Exercise 4: Metric embeddings

Lectures on metric embeddings will be available on the course website.

Exercises

Task 1: Data loading (10%)

Implement data loading to enable training models for metric embedding with triplet loss. To accomplish this, it is necessary to adapt the MNIST dataset so that when retrieving training examples (anchors), corresponding positive and negative examples are also retrieved.

```
from torch.utils.data import Dataset
from collections import defaultdict
from random import choice
import torchvision


class MNISTMetricDataset(Dataset):
    def __init__(self, root="/tmp/mnist/", split='train'):
        super().__init__()
        assert split in ['train', 'test', 'traineval']
        self.root = root
        self.split = split
        mnist_ds = torchvision.datasets.MNIST(self.root, train='train' in split, downloa
        self.images, self.targets = mnist_ds.data.float() / 255., mnist_ds.targets
        self.classes = list(range(10))

        self.target2indices = defaultdict(list)
        for i in range(len(self.images)):
            self.target2indices[self.targets[i].item()] += [i]

    def _sample_negative(self, index):
        # YOUR CODE HERE


    def _sample_positive(self, index):
        # YOUR CODE HERE


    def __getitem__(self, index):
        anchor = self.images[index].unsqueeze(0)
        target_id = self.targets[index].item()
        if self.split in ['traineval', 'val', 'test']:
            return anchor, target_id
        else:
            positive = self._sample_positive(index)
            negative = self._sample_negative(index)
            positive = self.images[positive]
            negative = self.images[negative]
            return anchor, positive.unsqueeze(0), negative.unsqueeze(0), target_id

    def __len__(self):
        return len(self.images)
```

Implement methods `_sample_positive` and `_sample_negative` so that their return values correspond to the indices of sampled images in the list `self.images`. For the purposes of this exercise, it is sufficient to implement a simple sampling strategy that randomly samples the positive from a subset of images that belong to the same class as the anchor and the negative from the subset of images that do not share the anchor's class.

Task 2: Defining a model for metric embedding (40%)

You are give a rough template of a model for metric embedding.

```
import torch
import torch.nn as nn
import torch.nn.functional as F


class _BNReluConv(nn.Sequential):
    def __init__(self, num_maps_in, num_maps_out, k=3, bias=True):
        super(_BNReluConv, self).__init__()
        # YOUR CODE HERE

class SimpleMetricEmbedding(nn.Module):
    def __init__(self, input_channels, emb_size=32):
        super().__init__()
        self.emb_size = emb_size
        # YOUR CODE HERE

    def get_features(self, img):
        # Returns tensor with dimensions BATCH_SIZE, EMB_SIZE
        # YOUR CODE HERE
        x = ...
        return x

    def loss(self, anchor, positive, negative):
        a_x = self.get_features(anchor)
        p_x = self.get_features(positive)
        n_x = self.get_features(negative)
        # YOUR CODE HERE
        loss = ...
        return loss
```

Fill in the missing code according to the following instructions:

a) loss

Implement a triplet loss similarly to the Pytorch `TripletMarginLoss`.

b) convolutional module BNReLUConv

In practice, we often extract a part of the model that repeats frequently into a shared differentiable module. Design the convolutional module `BNReLUConv` consisting of group normalization, ReLU activation, and convolution. Note that our template inherits from the class Sequential. This means that to add layers in the constructor, you can use the `append` method.

c) metric embedding

Complete the implementation of the metric embedding model. Let your model consist of 3 consecutive convolutional modules `BNReLUConv` (set kernel size to 3, and the number of feature maps to `emb_size`) separated by max-pooling (kernel size of 3, stride of 2). Finally, embed the image by global average pooling. Ensure that the output tensor in the `get_features` method retains the first dimension indicating the minibatch size, even when it is equal to 1.

Task 3: Training and evaluating (40%)

You are given a code for training a model for metric embedding on the MNIST dataset. You may use the `utils.py` script is available here.

```
import time
import torch.optim
from dataset import MNISTMetricDataset
from torch.utils.data import DataLoader
from model import SimpleMetricEmbedding
from utils import train, evaluate, compute_representations

EVAL_ON_TEST = True
EVAL_ON_TRAIN = False


if __name__ == '__main__':
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    print(f"= Using device {device}")

    # CHANGE ACCORDING TO YOUR PREFERENCE
    mnist_download_root = "./mnist/"
    ds_train = MNISTMetricDataset(mnist_download_root, split='train')
    ds_test = MNISTMetricDataset(mnist_download_root, split='test')
    ds_traineval = MNISTMetricDataset(mnist_download_root, split='traineval')

    num_classes = 10

    print(f"> Loaded {len(ds_train)} training images!")
    print(f"> Loaded {len(ds_test)} validation images!")

    train_loader = DataLoader(
        ds_train,
        batch_size=64,
        shuffle=True,
        pin_memory=True,
        num_workers=4,
        drop_last=True
    )

    test_loader = DataLoader(
        ds_test,
        batch_size=1,
        shuffle=False,
        pin_memory=True,
        num_workers=1
    )

    traineval_loader = DataLoader(
        ds_traineval,
        batch_size=1,
        shuffle=False,
        pin_memory=True,
        num_workers=1
    )

    emb_size = 32
    model = SimpleMetricEmbedding(1, emb_size).to(device)
    optimizer = torch.optim.Adam(
        model.parameters(),
        lr=1e-3
    )

    epochs = 3
    for epoch in range(epochs):
        print(f"Epoch: {epoch}")
        t0 = time.time_ns()
        train_loss = train(model, optimizer, train_loader, device)
        print(f"Mean Loss in Epoch {epoch}: {train_loss:.3f}")
        if EVAL_ON_TEST or EVAL_ON_TRAIN:
            print("Computing mean representations for evaluation...")
            representations = compute_representations(model, train_loader, num_classes,
        if EVAL_ON_TRAIN:
            print("Evaluating on training set...")
            acc1 = evaluate(model, representations, traineval_loader, device)
            print(f"Epoch {epoch}: Train Top1 Acc: {round(acc1 * 100, 2)}%")
        if EVAL_ON_TEST:
            print("Evaluating on test set...")
            acc1 = evaluate(model, representations, test_loader, device)
            print(f"Epoch {epoch}: Test Accuracy: {acc1 * 100:.2f}%")
        t1 = time.time_ns()
        print(f"Epoch time (sec): {(t1-t0)/10**9:.1f}")
```

