

linear_probing_with_pretraining

April 5, 2024

```
[1]: !pip install einops
```

Collecting einops

Downloading einops-0.7.0-py3-none-any.whl.metadata (13 kB)

Downloading einops-0.7.0-py3-none-any.whl (44 kB)

44.6/44.6 kB

1.8 MB/s eta 0:00:00

Installing collected packages: einops

Successfully installed einops-0.7.0

```
[14]: import os
import gc
import torch
import torch.nn as nn
import torch.nn.functional as F
import pandas as pd
import numpy as np
import h5py
import copy
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader, TensorDataset, \
↳ SubsetRandomSampler, ConcatDataset
from torchvision import transforms, utils, datasets
from torchmetrics import Accuracy
from sklearn.model_selection import KFold, StratifiedKFold, train_test_split
from torchvision.datasets import ImageFolder
from PIL import Image
import cv2
import pyarrow.parquet as pq
import seaborn as sns
from tqdm import tqdm
from statistics import mean
from sklearn.metrics import accuracy_score, roc_auc_score
from sklearn.preprocessing import StandardScaler
import csv
import torchvision
import ctypes
import torch.optim as optim
```

```

from torch.optim import Adam
from functools import partial
from einops import repeat, rearrange
from einops.layers.torch import Rearrange
from timm.models.vision_transformer import PatchEmbed, Block
from torch.optim import AdamW
from torch.optim.lr_scheduler import CosineAnnealingWarmRestarts, \
    ↪CosineAnnealingLR, StepLR, ReduceLROnPlateau
from torch.cuda.amp import autocast, GradScaler
from transformers import AutoModel, AutoTokenizer
from torch.utils.data.sampler import BatchSampler, Sampler
from skimage import io, transform
from torch.nn.utils import clip_grad_norm_

torch.manual_seed(42)
np.random.seed(42)
torch.cuda.manual_seed(42)

import warnings
warnings.filterwarnings("ignore")

```

```

[3]: with h5py.File('/kaggle/input/mae-labeled/Dataset_Specific_labelled.h5', 'r') \
    ↪as file:

```

```

print("Groups in the HDF5 file:")
for group in file:
    print(group)

dataset = file['jet']
print("Dataset shape:", dataset.shape)
print("Dataset dtype:", dataset.dtype)

dataset = file['Y']
print("Dataset shape:", dataset.shape)
print("Dataset dtype:", dataset.dtype)

print("Dataset attributes:")
for attr_name, attr_value in dataset.attrs.items():
    print(f"{attr_name}: {attr_value}")

X = np.array(file['jet'][:])
Y = np.array(file['Y'][:])

```

Groups in the HDF5 file:

Y

jet

Dataset shape: (10000, 125, 125, 8)

```
Dataset dtype: float32
Dataset shape: (10000, 1)
Dataset dtype: float32
Dataset attributes:
```

```
[4]: X.shape
```

```
[4]: (10000, 125, 125, 8)
```

```
[5]: def get_2d_sincos_pos_embed(embed_dim, grid_size, cls_token=False):

    grid_h = np.arange(grid_size, dtype=np.float32)
    grid_w = np.arange(grid_size, dtype=np.float32)
    grid = np.meshgrid(grid_w, grid_h) # here w goes first
    grid = np.stack(grid, axis=0)

    grid = grid.reshape([2, 1, grid_size, grid_size])
    pos_embed = get_2d_sincos_pos_embed_from_grid(embed_dim, grid)
    if cls_token:
        pos_embed = np.concatenate([np.zeros([1, embed_dim]), pos_embed],
        ↪axis=0)
    return pos_embed

def get_2d_sincos_pos_embed_from_grid(embed_dim, grid):
    assert embed_dim % 2 == 0

    emb_h = get_1d_sincos_pos_embed_from_grid(embed_dim // 2, grid[0]) # (H*W, ↪
    ↪D/2)
    emb_w = get_1d_sincos_pos_embed_from_grid(embed_dim // 2, grid[1]) # (H*W, ↪
    ↪D/2)

    emb = np.concatenate([emb_h, emb_w], axis=1) # (H*W, D)
    return emb

def get_1d_sincos_pos_embed_from_grid(embed_dim, pos):

    assert embed_dim % 2 == 0
    omega = np.arange(embed_dim // 2, dtype='float32')
    omega /= embed_dim / 2.
    omega = 1. / 10000**omega # (D/2,)

    pos = pos.reshape(-1) # (M,)
    out = np.einsum('m,d->md', pos, omega) # (M, D/2), outer product

    emb_sin = np.sin(out) # (M, D/2)
```

```

emb_cos = np.cos(out) # (M, D/2)

emb = np.concatenate([emb_sin, emb_cos], axis=1) # (M, D)
return emb

