1.a.i)
Mean of channel R: 124.495
Mean of channel G: 118.847
Mean of channel B: 95.293

Standard Deviation of channel R: 62.754
Standard Deviation of channel G: 59.597
Standard Deviation of channel B: 62.425

1.a.ii)
By extracting the mean and standard deviation from the training set as opposed to other data partitions, we are not taking into account the structure of the data partitions that we use to evaluate our model. The validation and test performance helps us evaluate our model by making sure our model generalizes well to unseen data.

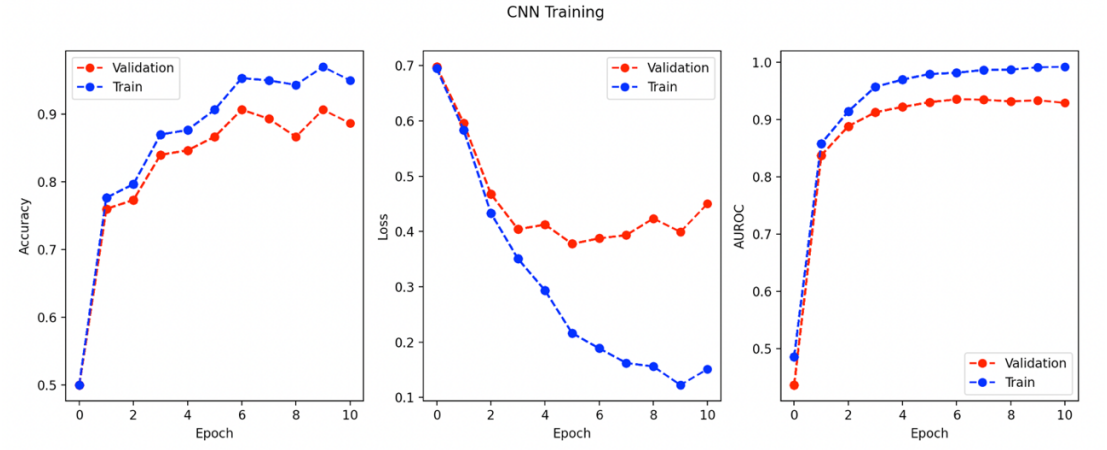
1.b)

2.a)
Conv1: 5 * 5 * 3 * 16 * 16 + 16 = 1,216
Pool: 0 + 0 = 0
Conv2: 5 * 5 * 16 * 64 + 64 = 25,664
Pool: 0 + 0 = 0
Conv3: 5 * 5 * 64 * 8 + 8 = 12,808
Fc_1: 32 * 2 + 2 = 66

There are 1,216 + 25,664 + 12,808 + 66 = 39,754 learnable float-valued parameters.

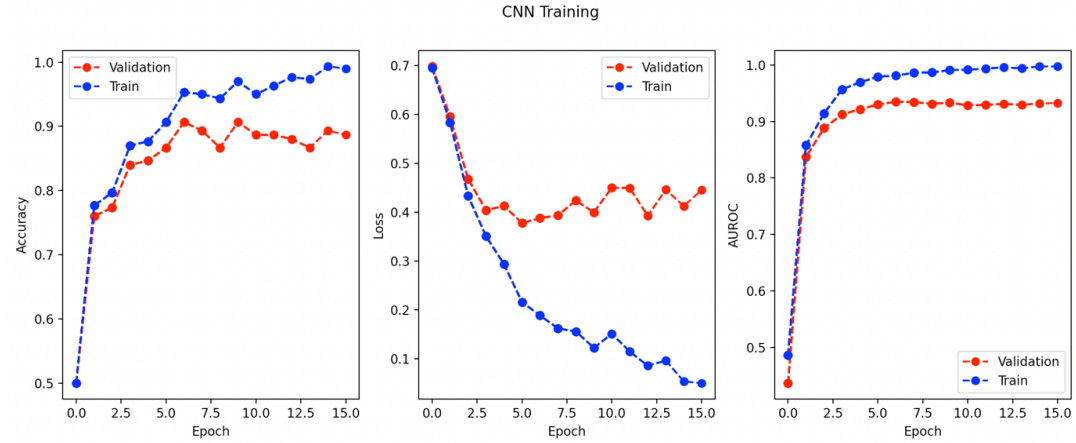
2.f.i)
We can observe that the validation loss doesn't monotonically decrease in the training plot. This is because we randomly shuffle training data at the end of every epoch. Also, the noise in validation loss can be induced from data as it contains noise such as color balance.

