```
# argument_data.py
"""
EECS 445 - Introduction to Machine Learning
Fall 2023 - 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.v3 import imread, imwrite

import rng_control


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)
        return rotate(
            input=img, angle=np.random.randint(-deg, deg), reshape=False)

    return _rotate



def Grayscale():
    """Return function to grayscale image."""

    def _grayscale(img):
        """Return 3-channel grayscale of image.
```

```
            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_img = np.mean(img, axis=2)
        grayscale_img = np.round(grayscale_img).astype(np.uint8)
        grayscale_img = np.stack([grayscale_img] * 3, axis=-1)
        return grayscale_img

    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
    :original: whether to include the original images in the augmented dataset or r
    :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_landmarks.csv", "w"),
        fieldnames=["filename", "semantic label", "partition", "numeric label", "ta
```

```
    )
    augment_partitions = set(args.partitions)

    # TODO: change `augmentations` to specify which augmentations to apply
    # augmentations = [Grayscale(), Rotate()]
    augmentations = [Grayscale()]
    # augmentations = [Rotate()]

    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:]))



# challenge_augment_data.py
"""
EECS 445 - Introduction to Machine Learning
Fall 2023 - 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
import torchvision.transforms as tortransformers
from imageio.v3 import imread, imwrite

import cv2

import rng_control


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)
        return rotate(
            input=img, angle=np.random.randint(-deg, deg), 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: implement _grayscale(img)
        grayscale_img = np.mean(img, axis=2)
        grayscale_img = np.round(grayscale_img).astype(np.uint8)
        grayscale_img = np.stack([grayscale_img] * 3, axis=-1)
        return grayscale_img

    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
    :original: whether to include the original images in the augmented dataset or r
    :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 []
```

```
    res = [img] if original else []
    for i in range(n):
        new = img
        for transform in transforms:
            new = transform(new)
        res.append(new)
    return res

transform_shape = tortransformers.Compose([
    tortransformers.RandomRotation(degrees=(-7, 7)),
    # tortransformers.RandomResizedCrop(
    #     (64, 64), scale=(0.7, 1), ratio=(0.5, 2)),
    tortransformers.RandomHorizontalFlip(),
    # tortransformers.ToTensor(),
])

transformer_color = tortransformers.ColorJitter(
  brightness=0.5, contrast=0.5, saturation=0.5, hue=0.5
)

def main(args):
    """Create augmented dataset."""

    use_feature_selection = input("do you want to use feature selection? y/n\n")

    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(), Rotate()]
    # augmentations = [Grayscale(), tortransformers.ToPILImage(), transformer_color
    augmentations = [tortransformers.ToPILImage(),
                     transform_shape, transformer_color]
    # augmentations = [Rotate()]

    writer.writeheader()
    os.makedirs(f"{args.datadir}/challenge_augmented/", exist_ok=True)
    for f in glob.glob(f"{args.datadir}/challenge_augmented/*"):
        print(f"Deleting {f}")
        os.remove(f)
    for row in reader:
```

```
    for row in reader:
        if row["partition"] not in augment_partitions:
            imwrite(
                f"{args.datadir}/challenge_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=True,  # 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}/challenge_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:]))
```

```
# challenge_target.py
"""
EECS 445 - Introduction to Machine Learning
Fall 2023 - 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
import torchvision.transforms as tortransformers
from imageio.v3 import imread, imwrite

import cv2

import rng_control


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)
        return rotate(
            input=img, angle=np.random.randint(-deg, deg), 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: implement _grayscale(img)
        grayscale_img = np.mean(img, axis=2)
        grayscale_img = np.round(grayscale_img).astype(np.uint8)
        grayscale_img = np.stack([grayscale_img] * 3, axis=-1)
        return grayscale_img

    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
    :original: whether to include the original images in the augmented dataset or r
    :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

transform_shape = tortransformers.Compose([
    tortransformers.RandomRotation(degrees=(-7, 7)),
