Hyperbolic Representation Learning

Release v1.0

Mina Ghadimi Atigh

Sep 28, 2022

TUTORIALS

Tutorial website: https://sites.google.com/view/hyperbolic-tutorial-eccv22
Tutorial edition: October 24th, 2022
Repository: https://github.com/MinaGhadimiAtigh/hyperbolic_representation_learning
Author: Mina Ghadimi Atigh

This website provides you access to the content that we will use for the tutorial session at the ECCV 2022, "Hyperbolic Representation Learning for Computer Vision".

Learning in computer vision is all about deep networks and such networks operate on Euclidean manifolds by design. But is Euclidean geometry the best choice for deep learning or simply a practical option? Recent literature in machine learning and computer vision has shown that hyperbolic geometry provides a strong alternative, with an improved ability to embed hierarchies, graphs, text, images, and videos.

In light of recent advancements in hyperbolic representation learning for computer vision, this tutorial seeks to advocate hyperbolic geometry and its strong potential for computer vision to a broader audience. The tutorials provides a theoretical and practical starting point for the field. At the conference, we will provide an easy-going introduction to hyperbolic geometry for non-mathematicians, where we focus on intuition and high-level understanding. We then outline the current state of hyperbolic geometry for vision from supervised and unsupervised perspectives. At the end, we dive into open research problems and future potential for hyperbolic geometry and visual understanding.

Unique for this tutorial is that we do not stop at a theoretical foundation. The tutorial website will also host a series of notebook-style code snippets with foundational works on hyperbolic geometry, to get a better understanding of its workings and lower the barrier to start your dive into this exciting new research direction in computer vision.

For any remaining questions regarding the notebooks, please contact us at m.ghadimiatigh@uva.nl.

CHAPTER

ONE

HOW TO RUN THE NOTEBOOKS

On this website, you will find the notebooks exported into a HTML format so that you can read them from whatever device you prefer.

There are three main ways of running the notebooks we recommend:

- **Locally on GPU**: If you have a laptop with a build-in NVIDIA GPU, we recommend that you run the notebooks on your own machine. All notebooks are stored on the github repository that also builds this website. You can find them here: to be updated.All notebooks require access to a GPU to keep the training times in a reasonable range. Nonetheless, if you prefer, you can code and test most of your code on a CPU-only system, i.e. your own laptop, and once your code is tested and ready, use one of the remaining options to train the model. To ensure that you have all the right python packages installed, we provide a conda environment in the same repository.
- **Google Colab**: If you do not have access to a GPU on your local machine, you can make use of Google Colab. Google Colab provides you access to GPUs for free, and you can activate the GPU support by `Runtime -> Change runtime type -> Hardware accelerator: GPU`. Each notebook on this documentation website has a badge with a link to directly open it on Google Colab. It is highly recommend to copy the notebook to your own Google Drive before starting, since when closing the session, changes might be lost if you don't save it to your local computer or have copied the notebook to your Google Drive beforehand. In addition, note that for free account, Google Colab is limited to one session at a time, and each session has a time limit.
- **Compute cluster**: If you have access to a compute cluster, we recommend to using it to do your final trainings. Depending on your preference, you can run the tutorials either locally or on Google Colab. Once your notebook is ready, you can first convert the notebooks to a script using `jupyter nbconvert --to script ...ipynb`, and then start a job on the cluster for running the script. A few advices when running on clusters:
  - Disable the tqdm statements in the notebook. Otherwise your slurm output file might overflow and be several MB large.
  - Comment out the matplotlib plotting statements, or change `plt.show()` to `plt.savefig(...)`.

1.1 Image Classification with Euclidean and Hyperbolic Neural Network

Welcome to our first notebook for the ECCV 2022 Tutorial "Hyperbolic Representation Learning for Computer Vision"!

Open notebook:

Author: Mina Ghadimi Atigh

In this notebook, you will perform a simple image classification task using two Neural Networks, one in Euclidean space and one in the hyperbolic space. The main goal is to go through the training and testing process in Euclidean and hyperbolic spaces and see the similarities and the differences.

If you prefer working with standard python scripts, feel free to convert this notebook into a python script. To open this notebook on Google Colab, use the button above. Note that you need to copy this notebook into your own Google Drive to save the notebook and trained models. Otherwise, your progress will be lost when you close the browser tab.

Let's start with importing the libraries and setting manual seed using `set_seed`.

[1]:
```
## standard libraries
import numpy as np
import warnings
from IPython.display import clear_output

## Imports for plotting
import matplotlib.pyplot as plt

## PyTorch
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

## PyTorch Torchvision
import torchvision

warnings.filterwarnings('ignore')
```

[2]:
```
# Function for setting the seed
def set_seed(seed):
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed(seed)
        torch.cuda.manual_seed_all(seed)
set_seed(42)

# Ensure that all operations are deterministic on GPU (if used) for reproducibility
torch.backends.cudnn.determinstic = True
torch.backends.cudnn.benchmark = False

# Fetching the device that will be used throughout this notebook
device = torch.device("cpu") if not torch.cuda.is_available() else torch.device("cuda:0")
print("Using device", device)
```

```
Using device cuda:0
```

For the Hyperbolic layers and functions, we're going to use geoopt library in this notebook.

[3]:
```
!pip install -q git+https://github.com/geoopt/geoopt.git
! [ ! -f mobius_linear_example.py ] && wget -q https://raw.githubusercontent.com/geoopt/
↪geoopt/master/examples/mobius_linear_example.py
```

[3]:
```
import geoopt
from mobius_linear_example import MobiusLinear
```

Here, we define the paths which will be used in this notebook.

[4]:
```
DATA_PATH = './data'
```

Let's start with setting up the dataset. In this notebook, you will work with `MNIST` dataset. MNIST consists of 70000 tiny (28*28) gray scale images of handwritten digits, from zero to nine. The goal is to recognize the digit corresponding to each image.

[5]:
```
transform=torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize((0.1307,), (0.3081,))
    ])

# Train dataset - downloading the training dataset. Training dataset is splitted into␣
↪train and val parts.
main_trainset = torchvision.datasets.MNIST(root=DATA_PATH, train = True, download=True,␣
↪transform=transform)
trainset, valset = torch.utils.data.random_split(main_trainset, [10000, 50000],␣
↪generator=torch.Generator().manual_seed(42))

# Test dataset - downloading and loading the testing dataset.
testset = torchvision.datasets.MNIST(root=DATA_PATH, train=False, download=True,␣
↪transform=transform)
testset_small, _ = torch.utils.data.random_split(main_trainset, [30000, 30000],␣
↪generator=torch.Generator().manual_seed(42))

# Create dataloaders for the train, val and test sets
batch_size = 8
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True,␣
↪num_workers=2, pin_memory=True)
valloader = torch.utils.data.DataLoader(valset, batch_size=batch_size, shuffle=True, num_
↪workers=2)
testloader = torch.utils.data.DataLoader(testset_small, batch_size=batch_size,␣
↪shuffle=False, num_workers=2)
```

Before starting the main task, let's visualize some of the images from the dataset.

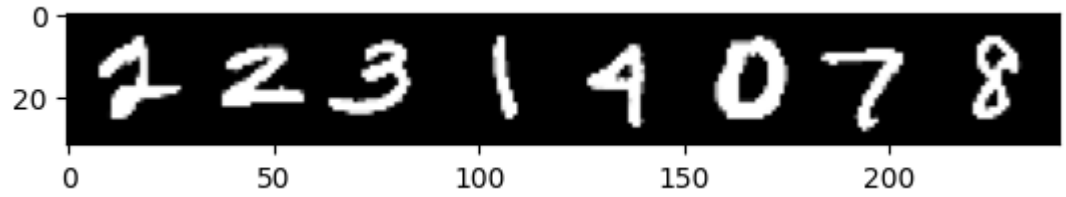
[6]:
```
# Extract class names
classes = torchvision.datasets.MNIST.classes
# Class names are like 0 - zero
classes_num = [c.split('-')[0] for c in classes]

# functions to show an image
def imshow(img):
    # What should I do here to unnormalize?
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()


# get some random training images
dataiter = iter(trainloader)
images, labels = dataiter.next()

# show images
```

(continues on next page)

(continued from previous page)

```
imshow(torchvision.utils.make_grid(images))
print('     '+' '.join(f'{classes_num[labels[j]]:4s}' for j in range(batch_size)))
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0.
↪.255] for integers).
```




```
    2    2    3    1    4    0    7    8
```

1.1.1 Implementing Neural Networks

Euclidean Neural Network

Let's start with defining a simple Euclidean neural network. Our simple neural network consists of three fully connected layers. To create the fully connected layers in Euclidean space, `nn.Linear` in used.

[15]:
```
class EuclideanNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(28 * 28, 750)
        self.relu1 = nn.ReLU()
        self.fc11 = nn.Linear(750, 20)
        self.relu11 = nn.ReLU()
        self.fc2 = nn.Linear(20, 10)

    def forward(self, x):
        out_x = self.fc1(x)
        out_x = self.relu1(out_x)
        out_x = self.fc11(out_x)
        out_x = self.relu11(out_x)
        out_x = self.fc2(out_x)
        return out_x

euclidean_net = EuclideanNet().to(device)
```

Hyperbolic Neural Network

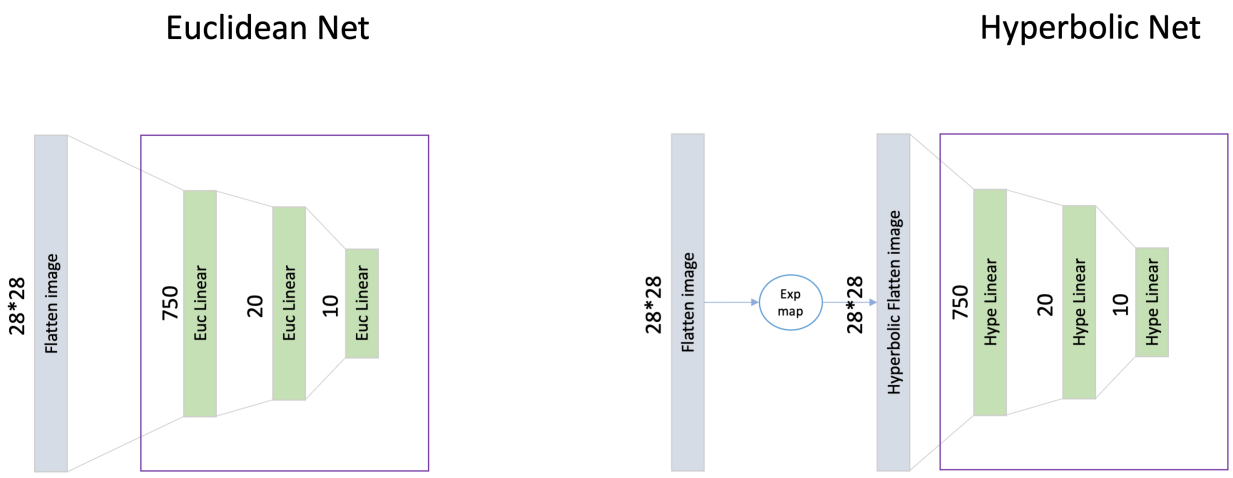
Next step is to create a neural network in hyperbolic space. This network consists of three fully connected layer, similar to the netowrk defined in the Euclidean space. To define a fully connected layer in the hyperbolic space, `MobiusLinear` is used.

[16]:
```
def hyperbolic_ReLU(hyperbolic_input, manifold):
    euclidean_input = manifold.logmap0(hyperbolic_input)
    euclidean_output = F.relu(euclidean_input)
    hyperbolic_output = manifold.expmap0(euclidean_output)
    return hyperbolic_output
```

[17]:
```
class HyperbolicNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = MobiusLinear(28 * 28, 750)
        self.fc11 = MobiusLinear(750, 20)
        self.fc2 = MobiusLinear(20, 10)

    def forward(self, x, manifold):
        out_x = self.fc1(x)
        out_x = hyperbolic_ReLU(out_x, manifold)
        out_x = self.fc11(out_x)
        out_x = hyperbolic_ReLU(out_x, manifold)
        out_x = self.fc2(out_x)
        return out_x

hyperbolic_net = HyperbolicNet().to(device)
```

Here you can see the overview of both Euclidean and hyperbolic Neural Networks. The main structure of the networks are similar, one consisting of *Euclidean* Linear layers and the other one of *hyperbolic* layers.

As the Hyperbolic Net consists of hyperbolic layers, the input must be in hyperbolic space, too. Therefore, `Exp map` which stands for `Exponential Map` is used to project the input of the Neural Network (flatten image) from Euclidean space to the hyperbolic space. `Exp map` will be performed later in the training and testing loop.



Optimizer and Loss function

Next step is to define the optimizer and loss function. For training a neural network with hyperbolic layers, the optimizer should be in hyperbolic space, too. Here we use `SGD` and `Riemannian SGD` as the optimizers.

