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.
```

```python
        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

```python
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)
```

```python
        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

```python
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

```python
def freeze_layers(model, num_layers=0):
    """Stop tracking gradients on selected layers."""
    # TODO: modify model with the given layers frozen
    #      e.g. if num_layers=2, freeze CONV1 and CONV2
    #      Hint: https://pytorch.org/docs/master/notes/autograd.html
    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()
```

```python
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.
```