2.f.ii)
Training plot generated using patience = 5:



The model stopped training at epoch = 10

Training plot generated using patience = 10:



The model stopped training at epoch = 15

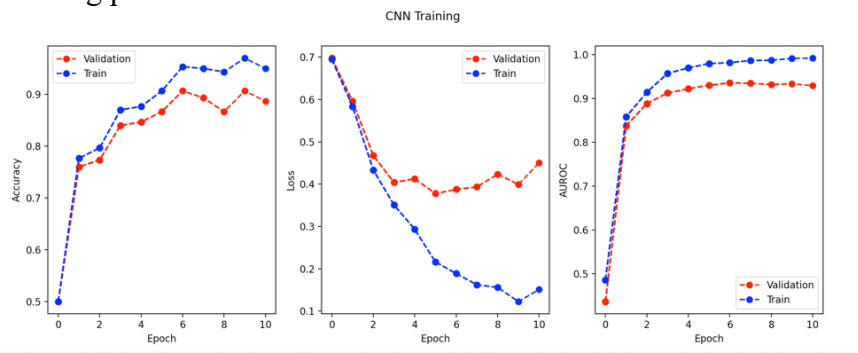
Patience = 10 works better for this data set as it converges better for all three graphs. The training plot generated using patience = 5 looks like it is still decreasing so increasing the patience ensures that we have a model that has better predictions.

Increased patience might work better if we haven't detected convergence to a specific value. Then by increasing patience, the objective function will be optimized.

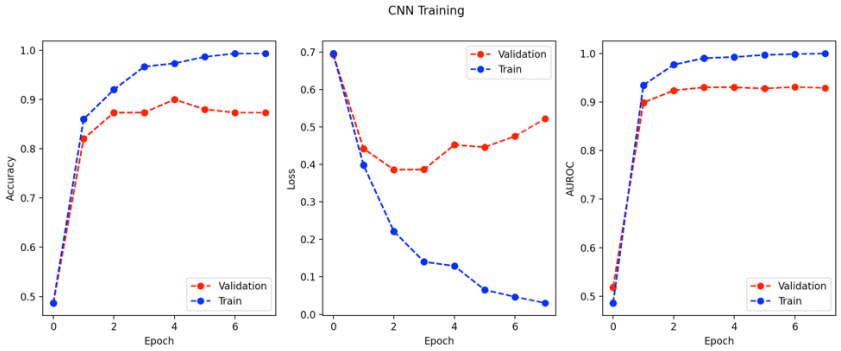
2.f.iii)
New size of the input to the fully connected layer = 256

	Epoch	**Training AUROC**	**Validation AUROC**
8 filters	5	0.9793	0.93
64 filters	2	0.9768	0.9239

Training plot for 8 filters:



Training plot for 64 filters:



As we increased the number of filters from 8 to 64, there is a larger gap between train and validation performance. The curves are smoother, and it was trained for fewer epochs.

Because the model has more parameters, it is more likely to overfit to the training data which resulted in a larger gap. Also, due to having more parameters, we are converging quicker so the curves appear smoother.

2.g.i)

	Training	**Validation**	**Testing**
Accuracy	0.9067	0.8667	0.6
AUROC	0.9793	0.93	0.6556

2.g.ii)
The training and validation performance only differs by .04 which means that there is no evidence of overfitting.

2.g.iii)
The testing performance is much lower than the validation performance. This indicates that our validation data was not representative of the testing data. A possible explanation for such a trend could be validation data containing features that are not included in the test data such as validation data containing all the white-colored dogs while training data doesn't contain any.