[18]:
```
criterion = nn.CrossEntropyLoss()
euclidean_optimizer = optim.SGD(euclidean_net.parameters(), lr=0.001, momentum=0.9)
hyperbolic_optimizer = geoopt.optim.RiemannianSGD(list(hyperbolic_net.parameters()),␣
↪lr=0.001, momentum=0.9)
```

Finally, it's time to implement the training loop.

Euclidean training loop

First, we start with training the Euclidean neural network.

[19]:
```
for epoch in range(1000):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data
        inputs = inputs.to(device)
        labels = labels.to(device)

        # zero the parameter gradients
        euclidean_optimizer.zero_grad()

        # forward + backward + optimize
```

(continues on next page)

(continued from previous page)

```
        flatten_inputs = torch.flatten(inputs, start_dim=1)
        outputs = euclidean_net(flatten_inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        euclidean_optimizer.step()

        # print statistics
        running_loss += loss.item()
        if (epoch % 100) == 0:
            if (i % len(trainloader)) == (len(trainloader) - 1):
                print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / len(trainloader):
↪.3f}')
                running_loss = 0.0
                #␣
↪==========================================================================
                correct = 0
                total = 0
                with torch.no_grad():
                    for data in valloader:
                        val_images, val_labels = data
                        val_images = val_images.to(device)
                        val_labels = val_labels.to(device)
                        # calculate outputs by running images through the network
                        flatten_val_images = torch.flatten(val_images, start_dim=1)
                        val_outputs = euclidean_net(flatten_val_images)
                        # the class with the highest energy is what we choose as␣
↪prediction
                        _, val_predicted = torch.max(val_outputs.data, 1)
                        total += val_labels.size(0)
                        correct += (val_predicted == val_labels).sum().item()
                print(f'Accuracy of the network on the 10000 val images: {100 * correct /
↪/ total} %')
                #␣
↪==========================================================================
print('Finished Training')
```

```
[1,  1250] loss: 0.732
Accuracy of the network on the 10000 val images: 89 %
[101,  1250] loss: 0.000
Accuracy of the network on the 10000 val images: 96 %
[201,  1250] loss: 0.000
Accuracy of the network on the 10000 val images: 96 %
[301,  1250] loss: 0.000
Accuracy of the network on the 10000 val images: 96 %
[401,  1250] loss: 0.000
Accuracy of the network on the 10000 val images: 96 %
[501,  1250] loss: 0.000
Accuracy of the network on the 10000 val images: 96 %
[601,  1250] loss: 0.000
Accuracy of the network on the 10000 val images: 96 %
[701,  1250] loss: 0.000
Accuracy of the network on the 10000 val images: 96 %
```

(continues on next page)

(continued from previous page)

```
[801,  1250] loss: 0.000
Accuracy of the network on the 10000 val images: 96 %
[901,  1250] loss: 0.000
Accuracy of the network on the 10000 val images: 96 %
Finished Training
```

Hyperbolic training loop

Next step is to train the hyperbolic neural network.

In addition to the optimizers, there is one more difference between Euclidean and hyperbolic training loops; `Exponential Map`. As the hyperbolic Net expects hyperbolic input, data is projected to the hyperbolic space using `manifold.expmap0(flat_inputs)`.

[20]:
```
manifold = geoopt.PoincareBall()

for epoch in range(1000):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data
        inputs = inputs.to(device)
        labels = labels.to(device)

        # zero the parameter gradients
        hyperbolic_optimizer.zero_grad()

        # forward + backward + optimize
        # ------------------------------
        # Hyperbolic inputs
        flat_inputs = torch.flatten(inputs, start_dim=1)
        hyp_inputs = manifold.expmap0(flat_inputs)
        # ------------------------------
        outputs = hyperbolic_net(hyp_inputs, manifold)
        loss = criterion(outputs, labels)
        loss.backward()
        hyperbolic_optimizer.step()

        # print statistics
        running_loss += loss.item()
        if (epoch % 100) == 0:
            # print statistics
            if (i % len(trainloader)) == (len(trainloader) - 1):
                print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / len(trainloader):
↪.3f}')
                running_loss = 0.0
                #␣
↪==============================================================================
                correct = 0
                total = 0
```

(continues on next page)

(continued from previous page)

```
                with torch.no_grad():
                    for data in valloader:
                        val_images, val_labels = data

                        val_images = val_images.to(device)
                        val_labels = val_labels.to(device)

                        # calculate outputs by running images through the network
                        flatten_val_images = torch.flatten(val_images, start_dim=1)
                        hyp_flatten_val_images = manifold.expmap0(flatten_val_images)
                        val_outputs = hyperbolic_net(hyp_flatten_val_images, manifold)
                        # the class with the highest energy is what we choose as␣
↪prediction
                        _, val_predicted = torch.max(val_outputs.data, 1)
                        total += val_labels.size(0)
                        correct += (val_predicted == val_labels).sum().item()
                print(f'Accuracy of the network on the 10000 val images: {100 * correct /
↪/ total} %')
                #␣
↪==========================================================================
print('Finished Training')
```

```
[1,  1250] loss: 1.975
Accuracy of the network on the 10000 val images: 82 %
[101,  1250] loss: 1.430
Accuracy of the network on the 10000 val images: 95 %
[201,  1250] loss: 1.427
Accuracy of the network on the 10000 val images: 95 %
[301,  1250] loss: 1.426
Accuracy of the network on the 10000 val images: 95 %
[401,  1250] loss: 1.425
Accuracy of the network on the 10000 val images: 95 %
[501,  1250] loss: 1.425
Accuracy of the network on the 10000 val images: 95 %
[601,  1250] loss: 1.424
Accuracy of the network on the 10000 val images: 95 %
[701,  1250] loss: 1.424
Accuracy of the network on the 10000 val images: 95 %
[801,  1250] loss: 1.424
Accuracy of the network on the 10000 val images: 95 %
[901,  1250] loss: 1.424
Accuracy of the network on the 10000 val images: 95 %
Finished Training
```

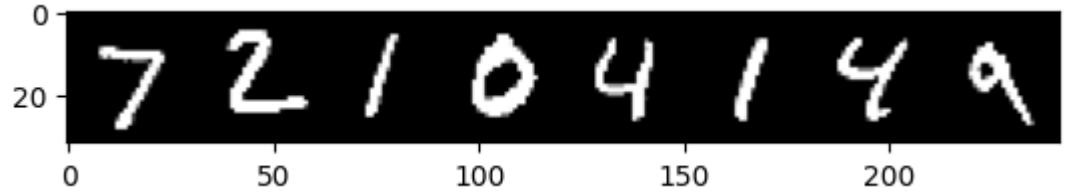
Testing function

Once the training loop is complete, it's time to see how the trained Euclidean and hyperbolic models work given the testing split.

First, we visualize a batch of test images and the predicted labels given the batch.

[21]:
```
dataiter = iter(testloader)
images, labels = dataiter.next()

flatten_images = torch.flatten(images, start_dim=1).to(device)
# ======================================
outputs = euclidean_net(flatten_images)
_, euclidean_predicted = torch.max(outputs, 1)
# ======================================
hyp_inputs = manifold.expmap0(flatten_images)
outputs = hyperbolic_net(hyp_inputs, manifold)
_, hyperbolic_predicted = torch.max(outputs, 1)

# print images
imshow(torchvision.utils.make_grid(images))
print('GroundTruth:', ' '.join(f'{classes_num[labels[j]]:3s}' for j in range(8)))
print('Euclidean:  ', ' '.join(f'{classes_num[euclidean_predicted[j]]:3s}' for j in␣
↪range(8)))
print('Hyperbolic: ', ' '.join(f'{classes_num[hyperbolic_predicted[j]]:3s}' for j in␣
↪range(8)))
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0.
↪.255] for integers).
```




```
GroundTruth: 7   2   1   0   4   1   4   9
Euclidean:   7   2   1   0   4   1   4   9
Hyperbolic:  7   2   1   0   4   1   4   9
```

[30]:
```
# Test dataset - downloading and loading the testing dataset.
testset = torchvision.datasets.MNIST(root=DATA_PATH, train=False, download=True,␣
↪transform=transform)
testset_small, _ = torch.utils.data.random_split(main_trainset, [30000, 30000],␣
↪generator=torch.Generator().manual_seed(42))

# Create dataloaders for the train, val and test sets
batch_size = 8
testloader = torch.utils.data.DataLoader(testset_small, batch_size=batch_size,␣
↪shuffle=False, num_workers=2)
# testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size, shuffle=False,
↪ num_workers=2)
```

(continues on next page)

(continued from previous page)

```
euclidean_correct = 0
hyperbolic_correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for our outputs
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images = images.to(device)
        labels = labels.to(device)
        # calculate outputs by running images through the network
        flatten_images = torch.flatten(images, start_dim=1)

        euclidean_outputs = euclidean_net(flatten_images)
        # ======================================
        hyp_inputs = manifold.expmap0(flatten_images)
        hyperbolic_outputs = hyperbolic_net(hyp_inputs, manifold)

        # the class with the highest energy is what we choose as prediction
        _, euclidean_predicted = torch.max(euclidean_outputs.data, 1)
        _, hyperbolic_predicted = torch.max(hyperbolic_outputs.data, 1)
        total += labels.size(0)

        euclidean_correct += (euclidean_predicted == labels).sum().item()
        hyperbolic_correct += (hyperbolic_predicted == labels).sum().item()

print(f'Accuracy of the Euclidean network on the 10000 test images: {100 * euclidean_
↪correct // total} %')
print(f'Accuracy of the Hyperbolic network on the 10000 test images: {100 * hyperbolic_
↪correct // total} %')
```

```
Accuracy of the Euclidean network on the 10000 test images: 97 %
Accuracy of the Hyperbolic network on the 10000 test images: 97 %
```

1.2 Hyperbolic Image Embeddings

Welcome to our second notebook for the ECCV 2022 Tutorial "Hyperbolic Representation Learning for Computer Vision"!

Open notebook:

Author: Mina Ghadimi Atigh

In this tutorial, you will go through Hyperbolic Image Embeddings, CVPR 2020 paper. The paper argues that hyperbolic spaces with negative curvature might often be more appropriate for learning the embedding of images. It shows that many image classification architectures, in particular the few-shot learning setting, can be easily modified to operate on hyperbolic embeddings, which in many cases also leads to improvements.

In this tutorial, you will go through the hyperbolic *Few-shot learning* task. Few-shot learning aims to classify new data given only a few training samples with supervised information. The few-shot learning task is formulated as N-way K-shot, in which N is the number of classes to classify and K is the number of samples given for each class.

Let's start with installing and importing libraries. Also, we set a manual seed using `set_seed`.

[1]:
```
!wget -q https://raw.githubusercontent.com/leymir/hyperbolic-image-embeddings/
↪6633edbbeffd6d90271f0963852a046c64f407d6/examples/fewshot/networks/convnet.py
!pip install -q git+https://github.com/geoopt/geoopt.git
```

[2]:
```
## standard libraries
import numpy as np
import os
import PIL
from PIL import Image
from PIL import ImageEnhance
import random
import warnings

## Imports for plotting
import matplotlib.pyplot as plt

## PyTorch
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader

## PyTorch Torchvision
import torchvision
from torchvision import transforms

## geoopt for hyperbolic
import geoopt

## Neural network downloaded from paper's GitHub
from convnet import ConvNet

warnings.filterwarnings('ignore')
```

[3]:
```
# Function for setting the seed
def set_seed(seed):
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed(seed)
        torch.cuda.manual_seed_all(seed)
set_seed(42)

# Ensure that all operations are deterministic on GPU (if used) for reproducibility
torch.backends.cudnn.determinstic = True
torch.backends.cudnn.benchmark = False

# Fetching the device that will be used throughout this notebook
device = torch.device("cpu") if not torch.cuda.is_available() else torch.device("cuda:0")
print("Using device", device)
```

```
Using device cpu
```

Throughout this tutorial, you will work with *CUB-200-2011* Dataset. The CUB dataset consists of 11788 images of 200 bird classes. In the experiments, 100 classes are used for training, 50 for validation, and 50 for testing. The code starts with downloading the dataset first.

Using the code, you can download the data to the path `data/cub`. Images will be saved in `data/cub/images`. Train, val, and test splits will be saved in `data/cub/split`. Also, there will be a `data/cub/classes.txt` file, containing the names of the 200 classes, which will be used for visualization.

[4]:
```
from IPython.display import clear_output
%mkdir data
%mkdir data/cub
%cd data/cub
!wget -q -O tmp.tgz https://data.caltech.edu/records/65de6-vp158/files/CUB_200_2011.tgz?
↪download=1
!tar -xzvf tmp.tgz
!rm tmp.tgz
!rm attributes.txt
%mkdir images
!mv CUB_200_2011/images/*/*.jpg images
!mv CUB_200_2011/classes.txt classes.txt
!rm -r CUB_200_2011
clear_output()

%mkdir split
%cd split
![ ! -f test.csv ] && wget -q https://raw.githubusercontent.com/leymir/hyperbolic-image-
↪embeddings/master/examples/fewshot/data/cub/split/test.csv
![ ! -f val.csv ] && wget -q https://raw.githubusercontent.com/leymir/hyperbolic-image-
↪embeddings/master/examples/fewshot/data/cub/split/val.csv
![ ! -f train.csv ] && wget -q https://raw.githubusercontent.com/leymir/hyperbolic-image-
↪embeddings/master/examples/fewshot/data/cub/split/train.csv
%cd ../../..
```

```
/content/data/cub/split
/content
```

The next step is to handle data loader functions.

