```
#%% md

PyTorch Lightning is a lightweight PyTorch wrapper that simplifies the process of training deep learning models. It provides a high-level interface for organizing and orchestrating training routines, making it easier for researchers and practitioners to focus on model development rather than low-level training details.

Advantages of using PyTorch Lightning to train models include:

### Abstraction of Training Loop:
PyTorch Lightning abstracts away the boilerplate code for training loops, validation loops, and testing loops. This allows developers to focus more on designing and implementing the model architecture rather than writing repetitive training code.

### Simplified Model Code:
By providing predefined hooks for common tasks such as forward passes, backward passes, and optimization steps, PyTorch Lightning reduces the complexity of model code. This results in cleaner, more organized code that is easier to understand and maintain.

### Standardization:
PyTorch Lightning encourages a standardized structure for organizing deep learning projects. This makes it easier for developers to collaborate, share code, and reproduce results across different projects and teams.

### Flexibility:
Despite providing a high-level interface, PyTorch Lightning remains flexible and customizable. Developers can override default behaviors and customize training routines according to their specific requirements.
#%%
```

```
!pip install pytorch-lightning
#%%
import numpy as np
import h5py
import math
import os
import numpy as np
import matplotlib.pyplot as plt


import torch
import torchvision
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader, random_split
import h5py
from torchvision import transforms
#%%
from google.colab import drive
drive.mount('/content/drive')
#%%
import numpy as np
import h5py

def load_h5(file_name, size):
    # Load the dataset from the HDF5 file
    with h5py.File(file_name, 'r') as f:
        X = np.array(f['X_jets'][:size])
        y = np.array(f['y'][:size])
    return X, y

#%%
file = '/content/drive/MyDrive/quark-gluon_data-set_n139306.hdf5'
size = 10000
X, y = load_h5(file, size)
#%%
from skimage.transform import resize
def data_preprocess(X_jets):
    #Normalizing the images

```

```python
    # Resizing images from (125, 125, 3) to (128, 128, 3)
    resized_images = np.zeros((X_jets.shape[0], 128, 128, 3), dtype=np.float32)
    for i in range(X_jets.shape[0]):
        resized_images[i] = resize(X_jets[i], (128, 128), anti_aliasing=True)

    X_jets = resized_images
    del resized_images

    # Normalizing the entire image across all channels
    mean = np.mean(X_jets)
    std = np.std(X_jets)
    X_jets = (X_jets - mean) / std

    # Assuming X_jets is your image array
    X_jets = np.clip(X_jets, 0, None)
    # X_jets[:,:,:,0] = X_jets[:,:,:,0]/np.max(X_jets[:,:,:,0])
    # X_jets[:,:,:,1] = X_jets[:,:,:,1]/np.max(X_jets[:,:,:,1])
    # X_jets[:,:,:,2] = X_jets[:,:,:,2]/np.max(X_jets[:,:,:,2])

    return X_jets
#%%
X_jets = data_preprocess(X)
#%%
import matplotlib.pyplot as plt
import numpy as np

def plot_images_with_combined_channels(dataset):
    # Extracting the initial 5 jet images from the dataset...
    images = dataset[:5]

    for i in range(5):
        X_sample = images[i]
        og_plot = plt.imshow(X_sample)
```

```


        fig, axs = plt.subplots(1, 3, figsize=(20, 20))

        im1 = axs[0].imshow(X_sample[:,:,0], cmap='viridis', vmin=-0.5, vmax=2.0, interpolation='nearest')
        axs[0].set_title(f'Track for {i}th image')

        im2 = axs[1].imshow(X_sample[:, :, 1], cmap='viridis', vmin=-0.5, vmax=2.0, interpolation='nearest')
        axs[1].set_title(f'ECAL for {i}th image')

        im3 = axs[2].imshow(X_sample[:, :,2], cmap='viridis', vmin=-0.5, vmax=2.0, interpolation='nearest')
        axs[2].set_title(f'HCAL for {i}th image')

        # Add colorbars
        fig.colorbar(im1, ax=axs[0], shrink=0.25)
        fig.colorbar(im2, ax=axs[1], shrink=0.25)
        fig.colorbar(im3, ax=axs[2], shrink=0.25)

        plt.show()
#%%
plot_images_with_combined_channels(X_jets)
#%% md
* The supervised contrastive loss contrasts the set of all samples from the same class as positives against the negatives from the remainder of the given data batch in training.