```

```

[6]: class Encoder(nn.Module):
    def __init__(self, img_size=224, patch_size=16, in_chans=8,
                  embed_dim=1024, depth=24, num_heads=16,
                  decoder_embed_dim=512, decoder_depth=8, decoder_num_heads=16,
                  mlp_ratio=4., norm_layer=nn.LayerNorm, norm_pix_loss=False):
        super().__init__()

        self.mask_ratio = 0.75
        self.patch_embed = PatchEmbed(img_size, patch_size, in_chans, embed_dim)
        num_patches = self.patch_embed.num_patches
        self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))
        self.pos_embed = nn.Parameter(torch.zeros(1, num_patches + 1,
↪ embed_dim), requires_grad=False) # fixed sin-cos embedding
        self.blocks = nn.ModuleList([
            Block(embed_dim, num_heads, mlp_ratio, qkv_bias=True,
↪ norm_layer=norm_layer)
            for i in range(depth)])
        self.norm = norm_layer(embed_dim)
        self.initialize_weights()

    def initialize_weights(self):
        pos_embed = get_2d_sincos_pos_embed(self.pos_embed.shape[-1], int(self.
↪ patch_embed.num_patches**.5), cls_token=True)
        self.pos_embed.data.copy_(torch.from_numpy(pos_embed).float().
↪ unsqueeze(0))
        w = self.patch_embed.proj.weight.data
        torch.nn.init.xavier_uniform_(w.view([w.shape[0], -1]))
        torch.nn.init.normal_(self.cls_token, std=.02)
        self.apply(self._init_weights)

    def _init_weights(self, m):
        if isinstance(m, nn.Linear):
            torch.nn.init.xavier_uniform_(m.weight)
            if isinstance(m, nn.Linear) and m.bias is not None:
                nn.init.constant_(m.bias, 0)
        elif isinstance(m, nn.LayerNorm):
            nn.init.constant_(m.bias, 0)
            nn.init.constant_(m.weight, 1.0)

    def patchify(self, imgs):
        p = self.patch_embed.patch_size[0]
        assert imgs.shape[2] == imgs.shape[3] and imgs.shape[2] % p == 0