[5]:
```
# ROOT to the images and split, data files are available here.
ROOT_PATH = os.getcwd()
DATA_PATH = os.path.join(ROOT_PATH, "data/cub/")
IMAGE_PATH = os.path.join(DATA_PATH, "images")
SPLIT_PATH = os.path.join(DATA_PATH, "split")

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
↪~~~~~~~~~~~~~~~
transformtypedict = dict(
    Brightness=ImageEnhance.Brightness,
    Contrast=ImageEnhance.Contrast,
    Sharpness=ImageEnhance.Sharpness,
    Color=ImageEnhance.Color,
)
```

(continues on next page)

(continued from previous page)

```
class ImageJitter(object):
    def __init__(self, transformdict):
        self.transforms = [(transformtypedict[k], transformdict[k]) for k in␣
↪transformdict]

    def __call__(self, img):
        out = img
        randtensor = torch.rand(len(self.transforms))
        for i, (transformer, alpha) in enumerate(self.transforms):
            r = alpha * (randtensor[i] * 2.0 - 1.0) + 1
            out = transformer(out).enhance(r).convert("RGB")
        return out


# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
↪~~~~~~~~~~~~~~~

# This is for the CUB dataset, which does not support the ResNet encoder now
# It is notable, we assume the cub images are cropped based on the given bounding boxes
# The concept labels are based on the attribute value, which are for further use (and␣
↪not used in this work)
class CUB(Dataset):
    def __init__(self, setname):
        txt_path = os.path.join(SPLIT_PATH, setname + ".csv")
        lines = [x.strip() for x in open(txt_path, "r").readlines()][1:]
        data = []
        label = []
        lb = -1
        self.wnids = []

        for l in lines:
            context = l.split(",")
            name = context[0]
            wnid = context[1]
            path = os.path.join(IMAGE_PATH, name)
            if wnid not in self.wnids:
                self.wnids.append(wnid)
                lb += 1

            data.append(path)
            label.append(lb)

        self.data = data
        self.label = label
        self.num_class = np.unique(np.array(label)).shape[0]

        normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,␣
↪0.225])

        if setname == "train":
```

(continues on next page)

(continued from previous page)

```
            self.transform = transforms.Compose([transforms.RandomResizedCrop(84),
                                                 ImageJitter(dict(Brightness=0.4,␣
↪Contrast=0.4, Color=0.4)),
                                                 transforms.RandomHorizontalFlip(),
                                                 transforms.ToTensor(),
                                                 transforms.Normalize(np.array([0.485, 0.
↪456, 0.406]),
                                                                      np.array([0.229, 0.
↪224, 0.225])),
                                                 ])

        else:
            self.transform = transforms.Compose([transforms.Resize(84, interpolation=PIL.
↪Image.BICUBIC),
                                                 transforms.CenterCrop(84),
                                                 transforms.ToTensor(),
                                                 normalize, ])

    def __len__(self):
        return len(self.data)

    def __getitem__(self, i):
        path, label = self.data[i], self.label[i]
        x = Image.open(path).convert("RGB")
        image = self.transform(Image.open(path).convert("RGB"))
        return image, label


# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
↪~~~~~~~~~~~~~~~
class CategoriesSampler:
    def __init__(self, label, n_batch, n_cls, n_per):
        self.n_batch = n_batch
        self.n_cls = n_cls
        self.n_per = n_per

        label = np.array(label)
        self.m_ind = []
        for i in range(max(label) + 1):
            ind = np.argwhere(label == i).reshape(-1)
            ind = torch.from_numpy(ind)
            self.m_ind.append(ind)

    def __len__(self):
        return self.n_batch

    def __iter__(self):
        for i_batch in range(self.n_batch):
            batch = []
            classes = torch.randperm(len(self.m_ind))[: self.n_cls]
            for c in classes:
                l = self.m_ind[c]
```

(continues on next page)

(continued from previous page)

```
                pos = torch.randperm(len(l))[: self.n_per]
                batch.append(l[pos])
            batch = torch.stack(batch).t().reshape(-1)
            yield batch
```

Exctract class names of the CUB dataset.

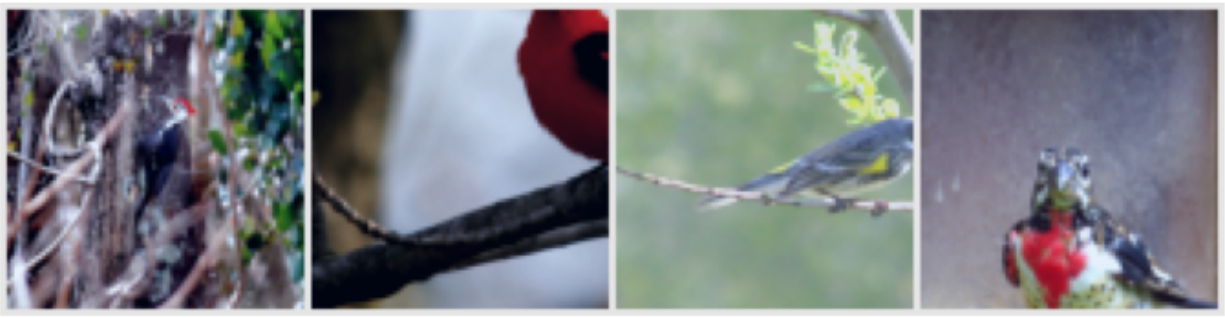
[6]:
```
classname_file = os.path.join(DATA_PATH, 'classes.txt')
class_dict = {}
with open(classname_file) as f:
    lines = f.readlines()
    for i, line in enumerate(lines):
        class_num = int(line.split(' ')[0])
        class_dict[class_num - 1] = line.split('.')[1].split('\n')[0].replace('_', ' ')
```

Before starting the main task, let's check how images from the dataset look like.

[7]:
```
# define a sample trainset
sample_trainset = CUB("train")
```

[8]:
```
# number of images to plot
NUM_IMAGES = 4
# Index of images to plot
indexes = random.sample(range(len(sample_trainset)), NUM_IMAGES)
images = [sample_trainset[idx][0] for idx in indexes]
labels = [sample_trainset[idx][1] for idx in indexes]
```

[9]:
```
img_grid = torchvision.utils.make_grid(images, nrow=8, normalize=True, pad_value=0.9)
img_grid = img_grid.permute(1, 2, 0)

plt.figure(figsize=(12,8))
plt.title("Examples of CUB dataset")
plt.imshow(img_grid)
plt.axis('off')
plt.show()
plt.close()
print('  |  '.join(f'{class_dict[labels[j]]:16s}' for j in range(NUM_IMAGES)))
```




```
Pied Kingfisher   |  White breasted Kingfisher  |  Laysan Albatross  |  Glaucous winged␣
↪Gull
```

1.2.1 Few Shot learning: Euclidean and Hyperbolic Space

First, we set the value of hyperparameters, assigned using `Config`. The hyperparameters are set for 5-way 1-shot task. Note that we train the model for 5 epochs, but to reproduce the final paper numbers, the model should be trained more.

[10]:
```
class Config:
    def __init__(self, shot=1, lr=0.001, step=50, gamma=0.8, c=0.05, model="convnet",␣
↪hyperbolic = True, dim=1600,
                 query=15, way=5, validation_way=5, temperature=1, dataset="CUB", lr_
↪decay=True, max_epoch=200):

        self.lr = lr                                    # learning rate
        self.lr_decay = lr_decay                        # Boolean, if to perform learning␣
↪rate scheduler
        self.step_size=step                             # Period of learning rate decay
        self.gamma = gamma                              # Multiplicative factor of␣
↪learning rate decay
        self.dataset= dataset                           # Dataset name
        self.model = "convnet"                          # Name of Base Model
        self.temperature = temperature                  # temperature used in calculating␣
↪logits
        self.max_epoch=max_epoch                        # number of epochs

        self.c = c                                      # Curvature of the hyperbolic space
        self.hyperbolic = hyperbolic                    # Boolean, if it is hyperbolic or␣
↪not
        self.dim = dim                                  # dimenionality of the output␣
↪vector

        self.shot = shot                                # Number of shots in Few shot␣
↪learning task
        self.query = query                              #
        self.way = way                                  # Number of ways in Few shot␣
↪learning task
        self.validation_way = validation_way            # Number of ways in Few shot␣
↪learning task for validation set


args = Config(dim=512, max_epoch = 5, hyperbolic= True)
```

Split training and validation set based on the `Config` for few-shot learning.

[11]:
```
trainset = CUB("train")
train_sampler = CategoriesSampler(trainset.label, 100, args.way, args.shot + args.query)
train_loader = DataLoader(dataset=trainset, batch_sampler=train_sampler, num_workers=8,␣
↪pin_memory=True)

valset = CUB("val")
val_sampler = CategoriesSampler(valset.label, 500, args.validation_way, args.shot + args.
↪query)
val_loader = DataLoader(dataset=valset, batch_sampler=val_sampler, num_workers=8, pin_
↪memory=True)
```

Now, it's time to define the model. Structure of the model is based on "Prototypical Networks for Few-shot Learning

(ProtoNet)", NeurIPS 2017 paper. Hyperbolic Image Embeddings has picked ProtoNet because it is simple in general and simple to convert to hyperbolic geometry. ProtoNets use the so-called **prototype representation** of a class, which is defined as a **mean of the embedded support set of a class**. To generalize this concept to hyperbolic space, Euclidean mean is substituted by *hyperbolic average*.

Once the prototypes are calculated based on the average of data points, it's time to calculate the distance of the data points with the prototypes and perform classificatin based on the calculated logits. To calculate the distance in the hyperbolic space, *Hyperbolic distance* is used.

Hyperbolic Average

Let's first define Hyperbolic average. Extension of Euclidean average to hyperbolic space is called *Einsterin midpoint*, which takes the most simple form in *Klein* coordinates.

$$HypAve(x_1, ..., x_N) = \frac{\sum_{i=1}^{N} \gamma_i x_i}{\sum_{i=1}^{N} \gamma_i} \tag{1.1}$$

where $\gamma_i = \frac{1}{\sqrt{1-c\|x_i\|^2}}$ are the Lorentz factors. The Lorentz factors is implemented in `lorenz_factor` function.

HypAve is in Klein coordinates. Therefore it is necessary to transfer the points to the Klein model first and then project the calculated average onto the Poincare model.

Let $x_{\mathbb{D}}$ and $x_{\mathbb{K}}$ denote the coordinates of the same point in the Poincare and Klein models. To project the points between these models, the following equations are needed.

$$x_{\mathbb{D}} = \frac{x_{\mathbb{K}}}{1 + \sqrt{1 - c\|x_{\mathbb{K}}\|^2}} \tag{1.2}$$

$$x_{\mathbb{K}} = \frac{2x_{\mathbb{D}}}{1 + c\|x_{\mathbb{D}}\|^2} \tag{1.3}$$

These projecttions are implemented in `k2p` and `p2k` functions. Finally, $HypAve$ is implements in `poincare_mean` function.

[12]:
```
def lorenz_factor(x, *, c=1.0, dim=-1, keepdim=False):
    """
    Parameters
    ----------
    x : tensor
        point on Klein disk
    c : float
        negative curvature
    dim : int
        dimension to calculate Lorenz factor
    keepdim : bool
        retain the last dim? (default: false)

    Returns
    -------
    tensor
        Lorenz factor
    """
    return 1 / torch.sqrt(1 - c * x.pow(2).sum(dim=dim, keepdim=keepdim))
```

(continues on next page)

(continued from previous page)

```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Project a point from Klein model to Poincare model
def k2p(x, c):
    denom = 1 + torch.sqrt(1 - c * x.pow(2).sum(-1, keepdim=True))
    return x / denom

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Project a point from Poincare model to Klein model
def p2k(x, c):
    denom = 1 + c * x.pow(2).sum(-1, keepdim=True)
    return 2 * x / denom

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

def poincare_mean(x, dim=0, c=1.0):
    # To calculate the mean, another model of hyperbolic space named Klein model is used.
    # 1. point is projected from Poincare model to Klein model using p2k, output x is a␣
↪point in Klein model
    x = p2k(x, c)
    # 2. mean is calculated
    lamb = lorenz_factor(x, c=c, keepdim=True)
    mean = torch.sum(lamb * x, dim=dim, keepdim=True) / torch.sum(lamb, dim=dim,␣
↪keepdim=True)
    # 3. Mean is projected from Klein model to Poincare model
    mean = k2p(mean, c)
    return mean.squeeze(dim)
```

Now, it's time to define the model `ProtoNet`. Whether `args.hyperbolic` is *True* of *False*, the model can be Hyperbolic or Euclidean.

