Code base reference: https://github.com/01-vyom/melanoma-classification/blob/main/src/BYOL/BYOL_SIIM-ISIC_RESNET101.py

Mounting Google Drive:

```
from google.colab import drive
drive.mount('/content/drive')
```

Importing pytorch lightning:

```
!pip install pytorch_lightning -qqq
import pytorch_lightning
```

Importing kornia data augmentation library:

```
!pip install kornia
```

Importing tensorboard visualisation tool:

```
%load_ext tensorboard
```

Importing relevant libraries:

```
import random
from kornia import augmentation as aug
from kornia import filters
from kornia.geometry import transform as tf
import torch
from torch import nn, Tensor
import os
import cv2
import numpy as np
import pandas as pd
import torch
from copy import deepcopy
from typing import Dict, List, Union, Callable, Tuple

import pytorch_lightning as pl
from torch import optim
import torch.nn.functional as f

from torch.utils.data import DataLoader, Dataset, random_split
from torchvision.models import resnet101
from numpy import random
import numpy as np
from sklearn.metrics import roc_auc_score, roc_curve, confusion_matrix, accuracy_score
from torchvision import datasets, models, transforms
from pytorch_lightning.callbacks import Callback
from torch.utils.data import DataLoader, TensorDataset, Subset
from pytorch_lightning.callbacks.early_stopping import EarlyStopping
from pytorch_lightning.callbacks import LearningRateMonitor
from pytorch_lightning.callbacks.model_checkpoint import ModelCheckpoint
from torch.utils.tensorboard import SummaryWriter
import albumentations
import os.path as osp
from abc import ABC, abstractmethod
from copy import deepcopy
from dataclasses import dataclass
from os import path
from typing import Any, Dict, List, Optional, Type

from torch.optim.lr_scheduler import ReduceLROnPlateau
from torch.utils.data import random_split
from torch.utils.data.dataloader import DataLoader
from torch.utils.data.dataset import Dataset, Subset
from torchmetrics.classification.accuracy import Accuracy

from pytorch_lightning import LightningDataModule, seed_everything, Trainer
from pytorch_lightning.core.module import LightningModule
from pytorch_lightning.demos.boring_classes import Net
from pytorch_lightning.loops.fit_loop import FitLoop
from pytorch_lightning.loops.loop import Loop
from pytorch_lightning.trainer.states import TrainerFn
import matplotlib.pyplot as plt
```

```
from torchvision import datasets, models, transforms


def seed_torch(seed=42):
    random.seed(seed)
    os.environ["PYTHONHASHSEED"] = str(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    #torch.cuda.manual_seed_all(seed)  # if you are using multi-GPU.
    torch.backends.cudnn.benchmark = True
    torch.backends.cudnn.deterministic = True


seed = 42
seed_torch(seed)
```

Metric calculation and evaluation functions:

Code reference: https://github.com/01-vyom/melanoma-classification/blob/main/src/BYOL/BYOL_SIIM-ISIC_RESNET101.py

```
def pretty_per_num(num):
    return round(num * 100, 2)


def calc_metrics(pred, target, model_nm):
    auc_score = roc_auc_score(target, pred)
    fpr, tpr, thr = roc_curve(target, pred)
    gmeans = np.sqrt(tpr * (1 - fpr))
    npv = []
    for i in range(len(thr)):
        temp_thr = thr[i]
        pred_thr = [True if p > temp_thr else False for p in pred]
        tn, fp, fn, tp = confusion_matrix(target, pred_thr).ravel()
        npv.append(tn / (tn + fn + 1e-12))
    npv = np.array(npv)
    # Here whatever we want to maximize, just write that. i.e. np.argmax(npv) will maximize npv, and np.argmax(gmeans) will maximize gmea
    ix = np.argmax(gmeans)
    bestg = gmeans[ix]
    bestthr = thr[ix]
    bestnpv = npv[ix]
    val = "Best Threshold=%f, G-Mean=%.3f, NPV=%.3f" % (
        bestthr,
        pretty_per_num(bestg),
        pretty_per_num(bestnpv),
    )
    pred_thr = [True if p > bestthr else False for p in pred]
    tn, fp, fn, tp = confusion_matrix(target, pred_thr).ravel()
    sensitivity = tp / (tp + fn + 1e-12)
    specificity = tn / (tn + fp + 1e-12)
    acc = accuracy_score(target, pred_thr)
    print(
        "\nFollowing are the metrics for " + model_nm + ": AUC_Score:",
        pretty_per_num(auc_score),
        val,
        "Sensitivity:",
        pretty_per_num(sensitivity),
        "Specificity:",
        pretty_per_num(specificity),
        "Accuracy:",
        pretty_per_num(acc),
    )
```

BYOL Self-Supervised and Supervised Structure codes:

Code reference: https://github.com/01-vyom/melanoma-classification/blob/main/src/BYOL/BYOL_SIIM-ISIC_RESNET101.py

```
class RandomApply(nn.Module):
    def __init__(self, fn: Callable, p: float):
        super().__init__()
        self.fn = fn
        self.p = p

    def forward(self, x: Tensor) -> Tensor:
        return x if random.random() > self.p else self.fn(x)
```

```
#data augmentations for BYOL models:
def default_augmentation(image_size: Tuple[int, int] = (224, 224)) -> nn.Module:
    return nn.Sequential(
        tf.Resize(size=image_size),
        RandomApply(aug.ColorJitter(0.8, 0.8, 0.8, 0.2), p=0.8),
        aug.RandomGrayscale(p=0.2),
        aug.RandomHorizontalFlip(),
        aug.RandomVerticalFlip(), #modification
        aug.RandomGaussianNoise(), #modification
        aug.RandomRotation(45), #modification
        aug.RandomPlasmaContrast(), #modification
        aug.RandomBoxBlur(), #modification
        aug.CenterCrop(size=125), #modification
        RandomApply(filters.GaussianBlur2d((3, 3), (1.5, 1.5)), p=0.1), #reapply
        aug.RandomResizedCrop(size=image_size),
        aug.Normalize(
            mean=torch.tensor([0.485, 0.456, 0.406]),
            std=torch.tensor([0.229, 0.224, 0.225]),
        ),

    )

#BYOL MLP structure function:
def mlp(dim: int, projection_size: int = 256, hidden_size: int = 4096) -> nn.Module:
    return nn.Sequential(
        nn.Linear(dim, hidden_size),
        nn.BatchNorm1d(hidden_size),
        nn.ReLU(inplace=True),
        nn.Linear(hidden_size, projection_size),
    )

#BYOL encoder and projector functions:
class EncoderWrapper(nn.Module):
    def __init__(
        self,
        model: nn.Module,
        projection_size: int = 256,
        hidden_size: int = 4096,
        layer: Union[str, int] = -2,
    ):
        super().__init__()
        self.model = model
        self.projection_size = projection_size
        self.hidden_size = hidden_size
        self.layer = layer

        self._projector = None
        self._projector_dim = None
        self._encoded = torch.empty(0)
        self._register_hook()

    @property
    def projector(self):
        if self._projector is None:
            self._projector = mlp(
                self._projector_dim, self.projection_size, self.hidden_size
            )
        return self._projector

    def _hook(self, _, __, output):
        output = output.flatten(start_dim=1)
        if self._projector_dim is None:
            self._projector_dim = output.shape[-1]
        self._encoded = self.projector(output)

    def _register_hook(self):
        if isinstance(self.layer, str):
            layer = dict([*self.model.named_modules()])[self.layer]
        else:
            layer = list(self.model.children())[self.layer]

        layer.register_forward_hook(self._hook)

    def forward(self, x: Tensor) -> Tensor:
        _ = self.model(x)
        return self._encoded

#normalising mean squared error function:
def normalized_mse(x: Tensor, y: Tensor) -> Tensor:
    x = f.normalize(x, dim=-1)
    y = f.normalize(y, dim=-1)
```

```
        return 2 - 2 * (x * y).sum(dim=-1)

    #main BYOL class
    class BYOL(pl.LightningModule):
        def __init__(
            self,
            model: nn.Module,
            image_size: Tuple[int, int] = (128, 128),
            hidden_layer: Union[str, int] = -2,
            projection_size: int = 256,
            hidden_size: int = 4096,
            augment_fn: Callable = None,
            beta: float = 0.999,
            **hparams,
        ):
            super().__init__()
            self.augment = (
                default_augmentation(image_size) if augment_fn is None else augment_fn
            )
            self.beta = beta
            self.encoder = EncoderWrapper(
                model, projection_size, hidden_size, layer=hidden_layer
            )
            self.predictor = nn.Linear(projection_size, projection_size, hidden_size)
            self.hparams.update(hparams)
            self._target = None

            self.encoder(torch.zeros(2, 3, *image_size))

        def forward(self, x: Tensor) -> Tensor:
            return self.predictor(self.encoder(x))

        @property
        def target(self):
            if self._target is None:
                self._target = deepcopy(self.encoder)
            return self._target

        def update_target(self):
            for p, pt in zip(self.encoder.parameters(), self.target.parameters()):
                pt.data = self.beta * pt.data + (1 - self.beta) * p.data

        # --- Methods required for PyTorch Lightning only! ---
        #optimisations
        def configure_optimizers(self):

            optimizer = getattr(optim, self.hparams.get("optimizer", "Adam"))
            lr = self.hparams.get("lr", 1e-4)
            weight_decay = self.hparams.get("weight_decay", 1e-6)
            return optimizer(self.parameters(), lr=lr, weight_decay=weight_decay)

            ###Learning Rate scheduler experiment:
            # optimizer = optim.Adam(
            #     self.parameters(),
            #     lr = self.hparams.get("lr", 1e-4),
            #     weight_decay = self.hparams.get("weight_decay", 1e-6)
            # )

            # scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer)
            # lr_scheduler_config = {
            #     "scheduler": scheduler,
            #     "monitor": "train_loss",
            #     "interval": "step",
            #     "frequency": 1,
            # }

            # return [optimizer], [lr_scheduler_config]


        def training_step(self, batch, *_) -> Dict[str, Union[Tensor, Dict]]:
            x = batch[0]
            with torch.no_grad():
                x1, x2 = self.augment(x), self.augment(x)

            pred1, pred2 = self.forward(x1), self.forward(x2)
            with torch.no_grad():
                targ1, targ2 = self.target(x1), self.target(x2)
            loss = torch.mean(normalized_mse(pred1, targ2) + normalized_mse(pred2, targ1))

            self.log("train_loss", loss.item())
            self.update_target()
            return {"loss": loss}

        @torch.no_grad()
        def validation_step(self, batch, *_) -> Dict[str, Union[Tensor, Dict]]:
            x = batch[0]
            x1, x2 = self.augment(x), self.augment(x)
            pred1, pred2 = self.forward(x1), self.forward(x2)
            targ1, targ2 = self.target(x1), self.target(x2)
            loss = torch.mean(normalized_mse(pred1, targ2) + normalized_mse(pred2, targ1))

            return {"loss": loss}

        @torch.no_grad()
        def validation_epoch_end(self, outputs: List[Dict]) -> Dict:
            val_loss = sum(x["loss"] for x in outputs) / len(outputs)
            self.log("val_loss", val_loss.item())

    #supervised model main class:
    class SupervisedLightningModule(pl.LightningModule):
        def __init__(self, model: nn.Module, **hparams):
            super().__init__()
            self.model = model


        def forward(self, x: Tensor) -> Tensor:
            return self.model(x)

        def configure_optimizers(self):
            optimizer = getattr(optim, self.hparams.get("optimizer", "Adam"))
            lr = self.hparams.get("lr", 1e-4)
            weight_decay = self.hparams.get("weight_decay", 1e-6)
            return optimizer(self.parameters(), lr=lr, weight_decay=weight_decay)

            ###Learning Rate scheduler experiment:
            # optimizer = optim.Adam(
            #     self.parameters(),
            #     lr = self.hparams.get("lr", 1e-4),
            #     weight_decay = self.hparams.get("weight_decay", 1e-6)
            # )

            # scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer)

            # lr_scheduler_config = {
            #     "scheduler": scheduler,
            #     "monitor": "train_loss",
            #     "interval": "step",
            #     "frequency": 1,
            # }

            # return [optimizer], [lr_scheduler_config]

        def training_step(self, batch, *_) -> Dict[str, Union[Tensor, Dict]]:
            x, y = batch
            loss = f.cross_entropy(self.forward(x), y)
            self.log("train_loss", loss.item())
            return {"loss": loss}

        @torch.no_grad()
        def validation_step(self, batch, *_) -> Dict[str, Union[Tensor, Dict]]:
            x, y = batch
            loss = f.cross_entropy(self.forward(x), y)
            return {"loss": loss}

        @torch.no_grad()
        def validation_epoch_end(self, outputs: List[Dict]) -> Dict:
            val_loss = sum(x["loss"] for x in outputs) / len(outputs)
            self.log("val_loss", val_loss.item())
```