3.a)

$$\alpha_1^1 = \frac{1}{16}\sum_{i=1}^{4}\sum_{j=1}^{4}\frac{\partial y^1}{\partial A_{ij}^{(1)}} = \frac{1}{16}(-1+1-2-1-1+1+1+1+2+2) = \frac{3}{16}$$

$$\alpha_2^1 = \frac{1}{16}\sum_{i=1}^{4}\sum_{j=1}^{4}\frac{\partial y^1}{\partial A_{ij}^{(2)}} = \frac{1}{16}(1+2+2+2+2+1+1+(-1)-2-1) = \frac{7}{16}$$

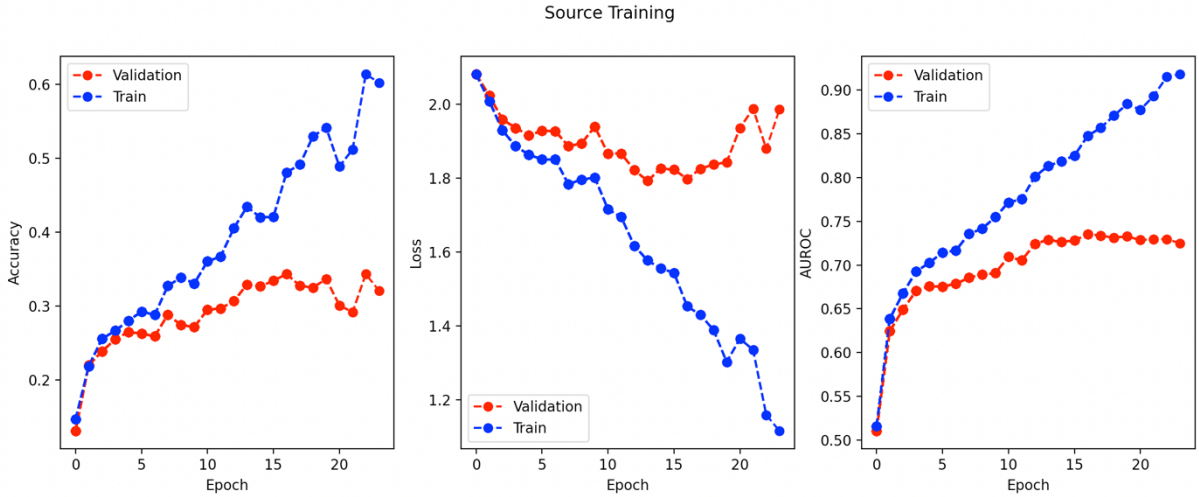
$$L^1 = ReLU(\alpha_1^1 \times A^{(1)} + \alpha_2^1 \times A^{(2)})$$

$$= \frac{1}{16} \times \begin{bmatrix} 10 & 10 & 13 & 10 \\ 17 & 20 & 17 & 14 \\ 14 & 17 & 7 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

3.b)
CNN appears to be using background features and using the features of the dog itself to identify the Collie class.

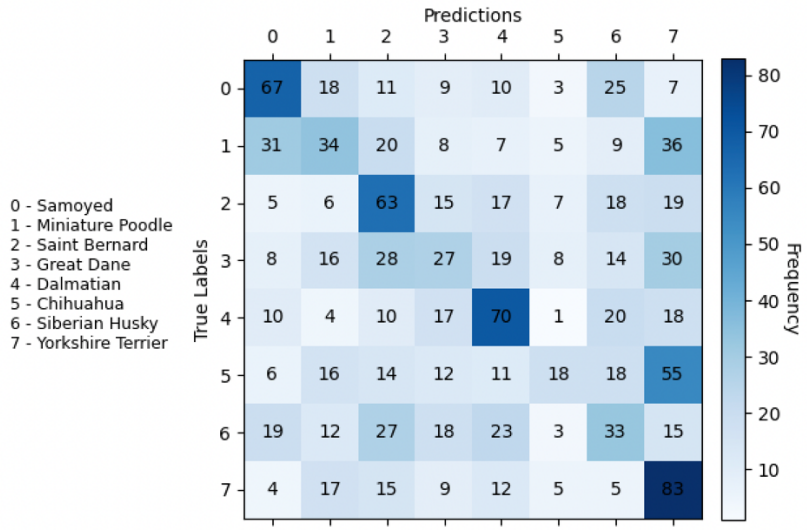
3.c)
The Grad-Cam visualizations confirm our hypothesis that the training and validation set fails to be representative of the test set. Because our model uses background features to identify the Collie class, we can infer that in the training and validation set, backgrounds are critical and helps our model classify the dogs. Therefore, our test performance was lower as classifying dogs based on backgrounds is not generalizable to other scenarios.