[13]:
```
class ProtoNet(nn.Module):
    def __init__(self, args):
        super().__init__()
        self.args = args

        # Base Model: ConvNet
        self.encoder = ConvNet(z_dim=args.dim)

        # If working in Hyperbolic Space
        if args.hyperbolic:
            self.manifold = geoopt.PoincareBall(c=args.c)

    def forward(self, data_shot, data_query):
        # 1. feed data to the model
        proto = self.encoder(data_shot)

        # Hyperbolic Space:
        if self.args.hyperbolic:
            # 2. encoder is Euclidean, so proto is in Euclidean space and should be␣
↪projected to Hyperboolic space using exponential map
```

(continues on next page)

(continued from previous page)

```
            proto = self.manifold.expmap0(proto)

            if self.training:
                proto = proto.reshape(self.args.shot, self.args.way, -1)
            else:
                proto = proto.reshape(self.args.shot, self.args.validation_way, -1)

            # 3. calculate prototypes based on mean of data
            proto = poincare_mean(proto, dim=0, c=self.manifold.c.item())

            # 4. query is projected to hyperbolic space too
            data_query = self.manifold.expmap0(self.encoder(data_query))

            # 5. Logits is calculated based on the Hyperbolic distance between data␣
↪query and proto
            logits = (-self.manifold.dist(data_query[:, None, :], proto) / self.args.
↪temperature)

        # Euclidean Space
        else:
            # 2. calculate prototypes based on mean of data
            if self.training:
                proto = proto.reshape(self.args.shot, self.args.way, -1).mean(dim=0)
            else:
                proto = proto.reshape(self.args.shot, self.args.validation_way, -1).
↪mean(dim=0)

            # 3. Logits is calculated based on the Euclidean distance between data query␣
↪and proto
            logits = (((self.encoder(data_query)[:, None, :] - proto)**2).sum(dim=-1) /␣
↪self.args.temperature)
        return logits


model = ProtoNet(args).to(device)
```

Once the model is defined, it's time for the training loop. Let's first define *optimizer* and *learning rate scheduler*, alongside with functions to calculate accuracy and average.

[14]:
```
optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)

if args.lr_decay:
    lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=args.step_size,␣
↪gamma=args.gamma)
```

[15]:
```
# Function to calculate the accuracy given logits and groundtruth labels
def count_acc(logits, label):
    pred = torch.argmax(logits, dim=1)
    if torch.cuda.is_available():
        return (pred == label).type(torch.cuda.FloatTensor).mean().item()
    else:
        return (pred == label).type(torch.FloatTensor).mean().item()
```

(continues on next page)

(continued from previous page)

```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Class defined to average the values passed to it
class Averager:
    def __init__(self):
        self.n = 0
        self.v = 0

    def add(self, x):
        self.v = (self.v * self.n + x) / (self.n + 1)
        self.n += 1

    def item(self):
        return self.v
```

Training loop

Next step is training the model.

[16]:
```
for epoch in range(1, args.max_epoch + 1):
    if args.lr_decay:
        lr_scheduler.step()
    model.train()

    tl = Averager()
    ta = Averager()

    label = torch.arange(args.way).repeat(args.query)
    if torch.cuda.is_available():
        label = label.type(torch.cuda.LongTensor)
    else:
        label = label.type(torch.LongTensor)

    for i, batch in enumerate(train_loader, 1):
        if torch.cuda.is_available():
            data, _ = [_.cuda() for _ in batch]
        else:
            data = batch[0]
        p = args.shot * args.way
        data_shot, data_query = data[:p], data[p:]
        logits = model(data_shot, data_query)
        loss = F.cross_entropy(logits, label)
        acc = count_acc(logits, label)
        if (i == 1) or (i == len(train_loader)):
            print("epoch {}, train {}/{}, loss={:.4f} acc={:.4f}".format(epoch, i,␣
↪len(train_loader), loss.item(), acc))

        tl.add(loss.item())
        ta.add(acc)

        optimizer.zero_grad()
        loss.backward()
```

(continues on next page)

(continued from previous page)

```
        optimizer.step()

    tl = tl.item()
    ta = ta.item()

    model.eval()

    vl = Averager()
    va = Averager()

    label = torch.arange(args.validation_way).repeat(args.query)
    if torch.cuda.is_available():
        label = label.type(torch.cuda.LongTensor)
    else:
        label = label.type(torch.LongTensor)

    with torch.no_grad():
        for i, batch in enumerate(val_loader, 1):
            if torch.cuda.is_available():
                data, _ = [_.cuda() for _ in batch]
            else:
                data = batch[0]
            p = args.shot * args.validation_way
            data_shot, data_query = data[:p], data[p:]
            logits = model(data_shot, data_query)
            loss = F.cross_entropy(logits, label)
            acc = count_acc(logits, label)

            vl.add(loss.item())
            va.add(acc)

    vl = vl.item()
    va = va.item()
    print("epoch {}, val, loss={:.4f} acc={:.4f}".format(epoch, vl, va))
```

```
epoch 1, train 1/100, loss=1.9849 acc=0.3467
epoch 1, train 100/100, loss=1.4964 acc=0.3467
epoch 1, val, loss=1.5653 acc=0.3891
epoch 2, train 1/100, loss=1.6622 acc=0.3067
epoch 2, train 100/100, loss=1.5009 acc=0.3867
epoch 2, val, loss=1.4661 acc=0.4106
epoch 3, train 1/100, loss=1.8146 acc=0.3067
epoch 3, train 100/100, loss=1.5077 acc=0.3067
epoch 3, val, loss=1.3767 acc=0.4407
epoch 4, train 1/100, loss=1.6395 acc=0.3200
epoch 4, train 100/100, loss=1.6156 acc=0.3333
epoch 4, val, loss=1.3271 acc=0.4650
epoch 5, train 1/100, loss=1.6648 acc=0.2933
epoch 5, train 100/100, loss=1.6426 acc=0.3867
epoch 5, val, loss=1.3222 acc=0.4531
```

Testing function

Once the training is complete, it's time to see model's performance on the test split.

First, let's create test dataloader.

[17]:
```
test_set = CUB("test")
test_sampler = CategoriesSampler(test_set.label, 1000, args.validation_way, args.shot +␣
↪args.query)
test_loader = DataLoader(test_set, batch_sampler=test_sampler, num_workers=8, pin_
↪memory=True)
```

[18]:
```
def compute_confidence_interval(data):
    """
    Compute 95% confidence interval
    :param data: An array of mean accuracy (or mAP) across a number of sampled episodes.
    :return: the 95% confidence interval for this data.
    """
    a = 1.0 * np.array(data)
    m = np.mean(a)
    std = np.std(a)
    pm = 1.96 * (std / np.sqrt(len(a)))
    return m, pm
```

[19]:
```
model.eval()

ave_acc = Averager()
test_acc_record = np.zeros((1000,))

label = torch.arange(args.validation_way).repeat(args.query)
if torch.cuda.is_available():
    label = label.type(torch.cuda.LongTensor)
else:
    label = label.type(torch.LongTensor)

# Testing loop
for i, batch in enumerate(test_loader, 1):
    if torch.cuda.is_available():
        data, _ = [_.cuda() for _ in batch]
    else:
        data = batch[0]
    k = args.validation_way * args.shot
    data_shot, data_query = data[:k], data[k:]

    logits = model(data_shot, data_query)
    acc = count_acc(logits, label)
    ave_acc.add(acc)
    test_acc_record[i - 1] = acc

m, pm = compute_confidence_interval(test_acc_record)
print("Test Acc {:.4f} + {:.4f}".format(m, pm))
```

```
Test Acc 0.3999 + 0.0061
```

As mentioned when defining `ProtoNet`, if `args.hyperbolic` is True, the model will be hyperbolic, Euclidean otherwise. To compare the performance of ProtoNet in hyperbolic and Euclidean manifolds, we train both of them for the same number of epochs. When training both models for 5 epochs, the test accuracy of hyperbolic ProtoNet is around $39.99 results provided in the paper, when training hyperbolic and Euclidean ProtoNet for 50$ epochs, the Euclidean version results in 51.31

1.3 Hyperbolic Busemann Learning with Ideal Prototypes

Welcome to our third notebook for the ECCV 2022 Tutorial "Hyperbolic Representation Learning for Computer Vision"!

Open notebook:

Authors: Mina Ghadimi Atigh

In this notebook, you will go through Hyperbolic Busemann Learning with Ideal prototypes, NeurIPS 2021 paper. The code is available in Github.

Building on the success of deep learning with prototypes in Euclidean and hyperspherical spaces, a few recent works have proposed hyperbolic prototypes for classification. Such approaches enable effective learning in low-dimensional output spaces and can exploit hierarchical relations amongst classes, but require privileged information about class labels to position the hyperbolic prototypes. This paper strives to combine the best of both non-Euclidean worlds, namely efficient low-dimensional embeddings from hyperbolic prototypes and the knowledge-free positioning from hyperspherical prototypes.

The paper makes three contributions. First, introduces a hyperbolic prototype network with class prototypes given as points on the ideal boundary of the Poincaré ball model of hyperbolic geometry. Second, proposes the penalized Busemann loss, which enables us to compute proximities between example outputs in hyperbolic space and prototypes at the ideal boundary, an impossible task for existing distance metrics, which put the ideal boundary at infinite distance from all other points in hyperbolic space. Third, we provide a theoretical link between our hyperbolic prototype approach and logistic regression.

Let's start with importing the libraries and setting manual seed using `set_seed`.

[1]:
```
## standard libraries
import math
import numpy as np
from PIL import Image
import warnings

## Imports for plotting
import matplotlib
import matplotlib.pyplot as plt

## PyTorch
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

## PyTorch Torchvision
import torchvision
from torchvision import datasets, transforms

warnings.filterwarnings('ignore')
```

[2]:
```
!wget -q https://raw.githubusercontent.com/MinaGhadimiAtigh/Hyperbolic-Busemann-Learning/
↪master/models/cifar/resnet.py
!pip install -q git+https://github.com/geoopt/geoopt.git
import resnet
import geoopt
```

[3]:
```
# Function for setting the seed
def set_seed(seed):
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed(seed)
        torch.cuda.manual_seed_all(seed)
set_seed(42)

# Ensure that all operations are deterministic on GPU (if used) for reproducibility
torch.backends.cudnn.determinstic = True
torch.backends.cudnn.benchmark = False

# Fetching the device that will be used throughout this notebook
device = torch.device("cpu") if not torch.cuda.is_available() else torch.device("cuda:0")
print("Using device", device)
```

```
Using device cuda:0
```

1.3.1 Prototype Learning - Ideal Prototypes

The central idea of this paper is to position class prototypes at ideal points, which represent points at infinity in hyperbolic geometry.

In the Poincare′ model, the ideal points form the boundary of the ball:

$$\mathbb{I}_d = \{\mathbf{z} \in \mathbb{R}^d : z_1^2 + \cdots + z_d^2 = 1\}. \tag{1.4}$$

Using the same approach as "Hyperspherical Prototype Networks", the paper places the prototypes uniformly on the ideal boundary of hyperbole to learn the prototypes.