    # tortransformers.RandomResizedCrop(
    #     (64, 64), scale=(0.7, 1), ratio=(0.5, 2)),
    tortransformers.RandomHorizontalFlip(),
```

```
    # tortransformers.ToTensor(),
])

transformer_color = tortransformers.ColorJitter(
  brightness=0.5, contrast=0.5, saturation=0.5, hue=0.5
)

def main(args):
    """Create augmented dataset."""

    use_feature_selection = input("do you want to use feature selection? y/n\n")

    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(), Rotate()]
    # augmentations = [Grayscale(), tortransformers.ToPILImage(), transformer_color
    augmentations = [tortransformers.ToPILImage(), transform_shape, transformer_col
    # augmentations = [Rotate()]

    writer.writeheader()
    os.makedirs(f"{args.datadir}/challenge_augmented/", exist_ok=True)
    for f in glob.glob(f"{args.datadir}/challenge_augmented/*"):
        print(f"Deleting {f}")
        os.remove(f)
    for row in reader:
        if row["partition"] not in augment_partitions:
            imwrite(
                f"{args.datadir}/challenge_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=True,  # 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}/challenge_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:]))



# confusion_matrix.py
"""
EECS 445 - Introduction to Machine Learning
Fall 2023 - Project 2

Generate confusion matrix graphs.
"""

import torch
import numpy as np
from dataset import get_train_val_test_loaders
from model.source import Source
from train_common import *
```

```
from utils import config
import utils
import os
import itertools
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt


def gen_labels(loader, model):
    """Return true and predicted values."""
    y_true, y_pred = [], []
    for X, y in loader:
        with torch.no_grad():
            output = model(X)
            predicted = predictions(output.data)
            y_true = np.append(y_true, y.numpy())
            y_pred = np.append(y_pred, predicted.numpy())
    return y_true, y_pred


def plot_conf(loader, model, sem_labels, png_name):
    """Draw confusion matrix."""
    y_true, y_pred = gen_labels(loader, model)
    cm = confusion_matrix(y_true, y_pred)
    fig, ax = plt.subplots()
    cax = ax.matshow(cm, cmap=plt.cm.Blues, interpolation="nearest")
    cbar = fig.colorbar(cax, fraction=0.046, pad=0.04)
    cbar.set_label("Frequency", rotation=270, labelpad=10)
    for (i, j), z in np.ndenumerate(cm):
        ax.text(j, i, z, ha="center", va="center")
    plt.gcf().text(0.02, 0.4, sem_labels, fontsize=9)
    plt.subplots_adjust(left=0.5)
    ax.set_xlabel("Predictions")
    ax.xaxis.set_label_position("top")
    ax.set_ylabel("True Labels")
    plt.savefig(png_name)


def main():
    """Create confusion matrix and save to file."""
    tr_loader, va_loader, te_loader, semantic_labels = get_train_val_test_loaders(
        task="source", batch_size=config("source.batch_size")
    )

    model = Source()
```

```
    print("Loading source...")
    model, epoch, stats = restore_checkpoint(model, config("source.checkpoint"))

    sem_labels =
        "0 - Colosseum\n1 - Petronas Towers\n2 - Rialto Bridge\n3 - Museu Nacional

    # Evaluate model
    plot_conf(va_loader, model, sem_labels, "conf_matrix.png")


if __name__ == "__main__":
    main()




# dataset_challenge.py
"""
EECS 445 - Introduction to Machine Learning
Fall 2023 - Project 2

Landmarks Dataset
    Class wrapper for interfacing with the dataset of landmark images
    Usage: python dataset.py
"""

import os
import random
import numpy as np
import pandas as pd
import torch
from matplotlib import pyplot as plt
from imageio.v3 import imread
from PIL import Image
from torch.utils.data import Dataset, DataLoader
from utils import config

import rng_control


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

    Any keyword arguments are forwarded to the LandmarksDataset constructor.
    """
```

```
    tr, va, te, _ = get_train_val_test_datasets(task, **kwargs)

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

    return tr_loader, va_loader, te_loader, tr.get_semantic_label


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

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

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

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

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

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


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

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

    # Resize
    # You may want to experiment with resizing images to be smaller
    # for the challenge portion. How might this affect your training?
    # tr.X = resize(tr.X)
    # va.X = resize(va.X)
    # te.X = resize(te.X)
```