BYOL metric calculator:

Code reference: https://github.com/01-vyom/melanoma-classification/blob/main/src/BYOL/BYOL_SIIM-ISIC_RESNET101.py

```
def pretty_per_num(num):
    return round(num * 100, 5)


def metric_byol(model_nm, model, val_loader):
    model.cuda()
    preds_out = []
    preds_probs = []
    classes_out = []
    with torch.no_grad():
        for i, (inputs, classes) in enumerate(val_loader):
            inputs = inputs.cuda()
            classes = classes.cuda()
            outputs = model(inputs)
            soft_outs = outputs.softmax(1)
            # Getting probability of the 1st index to be used for getting maximum threshold
            pred_prob = soft_outs[:, 1].float().cpu().detach().numpy()
            preds_probs.extend(pred_prob)
            _, preds = torch.max(soft_outs, 1)
            preds_out.extend(preds.float().cpu().detach().numpy())
            classes_out.extend(classes.float().cpu().detach().numpy())
    calc_metrics(preds_probs, classes_out, model_nm)
    tn, fp, fn, tp = confusion_matrix(classes_out, preds_out).ravel()
    accuracy = accuracy_score(classes_out, preds_out)
    sensitivity = tp / (tp + fn)
    specificity = tn / (tn + fp)
    npv = tn / (tn + fn)
    gmeans = np.sqrt(sensitivity * specificity)
    print(
        "\nFollowing are the metrics for " + model_nm + ": Sensitivity:",
        pretty_per_num(sensitivity),
        "Specificity:",
        pretty_per_num(specificity),
        "Accuracy:",
        pretty_per_num(accuracy),
        "NPV:",
        pretty_per_num(npv),
        "GMeans",
        pretty_per_num(gmeans),
    )
```

Metric tracker graphics for training and validation losses:

```
class MetricTracker(Callback):
    def __init__(self):
        self.val_losses = []


    def on_validation_epoch_end(self, trainer, module):
        self.val_losses.append(trainer._results['validation_epoch_end.val_loss'].value.cpu().numpy()) # track them

        if 0: #index==1:
            # live plotting of results during training, switched off
            ax.plot(self.val_losses, color="orange")
            ax.set_ylabel("Val loss", color="orange", fontsize=14)
            plt.show()
```

Dataset class generation:

Code reference: https://github.com/01-vyom/melanoma-classification/blob/main/src/BYOL/BYOL_SIIM-ISIC_RESNET101.py

```
class MelanomaDataset(Dataset):
    def __init__(self, csv, mode, meta_features, transform=None):

        self.csv = csv.reset_index(drop=True)
        self.mode = mode
        self.use_meta = meta_features is not None
        self.meta_features = meta_features
        self.transform = transform

    def __len__(self):
        return self.csv.shape[0]

    def __getitem__(self, index):

        row = self.csv.iloc[index]

        image = cv2.imread(row.filepath) #importing images
        image = cv2.resize(image, (256,256), interpolation = cv2.INTER_AREA) #resizing images
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        image = image.astype(np.float32)

        image = image.transpose(2, 0, 1)

        data = torch.tensor(image).float()
        if self.transform:
            data = self.transform(data)

        if self.mode == "test":
            return data
        else:
            return data, torch.tensor(self.csv.iloc[index].target).long()
```

importing ground truth data from Google Drive:

```
GOOGLE_DRIVE_PATH_DIS = 'Colab Notebooks/Dissertation'
GOOGLE_DRIVE_PATH = os.path.join('drive','My Drive', GOOGLE_DRIVE_PATH_DIS)

# 2020 data
path=os.path.join(GOOGLE_DRIVE_PATH, 'SIIM-ISIC 2020 image dataset')
df_train = pd.read_csv(os.path.join(GOOGLE_DRIVE_PATH, 'ISIC_2020_Training_GroundTruth_v2.csv'), header=None, sep=',')
```

Data wranglings for the ground truth table:

```
df_train.columns=df_train.iloc[0] #first row to column names
df_train = df_train.iloc[1: , :]
df_train
```