[4]:
```
# HYPERPARAMETER

## Number of classes to learn prototypes for
num_classes = 10
## dimension of the learned prototypes (d in the equation)
dims = 2


## Hyperparameters for prototype learning
epochs = 1000
learning_rate = 0.1
momentum = 0.9

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
↪~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

(continues on next page)

(continued from previous page)

```
# When prototype dimension >=3, the goal is to find the largest cosine similarity␣
↪between pairs of prototypes and minimize it.
def prototype_loss(prototype):
    # Dot product of normalized prototypes is cosine similarity.
    product = torch.matmul(prototype, prototypes.t()) + 1
    # Remove diagnonal from loss.
    product -= 2. * torch.diag(torch.diag(product))
    # Minimize maximum cosine similarity.
    loss = product.max(dim=1)[0]

    return loss.mean(), product.max()

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
↪~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# When prototype dimension=2, learning can be easily performed by splitting the unit-
↪cirlce into equal parts,
# seperated by an angle of 2/n, which n stands for the number of classes.
# Then, for each angle , the coordinates are obtained as (cos ,sin ).

def prototype_unify(num_classes):
    single_angle = 2 * math.pi / num_classes
    help_list = np.array(range(0, num_classes))
    angles = (help_list * single_angle).reshape(-1, 1)

    sin_points = np.sin(angles)
    cos_points = np.cos(angles)

    set_prototypes = torch.tensor(np.concatenate((cos_points, sin_points), axis=1))
    return set_prototypes

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
↪~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Now, prototype learning can be performed.
# While for d=2 prototype learning consists of only splitting the unit-circle, for d>2,␣
↪training and optimization is needed.
if dims == 2:
    prototypes = prototype_unify(num_classes)
    prototypes = nn.Parameter(F.normalize(prototypes, p=2, dim=1))

elif dims > 2:
    prototypes = torch.randn(num_classes, dims)
    prototypes = nn.Parameter(F.normalize(prototypes, p=2, dim=1))
    optimizer = optim.SGD([prototypes], lr=learning_rate, momentum=momentum)
    # Optimize for separation.
    for i in range(epochs):
        # Compute loss.
        loss, _ = prototype_loss(prototypes)
        # Update.
        loss.backward()
        optimizer.step()
```

(continues on next page)

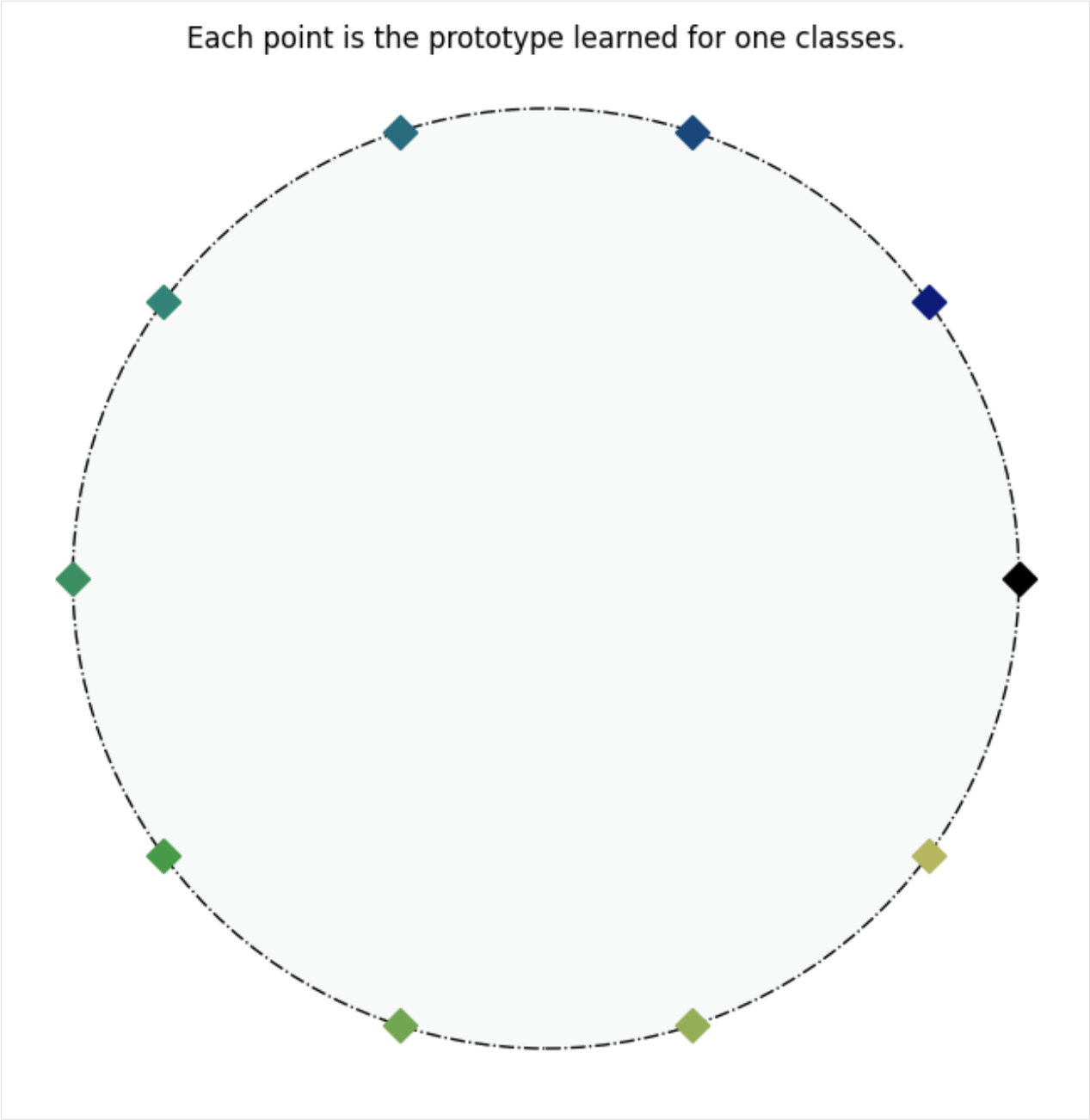
(continued from previous page)

```
        # Normalize prototypes again
        prototypes = nn.Parameter(F.normalize(prototypes, p=2, dim=1))
        optimizer = optim.SGD([prototypes], lr=learning_rate, momentum=momentum)
```

Now the prototypes are ready. As you may have noticed, the prototype learning is class agnostic. It only requires the number of classes and dimensions of the prototypes, and there is no need for data.

Let's visualize the prototypes when $d = 2$.

[5]:
```
if dims == 2:

    cmap = plt.get_cmap('gist_earth')
    colors_list = [cmap(i) for i in np.linspace(0, 1, num_classes + 4)]

    matplotlib.pyplot.figure(figsize=(8, 8))
    ax = plt.gca()

    for i in range(num_classes):
        plt.plot(prototypes[i, 0].detach().numpy(), prototypes[i, 1].detach().numpy(),␣
↪markersize = 10,  marker = 'D', linestyle='none', color = colors_list[i])

    ax.add_patch(plt.Circle((0, 0), 1, fill=True, ec='black', fc='#f8f9f9', linestyle='-.
↪', linewidth=1, zorder=0))

    ax.spines['bottom'].set_color('white')
    ax.spines['top'].set_color('white')
    ax.spines['right'].set_color('white')
    ax.spines['left'].set_color('white')

    ax.tick_params(
        axis='both',   # changes apply to the x-axis
        which='both',  # both major and minor ticks are affected
        bottom=False,  # ticks along the bottom edge are off
        top=False,     # ticks along the top edge are off
        left=False,
        labelbottom=False)

    # Turn off tick labels
    ax.set_yticklabels([])
    ax.set_xticklabels([])
    plt.title('Each point is the prototype learned for one classes.')
    plt.show()
```



Throught this tutorial, we will perform prototype learning using the Ideal prototypes. To this end, `CIFAR10` dataset is selected.

The first step is downloading and loading the dataset and dataloader.

[6]:
```
## Path to save the data
basedir = '.'

## batch size needed to define the dataloader
batch_size = 128
```

[7]:
```
mrgb = [0.507, 0.487, 0.441]
srgb = [0.267, 0.256, 0.276]


kwargs = {'num_workers': 32, 'pin_memory': True}
normalize = transforms.Normalize(mean=mrgb, std=srgb)

classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck
↪']

# Training Dataset
train_dataset = datasets.CIFAR10(root=basedir + 'cifar10/', train=True,
                                 transform=transforms.Compose([transforms.RandomCrop(32,␣
↪4),
                                                               transforms.
↪RandomHorizontalFlip(),
                                                               transforms.ToTensor(),
                                                               normalize,
                                                               ]), download=True)
trainloader = torch.utils.data.DataLoader(train_dataset,batch_size=batch_size,␣
↪shuffle=True, **kwargs)
trainloader.dataset.train_labels = torch.from_numpy(np.array(trainloader.dataset.
↪targets))

# Testing Dataset
test_dataset = datasets.CIFAR10(root=basedir + 'cifar10/', train=False,
                                transform=transforms.Compose([transforms.ToTensor(),
↪normalize,]))
testloader = torch.utils.data.DataLoader(test_dataset,batch_size=batch_size,␣
↪shuffle=True, **kwargs)
testloader.dataset.test_labels = torch.from_numpy(np.array(testloader.dataset.targets))
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to .cifar10/cifar-10-
↪python.tar.gz
```

```
100.0%
```

```
Extracting .cifar10/cifar-10-python.tar.gz to .cifar10/
```

Before starting the main task, let's check how images from the dataset look like.

[8]:
```
NUM_IMAGES = 8
images = [train_dataset[idx][0] for idx in range(NUM_IMAGES)]
labels = [train_dataset.targets[idx] for idx in range(NUM_IMAGES)]

img_grid = torchvision.utils.make_grid(images, nrow=8, normalize=True, pad_value=0.9)
img_grid = img_grid.permute(1, 2, 0)

plt.figure(figsize=(16,8))
plt.title("Examples of CIFAR10")
plt.imshow(img_grid)
plt.axis('off')
plt.show()
plt.close()
print('  ' + ' '.join(f'{classes[labels[j]]:13s}' for j in range(NUM_IMAGES)))
```


```
  frog       truck      truck      deer       car        car        ␣
↪bird       horse
```

1.3.2 Hyperbolic Image Classification using Ideal prototypes

The goal of hyperbolic Busemann learning with Ideal prototypes is to learn hyperbolic representations for the given data close to the corresponding ideal prototype. Initially, a Euclidean model, i.e., `ResNet`, is used to extract Euclidean representations. Then, the Euclidean representations are projected to the Hyperbolic space using `Exponential Map`.

Once the hyperbolic representations are ready, it's time to calculate the distance to the corresponding Ideal prototype and minimize it. Since the prototypes are on the ideal boundary of hyperbolic space, the Poincare distance results in infinity and cannot be calculated. Therefore, `Penalized Busemann Loss` is proposed in the paper.

Given `p` as the prototype and `z` as the hyperbolic data representation, `Busemann function` and `Penalized Busemann Loss` are calculated using $b_p(z)$ and $\ell(z, p)$,

$$b_p(z) = \log \frac{||p - z||^2}{(1 - ||z||^2)} \tag{1.5}$$

$$\ell(z, p) = b_p(z) - \phi(d) \cdot \log(1 - ||z||^2), \tag{1.6}$$

with $\phi(d)$ a scaling factor for the penalty term which is a function of the dimension of the hyperbolic space. In the penalized Busemann loss, $\phi(d)$ governs the amount of regularization. The paper shows that this function should be linear in the number of dimensions. Therefore, $\phi(d)$ is defined as $s.d$, which s is named as `mult` in the code.

[9]:
```
# Hyperbolic hyperparameters
c = 1.0
mult = 0.75
```

[10]:
```
class PeBusePenalty(nn.Module):
    def __init__(self, dimension, mult=1.0):
        super(PeBusePenalty, self).__init__()
        self.dimension = dimension
        self.penalty_constant = mult * self.dimension

    def forward(self, p, g):
        # first part of loss
        prediction_difference = g - p
        difference_norm = torch.norm(prediction_difference, dim=1)
        difference_log = 2 * torch.log(difference_norm)

        # second part of loss
        data_norm = torch.norm(p, dim=1)
        proto_difference = (1 - data_norm.pow(2) + 1e-6)
        proto_log = (1 + self.penalty_constant) * torch.log(proto_difference)
```

(continues on next page)

(continued from previous page)

```
        # second part of loss
        constant_loss = self.penalty_constant * math.log(2)

        one_loss = difference_log - proto_log + constant_loss
        total_loss = torch.mean(one_loss)

        return total_loss
```

The `dims` used here is the prototype and output dimensions, initialized when learning prototypes.

[11]:
```
f_loss = PeBusePenalty(dims, mult)
```

Let's first initialize hyperparamers and optimizers.

[12]:
```
# General hyperparameters
learning_rate = 0.0005
decay = 0.00005
epochs = 100
drop1 = 1000
drop2 = 1100
do_decay = True
```

`ResNet32` is initialized and used to extract Euclidean representations of the input data.

[13]:
```
model = resnet.ResNet(32, dims, 1, prototypes.float())
model = model.to(device)
```

[14]:
```
# Optimizer
optimizer = optim.Adam(model.parameters(), lr=learning_rate, weight_decay=decay)
```

Now, everything is ready to start training.

[15]:
```
manifold = geoopt.PoincareBall(c=c)
print('Training started:')
for i in range(epochs):
    # Learning rate decay.
    if i in [drop1, drop2] and do_decay:
        learning_rate *= 0.1
        for param_group in optimizer.param_groups:
            param_group['lr'] = learning_rate
    # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    avg_loss = 0
    sum_loss = 0
    count = 0
    acc = 0
    model.train()
    for bidx, (data, target) in enumerate(trainloader):
        target_tmp = target.to(device)
        target = model.polars[target]

        data = torch.autograd.Variable(data).to(device)
```

(continues on next page)

(continued from previous page)

```
        target = torch.autograd.Variable(target).to(device)

        output = model(data)
        output_exp_map = manifold.expmap0(output)

        loss_func = f_loss(output_exp_map, target)


        optimizer.zero_grad()
        loss_func.backward()
        optimizer.step()

        sum_loss += loss_func.item()
        count += 1.

        output = model.predict(output_exp_map).float()
        pred = output.max(1, keepdim=True)[1]
        acc += pred.eq(target_tmp.view_as(pred)).sum().item()

    avg_loss = sum_loss / float(len(trainloader.dataset))
    avg_acc = acc / float(len(trainloader.dataset))

    # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    if i != 0 and (i % 10 == 0 or i == epochs - 1):
        valid_acc = 0
        valid_loss = 0
        model.eval()
        with torch.no_grad():
            for data_val, target_val in testloader:
                data_val = torch.autograd.Variable(data_val).to(device)
                target_val = torch.autograd.Variable(target_val).to(device)
                target_loss_val = model.polars[target_val]

                output_val = model(data_val).float()
                output_val_exp_map = manifold.expmap0(output_val)

                output_val = model.predict(output_val_exp_map).float()
                pred_val = output_val.max(1, keepdim=True)[1]
                valid_acc += pred_val.eq(target_val.view_as(pred_val)).sum().item()

                valid_loss += f_loss(output_val_exp_map, target_loss_val.to(device)).
↪item()

        len_test = len(testloader.dataset)

        valid_avg_acc = valid_acc / float(len_test)
        valid_avg_loss = valid_loss / float(len_test)
        print('~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~')
        print('Epoch '+str(i)+' :' )
        print('Training Loss:' + str(avg_loss)+' , Training Accuracy: '+str(avg_acc))
        print('Val Loss:' + str(valid_avg_loss)+' , Val Accuracy: '+str(valid_avg_acc))
```

```
Training started:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Epoch 10 :
Training Loss:0.004980027570724487 , Training Accuracy: 0.64096
Val Loss:0.005297535902261734 , Val Accuracy: 0.6272
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Epoch 20 :
Training Loss:0.004105883414745331 , Training Accuracy: 0.78252
Val Loss:0.004605498462915421 , Val Accuracy: 0.7443
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Epoch 30 :
Training Loss:0.0037260478436946867 , Training Accuracy: 0.83498
Val Loss:0.004251863679289818 , Val Accuracy: 0.7945
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Epoch 40 :
Training Loss:0.003505309413075447 , Training Accuracy: 0.86814
Val Loss:0.0040145975530147555 , Val Accuracy: 0.8232
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Epoch 50 :
Training Loss:0.0033344823771715164 , Training Accuracy: 0.88842
Val Loss:0.003981673476099968 , Val Accuracy: 0.8303
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Epoch 60 :
Training Loss:0.0032073852705955504 , Training Accuracy: 0.90586
Val Loss:0.004107096675038338 , Val Accuracy: 0.8331
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Epoch 70 :
Training Loss:0.00313653585255146 , Training Accuracy: 0.91532
Val Loss:0.003802865654230118 , Val Accuracy: 0.8562
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Epoch 80 :
Training Loss:0.0030833755123615266 , Training Accuracy: 0.9211
Val Loss:0.003853413996100425 6 , Val Accuracy: 0.8557
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Epoch 90 :
Training Loss:0.0030213140720129015 , Training Accuracy: 0.92978
Val Loss:0.003996825054287911 , Val Accuracy: 0.8489
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Epoch 99 :
Training Loss:0.002990082864165306 , Training Accuracy: 0.93196
Val Loss:0.003868225568532943 7 , Val Accuracy: 0.8623
```

Let's visualize the output space.

[16]:
```
output_vectors = []
gt_labels = []

model.eval()
with torch.no_grad():
    for data_val, target_val in testloader:
        data_val = torch.autograd.Variable(data_val).to(device)
        target_val = torch.autograd.Variable(target_val).to(device)
        target_loss_val = model.polars[target_val]
```

(continues on next page)

(continued from previous page)

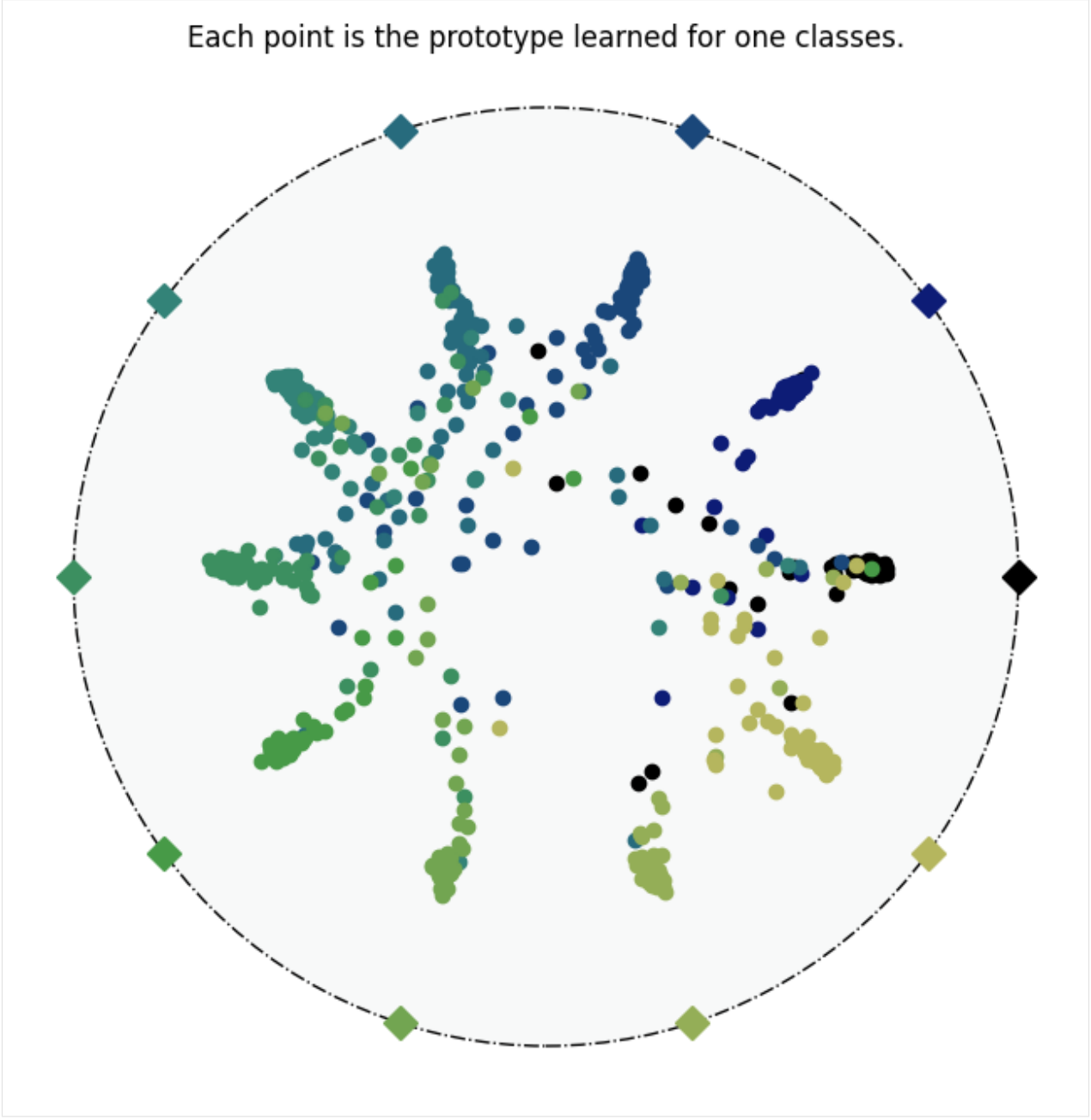
```
        output_val = model(data_val).float()
        output_val_exp_map = manifold.expmap0(output_val)

        output_vectors.extend(output_val_exp_map.cpu().numpy().tolist())
        gt_labels.extend(target_val.cpu().numpy().tolist())

output_vectors = np.array(output_vectors)
gt_labels = np.array(gt_labels).reshape(-1, 1)
# Let's select a subset of the data!
subset_index = np.random.choice(np.array(range(0,gt_labels.shape[0])), 1000,␣
↪replace=False)

output_vectors_small = output_vectors[subset_index, :]
gt_labels_small = gt_labels[subset_index]
```

[17]:
```
cmap = plt.get_cmap('gist_earth')
colors_list = [cmap(i) for i in np.linspace(0, 1, num_classes + 4)]

if dims == 2:
    matplotlib.pyplot.figure(figsize=(8, 8))
    ax = plt.gca()

    for i in range(num_classes):
        plt.plot(prototypes[i, 0].detach().numpy(), prototypes[i, 1].detach().numpy(),␣
↪markersize = 10,  marker = 'D', linestyle='none', color = colors_list[i])

        output_vectors_small_i = output_vectors_small[np.where(gt_labels_small == i)[0],␣
↪:]
        ax.scatter(output_vectors_small_i[:, 0], output_vectors_small_i[:, 1], color =␣
↪colors_list[i])

    ax.add_patch(plt.Circle((0, 0), 1, fill=True, ec='black', fc='#f8f9f9', linestyle='-.
↪', linewidth=1, zorder=0))

    ax.spines['bottom'].set_color('white')
    ax.spines['top'].set_color('white')
    ax.spines['right'].set_color('white')
    ax.spines['left'].set_color('white')

    ax.tick_params(
        axis='both',   # changes apply to the x-axis
        which='both',  # both major and minor ticks are affected
        bottom=False,  # ticks along the bottom edge are off
        top=False,     # ticks along the top edge are off
        left=False,
        labelbottom=False)

    # Turn off tick labels
    ax.set_yticklabels([])
    ax.set_xticklabels([])
    plt.title('Each point is the prototype learned for one classes.')
```

(continues on next page)

(continued from previous page)

```
    plt.show()
```



1.4 Clipped Hyperbolic Classifiers Are Super-Hyperbolic Classifiers

Welcome to our fourth notebook for the ECCV 2022 Tutorial "Hyperbolic Representation Learning for Computer Vision"!

Open notebook:

Author: Mina Ghadimi Atigh

In this tutorial, you will go through Clipped Hyperbolic Classifiers Are Super-Hyperbolic Classifiers, CVPR 2022 paper.

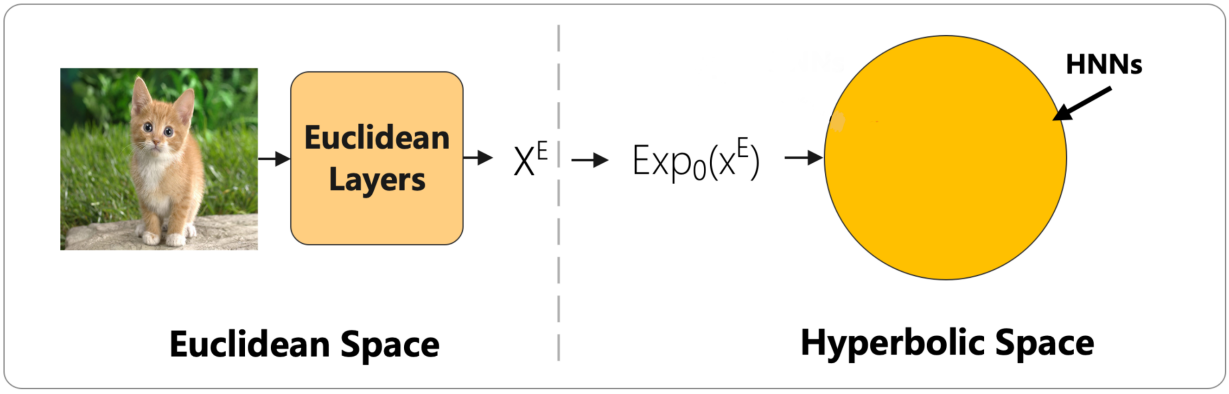
While many datasets and labels are hierarchical in nature, Euclidean space is suboptimal in learning the hierarchical structures. Hyperbolic space is a non-Euclidean space with constant negative curvature. As moving from the origin towards the boundary, the distances grow exponentially, as in hierarchical structures. When the dataset contains a known hierarchical structure, projecting the Euclidean features to hyperbolic space improves the performance. Hyperbolic neural networks. However, HNN suffers from several limitations.

HNN underperforms ENN when the dataset does not have a hierarchical structure.

There are some limitations even when the dataset contains hierarchical structure.

HNNs performance on tandard benchmarks is not investigated.

Here you can see an example of a hybrid HNN, in which the embedding of the input is extracted with an ENN as the first step. Then, the embedding is projected to the hyperbolic space using Exponential Map and loss function.



1.4.1 Vanishing gradients is the problem in hybrid HNN!

The goal is to find out why HNNs underperform ENNs in some cases. The key element may be backpropagating the **gradients** from the hyperbolic space to the Euclidean space. So let's look at how gradients are calculated.

$$\frac{\partial \ell}{\partial w^E} = (\frac{\partial x^H}{\partial w^E})^T \frac{\partial \ell}{\partial x^H} \tag{1.7}$$

where x^E and x^H are the Euclidean and Hyperbolic embeddings of the input x. $\frac{\partial \ell}{\partial x^H}$ in the gradient of the loss function ℓ with respect to the hyperbolic embedding x^H, which is the Riemannian gradient,

$$\frac{\partial \ell}{\partial x^H} = \frac{(1 - \|x^H\|^2)^2}{4} \nabla \ell(x^H) \tag{1.8}$$

where $\nabla \ell(x^H)$ is the Euclidean gradient.

During training and learning embeddings in hyperbolic space, the points move to the boundary of the hyperbolic space, resulting in $\|x^H\|^2$ getting close to 1. Consequently, the gradient diminishes during training, which indicates vanishing gradients as a problem occurring during the training of a hybrid HNN.

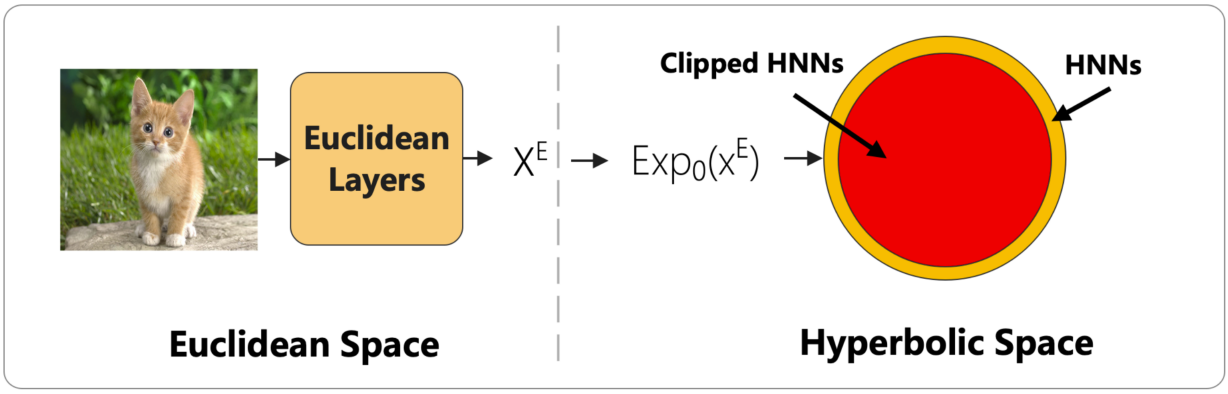
1.4.2 How can we avoid vanishing gradients?

Since approaching the boundary of hyperbolic space causes problems in training, one solution may be to impose a soft regularization on the Euclidean embedding before the exponential map so that the norm of the embedding does not exceed a certain threshold.

$$CLIP(x^E; r) = min\{1, \frac{r}{\|x^E\|}\}.x^E \tag{1.9}$$

where r is a hyperparameter. Also, clipped hyperbolic embedding will be $CLIP(x^H; r) = Exp_0^c(CLIP(x^E; r))$.

As a result of clipping, hyperbolic embeddings will not lie throughout the hyperbolic space. Here you can see the clipped hybrid HNN.



Here, you can find out the implementation of clipping, applied on top of Euclidean embedding. `clip(input_vector, r)` gets the Euclidean embedding and r which is the hyperparameter and outputs the clipped Euclidean embedding vector.

[1]:
```
def clip(input_vector, r):
    input_norm = torch.norm(input_vector, dim = -1)
    clip_value = float(r)/input_norm
    min_norm = torch.clamp(float(r)/input_norm, max = 1)
    return min_norm[:, None] * input_vector
```

1.4.3 Clipped Hyperbolic Embeddings

To check how the clipping affects the performance of HNN, we will go through Notebook2, Hyperbolic Image Embessings. Everything is same as Notebook2 and the only difference is feeding the clipped Euclidean features to the hyperbolic space.

Let's start with importing libraries, setting manual seed and downloading *CUB-200-2011* Dataset.

[2]:
```
!wget -q https://raw.githubusercontent.com/leymir/hyperbolic-image-embeddings/
↪6633edbbeffd6d90271f0963852a046c64f407d6/examples/fewshot/networks/convnet.py
!pip install -q git+https://github.com/geoopt/geoopt.git
```

[3]:
```
## standard libraries
import numpy as np
import os
import PIL
from PIL import Image
```

(continues on next page)

(continued from previous page)