```

```

h = w = imgs.shape[2] // p
x = imgs.reshape(shape=(imgs.shape[0], 8, h, p, w, p))
x = torch.einsum('nchpwq->nhwpqc', x)
x = x.reshape(shape=(imgs.shape[0], h * w, p**2 * 8))
return x

def unpatchify(self, x):
    p = self.patch_embed.patch_size[0]
    h = w = int(x.shape[1]**.5)
    assert h * w == x.shape[1]
    x = x.reshape(shape=(x.shape[0], h, w, p, p, 8))
    x = torch.einsum('nhwpqc->nchpwq', x)
    imgs = x.reshape(shape=(x.shape[0], 8, h * p, h * p))
    return imgs

def random_masking(self, x, mask_ratio):
    N, L, D = x.shape # batch, length, dim
    len_keep = int(L * (1 - mask_ratio))
    noise = torch.rand(N, L, device=x.device)
    ids_shuffle = torch.argsort(noise, dim=1) # ascend: small is keep,
    ↪ large is remove
    ids_restore = torch.argsort(ids_shuffle, dim=1)
    ids_keep = ids_shuffle[:, :len_keep]
    x_masked = torch.gather(x, dim=1, index=ids_keep.unsqueeze(-1).
    ↪ repeat(1, 1, D))
    mask = torch.ones([N, L], device=x.device)
    mask[:, :len_keep] = 0
    mask = torch.gather(mask, dim=1, index=ids_restore)
    return x_masked, mask, ids_restore

def forward(self, x):
    imgs = self.patchify(x)
    x = self.patch_embed(x)
    x = x + self.pos_embed[:, 1:, :]
    x, mask, ids_restore = self.random_masking(x, self.mask_ratio)
    cls_token = self.cls_token + self.pos_embed[:, :1, :]
    cls_tokens = cls_token.expand(x.shape[0], -1, -1)
    x = torch.cat((cls_tokens, x), dim=1)
    for blk in self.blocks:
        x = blk(x)
    x = self.norm(x)
    return x, mask, ids_restore, imgs

class Decoder(nn.Module):
    def __init__(self, img_size=224, patch_size=16, in_chans=8,
        embed_dim=1024, depth=24, num_heads=16,

```

```

        decoder_embed_dim=512, decoder_depth=8, decoder_num_heads=16,
        mlp_ratio=4., norm_layer=nn.LayerNorm, norm_pix_loss=False):
    super().__init__()

    self.num_patches = (img_size//patch_size)**2
    self.decoder_embed = nn.Linear(embed_dim, decoder_embed_dim, bias=True)
    self.mask_token = nn.Parameter(torch.zeros(1, 1, decoder_embed_dim))
    self.decoder_pos_embed = nn.Parameter(torch.zeros(1, self.num_patches +
↪1, decoder_embed_dim), requires_grad=False) # fixed sin-cos embedding
    self.decoder_blocks = nn.ModuleList([
        Block(decoder_embed_dim, decoder_num_heads, mlp_ratio,
↪qkv_bias=True, norm_layer=norm_layer)
        for i in range(decoder_depth)])

    self.decoder_norm = norm_layer(decoder_embed_dim)
    self.decoder_pred = nn.Linear(decoder_embed_dim, patch_size**2 *
↪in_chans, bias=True) # decoder to patch
    self.norm_pix_loss = norm_pix_loss

    self.initialize_weights()

    def initialize_weights(self):
        decoder_pos_embed = get_2d_sincos_pos_embed(self.decoder_pos_embed.
↪shape[-1], int(self.num_patches**.5), cls_token=True)
        self.decoder_pos_embed.data.copy_(torch.from_numpy(decoder_pos_embed).
↪float()).unsqueeze(0))
        torch.nn.init.normal_(self.mask_token, std=.02)
        self.apply(self._init_weights)

    def _init_weights(self, m):
        if isinstance(m, nn.Linear):
            torch.nn.init.xavier_uniform_(m.weight)
            if isinstance(m, nn.Linear) and m.bias is not None:
                nn.init.constant_(m.bias, 0)
        elif isinstance(m, nn.LayerNorm):
            nn.init.constant_(m.bias, 0)
            nn.init.constant_(m.weight, 1.0)

    def patchify(self, imgs):
        p = self.patch_embed.patch_size[0]
        assert imgs.shape[2] == imgs.shape[3] and imgs.shape[2] % p == 0
        h = w = imgs.shape[2] // p
        x = imgs.reshape(shape=(imgs.shape[0], 8, h, p, w, p))
        x = torch.einsum('nchpwq->nhwpqc', x)
        x = x.reshape(shape=(imgs.shape[0], h * w, p**2 * 8))
        return x