Counting each label:

```
df_train.groupby('benign_malignant').size()
```

Image file path names are taken:

```
df_train["filepath"] = df_train["image_name"].apply(
    lambda x: os.path.join(
        path, f"{x}.jpg"
    )
)

df_train = df_train.reset_index(drop=True)
df_train = df_train.sample(frac=1, random_state=seed).reset_index(drop=True)
df_train.target = df_train.target.astype(int)

df_train
```

Generating training dataset instance from Melanoma Dataset Class:

```
TRAIN_DATASET = MelanomaDataset(df_train, mode="train", meta_features=None)
```

Split work for training and validation sets, 80%-20% proportion

```
total = len(df_train)

train_ratio = 0.80

train_set, val_set = random_split(
    TRAIN_DATASET,
    [int(total * train_ratio), total - int(total * train_ratio)],
    generator=torch.Generator().manual_seed(seed),
)
```

Weighted random sampler function, code reference: https://stackoverflow.com/questions/67799246/weighted-random-sampler-oversample-or-undersample

```
def make_weights_for_balanced_classes(images, nclasses):
    n_images = len(images)
    count_per_class = [0] * nclasses
    for _, image_class in images:
        count_per_class[image_class] += 1
    weight_per_class = [0.] * nclasses
    for i in range(nclasses):
        weight_per_class[i] = float(n_images) / float(count_per_class[i])
    weights = [0] * n_images
    for idx, (image, image_class) in enumerate(images):
        weights[idx] = weight_per_class[image_class]
    return weights
```

Supervised Experiment with Wide Resnet-50 backbone

Loading data, model initiating and training:

Code reference: https://github.com/01-vyom/melanoma-classification/blob/main/src/BYOL/BYOL_SIIM-ISIC_RESNET101.py

```
batch_size=64

train_loader = DataLoader(
    train_set,
    batch_size=batch_size,
    shuffle=True,
    drop_last=True,
    num_workers=5,
    pin_memory=True,
)
val_loader = DataLoader(val_set, batch_size=batch_size, num_workers=5, pin_memory=True)

model = models.wide_resnet50_2(weights=True)
supervised = SupervisedLightningModule(model)
trainer = pl.Trainer(max_epochs=20, accelerator='gpu', devices=1, profiler="simple")
trainer.fit(supervised, train_loader, val_loader)

model_nm = "RESNET50"
metric_1=metric_byol(model_nm, model, val_loader)

savepath = "./"
torch.save(model.state_dict(), savepath + model_nm)

del model
```

Loss graphics:

```
writer = SummaryWriter()

for n_iter in range(100):
    writer.add_scalar('train_loss', np.random.random(), n_iter)
    writer.add_scalar('val_loss', np.random.random(), n_iter)
```

```
tensorboard --logdir=runs
```

BYOL-Wide Resnet-50

```
batch_size=64

train_loader = DataLoader(
    train_set,
    batch_size=batch_size,
    shuffle=True,
    drop_last=True,
    num_workers=5,
    pin_memory=True,
)
val_loader = DataLoader(val_set, batch_size=batch_size, num_workers=5, pin_memory=True)


savepath = "./"


##Self-Supervision

model = models.wide_resnet50_2(weights=True)
byol = BYOL(model, image_size=(300, 300))
trainer = pl.Trainer(
    max_epochs=20, accelerator='gpu', devices=1, accumulate_grad_batches=2048 // 128, profiler="simple"
)
trainer.fit(byol, train_loader, val_loader)
```

```
torch.save(model.state_dict(), savepath + "/BYOL_Selfsupervision_wideRESNET50")
del model
##Supervision


model = models.wide_resnet50_2()
model.load_state_dict(torch.load(savepath + "/BYOL_Selfsupervision_wideRESNET50"))

supervised = SupervisedLightningModule(model)
trainer = pl.Trainer(max_epochs=20, accelerator='gpu', devices=1, profiler="simple")
trainer.fit(supervised, train_loader, val_loader)

model_nm = "BYOL_wideRESNET50"
metric_2=metric_byol(model_nm, model, val_loader)

torch.save(model.state_dict(), savepath + model_nm)
```

```
from torch.utils.tensorboard import SummaryWriter
import numpy as np

writer = SummaryWriter()

for n_iter in range(100):
    writer.add_scalar('train_loss', np.random.random(), n_iter)
    writer.add_scalar('val_loss', np.random.random(), n_iter)
```

```
tensorboard --logdir=runs
```

Supervised Wide Resnet-50 experiment with weighted random sampler

Weighted random sampler code reference: https://stackoverflow.com/questions/67799246/weighted-random-sampler-oversample-or-undersample

```
# For unbalanced dataset we create a weighted sampler
weights = make_weights_for_balanced_classes(train_set, 2)
weights = torch.DoubleTensor(weights)
sampler = torch.utils.data.sampler.WeightedRandomSampler(weights, len(weights))
```

```
model = models.wide_resnet50_2(weights=True)
```

```
batch_size=64

train_loader = DataLoader(
    train_set,
    sampler = sampler,
    batch_size=batch_size,
    drop_last=True,
    num_workers=5,
    pin_memory=True,
)
val_loader = DataLoader(val_set, batch_size=batch_size, num_workers=5, pin_memory=True)
```

```
supervised = SupervisedLightningModule(model)
trainer = pl.Trainer(max_epochs=20, accelerator='gpu', devices=1, profiler="simple")
trainer.fit(supervised, train_loader, val_loader)

model_nm = "wideRESNET50"
metric_2=metric_byol(model_nm, model, val_loader)

savepath = "./"
torch.save(model.state_dict(), savepath + model_nm)
```

```
tensorboard --logdir=runs
```

BYOL-Wide Resnet-50 with weighted random sampling

```
weights = make_weights_for_balanced_classes(train_set, 2)
weights = torch.DoubleTensor(weights)
sampler = torch.utils.data.sampler.WeightedRandomSampler(weights, len(weights))
```

```
batch_size=64

train_loader = DataLoader(
    train_set,
    sampler = sampler,
    batch_size=batch_size,
    drop_last=True,
    num_workers=5,
    pin_memory=True,
)
val_loader = DataLoader(val_set, batch_size=batch_size, num_workers=5, pin_memory=True)
```

```
savepath = "./"
```

```
##Self-Supervision

model = models.wide_resnet50_2(weights=True)
byol = BYOL(model, image_size=(300, 300))
trainer = pl.Trainer(
    max_epochs=20, accelerator='gpu', devices=1, accumulate_grad_batches=2048 // 128, profiler="simple"
)
trainer.fit(byol, train_loader, val_loader)

torch.save(model.state_dict(), savepath + "/BYOL_Selfsupervision_wideRESNET50_weighted")
del model
##Supervision
```

```
model = models.wide_resnet50_2()
model.load_state_dict(torch.load(savepath + "/BYOL_Selfsupervision_wideRESNET50_weighted"))

supervised = SupervisedLightningModule(model)
trainer = pl.Trainer(max_epochs=20, accelerator='gpu', devices=1, profiler="simple")
trainer.fit(supervised, train_loader, val_loader)

model_nm = "BYOL_wideRESNET50_weighted"
metric_4=metric_byol(model_nm, model, val_loader)

torch.save(model.state_dict(), savepath + model_nm)
```

```
from torch.utils.tensorboard import SummaryWriter
import numpy as np

writer = SummaryWriter()

for n_iter in range(100):
    writer.add_scalar('train_loss', np.random.random(), n_iter)
    writer.add_scalar('val_loss', np.random.random(), n_iter)
```

```
tensorboard --logdir=runs
```

BYOL-Wide Resnet-50 with Extra Data Augmentation Experiment

```
batch_size=64

train_loader = DataLoader(
    train_set,
    batch_size=batch_size,
    shuffle=True,
    drop_last=True,
    num_workers=5,
    pin_memory=True,
)
val_loader = DataLoader(val_set, batch_size=batch_size, num_workers=5, pin_memory=True)
```

```
savepath = "./"
```

```
##Self-Supervision

model = models.wide_resnet50_2(weights=True)
byol = BYOL(model, image_size=(300, 300))
trainer = pl.Trainer(
    max_epochs=20, accelerator='gpu', devices=1, accumulate_grad_batches=2048 // 128, profiler="simple", callbacks=[EarlyStopping(monitor
)
```

```
trainer.fit(byol, train_loader, val_loader)

torch.save(model.state_dict(), savepath + "/BYOL_wideRESNET50_augmentation")
del model
##Supervision


model = models.wide_resnet50_2()
model.load_state_dict(torch.load(savepath + "/BYOL_wideRESNET50_augmentation"))

supervised = SupervisedLightningModule(model)
trainer = pl.Trainer(
    max_epochs=20, accelerator='gpu', devices=1, accumulate_grad_batches=2048 // 128, profiler="simple", callbacks=[EarlyStopping(monitor
)
trainer.fit(supervised, train_loader, val_loader)

model_nm = "BYOL_wideRESNET50_augmentation"
metric_7=metric_byol(model_nm, model, val_loader)

torch.save(model.state_dict(), savepath + model_nm)


writer = SummaryWriter()

for n_iter in range(100):
    writer.add_scalar('train_loss', np.random.random(), n_iter)
    writer.add_scalar('val_loss', np.random.random(), n_iter)


tensorboard --logdir=runs


n_splits=5
kf = KFold(n_splits=n_splits, shuffle=True)
val_losses = []
model_list = [] # A list of all final models
metric_list = []
index=0

model_wideResnet = models.wide_resnet50_2(weights=True)

for train_index, val_index in kf.split(TRAIN_DATASET):
    index+=1
    print("CV run {}...".format(index))
    train_ds, val_ds = Subset(TRAIN_DATASET, train_index), Subset(TRAIN_DATASET, val_index)

    train_loader = DataLoader(train_ds, batch_size=batch_size, num_workers=5, pin_memory=True)
    val_loader = DataLoader(val_ds, batch_size=batch_size, num_workers=5, pin_memory=True)

    supervised = SupervisedLightningModule(model_wideResnet)

    metricTracker = MetricTracker()
    trainer = pl.Trainer(accelerator='gpu', devices=1,
                         callbacks=[metricTracker,
                                    EarlyStopping(monitor="val_loss", patience=3),
                                    ModelCheckpoint(save_top_k=1, monitor="val_loss", save_on_train_epoch_end=False)],
                         max_epochs=10,
                         num_sanity_val_steps=0,)

    trainer.fit(supervised, train_loader, val_loader)


    # Load best model based on
    supervised = SupervisedLightningModule().load_from_checkpoint(trainer.checkpoint_callback.best_model_path)
    model_list.append(supervised)

    model_nm = "wideRESNET50"
    metric=metric_byol(model_nm, model, val_loader)
    metric_list.append(metric)

    # Show val results
    val_result = trainer.validate(model=supervised, dataloaders=val_loader)
    val_losses.append(val_result[0]['val_loss'])
    fig, ax = plt.subplots()
    ax.plot(metricTracker.val_losses, color="orange")
    ax.set_ylabel("Val loss", color="orange", fontsize=14)
    plt.show()

savepath = "./"
torch.save(model.state_dict(), savepath + model_list)
```

BYOL-RESNET 101

```
batch_size=32

train_loader = DataLoader(
    train_set,
    batch_size=batch_size,
    shuffle=True,
    drop_last=True,
    num_workers=5,
    pin_memory=True,
)
val_loader = DataLoader(val_set, batch_size=batch_size, num_workers=5, pin_memory=True)
```

```
from pytorch_lightning.callbacks.early_stopping import EarlyStopping
```

```
savepath = "./"
```

```
##Self-Supervision

model = models.resnet101(weights=True)
byol = BYOL(model, image_size=(300, 300))
trainer = pl.Trainer(
    max_epochs=20, accelerator='gpu', devices=1, accumulate_grad_batches=2048 // 128, profiler="simple", callbacks=[EarlyStopping(monitor
)
trainer.fit(byol, train_loader, val_loader)

torch.save(model.state_dict(), savepath + "/BYOL_Selfsupervision_RESNET101")
del model
##Supervision
```

```
model = models.resnet101()
model.load_state_dict(torch.load(savepath + "/BYOL_Selfsupervision_RESNET101"))

supervised = SupervisedLightningModule(model)
trainer = pl.Trainer(max_epochs=20, accelerator='gpu', devices=1, profiler="simple")
trainer.fit(supervised, train_loader, val_loader)

model_nm = "BYOL_RESNET101"
metric_2=metric_byol(model_nm, model, val_loader)

torch.save(model.state_dict(), savepath + model_nm)
```

```
from torch.utils.tensorboard import SummaryWriter
import numpy as np

writer = SummaryWriter()

for n_iter in range(100):
    writer.add_scalar('train_loss', np.random.random(), n_iter)
    writer.add_scalar('val_loss', np.random.random(), n_iter)
```

```
tensorboard --logdir=runs
```

BYOL-RESNET-101 with Weighted Random Sampling and Extra Data Augmentation Experiment

```
weights = make_weights_for_balanced_classes(train_set, 2)
weights = torch.DoubleTensor(weights)
sampler = torch.utils.data.sampler.WeightedRandomSampler(weights, len(weights))
```

```
batch_size=32

train_loader = DataLoader(
    train_set,
    sampler = sampler,
    batch_size=batch_size,
    drop_last=True,
    num_workers=5,
    pin_memory=True,
)
val_loader = DataLoader(val_set, batch_size=batch_size, num_workers=5, pin_memory=True)

savepath = "./"


##Self-Supervision

model = models.resnet101(weights=True)
byol = BYOL(model, image_size=(300, 300))
trainer = pl.Trainer(
    max_epochs=20, accelerator='gpu', devices=1, accumulate_grad_batches=2048 // 128, profiler="simple", callbacks=[EarlyStopping(monitor
)
trainer.fit(byol, train_loader, val_loader)

torch.save(model.state_dict(), savepath + "/BYOL_Selfsupervision_RESNET101_weighted")
del model
##Supervision


model = models.resnet101()
model.load_state_dict(torch.load(savepath + "/BYOL_Selfsupervision_RESNET101_weighted"))

supervised = SupervisedLightningModule(model)
trainer = pl.Trainer(
    max_epochs=20, accelerator='gpu', devices=1, accumulate_grad_batches=2048 // 128, profiler="simple", callbacks=[EarlyStopping(monitor
)
trainer.fit(supervised, train_loader, val_loader)

model_nm = "BYOL_Selfsupervision_RESNET101"

torch.save(model.state_dict(), savepath + model_nm)


metric_6=metric_byol(model_nm, model, val_loader)


writer = SummaryWriter()

for n_iter in range(100):
    writer.add_scalar('train_loss', np.random.random(), n_iter)
    writer.add_scalar('val_loss', np.random.random(), n_iter)


tensorboard --logdir=runs
```

BYOL-RESNET-101 with Extra Data Augmentation and 64 batch size

```
batch_size=64

train_loader = DataLoader(
    train_set,
    batch_size=batch_size,
    shuffle=True,
    drop_last=True,
    num_workers=5,
    pin_memory=True,
)
val_loader = DataLoader(val_set, batch_size=batch_size, num_workers=5, pin_memory=True)


savepath = "./"


##Self-Supervision

model = models.resnet101(weights=True)
byol = BYOL(model, image_size=(300, 300))
trainer = pl.Trainer(
    max_epochs=20, accelerator='gpu', devices=1, accumulate_grad_batches=2048 // 128, profiler="simple", callbacks=[EarlyStopping(monitor
)
trainer.fit(byol, train_loader, val_loader)

torch.save(model.state_dict(), savepath + "/BYOL_RESNET101_aug_bs64")
del model
##Supervision


model = models.resnet101()
```

```
model.load_state_dict(torch.load(savepath + "/BYOL_RESNET101_aug_bs64"))

supervised = SupervisedLightningModule(model)
trainer = pl.Trainer(
    max_epochs=20, accelerator='gpu', devices=1, accumulate_grad_batches=2048 // 128, profiler="simple", callbacks=[EarlyStopping(monitor
)
trainer.fit(supervised, train_loader, val_loader)

model_nm = "BYOL_RESNET101_aug_bs64"
metric_9=metric_byol(model_nm, model, val_loader)

torch.save(model.state_dict(), savepath + model_nm)
```

```
writer = SummaryWriter()

for n_iter in range(100):
    writer.add_scalar('train_loss', np.random.random(), n_iter)
    writer.add_scalar('val_loss', np.random.random(), n_iter)
```

```
tensorboard --logdir=runs
```

BYOL-RESNET 101 Merged dataset training with Extra Data augmentation experiment

Firstly I merged SIIM-ISIC 2020 and SIIM-ISIC 2019 datasets to obtain a more balanced dataset.

```
df_train2 = pd.read_csv(os.path.join(GOOGLE_DRIVE_PATH, 'isic 2019 ground truth.csv'), header=None, sep=',')
df_train2
```

```
df_train2.columns=df_train2.iloc[0]
df_train2 = df_train2.iloc[1: , :]
df_train2
```

Counting each label:

```
df_train2.groupby('benign_malignant').size()
```

```
path2=os.path.join(GOOGLE_DRIVE_PATH, 'SIIM-ISIC 2019 image dataset')

df_train2["filepath"] = df_train2["image_name"].apply(
    lambda x: os.path.join(
        path2, f"{x}.jpg"
    )
)

df_train2 = df_train2.reset_index(drop=True)
df_train2 = df_train2.sample(frac=1, random_state=seed).reset_index(drop=True)
df_train2.target = df_train2.target.astype(int)
df_train2
```

```
df_train_sq=pd.DataFrame()
df_train_sq['target']=df_train['target']
df_train_sq['filepath']=df_train['filepath']

df_train_sq2=pd.DataFrame()
df_train_sq2['target']=df_train2['target']
df_train_sq2['filepath']=df_train2['filepath']
df_train_sq2
```

```
df_train_merged = pd.concat([df_train_sq, df_train_sq2]).reset_index(drop=True)
df_train_merged
```

Counting each label:

```
df_train_merged.groupby('target').size()
```

```
train_ds_merged = MelanomaDataset(df_train_merged, mode="train", meta_features=None)
```

```
total = len(df_train_merged)

train_ratio = 0.80
```

```
train_set, val_set = random_split(
    train_ds_merged,
    [int(total * train_ratio), total - int(total * train_ratio)],
    generator=torch.Generator().manual_seed(seed),
)

batch_size=32

train_loader = DataLoader(
    train_set,
    batch_size=batch_size,
    shuffle=True,
    drop_last=True,
    num_workers=5,
    pin_memory=True,
)
val_loader = DataLoader(val_set, batch_size=batch_size, num_workers=5, pin_memory=True)


savepath = "./"


##Self-Supervision

model = models.resnet101(weights=True)
byol = BYOL(model, image_size=(256, 256))
trainer = pl.Trainer(
    max_epochs=20, accelerator='gpu', devices=1, accumulate_grad_batches=2048 // 128, profiler="simple", callbacks=[EarlyStopping(monitor
)
trainer.fit(byol, train_loader, val_loader)

torch.save(model.state_dict(), savepath + "/merged_BYOL_RESNET101_aug_bs64")
del model
##Supervision


model = models.resnet101()
model.load_state_dict(torch.load(savepath + "/merged_BYOL_RESNET101_aug_bs64"))

supervised = SupervisedLightningModule(model)
trainer = pl.Trainer(
    max_epochs=20, accelerator='gpu', devices=1, accumulate_grad_batches=2048 // 128, profiler="simple", callbacks=[EarlyStopping(monitor
)
trainer.fit(supervised, train_loader, val_loader)

model_nm = "merged_BYOL_RESNET101_aug_bs64"
metric_10=metric_byol(model_nm, model, val_loader)

torch.save(model.state_dict(), savepath + model_nm)


writer = SummaryWriter()

for n_iter in range(100):
    writer.add_scalar('train_loss', np.random.random(), n_iter)
    writer.add_scalar('val_loss', np.random.random(), n_iter)


tensorboard --logdir=runs
```

BYOL-RESNET 101 Merged dataset training with Extra Data augmentation and learning rate scheduler

```
savepath = "./"


##Self-Supervision

model = models.resnet101(weights=True)
byol = BYOL(model, image_size=(256, 256))
trainer = pl.Trainer(
    max_epochs=20, accelerator='gpu', devices=1, profiler="simple", callbacks=[LearningRateMonitor(logging_interval='step'),
                                                                           EarlyStopping(monitor='val_loss', patience=3)]
)
trainer.fit(byol, train_loader, val_loader)

torch.save(model.state_dict(), savepath + "/merged_BYOL_RESNET101_aug_bs64_lr_sch")
del model
```

```
##Supervision


model = models.resnet101()
model.load_state_dict(torch.load(savepath + "/merged_BYOL_RESNET101_aug_bs64_lr_sch"))

supervised = SupervisedLightningModule(model)
trainer = pl.Trainer(
    max_epochs=20, accelerator='gpu', devices=1, profiler="simple", callbacks=[LearningRateMonitor(logging_interval='step'),
                                                                                                     EarlyStopping(monitor='val_loss', patie
)
trainer.fit(supervised, train_loader, val_loader)

model_nm = "merged_BYOL_RESNET101_aug_bs64_lr_sch"
metric_lr=metric_byol(model_nm, model, val_loader)

torch.save(model.state_dict(), savepath + model_nm)
```

```
writer = SummaryWriter()

for n_iter in range(100):
    writer.add_scalar('train_loss', np.random.random(), n_iter)
    writer.add_scalar('val_loss', np.random.random(), n_iter)
```

```
tensorboard --logdir=runs
```

BYOL-RESNET 101 Merged dataset training with Extra Data augmentation and Weighted Random Sampler

```
weights = make_weights_for_balanced_classes(train_set, 2)
weights = torch.DoubleTensor(weights)
sampler = torch.utils.data.sampler.WeightedRandomSampler(weights, len(weights))
```

```
batch_size=32

train_loader = DataLoader(
    train_set,
    sampler = sampler,
    batch_size=batch_size,
    drop_last=True,
    num_workers=5,
    pin_memory=True,
)
val_loader = DataLoader(val_set, batch_size=batch_size, num_workers=5, pin_memory=True)
```

```
savepath = "./"
```

```
##Self-Supervision

model = models.resnet101(weights=True)
byol = BYOL(model, image_size=(256, 256))
trainer = pl.Trainer(
    max_epochs=20, accelerator='gpu', devices=1, accumulate_grad_batches=2048 // 128, profiler="simple", callbacks=[EarlyStopping(monitor
)
trainer.fit(byol, train_loader, val_loader)

torch.save(model.state_dict(), savepath + "/BYOL_RESNET101_mergedds_weighted")
del model
##Supervision


model = models.resnet101()
model.load_state_dict(torch.load(savepath + "/BYOL_RESNET101_mergedds_weighted"))

supervised = SupervisedLightningModule(model)
trainer = pl.Trainer(
    max_epochs=20, accelerator='gpu', devices=1, accumulate_grad_batches=2048 // 128, profiler="simple", callbacks=[EarlyStopping(monitor
)
trainer.fit(supervised, train_loader, val_loader)

model_nm = "BYOL_RESNET101_mergedds_weightd"

torch.save(model.state_dict(), savepath + model_nm)
```

```
metric_byol(model_nm, model, val_loader)
```

```
writer = SummaryWriter()

for n_iter in range(100):
    writer.add_scalar('train_loss', np.random.random(), n_iter)
    writer.add_scalar('val_loss', np.random.random(), n_iter)
```

```
tensorboard --logdir=runs
```

BYOL-RESNET 152 Merged dataset training with Extra Data augmentation and Weighted Random Sampler

```
savepath = "./"
```

```
##Self-Supervision

model = models.resnet152(weights=True)
byol = BYOL(model, image_size=(256, 256))
trainer = pl.Trainer(
    max_epochs=20, accelerator='gpu', devices=1, accumulate_grad_batches=2048 // 128, profiler="simple", callbacks=[EarlyStopping(monitor
)
trainer.fit(byol, train_loader, val_loader)

torch.save(model.state_dict(), savepath + "/BYOL_RESNET152_mergedds_weighted")
del model
##Supervision
```

```
model = models.resnet152()
model.load_state_dict(torch.load(savepath + "/BYOL_RESNET152_mergedds_weighted"))

supervised = SupervisedLightningModule(model)
trainer = pl.Trainer(
    max_epochs=20, accelerator='gpu', devices=1, accumulate_grad_batches=2048 // 128, profiler="simple", callbacks=[EarlyStopping(monitor
)
trainer.fit(supervised, train_loader, val_loader)

model_nm = "BYOL_RESNET152_mergedds_weightd"

torch.save(model.state_dict(), savepath + model_nm)
```

```
metric_byol(model_nm, model, val_loader)
```

```
writer = SummaryWriter()

for n_iter in range(100):
    writer.add_scalar('train_loss', np.random.random(), n_iter)
    writer.add_scalar('val_loss', np.random.random(), n_iter)
```

```
tensorboard --logdir=runs
```

Calculating statistical significancy bounds:

Confidence interval formula and code: https://machinelearningmastery.com/confidence-intervals-for-machine-learning/

To compare the results with Pathak's work, we need to calculate Pathak's work's confidence intervals:

```
pathak_result={
    'BYOL&Resnet-101 best threshold AUC value':0.907,
    'BYOL&Resnet-101 best threshold G-mean value':0.822,
    'BYOL&Resnet-101 best threshold sensitivity value':0.796,
    'BYOL&Resnet-101 standard threshold G-mean value':0.648,
    'BYOL&Resnet-101 standard threshold sensitivity value':0.428,
    'BYOL&EfficientNet-B5 best threshold AUC value':0.932,
    'BYOL&EfficientNet-B5 best threshold G-Mean value':0.857,
    'BYOL&EfficientNet-B5 best threshold sensitivity':0.864,
    'BYOL&EfficientNet-B5 standard threshold G-mean':0.774,
    'BYOL&EfficientNet-B5 standard threshold sensitivity':0.608
}

# binomial confidence interval for %95 confidence
from math import sqrt

for x in pathak_result:
  a=pathak_result[x]
  interval = 1.96 * sqrt((a * (1 - a)) / 11692)
  print('The confidence interval for '+x+ ' is between','%.3f' % (a+interval), 'and', '%.3f' % (a-interval))
```

Data augmentation sample visualisation for the report

I imported one sample image from the ISIC 2020 dataset

```
path_sample=os.path.join(GOOGLE_DRIVE_PATH, 'ISIC_0075663.jpg')
path_sample
```

Code reference for the rest of the code: https://pytorch.org/vision/stable/auto_examples/plot_transforms.html#sphx-glr-auto-examples-plot-transforms-py

```
from PIL import Image
from pathlib import Path
import matplotlib.pyplot as plt
import numpy as np

import torch
import torchvision.transforms as T


plt.rcParams["savefig.bbox"] = 'tight'
orig_img = Image.open(path_sample)

torch.manual_seed(0)


def plot(imgs, with_orig=True, row_title=None, **imshow_kwargs):
    if not isinstance(imgs[0], list):
        # Make a 2d grid even if there's just 1 row
        imgs = [imgs]

    num_rows = len(imgs)
    num_cols = len(imgs[0]) + with_orig
    fig, axs = plt.subplots(nrows=num_rows, ncols=num_cols, squeeze=False)
    for row_idx, row in enumerate(imgs):
        row = [orig_img] + row if with_orig else row
        for col_idx, img in enumerate(row):
            ax = axs[row_idx, col_idx]
            ax.imshow(np.asarray(img), **imshow_kwargs)
            ax.set(xticklabels=[], yticklabels=[], xticks=[], yticks=[])

    if with_orig:
        axs[0, 0].set(title='Original image')
        axs[0, 0].title.set_size(8)
    if row_title is not None:
        for row_idx in range(num_rows):
            axs[row_idx, 0].set(ylabel=row_title[row_idx])

    plt.tight_layout()
```

Colour jittering visualisation:

```
jitter = T.ColorJitter(brightness=.5, hue=.3)
jitted_imgs = [jitter(orig_img) for _ in range(4)]
plot(jitted_imgs)
```

Center crop visualisation:

```
center_crops = [T.CenterCrop(size=size)(orig_img) for size in (30, 50, 100, orig_img.size)]
plot(center_crops)
```