4.1.c)



Epoch = 13 resulted in lowest validation loss of 1.7923

4.1.d)

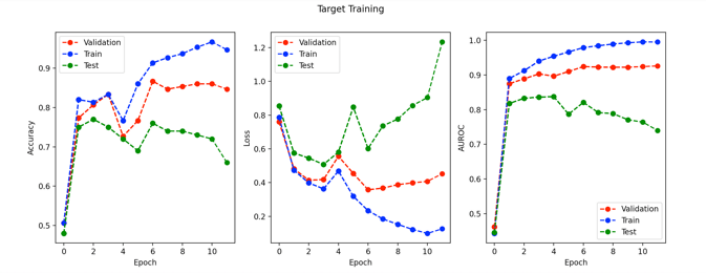


The classifier was most accurate for Yorkshire Terrier, Dalmatian, and Samoyed while it was least accurate for Chihuahua. This might be the case as our model classified most images as either a Yorkshire Terrier or a Dalmatian. Dogs of small sizes such as a Miniature Poodle and Chihuahua were mostly classified as a Yorkshire Terrier while dogs of bigger sizes such as Siberian Husky were classified as a Dalmatian. The classifier was least accurate for Chihuahua as it has similar size as a Yorkshire Terrier while not having a distinct color as a Dalmatian.

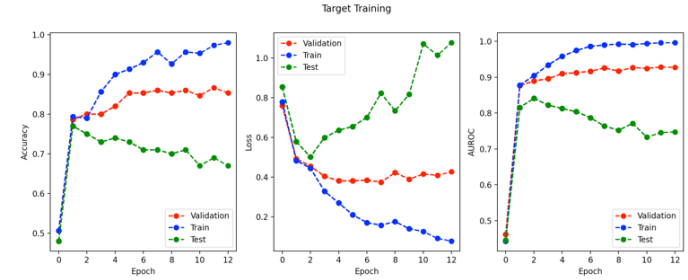
4.1.f)

	AUROC		
	TRAIN	**VAL**	**TEST**
Freeze all CONV layers	0.9783	0.9145	0.8116
Freeze first two CONV layers	0.9822	0.9116	0.7964
Freeze first CONV layer	0.9898	0.9259	0.7636
Freeze no layer	0.9789	0.9241	0.8208
No Pretraining or Transfer Learning (Section 2)	0.9793	0.93	0.6556

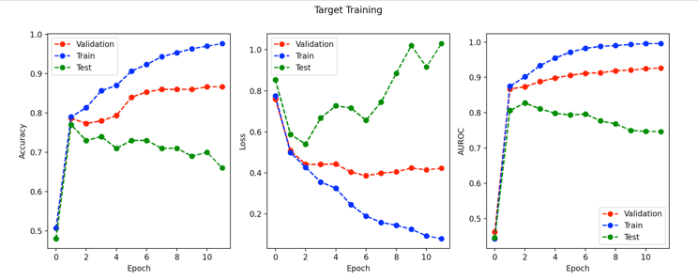
Plot for Freeze no layers (Fine-tune all layers):



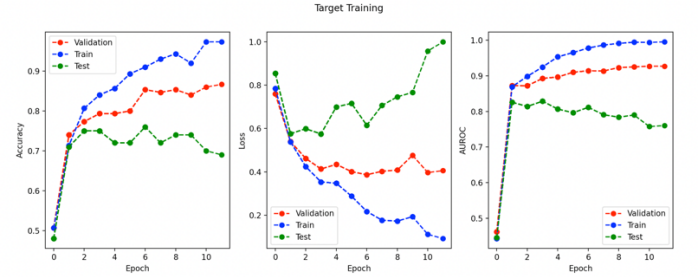
Plot for Freeze first CONV layer (Fine-tune last 2 conv. and fc layers):