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

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

    return tr, va, te, standardizer


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

    Use bicubic interpolation for resizing.

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

    return resized


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

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

    X has shape (N, image_height, image_width, color_channel), where N is
    the number of images in the set.
```

```
        """

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

        def fit(self, X):
            """Calculate per-channel mean and standard deviation from dataset X.
            Hint: you may find the axis parameter helpful"""
            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."""
            X1 = X - self.image_mean
            X2 = X1 / self.image_std
            return X2




    class LandmarksDataset(Dataset):
        """Dataset class for landmark images."""

        def __init__(self, partition, task="target", augment=False):
            """Read in the necessary data from disk.

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

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

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

            np.random.seed(42)
            torch.manual_seed(42)
            random.seed(42)
            self.partition = partition
            self.task = task
```

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

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

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

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

    def _load_data(self):
        """Load a single data partition from file."""
        print("loading %s..." % self.partition)
        df = self.metadata[
            (self.metadata.task == self.task)
            & (self.metadata.partition == self.partition)
        ]
        if self.augment:
            path = config("augmented_image_path")
        else:
            path = config("image_path")

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

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

        (e.g., the numeric label 1 maps to the semantic label 'hofburg_imperial_pal
        """
        return self.semantic_labels[numeric_label]


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



# dataset.py
"""
EECS 445 - Introduction to Machine Learning
Fall 2023 - Project 2

Landmarks Dataset
    Class wrapper for interfacing with the dataset of landmark images
    Usage: python dataset.py
"""

import os
import random
import numpy as np
import pandas as pd
import torch
from matplotlib import pyplot as plt
from imageio.v3 import imread
from PIL import Image
from torch.utils.data import Dataset, DataLoader
from utils import config

import rng_control


def get_train_val_test_loaders(task, batch_size, **kwargs):
```

```
    """Return DataLoaders for train, val and test splits.

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

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

    return tr_loader, va_loader, te_loader, tr.get_semantic_label


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

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

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

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

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

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


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

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

    # Resize
    # You may want to experiment with resizing images to be smaller
```

```
    # for the challenge portion. How might this affect your training?
    # tr.X = resize(tr.X)
    # va.X = resize(va.X)
    # te.X = resize(te.X)

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

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

    return tr, va, te, standardizer


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

    Use bicubic interpolation for resizing.

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

    return resized


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

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

```
    applied using `transform(X)`.

    X has shape (N, image_height, image_width, color_channel), where N is
    the number of images in the set.
    """

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

    def fit(self, X):
        """Calculate per-channel mean and standard deviation from dataset X.
        Hint: you may find the axis parameter helpful"""
        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."""
        X1 = X - self.image_mean
        X2 = X1 / self.image_std
        return X2




class LandmarksDataset(Dataset):
    """Dataset class for landmark images."""

    def __init__(self, partition, task="target", augment=False):
        """Read in the necessary data from disk.

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

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

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

        np.random.seed(42)
```

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

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

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

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

    def _load_data(self):
        """Load a single data partition from file."""
        print("loading %s..." % self.partition)
        df = self.metadata[
            (self.metadata.task == self.task)
            & (self.metadata.partition == self.partition)
        ]
        if self.augment:
            path = config("augmented_image_path")
        else:
            path = config("image_path")

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

```
            X.append(image)
            y.append(row["numeric_label"])
        return np.array(X), np.array(y)

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

        (e.g., the numeric label 1 maps to the semantic label 'hofburg_imperial_pal
        """
        return self.semantic_labels[numeric_label]


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

```
# feature_selection.py
import cv2
import glob

dataset_dir = "data/challenge_augmented"

image_file_paths = []

for file_path in glob.glob(dataset_dir + "/*.png"):
    image_file_paths.append(file_path)

for image_file_path in image_file_paths:

    image = cv2.imread(image_file_path)
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    ret, thresh = cv2.threshold(gray_image, 150, 255, cv2.THRESH_BINARY)
    contours, _ = cv2.findContours(thresh,
                                   cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
    cv2.drawContours(image, contours, -1, (128, 128, 128), 1)
    cv2.imwrite(image_file_path, image)



# predict_challenge.py
"""
EECS 445 - Introduction to Machine Learning
Fall 2023 - Project 2

Predict Challenge
    Runs the challenge model inference on the test dataset and saves the
    predictions to disk
    Usage: python predict_challenge.py --uniqname=<uniqname>
"""

import argparse
import torch
import numpy as np
import pandas as pd
import utils
from dataset import get_challenge
from model.challenge import Challenge
from train_common import *
from utils import config
```

```
import utils
from sklearn import metrics
from torch.nn.functional import softmax


def predict_challenge(data_loader, model):
    """
    Runs the model inference on the test set and outputs the predictions
    """
    y_score = []
    for X, y in data_loader:
        output = model(X)
        y_score.append(softmax(output.data, dim=1)[:, 1])
    return torch.cat(y_score)

def main(uniqname):
    """Train challenge model."""
    # data loaders
    if check_for_augmented_data("./data"):
        ch_loader, get_semantic_label = get_challenge(
            task="target",
            batch_size=config("challenge.batch_size"), augment = True
        )
    else:
        ch_loader, get_semantic_label = get_challenge(
            task="target",
            batch_size=config("challenge.batch_size"),
        )

    model = Challenge()

    # Attempts to restore the latest checkpoint if exists
    model, _, _ = restore_checkpoint(model, config("challenge.checkpoint"))

    # Evaluate model
    model_pred = predict_challenge(ch_loader, model)
    print("saving challenge predictions...\n")
    pd_writer = pd.DataFrame(model_pred, columns=["predictions"])
    pd_writer.to_csv(uniqname + ".csv", index=False, header=False)


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--uniqname", required=True)
    args = parser.parse_args()
```

```
    main(args.uniqname)
```

```
# rng_control.py
import torch
import numpy as np
import random

SEED = 42

np.random.seed(SEED)

torch.manual_seed(SEED)
random.seed(SEED)
torch.use_deterministic_algorithms(True)
torch.cuda.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
```

```
# test_cnn.py
"""
EECS 445 - Introduction to Machine Learning
Fall 2023 - 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

import rng_control
```

```
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()



# train_challenge.py
"""
EECS 445 - Introduction to Machine Learning
Fall 2023 - 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 model.challenge import *
from train_common import *
from challenge_target import *
# import challenge_source as chas
from model.challenge_source import *
from utils import config
import utils
import copy

def freeze_layers(model, size=0):
    """
    Args:
        model (challenge): a model built in challenge.py
        size (int, optional): the size of layers to be frozen. Defaults to 0.
    Due to the time limit, I only implemented
    1. half of the conv layers are freezed: size=1
    2. the whole conv layers are freezed: size=2
    """
    if size <= 0:
        return
    num = size
    for param in model.parameters():
        if num == 0:
            break
        param.requires_grad = False
        num -= 0.5

def train(tr_loader, va_loader, te_loader, model, model_name, num_layers=0):
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

    # Attempts to restore the latest checkpoint if exists
    print("Loading challenge target model with type", num_layers, "frozen")
    model, start_epoch, stats = restore_checkpoint(
        model, model_name
```

```
        model, model_name
    )

    axes = utils.make_training_plot("Challenge Target Training")

    # Evaluate the randomly initialized model
    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 va
    patience = 12
    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)

        # TODO: Implement early stopping
        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()


def main():
```