```

```

def unpatchify(self, x):
    p = self.patch_embed.patch_size[0]
    h = w = int(x.shape[1]**.5)
    assert h * w == x.shape[1]
    x = x.reshape(shape=(x.shape[0], h, w, p, p, 8))
    x = torch.einsum('nhwpqc->nchpwq', x)
    imgs = x.reshape(shape=(x.shape[0], 8, h * p, h * p))
    return imgs

def forward(self, x, ids_restore):
    x = self.decoder_embed(x)
    mask_tokens = self.mask_token.repeat(x.shape[0], ids_restore.shape[1] +
↪1 - x.shape[1], 1)
    x_ = torch.cat([x[:, 1:, :], mask_tokens], dim=1) # no cls token
    x_ = torch.gather(x_, dim=1, index=ids_restore.unsqueeze(-1).repeat(1,
↪1, x.shape[2])) # unshuffle
    x = torch.cat([x[:, :1, :], x_], dim=1)
    x = x + self.decoder_pos_embed
    for blk in self.decoder_blocks:
        x = blk(x)
    x = self.decoder_norm(x)
    x = self.decoder_pred(x)
    x = x[:, 1:, :]
    return x

class Masked_ViT(nn.Module):
    def __init__(self, encoder, decoder, mask_ratio):
        super().__init__()

        self.encoder = encoder
        self.decoder = decoder
        self.mask_ratio = mask_ratio

    def forward(self, x):
        x, mask, ids_restore, imgs = self.encoder(x)
        pred = self.decoder(x, ids_restore)

        return imgs, pred, mask

def mae_vit_base_patch16_dec512d8b(img_size=125, mask_ratio = 0.75, **kwargs):
    encoder = Encoder(
        img_size=img_size, patch_size=5, embed_dim=768, depth=8, num_heads=12,
        decoder_embed_dim=512, decoder_depth=4, decoder_num_heads=16,
        mlp_ratio=4, norm_layer=partial(nn.LayerNorm, eps=1e-6), **kwargs)

    decoder = Decoder(
        img_size=img_size, patch_size=5, embed_dim=768, depth=8, num_heads=12,

```

```

        decoder_embed_dim=512, decoder_depth=4, decoder_num_heads=16,
        mlp_ratio=4, norm_layer=partial(nn.LayerNorm, eps=1e-6), **kwargs)

    model = Masked_VIT(encoder, decoder, mask_ratio)
    return model

model = mae_vit_base_patch16_dec512d8b(img_size=125, mask_ratio = 0.75)

```

```

[7]: class VIT_classifier(nn.Module):
    def __init__(self, encoder, num_classes):
        super().__init__()
        self.encoder = encoder
        self.patch_embed = encoder.patch_embed
        self.cls_token = encoder.cls_token
        self.pos_embed = encoder.pos_embed
        self.patchify = encoder.patchify
        self.transformer = encoder.blocks
        self.layer_norm = encoder.norm
        self.head = torch.nn.Linear(self.pos_embed.shape[-1], num_classes)
        self.blocks = encoder.blocks

        self.avg_pool = nn.AdaptiveAvgPool1d((1))
        self.flatten = nn.Flatten()
        self.fc = nn.Linear(in_features=625, out_features=64)
        self.fc_1 = nn.Linear(in_features=64, out_features=1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):

        x = self.patch_embed(x)
        x = x + self.pos_embed[:, 1:, :]
        cls_token = self.cls_token + self.pos_embed[:, :1, :]
        cls_tokens = cls_token.expand(x.shape[0], -1, -1)
        x = torch.cat((cls_tokens, x), dim=1)
        for blk in self.blocks:
            x = blk(x)
        x = self.layer_norm(x)
        x = x[:, 1:, :]
        x = self.avg_pool(x)
        x = self.flatten(x)
        x = self.fc(x)
        x = self.fc_1(x)
        x = self.sigmoid(x)

        return x

encoder = model.encoder