```
from PIL import ImageEnhance
import random
import warnings

## Imports for plotting
import matplotlib.pyplot as plt

## PyTorch
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader

## PyTorch Torchvision
import torchvision
from torchvision import transforms

## geoopt for hyperbolic
import geoopt

## Neural network downloaded from paper's GitHub
from convnet import ConvNet

warnings.filterwarnings('ignore')
```

[4]:
```
# Function for setting the seed
def set_seed(seed):
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed(seed)
        torch.cuda.manual_seed_all(seed)
set_seed(42)

# Ensure that all operations are deterministic on GPU (if used) for reproducibility
torch.backends.cudnn.determinstic = True
torch.backends.cudnn.benchmark = False

# Fetching the device that will be used throughout this notebook
device = torch.device("cpu") if not torch.cuda.is_available() else torch.device("cuda:0")
print("Using device", device)
```

```
Using device cuda:0
```

[5]:
```
from IPython.display import clear_output
%mkdir data
%mkdir data/cub
%cd data/cub
!wget -q -O tmp.tgz https://data.caltech.edu/records/65de6-vp158/files/CUB_200_2011.tgz?
↪download=1
!tar -xzvf tmp.tgz
!rm tmp.tgz
```

(continues on next page)

(continued from previous page)

```
!rm attributes.txt
%mkdir images
!mv CUB_200_2011/images/*/*.jpg images
!mv CUB_200_2011/classes.txt classes.txt
!rm -r CUB_200_2011
clear_output()

%mkdir split
%cd split
![ ! -f test.csv ] && wget -q https://raw.githubusercontent.com/leymir/hyperbolic-image-
↪embeddings/master/examples/fewshot/data/cub/split/test.csv
![ ! -f val.csv ] && wget -q https://raw.githubusercontent.com/leymir/hyperbolic-image-
↪embeddings/master/examples/fewshot/data/cub/split/val.csv
![ ! -f train.csv ] && wget -q https://raw.githubusercontent.com/leymir/hyperbolic-image-
↪embeddings/master/examples/fewshot/data/cub/split/train.csv
%cd ../../..
```

```
mkdir: cannot create directory ‘split’: File exists
/home/mina/workspace/hyperbolic_representation_learning/notebooks/data/cub/split
/home/mina/workspace/hyperbolic_representation_learning/notebooks
```

The next step is to handle data loader functions.

[5]:
```
# ROOT to the images and split, data files are available here.
ROOT_PATH = os.getcwd()
DATA_PATH = os.path.join(ROOT_PATH, "data/cub/")
IMAGE_PATH = os.path.join(DATA_PATH, "images")
SPLIT_PATH = os.path.join(DATA_PATH, "split")

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
↪~~~~~~~~~~~~~~~
transformtypedict = dict(
    Brightness=ImageEnhance.Brightness,
    Contrast=ImageEnhance.Contrast,
    Sharpness=ImageEnhance.Sharpness,
    Color=ImageEnhance.Color,
)


class ImageJitter(object):
    def __init__(self, transformdict):
        self.transforms = [(transformtypedict[k], transformdict[k]) for k in␣
↪transformdict]

    def __call__(self, img):
        out = img
        randtensor = torch.rand(len(self.transforms))
        for i, (transformer, alpha) in enumerate(self.transforms):
            r = alpha * (randtensor[i] * 2.0 - 1.0) + 1
            out = transformer(out).enhance(r).convert("RGB")
        return out
```

(continues on next page)

(continued from previous page)

```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
↪~~~~~~~~~~~~~~~

# This is for the CUB dataset, which does not support the ResNet encoder now
# It is notable, we assume the cub images are cropped based on the given bounding boxes
# The concept labels are based on the attribute value, which are for further use (and␣
↪not used in this work)
class CUB(Dataset):
    def __init__(self, setname):
        txt_path = os.path.join(SPLIT_PATH, setname + ".csv")
        lines = [x.strip() for x in open(txt_path, "r").readlines()][1:]
        data = []
        label = []
        lb = -1
        self.wnids = []

        for l in lines:
            context = l.split(",")
            name = context[0]
            wnid = context[1]
            path = os.path.join(IMAGE_PATH, name)
            if wnid not in self.wnids:
                self.wnids.append(wnid)
                lb += 1

            data.append(path)
            label.append(lb)

        self.data = data
        self.label = label
        self.num_class = np.unique(np.array(label)).shape[0]

        normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,␣
↪0.225])

        if setname == "train":
            self.transform = transforms.Compose([transforms.RandomResizedCrop(84),
                                                 ImageJitter(dict(Brightness=0.4,␣
↪Contrast=0.4, Color=0.4)),
                                                 transforms.RandomHorizontalFlip(),
                                                 transforms.ToTensor(),
                                                 transforms.Normalize(np.array([0.485, 0.
↪456, 0.406]),
                                                                      np.array([0.229, 0.
↪224, 0.225])),
                                                 ])

        else:
            self.transform = transforms.Compose([transforms.Resize(84, interpolation=PIL.
↪Image.BICUBIC),
                                                 transforms.CenterCrop(84),
                                                 transforms.ToTensor(),
```

(continues on next page)

(continued from previous page)

```
                                                normalize, ])

    def __len__(self):
        return len(self.data)

    def __getitem__(self, i):
        path, label = self.data[i], self.label[i]
        x = Image.open(path).convert("RGB")
        image = self.transform(Image.open(path).convert("RGB"))
        return image, label


# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
↪~~~~~~~~~~~~~~
class CategoriesSampler:
    def __init__(self, label, n_batch, n_cls, n_per):
        self.n_batch = n_batch
        self.n_cls = n_cls
        self.n_per = n_per

        label = np.array(label)
        self.m_ind = []
        for i in range(max(label) + 1):
            ind = np.argwhere(label == i).reshape(-1)
            ind = torch.from_numpy(ind)
            self.m_ind.append(ind)

    def __len__(self):
        return self.n_batch

    def __iter__(self):
        for i_batch in range(self.n_batch):
            batch = []
            classes = torch.randperm(len(self.m_ind))[: self.n_cls]
            for c in classes:
                l = self.m_ind[c]
                pos = torch.randperm(len(l))[: self.n_per]
                batch.append(l[pos])
            batch = torch.stack(batch).t().reshape(-1)
            yield batch
```

Exctract class names of the CUB dataset.

[6]:
```
DATA_PATH
```

[6]:
```
'/home/mina/workspace/hyperbolic_representation_learning/notebooks/data/cub/'
```

[7]:
```
classname_file = os.path.join(DATA_PATH, 'classes.txt')
class_dict = {}
with open(classname_file) as f:
    lines = f.readlines()
```

(continues on next page)

(continued from previous page)

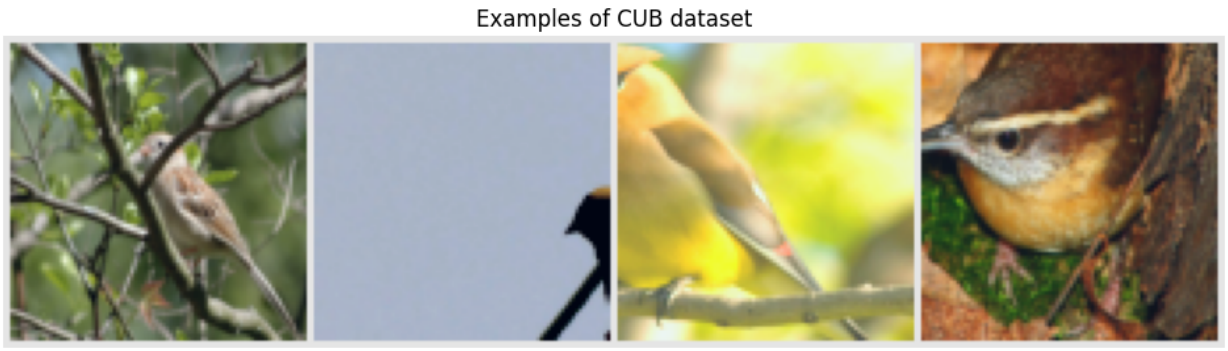
```
    for i, line in enumerate(lines):
        class_num = int(line.split(' ')[0])
        class_dict[class_num - 1] = line.split('.')[1].split('\n')[0].replace('_', ' ')
```

Before starting the main task, let's check how images from the dataset look like.

[8]:
```
# define a sample trainset
sample_trainset = CUB("train")
```

[9]:
```
# number of images to plot
NUM_IMAGES = 4
# Index of images to plot
indexes = random.sample(range(len(sample_trainset)), NUM_IMAGES)
images = [sample_trainset[idx][0] for idx in indexes]
labels = [sample_trainset[idx][1] for idx in indexes]
```

[10]:
```
img_grid = torchvision.utils.make_grid(images, nrow=8, normalize=True, pad_value=0.9)
img_grid = img_grid.permute(1, 2, 0)

plt.figure(figsize=(12,8))
plt.title("Examples of CUB dataset")
plt.imshow(img_grid)
plt.axis('off')
plt.show()
plt.close()
print('  |  '.join(f'{class_dict[labels[j]]:16s}' for j in range(NUM_IMAGES)))
```