* Taking class label information into account results in an embedding space where elements of the same class are more closely aligned than in the self-supervised case.
* Although we already know that the loss function is pushing to map the given pairs of positives close together, and the pair of negative and postive samples further apart.
```

```
#%% md
In a Supervised Contrastive Loss setting, label information is used to sample positives in addition to augmentations of the same image.
#%%
# We can have an encoder, to which we pass the image (ResNet50 etc.), and obtain the underlying hidden representation of the specified dimension.
# Then, we apply a non-linear transformation on it to further apply contrastive layer.

# Then, normalize the obtianed hidden representations and apply loss function.
#%%
def nt_xent_loss(out_1, out_2, temperature):
  out  = torch.cat((out_1, out_2), dim=0)
  n_samples = len(out)

  #Full similarity matrix
  cov = torch.mm(out, out.t().contiguous())
  sim = torch.exp(cov/temperature)

  mask = ~torch.eye(n_samples, device = sim.device).bool()
  neg = sim.masked_select(mask).view(n_samples, -1).sum(dim=-1)

  #Positive similarity
  pos = torch.exp(torch.sum(out_1 * out_2, dim = -1) / temperature)
  pos = torch.cat([pos, pos], dim = 0)

  loss =  -(torch.log(pos/neg).mean())
  return loss
#%%
# class Projection(nn.Module):
#     def __init__(self, dim_in = 125*125, dim_hid = 125*125, dim_out = 128):
#         super(Projection, self).__init__()
#         self.dim_in = dim_in
#         self.dim_hid = dim_hid
```

```
#         self.dim_out = dim_out
#         self.linear = nn.Linear(dim_in, dim_out)
#%%
import torch
import torch.nn as nn
import torch.nn.functional as F
import pytorch_lightning as pl
import torchvision.models as models

class SimCLR(pl.LightningModule):
    def __init__(self, hidden_dim, lr, temperature, weight_decay, max_epochs=500):
        super().__init__()
        self.save_hyperparameters()
        assert self.hparams.temperature > 0.0, "The temperature must be a positive float!"

        # Base model for feature extraction
        self.convnet = models.resnet18(pretrained=True)
        num_ftrs = self.convnet.fc.in_features
        self.convnet.fc = nn.Linear(num_ftrs, hidden_dim)
        # self.lin1 = nn.Linear(self.convnet.fc.in_features, hidden_dim)
        # self.lin2 = nn.Linear(hidden_dim, hidden_dim)
        # self.relu = nn.ReLU(inplace=True)

    # def forward(self, x):
    #     x = self.convnet(x)
    #     x = self.lin1(x)
    #     x = self.relu(x)
    #     x = self.lin2(x)

    def configure_optimizers(self):
        optimizer = torch.optim.AdamW(self.parameters(), lr=self.hparams.lr, weight_decay=self.hparams.weight_decay)
        lr_scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
```

```
                optimizer, T_max=self.hparams.max_epochs, eta_min=self.hparams.lr / 50
            )
        return [optimizer], [lr_scheduler]

    def info_nce_loss(self, batch, mode="train"):
        imgs, labels = batch
#         print(imgs.shape)
#         print(labels.shape)
#         imgs = torch.cat(imgs, dim=0)

        # Encode all images
        feats = self.convnet(imgs)

        # Normalize features
        feats = F.normalize(feats, dim=1)

        # Calculate cosine similarity
        cos_sim = F.cosine_similarity(feats[:, None, :], feats[None, :, :], dim=-1)
        # Mask out cosine similarity to itself
        self_mask = torch.eye(cos_sim.shape[0], dtype=torch.bool, device=cos_sim.device)
        cos_sim.masked_fill_(self_mask, -9e15)
        # Find positive example -> batch_size//2 away from the original example
        pos_mask = self_mask.roll(shifts=cos_sim.shape[0] // 2, dims=0)
        # InfoNCE loss
        cos_sim = cos_sim / self.hparams.temperature
        nll = -cos_sim[pos_mask] + torch.logsumexp(cos_sim, dim=-1)
        nll = nll.mean()