a) Analyze the module `utils.py`

Study the functions for training and evaluation in `utils.py`. How are class representations calculated? How is the classification of examples carried out? Try to come up with alternative approaches for classifying examples.

b) Classification based on metric embeddings

Learn the metric embedding model from task 2.c on a subset of the MNIST training set. Perform the classification of images from the validation subset and measure the accuracy.

c) Classification based on distances in image space

Perform classification on the validation subset, but this time in the image space. Yo may accomplish this by designing a module that performs simple image vectorization in the `get_features` method.

```
class IdentityModel(nn.Module):
    def __init__(self):
        super(IdentityModel, self).__init__()

    def get_features(self, img):
        # YOUR CODE HERE
        feats = ...
        return feats
```

Implement the `IdentityModel` module according to the provided template. Modify the training function so that classification is performed in the image space. Note that `IdentityModel` cannot be trained. Measure the classification accuracy on the validation subset.

d) Storing model parameters

In practice, it is practical to store the parameters of the trained model for later use in the inference phase. Modify the training function so that you store the learned parameters using the 'torch.save' method. Re-train the metric embedding model and save the obtained parameters.

e) Classification of new classes

One of the advantages of metric embeddings over standard classification models is the ability to add new classes in the inference phase. Modify the constructor of `MNISTMetricDataset` to enable the removal of examples from the selected class in the training set:

```
    def __init__(self, root="/tmp/mnist/", split='train', remove_class=None):
        super().__init__()
        assert split in ['train', 'test', 'traineval']
        self.root = root
        self.split = split
        mnist_ds = torchvision.datasets.MNIST(self.root, train='train' in split, downloa
        self.images, self.targets = mnist_ds.data.float() / 255., mnist_ds.targets
        self.classes = list(range(10))

        if remove_class is not None:
            # Filter out images with target class equal to remove_class
            # YOUR CODE HERE

        self.target2indices = defaultdict(list)
        for i in range(len(self.images)):
            self.target2indices[self.targets[i].item()] += [i]
```

Remove class 0 from the training subset and train a new metric embedding model from task 2. Classify all images (including class 0) from the validation subset based on similarity in the feature space. Note that you will need to have two loaders for the training subset. The first loader will ignore images of digit 0 and will be used to train the model. The second loader will read images with all digits, and you will use it to obtain the average representation of digits from all classes. Save the parameters of the trained model and display the achieved classification accuracy.

Task 4: Data visualization (10%)

The quality of metric embedding can also be qualitatively assessed by comparing the arrangement of data in the feature space and the image space. Since it is impossible to visualize high-dimensional data in the original space, examples need to be projected into a 2D space. This can be done with principal component analysis. Note that you may use the pytorch pca_lowrank method.

```
import numpy as np
import torch

from dataset import MNISTMetricDataset
from model import SimpleMetricEmbedding
from matplotlib import pyplot as plt


def get_colormap():
    # Cityscapes colormap for first 10 classes
    colormap = np.zeros((10, 3), dtype=np.uint8)
    colormap[0] = [128, 64, 128]
    colormap[1] = [244, 35, 232]
    colormap[2] = [70, 70, 70]
    colormap[3] = [102, 102, 156]
    colormap[4] = [190, 153, 153]
    colormap[5] = [153, 153, 153]
    colormap[6] = [250, 170, 30]
    colormap[7] = [220, 220, 0]
    colormap[8] = [107, 142, 35]
    colormap[9] = [152, 251, 152]
    return colormap


if __name__ == '__main__':
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    print(f"= Using device {device}")
    emb_size = 32
    model = SimpleMetricEmbedding(1, emb_size).to(device)
    # YOUR CODE HERE
    # LOAD TRAINED PARAMS

    colormap = get_colormap()
    mnist_download_root = "./mnist/"
    ds_test = MNISTMetricDataset(mnist_download_root, split='test')
    X = ds_test.images
    Y = ds_test.targets
    print("Fitting PCA directly from images...")
    test_img_rep2d = torch.pca_lowrank(ds_test.images.view(-1, 28 * 28), 2)[0]
    plt.scatter(test_img_rep2d[:, 0], test_img_rep2d[:, 1], color=colormap[Y[:]] / 255.,
    plt.show()
    plt.figure()

    print("Fitting PCA from feature representation")
    with torch.no_grad():
        model.eval()
        test_rep = model.get_features(X.unsqueeze(1))
        test_rep2d = torch.pca_lowrank(test_rep, 2)[0]
        plt.scatter(test_rep2d[:, 0], test_rep2d[:, 1], color=colormap[Y[:]] / 255., s=
        plt.show()
```

Modify the code to load the parameters trained in the previous task. You can find more about saving and loading parameters in Pytorch documentation. Visualize examples in the image space and the feature space for the model trained with all digits and the model that, during training, did not see images with digit 0.

dlunizg
ivan.kreso@fer.hr