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

    do_transfer_learning = input("use transfer learning?y/n\n")
    if do_transfer_learning == 'n':
        use_which_model = input("use which model? resnet/original\n")
        if use_which_model == 'original':
            model = Challenge2()
        if use_which_model == 'resnet':
            model = getResNet18()

        # TODO: Define loss function and optimizer. Replace "None" with the appropr
        criterion = torch.nn.CrossEntropyLoss()
        optimizer = torch.optim.Adam(model.parameters(),
                                     lr=1e-3, weight_decay=0.009)

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

        axes = utils.make_training_plot()

        # Evaluate the randomly initialized model
        evaluate_epoch(
            axes, tr_loader, va_loader,
            te_loader, model, criterion, start_epoch, stats, 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 patienc
        patience = 5
        curr_count_to_patience = 0
```

```
        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,
                            config("challenge.checkpoint"), stats)

            # TODO: Implement early stopping
            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()
    else:
        # freeze_none = getResNet18_target()
        freeze_none = getResNet18_source()
        print("Loading source ...")
        freeze_none, _, _ = restore_checkpoint(
            freeze_none, config("challenge_source.checkpoint"),
            force=True, pretrain=True
        )

        freeze_whole = copy.deepcopy(freeze_none)
        freeze_layers(freeze_whole, 10)

        # modify the last layer:
        num_class = 2
        freeze_none.fc = torch.nn.Linear(freeze_none.fc.in_features, num_class)
        freeze_whole.fc = torch.nn.Linear(freeze_whole.fc.in_features, num_class)
        # freeze_layers.fc = torch.nn.Linear(freeze_layers.fc.in_features, num_clas
```

```
        # train(tr_loader, va_loader,
        # te_loader, freeze_none, "./checkpoints/challenge_target0/", 0)
        train(tr_loader, va_loader,
              te_loader, freeze_whole, "./checkpoints/challenge_target1/", 1)

    # Model
    # model = Challenge()
    # resnet_8class, _, _ = restore_checkpoint(
    #     freeze_none,
    #     config("challenge_source.checkpoint"), force=True, pretrain=True
    # )

    # resnet_8class = nn.Sequential(*list(resnet_8class.children())[:-1])

    # add own classifier
    # num_class = 2
    # classifier = nn.Sequential(
    #     nn.Flatten(),
    #     nn.Linear(512, num_classes)  # 512 is the number of features in the ResNe
    # )

    # Combine pre-trained ResNet and new classifier
    # model = nn.Sequential(resnet_8class, classifier)




if __name__ == "__main__":
    main()



# train_challenge_source.py
"""
EECS 445 - Introduction to Machine Learning
Fall 2023 - Project 2
Train Source CNN
    Train a convolutional neural network to classify images.
    Periodically output training information, and saves model checkpoints
    Usage: python 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 *
import model.challenge_source as mcs
from utils import config
import utils

import rng_control


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 = mcs.getResNet18_source()

    # TODO: Define loss function and optimizer. Replace "None" with the appropriate
    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...")
    clear_checkpoint(config("challenge_source.checkpoint"))
    model, start_epoch, stats = restore_checkpoint(
        model, config("challenge_source.checkpoint"))

    axes = utils.make_training_plot("Challenge Source Training")

    # Evaluate the randomly initialized model
    evaluate_epoch(
        axes,
        tr_loader,
```

```
        va_loader,
        te_loader,
        model,
        criterion,
        start_epoch,
        stats,
        multiclass=True,
        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 va
    patience = 12
    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,
            include_test=True
        )

        # Save model parameters
        save_checkpoint(model, epoch + 1,
                        config("challenge_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()