```
Boat tailed Grackle  |  Brewer Blackbird  |  Brown Creeper     |  Clark Nutcracker
```

1.4.4 Clipped Few Shot learning in hyperbolic space

First, we set the value of hyperparameters, assigned using `Config`. The hyperparameters are set for 5-way 1-shot task (same as Notebook 2).

Note that we train the model for 5 epochs, but to reproduce the final paper numbers, the model should be trained more.

What's new?

Two values are added to the `Config` to perform clipping. *First*, `do_clip` is used to show whether to use clipping or not. When `do_clip` is `False`, model will be same as Notebook2. *Second*, `r` is the hyperparameter used in $CLIP(x^E, r)$.

[26]:
```
class Config:
    def __init__(self, shot=1, lr=0.001, step=50, gamma=0.8, c=0.05, model="convnet",␣
↪hyperbolic = True, dim=1600,
                 query=15, way=5, validation_way=5, temperature=1, dataset="CUB", lr_
↪decay=True, max_epoch=200,
                 do_clip = True, r = 1.0):

        self.lr = lr                                    # learning rate
        self.lr_decay = lr_decay                        # Boolean, whether to perform␣
↪learning rate scheduler or not
        self.step_size=step                             # Period of learning rate decay
        self.gamma = gamma                              # Multiplicative factor of␣
↪learning rate decay
        self.dataset= dataset                           # Dataset name
        self.model = "convnet"                          # Name of Base Model
        self.temperature = temperature                  # temperature used in calculating␣
↪logits
        self.max_epoch=max_epoch                        # number of epochs

        self.c = c                                      # Curvature of the hyperbolic space
        self.hyperbolic = hyperbolic                    # Boolean, if it is hyperbolic or␣
↪not
        self.dim = dim                                  # dimenionality of the output␣
↪vector

        self.shot = shot                                # Number of shots in Few shot␣
↪learning task
        self.query = query                              #
        self.way = way                                  # Number of ways in Few shot␣
↪learning task
        self.validation_way = validation_way            # Number of ways in Few shot␣
↪learning task for validation set

        self.do_clip = do_clip                          # Boolean, whether to perform␣
↪clipping or not
        self.r = r                                      # float, hyperparameter used in␣
↪CLIP(xE, r)


args = Config(dim=512, max_epoch = 5, do_clip = True, r = 1.5)
```

Split training and validation set based on the `Config` for few-shot learning.

[12]:
```
trainset = CUB("train")
train_sampler = CategoriesSampler(trainset.label, 100, args.way, args.shot + args.query)
train_loader = DataLoader(dataset=trainset, batch_sampler=train_sampler, num_workers=8,␣
↪pin_memory=True)

valset = CUB("val")
val_sampler = CategoriesSampler(valset.label, 500, args.validation_way, args.shot + args.
↪query)
val_loader = DataLoader(dataset=valset, batch_sampler=val_sampler, num_workers=8, pin_
↪memory=True)
```

[13]:
```
def lorenz_factor(x, *, c=1.0, dim=-1, keepdim=False):
    """
    Parameters
    ----------
    x : tensor
        point on Klein disk
    c : float
        negative curvature
    dim : int
        dimension to calculate Lorenz factor
    keepdim : bool
        retain the last dim? (default: false)

    Returns
    -------
    tensor
        Lorenz factor
    """
    return 1 / torch.sqrt(1 - c * x.pow(2).sum(dim=dim, keepdim=keepdim))

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Project a point from Klein model to Poincare model
def k2p(x, c):
    denom = 1 + torch.sqrt(1 - c * x.pow(2).sum(-1, keepdim=True))
    return x / denom

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Project a point from Poincare model to Klein model
def p2k(x, c):
    denom = 1 + c * x.pow(2).sum(-1, keepdim=True)
    return 2 * x / denom

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

def poincare_mean(x, dim=0, c=1.0):
    # To calculate the mean, another model of hyperbolic space named Klein model is used.
    # 1. point is projected from Poincare model to Klein model using p2k, output x is a␣
↪point in Klein model
    x = p2k(x, c)
    # 2. mean is calculated
    lamb = lorenz_factor(x, c=c, keepdim=True)
    mean = torch.sum(lamb * x, dim=dim, keepdim=True) / torch.sum(lamb, dim=dim,␣
↪keepdim=True)
    # 3. Mean is projected from Klein model to Poincare model
    mean = k2p(mean, c)
    return mean.squeeze(dim)
```

What is the main change on `ProtoNet`?

The main difference between the *clipped* and *normal* `ProtoNet` lies in the input of the `exp_map`. Therefore, when `args.do_clip` is `True`, everything should be clipped using the `clip`. This results ends in clipping the output of the `self.encoder`, which is the Euclidean embedding.

[27]:
```
class ProtoNet(nn.Module):
    def __init__(self, args):
        super().__init__()
        self.args = args

        # Base Model: ConvNet
        self.encoder = ConvNet(z_dim=args.dim)

        # If working in Hyperbolic Space
        if args.hyperbolic:
            self.manifold = geoopt.PoincareBall(c=args.c)

    def forward(self, data_shot, data_query):
        # 1. feed data to the model
        proto = self.encoder(data_shot)


        # Hyperbolic Space:
        if self.args.hyperbolic:
            ###################### CLIP ######################
            if args.do_clip:
                proto = clip(proto, args.r)
            ##################################################

            # 2. encoder is Euclidean, output should be projected to Hyperboolic space␣
↪using exponential map
            proto = self.manifold.expmap0(proto)

            if self.training:
                proto = proto.reshape(self.args.shot, self.args.way, -1)
            else:
                proto = proto.reshape(self.args.shot, self.args.validation_way, -1)

            # 3. calculate prototypes based on mean of data
            proto = poincare_mean(proto, dim=0, c=self.manifold.c.item())

            # 4. query if projected to hyperbolic space too
            data_query = self.encoder(data_query)
            ###################### CLIP ######################
            if args.do_clip:
                data_query = clip(data_query, args.r)
            ##################################################
            data_query = self.manifold.expmap0(data_query)

            # 5. Logits is calculated based on the Hyperbolic distance between data␣
↪query and proto
            logits = (-self.manifold.dist(data_query[:, None, :], proto) / self.args.
↪temperature)
```

(continues on next page)

(continued from previous page)

```
        # Euclidean Space
        else:
            # 2. calculate prototypes based on mean of data
            if self.training:
                proto = proto.reshape(self.args.shot, self.args.way, -1).mean(dim=0)
            else:
                proto = proto.reshape(self.args.shot, self.args.validation_way, -1).
↪mean(dim=0)

            # 3. Logits is calculated based on the Euclidean distance between data query␣
↪and proto
            logits = (((self.encoder(data_query)[:, None, :] - proto)**2).sum(dim=-1) /␣
↪self.args.temperature)
        return logits


model = ProtoNet(args).to(device)
```

Once the model is defined, it's time for the training loop. Let's first define *optimizer* and *learning rate scheduler*.

[28]:
```
optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)

if args.lr_decay:
    lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=args.step_size,␣
↪gamma=args.gamma)
```

[16]:
```
# Function to calculate the accuracy given logits and groundtruth labels
def count_acc(logits, label):
    pred = torch.argmax(logits, dim=1)
    if torch.cuda.is_available():
        return (pred == label).type(torch.cuda.FloatTensor).mean().item()
    else:
        return (pred == label).type(torch.FloatTensor).mean().item()
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Class defined to average the values passed to it
class Averager:
    def __init__(self):
        self.n = 0
        self.v = 0

    def add(self, x):
        self.v = (self.v * self.n + x) / (self.n + 1)
        self.n += 1

    def item(self):
        return self.v
```

Training loop

Next step is training the model.

[29]:
```
for epoch in range(1, args.max_epoch + 1):
    if args.lr_decay:
        lr_scheduler.step()
    model.train()

    tl = Averager()
    ta = Averager()

    label = torch.arange(args.way).repeat(args.query)
    if torch.cuda.is_available():
        label = label.type(torch.cuda.LongTensor)
    else:
        label = label.type(torch.LongTensor)

    for i, batch in enumerate(train_loader, 1):
        if torch.cuda.is_available():
            data, _ = [_.cuda() for _ in batch]
        else:
            data = batch[0]
        p = args.shot * args.way
        data_shot, data_query = data[:p], data[p:]
        logits = model(data_shot, data_query)
        loss = F.cross_entropy(logits, label)
        acc = count_acc(logits, label)

        if (i == 1) or (i == len(train_loader)):
            print("epoch {}, train {}/{}, loss={:.4f} acc={:.4f}".format(epoch, i,␣
↪len(train_loader), loss.item(), acc))

        tl.add(loss.item())
        ta.add(acc)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    tl = tl.item()
    ta = ta.item()

    model.eval()

    vl = Averager()
    va = Averager()

    label = torch.arange(args.validation_way).repeat(args.query)
    if torch.cuda.is_available():
        label = label.type(torch.cuda.LongTensor)
    else:
        label = label.type(torch.LongTensor)
```

(continues on next page)

(continued from previous page)

```
    with torch.no_grad():
        for i, batch in enumerate(val_loader, 1):
            if torch.cuda.is_available():
                data, _ = [_.cuda() for _ in batch]
            else:
                data = batch[0]
            p = args.shot * args.validation_way
            data_shot, data_query = data[:p], data[p:]
            logits = model(data_shot, data_query)
            loss = F.cross_entropy(logits, label)
            acc = count_acc(logits, label)

            vl.add(loss.item())
            va.add(acc)

    vl = vl.item()
    va = va.item()
    print("epoch {}, val, loss={:.4f} acc={:.4f}".format(epoch, vl, va))
```

```
epoch 1, train 1/100, loss=1.5794 acc=0.2933
epoch 1, train 100/100, loss=1.6932 acc=0.2133
epoch 1, val, loss=1.4254 acc=0.3972
epoch 2, train 1/100, loss=1.2952 acc=0.4933
epoch 2, train 100/100, loss=1.6063 acc=0.2400
epoch 2, val, loss=1.4004 acc=0.4254
epoch 3, train 1/100, loss=1.4594 acc=0.4000
epoch 3, train 100/100, loss=1.6920 acc=0.2133
epoch 3, val, loss=1.3755 acc=0.4182
epoch 4, train 1/100, loss=1.5255 acc=0.3867
epoch 4, train 100/100, loss=1.6323 acc=0.3067
epoch 4, val, loss=1.3500 acc=0.4370
epoch 5, train 1/100, loss=1.3717 acc=0.4800
epoch 5, train 100/100, loss=1.1023 acc=0.6667
epoch 5, val, loss=1.3338 acc=0.4510
```

Testing function

Once the training is complete, it's time to see model's performance on the test split.

First, let's create test dataloader.

[18]:
```
def compute_confidence_interval(data):
    """
    Compute 95% confidence interval
    :param data: An array of mean accuracy (or mAP) across a number of sampled episodes.
    :return: the 95% confidence interval for this data.
    """
    a = 1.0 * np.array(data)
    m = np.mean(a)
    std = np.std(a)
```

(continues on next page)

(continued from previous page)

```
    pm = 1.96 * (std / np.sqrt(len(a)))
    return m, pm
```

[19]:
```
test_set = CUB("test")
test_sampler = CategoriesSampler(test_set.label, 10000, args.validation_way, args.shot +␣
↪args.query)
test_loader = DataLoader(test_set, batch_sampler=test_sampler, num_workers=8, pin_
↪memory=True)
```

[20]:
```
model.eval()

ave_acc = Averager()
test_acc_record = np.zeros((10000,))

label = torch.arange(args.validation_way).repeat(args.query)
if torch.cuda.is_available():
    label = label.type(torch.cuda.LongTensor)
else:
    label = label.type(torch.LongTensor)

# Testing loop
for i, batch in enumerate(test_loader, 1):
    if torch.cuda.is_available():
        data, _ = [_.cuda() for _ in batch]
    else:
        data = batch[0]
    k = args.validation_way * args.shot
    data_shot, data_query = data[:k], data[k:]

    logits = model(data_shot, data_query)
    acc = count_acc(logits, label)
    ave_acc.add(acc)
    test_acc_record[i - 1] = acc

m, pm = compute_confidence_interval(test_acc_record)
print("Test Acc {:.4f} + {:.4f}".format(m, pm))
```

```
Test Acc 0.4033 + 0.0020
```

[30]:
```
model.eval()

ave_acc = Averager()
test_acc_record = np.zeros((10000,))

label = torch.arange(args.validation_way).repeat(args.query)
if torch.cuda.is_available():
    label = label.type(torch.cuda.LongTensor)
else:
    label = label.type(torch.LongTensor)

# Testing loop
for i, batch in enumerate(test_loader, 1):
```

(continues on next page)

(continued from previous page)

```
    if torch.cuda.is_available():
        data, _ = [_.cuda() for _ in batch]
    else:
        data = batch[0]
    k = args.validation_way * args.shot
    data_shot, data_query = data[:k], data[k:]

    logits = model(data_shot, data_query)
    acc = count_acc(logits, label)
    ave_acc.add(acc)
    test_acc_record[i - 1] = acc

m, pm = compute_confidence_interval(test_acc_record)
print("Test Acc {:.4f} + {:.4f}".format(m, pm))
```

```
Test Acc 0.4160 + 0.0021
```

When training `ProtoNet` with hyperparameter `r = 1.5`, the performance increases from 40.33 to 41.60.