        # Supervised contrastive loss, calculating class similarity,
        # class_sim = cos_sim[labels == labels[:, None]]
        # class_sim = class_sim / self.hparams.temperature
        # class_nll = -class_sim.diag() + torch.
```

```
logsumexp(class_sim, dim = -1)
        # class_nll = class_nll.mean()

        #Combining both the losses,
        total_loss = nll


        # Logging loss
        self.log(mode + "_loss", total_loss)
        self.log(mode + "_nll", nll)
#         self.log(mode + "_class_nll", class_nll)
        # Get ranking position of positive example
        comb_sim = torch.cat(
            [cos_sim[pos_mask][:, None], cos_sim.masked_fill(pos_mask, -9e15)],
            dim=-1,
        )
        sim_argsort = comb_sim.argsort(dim=-1, descending=True).argmin(dim=-1)
        # Logging ranking metrics
        self.log(mode + "_acc_top1", (sim_argsort == 0).float().mean())
        self.log(mode + "_acc_top5", (sim_argsort < 5).float().mean())
        self.log(mode + "_acc_mean_pos", 1 + sim_argsort.float().mean())
        return total_loss

    def training_step(self, batch, batch_idx):
        return self.info_nce_loss(batch, mode="train")

    def validation_step(self, batch, batch_idx):
        self.info_nce_loss(batch, mode="val")

    # def testing_step(self, batch, batch_idx):
    #     self.info_nce_loss(batch, mode="test")

    def on_epoch_end(self):
            if self.current_epoch % 5 == 0:
                print(f"Epoch: {self.current_epoch}
```

```python
, Train Loss: {self.trainer.callback_metrics['train_loss']}, Val Loss: {self.trainer.callback_metrics['val_loss']}")
#%%
from torch.utils.data import DataLoader, Dataset
from torchvision import transforms
from PIL import Image
from matplotlib import cm

class JetImageDataset(Dataset):
    def __init__(self, images, labels, transform=None):
        self.images = images
        self.labels = labels
        self.transform = transform

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

    def __getitem__(self, idx):
        image = self.images[idx]
#         image = Image.fromarray(np.uint8(cm.gist_earth(image)*255))
        label = self.labels[idx]
        if self.transform:
            image = self.transform(image)
        return image, label

# Define your data augmentation pipeline
transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.ToTensor(),
])

# Load your data
# Assuming `images` is a list of ndarrays and `labels` is a list of labels
dataset = JetImageDataset(X_jets, y, transform=transform)
```

```
#dataloader = DataLoader(dataset, batch_size=32, shuffle=True)
#%%

#%%
print(dataset.labels[:20])
#%%
from torch.utils.data import random_split

# Splitting data as per the following scheme :  80% for training, 10% for validation, and 10% for testing
train_size = int(0.8 * len(dataset))
valid_size = int(0.1 * len(dataset))
test_size = len(dataset) - train_size - valid_size

# Splitting the dataset
train_dataset, valid_dataset, test_dataset = random_split(dataset, [train_size, valid_size, test_size])
#%%
train_loader = DataLoader(train_dataset, batch_size=256, shuffle=True)
valid_loader = DataLoader(valid_dataset, batch_size=256, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=256, shuffle=False)
#%%
from pytorch_lightning.loggers import TensorBoardLogger

# Create a logger
logger = TensorBoardLogger('logs/', name='simclr')
model = SimCLR(hidden_dim=256, lr=0.001, temperature=10, weight_decay=0.1, max_epochs=50)
#%%
torch.cuda.device_count()
#%%
from pytorch_lightning import Trainer

logger = TensorBoardLogger('logs/', name='simclr')
```

```

trainer = Trainer(devices=1, accelerator="gpu", logger=logger, enable_checkpointing=True, max_time="00:00:45:00", max_epochs=100)
trainer.fit(model, train_loader, valid_loader)
#%%
%load_ext tensorboard
#%%
%tensorboard --logdir logs/simclr/
#%%

```