. It can achieve better performance because it leverages more knowledge gained from the source than just from the limited target data. When all layers

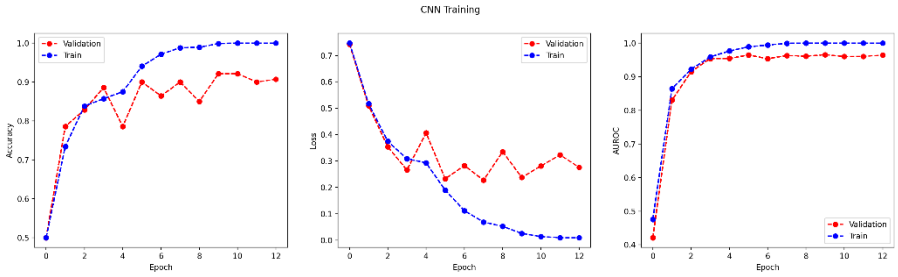
are frozen, the training curve takes very long time to converge and the training loss is lower than both validation and testing loss, whereas for other models, the convergence rate is faster and the validation losses are lower than the training losses. As more layers are frozen, the training and validation AUROC decreases whereas the testing AUROC increases. There are a few possible explanations for this: The original model is overfitted to the target. As more layers are frozen, less weight and bias are influenced by training, but this causes a higher testing result, indicating that the original model fits too closely to the training data. Another reason is the lack of relevance for the task. By freezing more layers, we restrict the model from learning task-specific features, causing a decrease in AUROC for both training and validation. However, the testing AUROC still improves because the model is trained more can still provide a valuable information for classification.

4.2 Data Augmentation

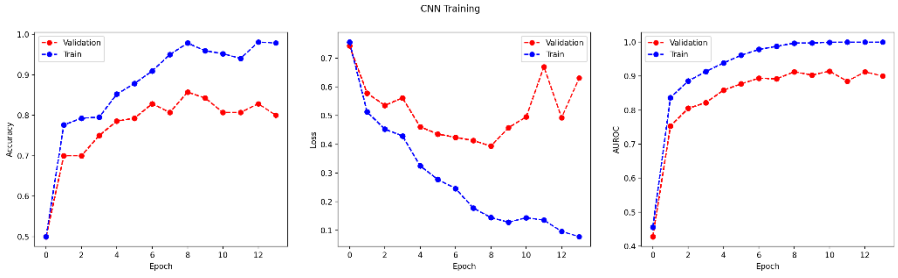
(b) iii.




Rotation (keep original)




Greyscale (keep original)




Grayscale (discard original)

| | AUROC | | |
|---|---|---|---|
| | TRAIN | VAL | TEST |
| Rotation (keep original) | 1.0 | 0.9816 | 0.7423 |
| Grayscale (keep original) | 0.9993 | 0.9631 | 0.7682 |
| Grayscale (discard original) | 0.997 | 0.9129 | 0.8058 |
| No augmentation (Section 2(g) performance) | 0.9987 | 0.9761 | 0.684 |

All of these augmentations provide better performance than the original model. Grayscale (discard original) has the highest test AUROC comparing with other augmentations, but also have the lowest training and validation AUROC.

(c) Yes, the validation and training plots change as I apply these augmentation techniques. One possible reason for this might be that these augmentations increase data variability. When modified data is introduced to the original set, the model may have stronger interpretability when dealing with a specific task because it has strong generalization power through training with the new data. Another possible reason is that the augmentation has regularization effect that prevents overfitting: The larger variety of data prevents the model to overfit to a specific form and also results in more changes in the training curve.

Question 5 Challenge

In my model, I used the same criterion and optimizer as Appendix B. I tried to use weight decay, but the result is better without it. I also tried patience value between 5 and 10 and finally decided on 7, a value in-between. Patience of 5 converges too quickly, leading to a lot of variations in the learning curves and patience of 10 takes too many epochs to converge (about 70). For the model architectures, I added more convolutional layers with larger input and output sizes, added more pooling layers and reduces the sizes of kernel, stride and padding. I gained inspiration from the architecture that I learned from another course I'm currently taking (EECS 442). Large input and output dimensions provide a richer source of information which allow for more complex features to be learned. Smaller kernel sizes also allow for more intricate feature extraction at different levels of abstraction. Combining these would potentially increase predictive power. For transfer learning, I tried to freeze different numbers of layers but they didn't produce better result than the original model, so I decide not to use it. For Data Augmentation, I tried all three kinds of augmentation we did in 4.2 and discovered that Grayscale (discard original) produces the best result, So I used it in my model. For model evaluation, I used lowest validation loss to find the best checkpoint and used test AUROC to determine which model is the best because it is a comprehensive metric that generalizes all aspects of the model's performance.

