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

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

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

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.