```



```
classifier = VIT_classifier(encoder, 2)
```

```
[8]: for _z in range(8):  
      X[:, :, :, _z] = (X[:, :, :, _z] - X[:, :, :, _z].mean()) / (X[:, :, :, _z].std())
```

```
[9]: class Custom_Dataset(Dataset):  
      def __init__(self, x, y, transform):  
          self.x = x  
          self.y = y  
          self.transform = transform  
  
      def __len__(self):  
          return self.x.shape[0]  
  
      def __getitem__(self, idx):  
          if torch.is_tensor(idx):  
              idx = idx.tolist()  
          img_1 = self.x[idx]  
          label = self.y[idx]  
  
          if self.transform:  
              img_1 = self.transform(img_1)  
          sample = {'img' : img_1, 'label' : label}  
          return sample  
  
transform = torchvision.transforms.Compose([torchvision.transforms.ToTensor()])  
dataset = Custom_Dataset(X, Y, transform = transform)  
sample = dataset.__getitem__(0)  
print((sample['img']).shape)  
print(sample['label'].shape)
```

```
torch.Size([8, 125, 125])  
(1,)
```

```
[10]: device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

```
[15]: def model_train(fold, model, epochs, train_dataloader, test_dataloader):  
  
      criterion = nn.BCELoss()  
      optimizer = optim.AdamW(model.parameters(), lr=1.5e-5)  
  
      best_acc = -np.inf  
      best_weights = None  
      accuracy = Accuracy(task = 'binary').to(device)  
  
      train_losses = []  
      val_losses = []  
      train_accuracies = []
```

```

val_accuracies = []

for epoch in range(epochs):
    train_pred = []
    val_pred = []

    model.train()
    for batch in tqdm(train_dataloader):
        images, labels = batch['img'], batch['label']
        images = images.to(device)
        labels = labels.to(device)

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        train_pred.append(loss.item())

        # Calculate training accuracy
        train_acc = accuracy(outputs, labels)
        train_accuracies.append(train_acc.item())

    train_loss = np.mean(train_pred)
    model.eval()
    with torch.no_grad():
        for val_batch in tqdm(test_dataloader):
            val_images, val_labels = val_batch['img'], val_batch['label']
            val_images = val_images.to(device)
            val_labels = val_labels.to(device)

            val_outputs = model(val_images)
            val_loss = criterion(val_outputs, val_labels)
            val_pred.append(val_loss.item())

            val_acc = accuracy(val_outputs, val_labels)
            val_accuracies.append(val_acc.item())

    val_loss = np.mean(val_pred)
    print(f'Epoch {epoch+1}/{epochs}, Train Loss: {train_loss:.4f}, Train_
↪Accuracy: {np.mean(train_accuracies):.4f}, Valid Loss: {val_loss:.4f}, Valid_
↪Accuracy: {np.mean(val_accuracies):.4f}')
    train_losses.append(train_loss)
    val_losses.append(val_loss)

    # Save best model

```

```

        if max(train_accuracies) > best_acc:
            best_acc = max(train_accuracies)
            best_weights = copy.deepcopy(model.state_dict())

    # Save the best model
    torch.save(best_weights, f'./best_model_{fold}.pth')

    return train_losses, val_losses, train_accuracies, val_accuracies

```

```

[16]: del classifier
      gc.collect()
      torch.cuda.empty_cache()

```

```

[19]: train_size = int(0.8 * len(dataset))
      val_size = len(dataset) - train_size
      train_dataset, val_dataset = torch.utils.data.random_split(dataset, [
          ↪ train_size, val_size])
      training_loss = []
      validation_loss = []
      train_dataloader = DataLoader(train_dataset, batch_size=64, shuffle=True)
      val_dataloader = DataLoader(val_dataset, batch_size=64, shuffle=False)
      classifier = VIT_classifier(model.encoder, 2)
      NUM_GPU = torch.cuda.device_count()
      if NUM_GPU > 1:
          classifier = nn.DataParallel(classifier)
      classifier = classifier.to(device)
      train_losses, val_losses, train_accs, val_accs = model_train(1, classifier, 20, [
          ↪ train_dataloader, val_dataloader])

```

```

100%|      | 125/125 [05:37<00:00,  2.70s/it]
100%|      | 32/32 [00:26<00:00,  1.19it/s]

```

Epoch 1/20, Train Loss: 0.6934, Train Accuracy: 0.4868, Valid Loss: 0.6933, Valid Accuracy: 0.4922