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



Plot for Freeze all CONV layers (Fine-tune FC layer):

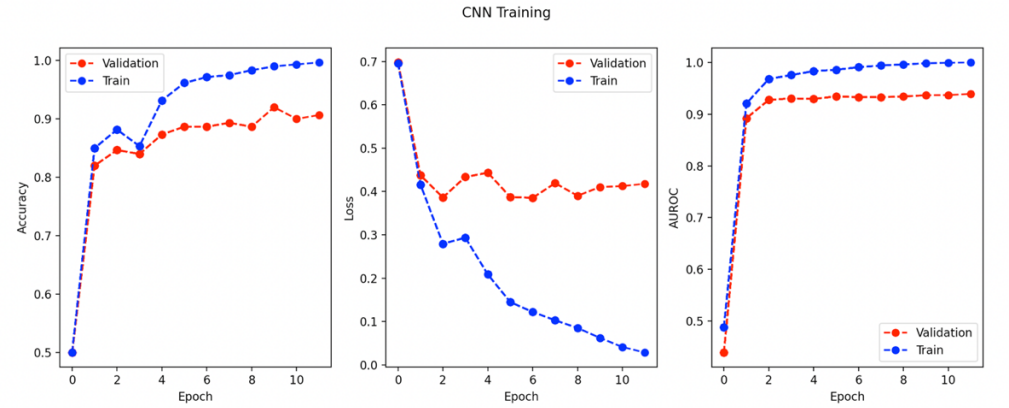


Looking at the test performance, we can see that the test AUROC isn't significantly smaller than the train and validation AUROC. Therefore, we can infer that the source task was helpful as train and validation data were representative of the test data. The transfer learning resulted in better performance than the Section 2 performance, even with freezing none. Freezing all convolutional layers resulted in a decrease of test AUROC compared to freezing just a subset or freezing none. This is because the model isn't able to learn the bias.

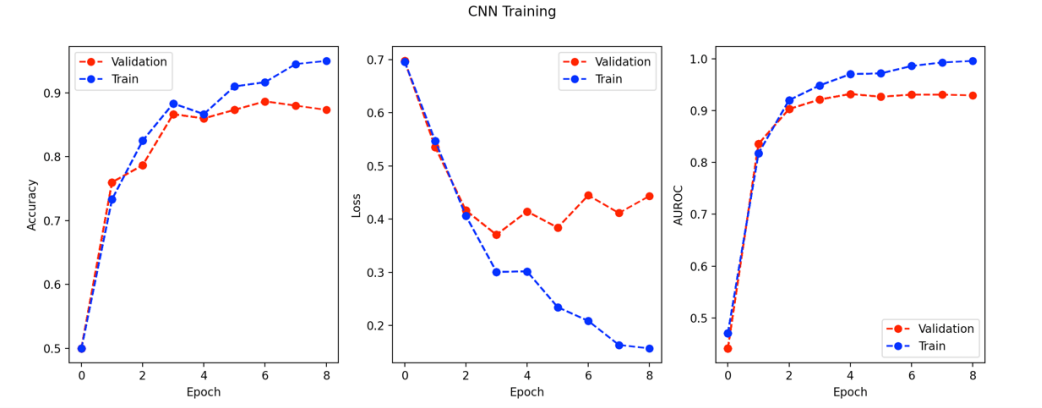
4.2.iii)

	AUROC		
	TRAIN	**VAL**	**TEST**
Rotation (keep original)	0.9913	0.9323	0.6608
Grayscale (keep original)	0.9761	0.9193	0.7248
Grayscale (discard original)	0.8829	0.7913	0.7752
No augmentation (Section 2 performance)	0.9793	0.93	0.6556

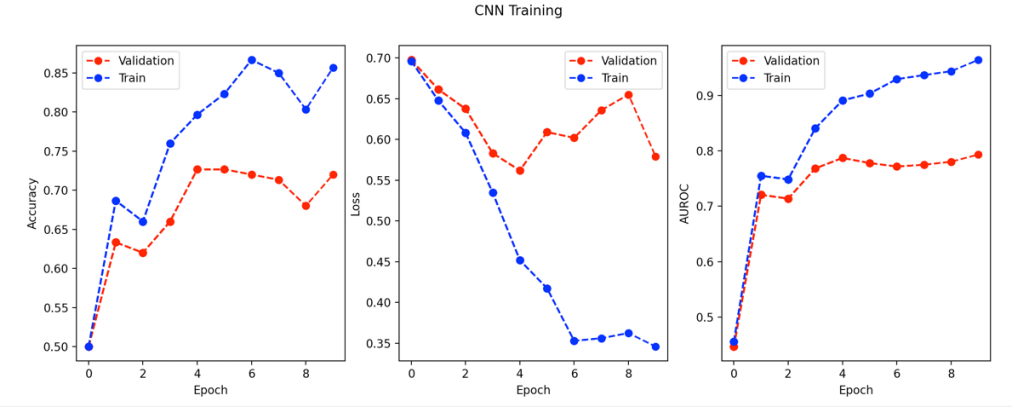
Training Plot for Rotation (keep original):



Training Plot for Grayscale (keep original):



Training Plot for Grayscale (discard original):



4.2.c)
Rotation and Grayscale (keep original) training plots are similar to that of the Section 2. However, Grayscale (discard original) training plots display much worse performance than that of Section 2. This is because the Grayscale (discard original) hinders the model from classifying dogs based off on irrelevant features such as background while Rotation and Grayscale (keep original) doesn't.

5)
I have decided to keep the model architecture and use transfer learning and data augmentation for this part. Even when I tried to increase the number of convolutional layers, the performance wasn't any different from what we had implemented using Appendix B and C. Therefore, I assumed that the model architecture was complex enough to classify Collies and Golden Retrievers.

From (4.1.f), I discovered that Transfer learning gave a much better performance result even when no layers were frozen compared to the performance when no transfer learning was conducted. Although the chart from (4.1.f) shows that freezing all three layers yielded highest test performance, I got the same performance result for freezing one, two, and all layers. Therefore, I froze all layers because as we freeze more layers, the model will not be able to learn any biases in the dataset.

For the data augmentation, from (4.2.iii) we learned that if the training and the validation set is not representative of the test data, data augmentation helps our model generalize to the test dataset. Therefore, I used grayscale and discarded the original image for the data augmentation part to ensure that our model generalizes to the test dataset and don't classify dogs based on irrelevant features such as the background of the image.

Because the spec mentions that we will be evaluated on the AUROC of my classifier's predictions, I used AUROC to determine which model is best.

The model was trained on MacBook Pro, M1, 2020.

Running train_challenge.py, the epoch with the lowest validation loss yielded `Test AUROC:0.8208`. Running test_cnn_challenge.py yielded `Test AUROC:0.7604`.

```
"""
EECS 445 - Introduction to Machine Learning
Winter 2022 - 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
        padding_size = 2
        self.conv1 = nn.Conv2d(3, 16, (5,5), stride = (2,2), padding = padding_size)
        self.pool = nn.MaxPool2d((2, 2), stride = (2,2))
        self.conv2 = nn.Conv2d(16, 64, (5,5), stride = (2,2), padding = padding_size)
        self.conv3 = nn.Conv2d(64, 8, (5,5), stride = (2,2), padding = padding_size)
        self.fc_1 = nn.Linear(32, 2)

        # 2.f.iii
        # self.conv3 = nn.Conv2d(64, 64, (5,5), stride = (2,2), padding =
padding_size)
        # self.fc_1 = nn.Linear(256, 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]
        nn.init.normal_(self.fc_1.weight, 0.0, 1/32)
        nn.init.constant_(self.fc_1.bias, 0.0)
        ##

    def forward(self, x):
```

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

        ## TODO: forward pass
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = F.relu(self.conv3(x))
        x = self.fc_1(torch.flatten(x, 1))
        ##

        return x

"""
EECS 445 - Introduction to Machine Learning
Winter 2022 - 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
        padding_size = 2
        self.conv1 = nn.Conv2d(3, 16, (5,5), stride = (2,2), padding = padding_size)
        self.pool = nn.MaxPool2d((2, 2), stride = (2,2))
        self.conv2 = nn.Conv2d(16, 64, (5,5), stride = (2,2), padding = padding_size)
        self.conv3 = nn.Conv2d(64, 8, (5,5), stride = (2,2), padding = padding_size)
        self.fc1 = 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]
        nn.init.normal_(self.fc1.weight, 0.0, 1/32)
        nn.init.constant_(self.fc1.bias, 0.0)
        ##

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

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

        return x

"""
EECS 445 - Introduction to Machine Learning
Winter 2022 - 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
from utils import config

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

        ## TODO: define each layer of your network
        padding_size = 2
        self.conv1 = nn.Conv2d(3, 16, (5,5), stride = (2,2), padding = padding_size)
        self.pool = nn.MaxPool2d((2, 2), stride = (2,2))
        self.conv2 = nn.Conv2d(16, 64, (5,5), stride = (2,2), padding = padding_size)
        self.conv3 = nn.Conv2d(64, 8, (5,5), stride = (2,2), padding = padding_size)
        self.fc_1 = nn.Linear(32, 2)
```

```
        ##

        self.init_weights()

    def init_weights(self):
        ## TODO: initialize the parameters for your network
        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)

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

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

        ## TODO: forward pass
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = F.relu(self.conv3(x))
        x = self.fc_1(torch.flatten(x, 1))
        ##

        return x

"""
EECS 445 - Introduction to Machine Learning
Winter 2022 - Project 2
Source CNN Challenge
    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
        padding_size = 2
        self.conv1 = nn.Conv2d(3, 16, (5,5), stride = (2,2), padding = padding_size)
        self.pool = nn.MaxPool2d((2, 2), stride = (2,2))
        self.conv2 = nn.Conv2d(16, 64, (5,5), stride = (2,2), padding = padding_size)
        self.conv3 = nn.Conv2d(64, 8, (5,5), stride = (2,2), padding = padding_size)
        self.fc1 = 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]
        nn.init.normal_(self.fc1.weight, 0.0, 1/32)
        nn.init.constant_(self.fc1.bias, 0.0)
        ##

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

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

        return x

"""
EECS 445 - Introduction to Machine Learning
Winter 2022  - 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


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
        degree = np.random.randint(-deg, deg)
        return rotate(img, degree, 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.

        Round to the nearest integer.

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

        """
        # TODO
        # gray = np.zeros((img.shape[0], img.shape[1]))
        # for i in range(img.shape[0]):
```

```
        #     for j in range(img.shape[1]):
        #         R = img[i][j][0]
        #         G = img[i][j][1]
        #         B = img[i][j][2]
        #         mean = np.mean(R,G,B)
        #         gray[i][j] = mean
        gray = img.mean(axis = 2)


        # G = np.stack((R, G, B), axis = 2).astype(np.uint8)
        return np.stack((gray, gray, gray), axis = 2).astype(np.uint8)

    return _grayscale


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

    :filename: name of image to be augmented
    :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)
    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(), Rotate()]
    # augmentations = [Rotate()]
```

```
    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:]))
```

```
"""
EECS 445 - Introduction to Machine Learning
Winter 2022  - Project 2
Test CNN Challenge
    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.challenge import Challenge
from train_common import *
from utils import config
import utils

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("target.batch_size"),
    )

    # Model
    model = Challenge()

    # 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("target.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()

"""
EECS 445 - Introduction to Machine Learning
Winter 2022  - 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

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("target.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("target.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()

"""
EECS 445 - Introduction to Machine Learning
Winter 2022 - 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 import get_train_val_test_loaders
from model.challenge import Challenge
from train_common import *
from train_target import train
from utils import config
import utils
import copy
```

```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
    layer = num_layers * 2
    for name, param in model.named_parameters():
        if layer != 0:
            param.requires_grad_=False
            layer -= 1
        else:
            break

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


    freeze_three = copy.deepcopy(freeze_none)
    freeze_layers(freeze_three, 3)
    train(tr_loader, va_loader, te_loader, freeze_three, "./checkpoints/target3/", 3)



    # # 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()

    # # Transfer learning freeze layer
    # freeze_none = Challenge()
    # freeze_none, _, _ = restore_checkpoint(
    #     freeze_none, config("source.checkpoint"), force=True, pretrain=True
    # )
    # freeze_one = copy.deepcopy(freeze_none)
    # freeze_layers(freeze_one, 1)
    # train(tr_loader, va_loader, te_loader, freeze_none, "./checkpoints/target0/", 0)

    # # 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
    # patience = 10
    # curr_count_to_patience = 0
    # #

    # # Loop over the entire dataset multiple times
    # epoch = start_epoch
    # while curr_count_to_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)

    #     # Updates early stopping parameters
    #     curr_count_to_patience, global_min_loss = early_stopping(
    #         stats, curr_count_to_patience, global_min_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()

"""
EECS 445 - Introduction to Machine Learning
Winter 2022  - 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("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
    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("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
    patience = 5
    curr_count_to_patience = 0
    #

    # Loop over the entire dataset multiple times
    epoch = start_epoch
    while curr_count_to_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("target.checkpoint"), stats)
```

```
        # update early stopping parameters
        curr_count_to_patience, global_min_loss = early_stopping(
            stats, curr_count_to_patience, global_min_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()

"""
EECS 445 - Introduction to Machine Learning
Winter 2022  - 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_count_to_patience, global_min_loss):
    """Calculate new patience and validation loss.

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

    Returns: new values of curr_count_to_patience and global_min_loss
    """
    # TODO implement early stopping

    #

    if stats[-1][1] >= global_min_loss:
        curr_count_to_patience += 1
    else:
        global_min_loss = stats[-1][1]
        curr_count_to_patience = 0

    return curr_count_to_patience, global_min_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
        predictions = model(X)
        loss = criterion(predictions, y)
        optimizer.zero_grad()
        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
    max = np.argmax(np.array(logits), axis = 1)
    max = torch.tensor(max)
    return max

"""
EECS 445 - Introduction to Machine Learning
Winter 2022 - Project 2
Train Source CNN Challenge
    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

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
    global_min_loss = stats[0][1]

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

    # Loop over the entire dataset multiple times
    epoch = start_epoch
    while curr_count_to_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_count_to_patience, global_min_loss = early_stopping(
            stats, curr_count_to_patience, global_min_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()

"""
EECS 445 - Introduction to Machine Learning
Winter 2022 - 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

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
    global_min_loss = stats[0][1]

    # TODO: patience for early stopping
    patience = 10
    curr_count_to_patience = 0
    #
```

```
    # Loop over the entire dataset multiple times
    epoch = start_epoch
    while curr_count_to_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_count_to_patience, global_min_loss = early_stopping(
            stats, curr_count_to_patience, global_min_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()

"""
EECS 445 - Introduction to Machine Learning
Winter 2022 - 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
"""

from gc import freeze
import torch
import numpy as np
```

```python
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
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 CONV2
    #      Hint: https://pytorch.org/docs/master/notes/autograd.html
    layer = num_layers * 2
    for name, param in model.named_parameters():
        if layer != 0:
            param.requires_grad_=False
            layer -= 1
        else:
            break




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
    global_min_loss = stats[0][1]

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

    # Loop over the entire dataset multiple times
    epoch = start_epoch
    while curr_count_to_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_count_to_patience, global_min_loss = early_stopping(
            stats, curr_count_to_patience, global_min_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)
    print("zero")
    train(tr_loader, va_loader, te_loader, freeze_none, "./checkpoints/target0/", 0)
    print("one")
    train(tr_loader, va_loader, te_loader, freeze_one, "./checkpoints/target1/", 1)
    print("two")
    train(tr_loader, va_loader, te_loader, freeze_two, "./checkpoints/target2/", 2)
    print("three")
    train(tr_loader, va_loader, te_loader, freeze_three, "./checkpoints/target3/", 3)


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