# train_cnn.py
"""
EECS 445 - Introduction to Machine Learning
Fall 2023 - Project 2

Train CNN
    Train a convolutional neural network to classify images
    Periodically output training information, and save 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

import rng_control

def main():
    """Train CNN and show training plots."""
    # Data loaders
    if check_for_augmented_data("./data"): # if "augmented_landmarks.csv" exists ir
        # if go into this if statement, shows the user decides to use augmented dat
        tr_loader, va_loader, te_loader,
            _ = get_train_val_test_loaders(
            task="target", batch_size=config("target.batch_size"), augment=True
        )
    else: # do not want to use augmented data
        tr_loader, va_loader, te_loader, _ = get_train_val_test_loaders(
            task="target",
```

```
        batch_size=config("target.batch_size"),
    )
# Model
model = Target() # target is a class(convolutional neural network)

# TODO: Define loss function and optimizer. Replace "None" with the appropriate
criterion = torch.nn.CrossEntropyLoss() # use cross entropy loss function
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3) # use adam optimizer

print("Number of float-valued parameters:",
        count_parameters(model)) # output number of learnable parameters in the

# Attempts to restore the latest checkpoint if exists
print("Loading cnn...")

# add clear_checkpoint for 1.f.iii >>>
clear_checkpoint(config("target.checkpoint")) # remove all the original checkpo
# <<<

model, start_epoch, stats = restore_checkpoint(
    model, config("target.checkpoint")) # restore model from checkpoint if exis

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 va
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=False,
        )

        # 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()



# train_common.py
"""
EECS 445 - Introduction to Machine Learning
Fall 2023  - 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
```

```
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`.
    We save the model parameters after each epoch. The periodic saving of model par
    checkpointing. Checking is an important technique for training large models in
    if a hardware failure occurs due to a power outage or our code fails for whatev
    don't want to lose all of our progress.
    """
    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_landmarks.csv` exists in the data di
    if "augmented_landmarks.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
```

```
            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:
```

```
        if cuda:
            checkpoint = torch.load(filename)
        else:
            # Load GPU model on CPU
            checkpoint = torch.load(filename,
                                    map_location=lambda storage, loc: storage)

        print("the check point is:\n", checkpoint)

        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_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_cou