```

100%|      | 125/125 [05:38<00:00,  2.71s/it]
100%|      | 32/32 [00:27<00:00,  1.18it/s]

```

Epoch 2/20, Train Loss: 0.6928, Train Accuracy: 0.5047, Valid Loss: 0.6901, Valid Accuracy: 0.5776

```

100%|      | 125/125 [05:37<00:00,  2.70s/it]
100%|      | 32/32 [00:26<00:00,  1.19it/s]

```

Epoch 3/20, Train Loss: 0.6879, Train Accuracy: 0.5407, Valid Loss: 0.6941, Valid Accuracy: 0.5495

```

100%|      | 125/125 [05:36<00:00,  2.69s/it]
100%|      | 32/32 [00:26<00:00,  1.19it/s]

```

Epoch 4/20, Train Loss: 0.6901, Train Accuracy: 0.5405, Valid Loss: 0.6828,
Valid Accuracy: 0.5631

100%| | 125/125 [05:36<00:00, 2.69s/it]
100%| | 32/32 [00:26<00:00, 1.20it/s]

Epoch 5/20, Train Loss: 0.6894, Train Accuracy: 0.5393, Valid Loss: 0.6937,
Valid Accuracy: 0.5489

100%| | 125/125 [05:35<00:00, 2.68s/it]
100%| | 32/32 [00:26<00:00, 1.19it/s]

Epoch 6/20, Train Loss: 0.6928, Train Accuracy: 0.5334, Valid Loss: 0.6915,
Valid Accuracy: 0.5421

100%| | 125/125 [05:36<00:00, 2.69s/it]
100%| | 32/32 [00:26<00:00, 1.19it/s]

Epoch 7/20, Train Loss: 0.6703, Train Accuracy: 0.5483, Valid Loss: 0.6546,
Valid Accuracy: 0.5621

100%| | 125/125 [05:37<00:00, 2.70s/it]
100%| | 32/32 [00:26<00:00, 1.19it/s]

Epoch 8/20, Train Loss: 0.6446, Train Accuracy: 0.5643, Valid Loss: 0.6231,
Valid Accuracy: 0.5795

100%| | 125/125 [05:38<00:00, 2.71s/it]
100%| | 32/32 [00:26<00:00, 1.19it/s]

Epoch 9/20, Train Loss: 0.6056, Train Accuracy: 0.5814, Valid Loss: 0.5856,
Valid Accuracy: 0.5966

100%| | 125/125 [05:38<00:00, 2.71s/it]
100%| | 32/32 [00:27<00:00, 1.18it/s]

Epoch 10/20, Train Loss: 0.5959, Train Accuracy: 0.5948, Valid Loss: 0.5771,
Valid Accuracy: 0.6102

100%| | 125/125 [05:39<00:00, 2.71s/it]
100%| | 32/32 [00:27<00:00, 1.18it/s]

Epoch 11/20, Train Loss: 0.5343, Train Accuracy: 0.6106, Valid Loss: 0.5304,
Valid Accuracy: 0.6239

100%| | 125/125 [05:39<00:00, 2.71s/it]
100%| | 32/32 [00:27<00:00, 1.17it/s]

Epoch 12/20, Train Loss: 0.5016, Train Accuracy: 0.6249, Valid Loss: 0.4792,
Valid Accuracy: 0.6381

100%| | 125/125 [05:39<00:00, 2.71s/it]
100%| | 32/32 [00:26<00:00, 1.19it/s]

Epoch 13/20, Train Loss: 0.4699, Train Accuracy: 0.6382, Valid Loss: 0.4822,
Valid Accuracy: 0.6500

```

100%|      | 125/125 [05:39<00:00,  2.72s/it]
100%|      | 32/32 [00:27<00:00,  1.17it/s]

Epoch 14/20, Train Loss: 0.4370, Train Accuracy: 0.6509, Valid Loss: 0.4358,
Valid Accuracy: 0.6616

100%|      | 125/125 [05:39<00:00,  2.72s/it]
100%|      | 32/32 [00:26<00:00,  1.19it/s]