dataset.py

```
def fit(self, X):
        """Calculate per-channel mean and standard deviation from dataset X.
        Hint: you may find the axis parameter helpful"""
        # TODO: Complete this function
        self.image_mean = np.mean(X, axis=(0, 1, 2))
        self.image_std = np.std(X, axis=(0, 1, 2))

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

model/target.py

```
   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 = nn.Conv2d(3, 16, 5, 2, 2)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(16, 64, 5, 2, 2)
        self.conv3 = nn.Conv2d(64, 8, 5, 2, 2)
        self.fc_1 = nn.Linear(in_features=32, out_features=2)

        self.init_weights()

    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]
        input_size = self.fc_1.weight.size(1)
        nn.init.normal_(self.fc_1.weight, 0.0, 1 / sqrt(input_size))
        nn.init.constant_(self.fc_1.bias, 0.0)

    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.conv1(x)
        x = nn.functional.relu(x)
        x = self.pool(x)

        x = self.conv2(x)
        x = nn.functional.relu(x)
        x = self.pool(x)

        x = self.conv3(x)
        x = nn.functional.relu(x)

        x = x.view(x.size(0), -1)
        x = self.fc_1(x)
        return x
```

train_common.py

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

    Example: Given tensor([[0.2, -0.8], [-0.9, -3.1], [0.5, 2.3]]), return tensor([0, 0, 1])

    Returns:
        the predicted class output as a PyTorch Tensor
    """
    # TODO implement predictions
    pred = torch.argmax(logits, dim=1)
    return pred

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

    Increment curr_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_patience and global_min_loss
    """
    # TODO implement early stopping
    val_loss = stats[-1][1]

    if val_loss < global_min_loss:
        global_min_loss = val_loss
        curr_count_to_patience = 0
    else:
        curr_count_to_patience += 1
    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()
        forward = model(X)
        loss = criterion(forward, y)
        loss.backward()
        optimizer.step()
```

train_cnn.py

```python
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("target.batch_size"), augment=True
        )
    else:
        tr_loader, va_loader, te_loader, _ = get_train_val_test_loaders(
            task="target",
            batch_size=config("target.batch_size"),
        )
    # Model
    model = Target()

    # TODO: Define loss function and optimizer. Replace "None" with the appropriate definitions.
    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
    print("Loading cnn...")
    model, start_epoch, stats = restore_checkpoint(model, config("target.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
    global_min_loss = stats[0][1]

    # TODO: Define patience for early stopping. Replace "None" with the patience value.
    patience = 5
    curr_count_to_patience = 0
```

model/source.py

```
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 = nn.Conv2d(3, 16, 5, 2, 2)
        self.conv2 = nn.Conv2d(16, 64, 5, 2, 2)
        self.conv3 = nn.Conv2d(64, 8, 5, 2, 2)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(in_features=32, out_features=8)

        self.init_weights()

    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]
        input_size = self.fc1.weight.size(1)
        nn.init.normal_(self.fc1.weight, 0.0, 1 / sqrt(input_size))
        nn.init.constant_(self.fc1.bias, 0.0)

    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.conv1(x)
        x = nn.functional.relu(x)
        x = self.pool(x)

        x = self.conv2(x)
        x = nn.functional.relu(x)
```

```
        x = self.pool(x)

        x = self.conv3(x)
        x = nn.functional.relu(x)

        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        return x
```

train_source.py

```
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. Replace "None" with the appropriate definitions.
    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))

    # 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
    global_min_loss = stats[0][1]

    # TODO: Define patience for early stopping. Replace "None" with the patience value.
    patience = 10
    curr_count_to_patience = 0
```

train_target.py

```
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
    layers = 0
    # print(num_layers)
    for name, param in model.named_parameters():
        if 'conv' in name and layers < num_layers * 2:
            # print(name)
            param.requires_grad = False
            layers += 1


def train(tr_loader, va_loader, te_loader, model, model_name, num_layers=0):
    """Train transfer learning model."""
    # TODO: Define loss function and optimizer. Replace "None" with the appropriate definitions.
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

    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
    global_min_loss = stats[0][1]

    # TODO: Define patience for early stopping. Replace "None" with the patience value.
    patience = 5
    curr_count_to_patience = 0
```

augment_data.py

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

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

        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: implement _rotate(img)
        angle = np.random.randint(-deg, deg + 1)
        rotated_im = rotate(img, angle=angle, reshape=False, mode='constant', cval=0)
        return rotated_im

    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.

        Round to the nearest integer.

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

        """
        # TODO: implement _grayscale(img)
        grayscale = np.mean(img, axis=2, keepdims=True)
        grayscale = np.round(grayscale).astype(np.uint8)
        return np.repeat(grayscale, 3, axis=2)

    return _grayscale

def main(args):
    """Create augmented dataset."""
    reader = csv.DictReader(open(args.input, "r"), delimiter=",")
    writer = csv.DictWriter(
        open(f"{args.datadir}/augmented_landmarks.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.
```