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

```
    # 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."""
    """
    pass the entire validation set (in batches) through the network
    and get the model's predictions, and compare these with the true
    labels to get and evaluation metric
    """

    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`
    Definition: within one epoch, we pass batches of training examples through the
        use back propagation to compute gradients, and update model weights using t
    """
    for i, (X, y) in enumerate(data_loader):
        optimizer.zero_grad()
        predict = model(X)
        loss = criterion(predict, y)
        loss.backward()
        optimizer.step()
```

```
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(|

    Returns:
        the predicted class output as a PyTorch Tensor
        the set of outputs is called logits
    """
    pred = []
    for i in range(logits.shape[0]):
        ans = np.argmax(logits[i])
        pred.append(ans)
    return torch.tensor(pred)



# train_source.py
"""
EECS 445 - Introduction to Machine Learning
Fall 2023 - Project 2
Train Source CNN
    Train a convolutional neural network to classify images.
    Periodically output training information, and saves model checkpoints
    Usage: python 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

import rng_control


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
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...")
clear_checkpoint(config("source.checkpoint"))
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 va
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()



# train_target.py
"""
EECS 445 - Introduction to Machine Learning
Fall 2023 - Project 2
Train Target
    Train a convolutional neural network to classify images.
    Periodically output training information, and saves model checkpoints
    Usage: python train_target.py
"""
```

```
import torch
import numpy as np
import random
from dataset import get_train_val_test_loaders
from model.target import Target
from train_common import *
from utils import config
import utils
import copy

import rng_control


def freeze_layers(model, num_layers=0):
    """Stop tracking gradients on selected layers."""
    # if num_layers == 1:
    #     model.conv1.requires_grad = False
    # if num_layers == 2:
    #     model.conv1.requires_grad = False
    #     model.conv2.requires_grad = False
    # if num_layers == 3:
    #     model.conv1.requires_grad = False
    #     model.conv2.requires_grad = False
    #     model.conv3.requires_grad = False
    if num_layers <= 0:
        return
    new_num_layers = num_layers
    for name, param in model.named_parameters():
        if new_num_layers == 0:
            break
        param.requires_grad = False
        new_num_layers -= 0.5

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
    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: Define patience for early stopping. Replace "None" with the patience va
    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() # class:2
    print("Loading source...")
    freeze_none, _, _ = restore_checkpoint( # get trained parameters here
        freeze_none, config("source.checkpoint"), force=True, pretrain=True
    )

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

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

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


if __name__ == "__main__":
    main()



# utils.py
"""
```

```
EECS 445 - Introduction to Machine Learning
Fall 2023  - Project 2
Utility functions
"""

import os
import numpy as np
import matplotlib.pyplot as plt


def config(attr):
    """
    Retrieves the queried attribute value from the config file. Loads the
    config file on first call.
    """
    if not hasattr(config, "config"): # hasattr: return 1 if config has attribute '
        with open("config.json") as f:
            config.config = eval(f.read())
    node = config.config
    for part in attr.split("."):
        node = node[part]
    return node


def denormalize_image(image):
    """Rescale the image's color space from (min, max) to (0, 1)"""
    ptp = np.max(image, axis=(0, 1)) - np.min(image, axis=(0, 1))
    return (image - np.min(image, axis=(0, 1))) / ptp


def hold_training_plot():
    """
    Keep the program alive to display the training plot
    """
    plt.ioff()
    plt.show()


def log_training(epoch, stats):
    """Print the train, validation, test accuracy/loss/auroc.

    Each epoch in `stats` should have order
        [val_acc, val_loss, val_auc, train_acc, ...]
    Test accuracy is optional and will only be logged if stats is length 9.
    """
```

```
    splits = ["Validation", "Train", "Test"]
    metrics = ["Accuracy", "Loss", "AUROC"]
    print("Epoch {}".format(epoch))
    for j, split in enumerate(splits):
        for i, metric in enumerate(metrics):
            idx = len(metrics) * j + i
            if idx >= len(stats[-1]):
                continue
            print(f"\t{split} {metric}:{round(stats[-1][idx],4)}")


def make_training_plot(name="CNN Training"):
    """Set up an interactive matplotlib graph to log metrics during training."""
    plt.ion()
    fig, axes = plt.subplots(1, 3, figsize=(20, 5))
    plt.suptitle(name)
    axes[0].set_xlabel("Epoch")
    axes[0].set_ylabel("Accuracy")
    axes[1].set_xlabel("Epoch")
    axes[1].set_ylabel("Loss")
    axes[2].set_xlabel("Epoch")
    axes[2].set_ylabel("AUROC")

    return axes


def update_training_plot(axes, epoch, stats):
    """Update the training plot with a new data point for loss and accuracy."""
    splits = ["Validation", "Train", "Test"]
    metrics = ["Accuracy", "Loss", "AUROC"]
    colors = ["r", "b", "g"]
    for i, metric in enumerate(metrics):
        for j, split in enumerate(splits):
            idx = len(metrics) * j + i
            if idx >= len(stats[-1]):
                continue
            # __import__('pdb').set_trace()
            axes[i].plot(
                range(epoch - len(stats) + 1, epoch + 1),
                [stat[idx] for stat in stats],
                linestyle="--",
                marker="o",
                color=colors[j],
            )
        axes[i].legend(splits[: int(len(stats[-1]) / len(metrics))])
```

```
    plt.pause(0.00001)


def save_cnn_training_plot():
    """Save the training plot to a file."""
    plt.savefig("cnn_training_plot.png", dpi=200)


def save_tl_training_plot(num_layers):
    """Save the transfer learning training plot to a file."""
    if num_layers == 0:
        plt.savefig("TL_0_layers.png", dpi=200)
    elif num_layers == 1:
        plt.savefig("TL_1_layers.png", dpi=200)
    elif num_layers == 2:
        plt.savefig("TL_2_layers.png", dpi=200)
    elif num_layers == 3:
        plt.savefig("TL_3_layers.png", dpi=200)


def save_source_training_plot():
    """Save the source learning training plot to a file."""
    plt.savefig("source_training_plot.png", dpi=200)


def save_challenge_training_plot():
    """Save the challenge learning training plot to a file."""
    plt.savefig("challenge_training_plot.png", dpi=200)



# visualize_cnn.py
"""
EECS 445 - Introduction to Machine Learning
Fall 2023  - Project 2
Grad-CAM Visualization
    This script generates a heat map on top of the original image for 20
    sample images in the dataset to help visualize what is learned by
    the convolutional networks.

    Output files will be titled CNN_viz1_<number>.png

    Usage: python visualize_cc.py

Original credit to:
```

```
Author:   Kazuto Nakashima
URL:      http://kazuto1011.github.io
Created:  2017-05-26
"""

import numpy as np
import torch
from matplotlib import pyplot as plt
import matplotlib.cm as cm
from dataset import get_train_val_test_loaders
from model.target import Target
from model.source import Source
from train_common import *
from utils import config
import utils
from collections import OrderedDict
from torch.nn import functional as F
from imageio.v3 import imread


def save_gradcam(gcam, original_image, axarr, i):
    cmap = cm.viridis(np.squeeze(gcam.numpy()))[..., :3] * 255.0
    raw_image = (
        (
            (original_image - original_image.min())
            / (original_image.max() - original_image.min())
        )
        * 255
    ).astype("uint8")
    gcam = (cmap.astype(np.float64) + raw_image.astype(np.float64)) / 2
    axarr[1].imshow(np.uint8(gcam))
    axarr[1].axis("off")
    plt.savefig("CNN_viz1_{}.png".format(i), dpi=200, bbox_inches="tight")


class _BaseWrapper(object):
    """
    Please modify forward() and backward() according to your task.
    """

    def __init__(self, model, fmaps=None):
        super(_BaseWrapper, self).__init__()
        self.device = next(model.parameters()).device
        self.model = model
        self.handlers = []  # a set of hook function handlers
```

```
    def _encode_one_hot(self, ids):
        one_hot = torch.zeros_like(self.logits).to(self.device)
        one_hot.scatter_(1, ids, 1.0)
        return one_hot

    def forward(self, image):
        """
        Simple classification
        """
        self.model.zero_grad()
        self.logits = self.model(image)
        if type(self.logits) is tuple:
            self.logits = self.logits[0]
        self.probs = torch.nn.Sigmoid()(self.logits)
        return self.probs.sort(dim=1, descending=True)

    def backward(self, ids):
        """
        Class-specific backpropagation
        Either way works:
        1. self.logits.backward(gradient=one_hot, retain_graph=True)
        2. (self.logits * one_hot).sum().backward(retain_graph=True)
        """
        one_hot = self._encode_one_hot(ids)
        self.logits.backward(gradient=one_hot, retain_graph=True)

    def generate(self):
        raise NotImplementedError

    def remove_hook(self):
        """
        Remove all the forward/backward hook functions
        """
        for handle in self.handlers:
            handle.remove()


class BackPropagation(_BaseWrapper):
    def forward(self, image):
        self.image = image
        self.image.requires_grad = False
        return super(BackPropagation, self).forward(self.image)

    def generate(self):
```

```
        gradient = self.image.grad.clone()
        self.image.grad.zero_()
        return gradient


class GradCAM(_BaseWrapper):
    """
    "Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localizati
    https://arxiv.org/pdf/1610.02391.pdf
    Look at Figure 2 on page 4
    """

    def __init__(self, model, candidate_layers=None):
        super(GradCAM, self).__init__(model)
        self.fmap_pool = OrderedDict()
        self.grad_pool = OrderedDict()
        self.candidate_layers = candidate_layers  # list

        def forward_hook(key):
            def forward_hook_(module, input, output):
                # Save featuremaps
                a = output.detach().cpu()
                self.fmap_pool[key] = a
                del output
                del a
                torch.cuda.empty_cache()

            return forward_hook_

        def backward_hook(key):
            def backward_hook_(module, grad_in, grad_out):
                # Save the gradients correspond to the featuremaps
                a = grad_out[0].detach().cpu()
                self.grad_pool[key] = a
                torch.cuda.empty_cache()

            return backward_hook_

        # If any candidates are not specified, the hook is registered to all the la
        for name, module in self.model.named_modules():
            if self.candidate_layers is None or name in self.candidate_layers:
                self.handlers.append(
                    module.register_forward_hook(forward_hook(name)))
                self.handlers.append(
                    module.register_full_backward_hook(backward_hook(name))
```

```
                )

    def find(self, pool, target_layer):
        if target_layer in pool.keys():
            return pool[target_layer]
        else:
            raise ValueError("Invalid layer name: {}".format(target_layer))

    def _compute_grad_weights(self, grads):
        return F.adaptive_avg_pool2d(grads, 1)

    def forward(self, image):
        self.image = image
        self.image_shape = image.shape[2:]
        return super(GradCAM, self).forward(self.image)

    def generate(self, target_layer):
        fmaps = self.find(self.fmap_pool, target_layer)
        grads = self.find(self.grad_pool, target_layer)
        weights = self._compute_grad_weights(grads)
        gcam = torch.mul(fmaps, weights).sum(dim=1, keepdim=True)
        gcam = F.relu(gcam)
        gcam = F.interpolate(
            gcam, self.image_shape, mode="bilinear", align_corners=False
        )
        B, C, H, W = gcam.shape
        gcam = gcam.view(B, -1)
        gcam -= gcam.min(dim=1, keepdim=True)[0]
        gcam /= gcam.max(dim=1, keepdim=True)[0]
        gcam = gcam.view(B, C, H, W)
        return gcam


device = torch.device("cpu")


def get_image(img_num):
    img_path = "data/images/" + img_num + ".png"
    img = imread(img_path)
    return img


def visualize_input(img_num, axarr):
    xi = get_image(img_num)
    axarr[0].imshow(utils.denormalize_image(xi))
```

```
    axarr[0].axis("off")


def visualize_layer1_activations(img_num, i, axarr):
    xi = get_image(img_num)
    xi = xi.transpose(2, 0, 1)
    xi = torch.from_numpy(xi).float()
    xi = xi.view((1, 3, 64, 64))
    bp = BackPropagation(model=model)
    gcam = GradCAM(model=model)
    target_layer = "conv1"
    target_class = 1
    _ = gcam.forward(xi)
    gcam.backward(ids=torch.tensor([[target_class]]).to(device))
    regions = gcam.generate(target_layer=target_layer)
    activation = regions.detach()
    save_gradcam(
        np.squeeze(activation),
        utils.denormalize_image(np.squeeze(xi.numpy()).transpose(1, 2, 0)),
        axarr,
        i,
    )


if __name__ == "__main__":
    # Attempts to restore from checkpoint
    print("Loading cnn...")
    model = Target()
    model, start_epoch, _ = restore_checkpoint(
        model, config("target.checkpoint"), force=True
    )

    tr_loader, va_loader, te_loader, _ = get_train_val_test_loaders(
        task="target",
        batch_size=config("target.batch_size"),
    )

    # img_list contains the ids for a sample of images from the training set
    img_list = ['01105', '00024', '03545', '05934', '03165',
                '05354', '01295', '00914', '03035', '07984',
                '09125', '05654', '05125', '05174', '04685',
                '12235', '00975', '01074', '02995', '01134']
    for i, img_num in enumerate(img_list):
        plt.clf()
        f, axarr = plt.subplots(1, 2)
```

```
        visualize_input(img_num, axarr)
        visualize_layer1_activations(img_num, i, axarr)
        plt.close()



# visualize_data.py
"""
EECS 445 - Introduction to Machine Learning
Fall 2023  - Project 2
Visualize Landmarks
    This will open up a window displaying randomly selected training
    images. The label of the image is shown. Click on the figure to
    refresh with a set of new images. You can save the images using
    the save button. Close the window to break out of the loop.

    The success of this script is a good indication that the data flow
    part of this project is running smoothly.

    Usage: python visualize_data.py
"""
import matplotlib.pyplot as plt
import numpy as np
import os
import platform
import pandas as pd

from dataset import resize, ImageStandardizer, LandmarksDataset
from imageio.v3 import imread
from utils import config, denormalize_image

OUT_FILENAME = "visualize_data.png"

training_set = LandmarksDataset("train")
training_set.X = resize(training_set.X)
standardizer = ImageStandardizer()
standardizer.fit(training_set.X)

metadata = pd.read_csv(config("csv_file"))
metadata = metadata[metadata["partition"] != "challenge"].reset_index(drop=True)

N = 4
fig, axes = plt.subplots(nrows=2, ncols=N, figsize=(2 * N, 2 * 2))

pad = 3
axes[0, 0].annotate(
```

```
axes[0, 0].annotate(
    "Original",
    xy=(0, 0.5),
    xytext=(-axes[0, 0].yaxis.labelpad - pad, 0),
    xycoords=axes[0, 0].yaxis.label,
    textcoords="offset points",
    size="large",
    ha="right",
    va="center",
    rotation="vertical",
)
axes[1, 0].annotate(
    "Preprocessed",
    xy=(0, 0.5),
    xytext=(-axes[1, 0].yaxis.labelpad - pad, 0),
    xycoords=axes[1, 0].yaxis.label,
    textcoords="offset points",
    size="large",
    ha="right",
    va="center",
    rotation="vertical",
)

for ax in axes.flatten():
    ax.set_xticks([])
    ax.set_yticks([])


def display_imgs():
    rand_idx = np.random.choice(np.arange(len(metadata)), size=N, replace=False)
    X, y = [], []
    for idx in rand_idx:
        filename = os.path.join(config("image_path"),
                                metadata.loc[idx, "filename"])
        X.append(imread(filename))
        y.append(metadata.loc[idx, "semantic_label"])

    for i, (xi, yi) in enumerate(zip(X, y)):
        axes[0, i].imshow(xi)
        axes[0, i].set_title(yi.replace("_", "\n"))

    X_ = resize(np.array(X))
    X_ = standardizer.transform(X_)
    for i, (xi, yi) in enumerate(zip(X_, y)):
        axes[1, i].imshow(denormalize_image(xi),
                          interpolation="bicubic")
```

```
                                interpolation='bicubic')


if platform.system() == "Linux" and "DISPLAY" not in os.environ:
    print(
        f'No window server found, switching \
            to writing first {N} images to file "{OUT_FILENAME}"'
    )
    display_imgs()
    fig.savefig(OUT_FILENAME, bbox_inches="tight")
    exit(0)

print(
    "I will display some images. \
        Click on the figure to refresh. Close the figure to exit."
)

while True:
    display_imgs()
    plt.draw()
    if plt.get_fignums():
        print(plt.get_fignums())
    else:
        print(None)
    fig.savefig(OUT_FILENAME, bbox_inches="tight")
    if plt.waitforbuttonpress(0) == None:
        break

print("OK, bye!")
```