Epoch 15/20, Train Loss: 0.4160, Train Accuracy: 0.6626, Valid Loss: 0.4298,
Valid Accuracy: 0.6719

100%|      | 125/125 [05:39<00:00,  2.72s/it]
100%|      | 32/32 [00:27<00:00,  1.18it/s]

Epoch 16/20, Train Loss: 0.3873, Train Accuracy: 0.6740, Valid Loss: 0.4707,
Valid Accuracy: 0.6790

100%|      | 125/125 [05:39<00:00,  2.72s/it]
100%|      | 32/32 [00:27<00:00,  1.18it/s]

Epoch 17/20, Train Loss: 0.3706, Train Accuracy: 0.6845, Valid Loss: 0.4307,
Valid Accuracy: 0.6871

100%|      | 125/125 [05:39<00:00,  2.72s/it]
100%|      | 32/32 [00:27<00:00,  1.17it/s]

Epoch 18/20, Train Loss: 0.3548, Train Accuracy: 0.6943, Valid Loss: 0.4111,
Valid Accuracy: 0.6949

100%|      | 125/125 [05:39<00:00,  2.72s/it]
100%|      | 32/32 [00:27<00:00,  1.18it/s]

Epoch 19/20, Train Loss: 0.3353, Train Accuracy: 0.7035, Valid Loss: 0.4074,
Valid Accuracy: 0.7022

100%|      | 125/125 [05:39<00:00,  2.72s/it]
100%|      | 32/32 [00:27<00:00,  1.18it/s]

Epoch 20/20, Train Loss: 0.3123, Train Accuracy: 0.7125, Valid Loss: 0.4241,
Valid Accuracy: 0.7086

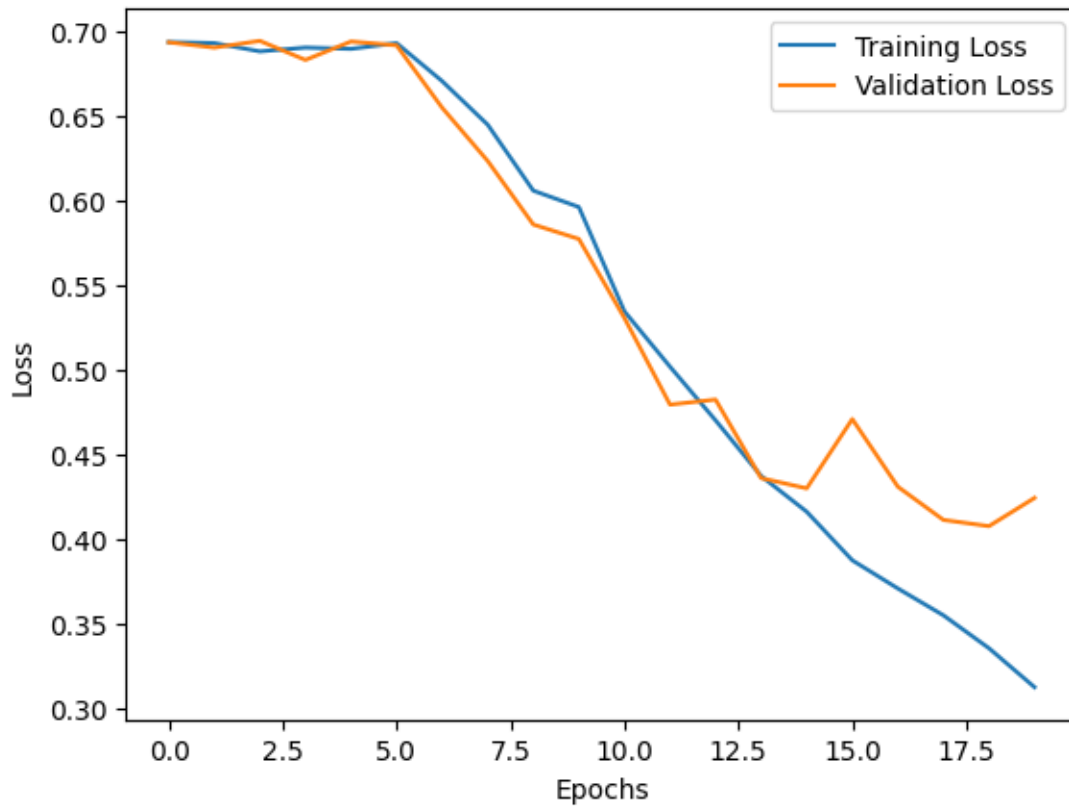
```

```
[20]: encoder = classifier.module.encoder
```

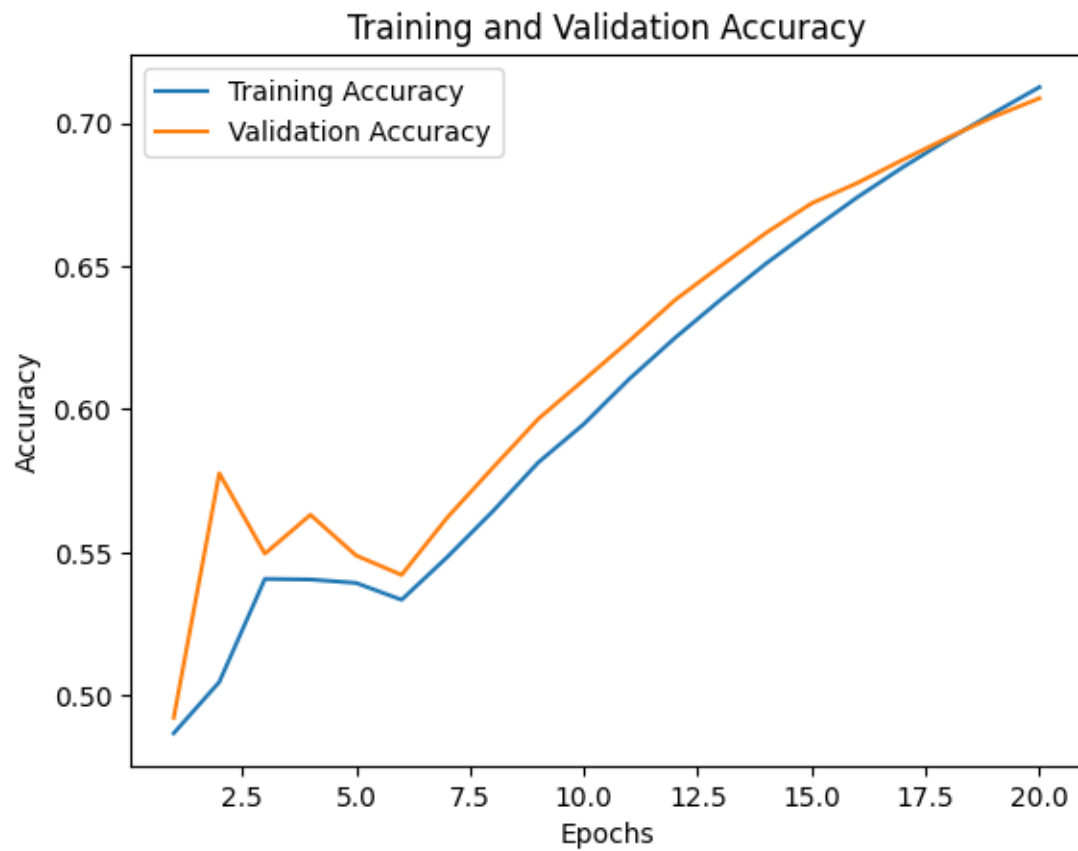
```
[21]: torch.save(encoder, 'encoder.pth')
```

```
[22]: torch.save(classifier.module, 'model.pth')
```

```
[23]: plt.plot(train_losses, label='Training Loss')
plt.plot(val_losses, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



```
[25]: plt.plot(train_accs, label='Training Accuracy')
plt.plot(val_accs, label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



[]: