Specific Task 1

(if you are interested in “Deep Graph Anomaly Detection with Contrastive Learning” project):

1. Classify the quark/gluon data with a model that learns data representation with a contrastive loss.
2. Evaluate the classification performance on a test dataset.

In [ ]:
```
import torch
from torch.utils.data import Dataset, DataLoader, random_split
from torch import nn, optim
import torch.functional as F
import torchvision
import h5py
import os
import multiprocessing as mp
from torchvision import transforms
import pytorch_lightning as pl
import numpy as np
import torchmetrics as tm
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint
%load_ext tensorboard

device = "cuda"
```

In [ ]:
```
def subset_dataset(raw_path, processed_path, subset_len = 6000):
    with h5py.File(raw_path, 'r') as f, h5py.File(processed_path, 'w') as p:
        keys = list(f.keys())
        total_events = f[keys[1]].shape[0]
        for key in keys:
            shape = (subset_len,)
            if len(f[key].shape) > 1:
                shape = (subset_len, 125, 125, 3)
            p.create_dataset(key, shape=shape)
        quark_count = 0
        gluon_count = 0
        idx = 0
        for i in range(total_events):
            if quark_count < subset_len // 2:
                for key in keys:
                    p[key][idx] = f[key][idx]
                quark_count += 1
                idx += 1
            elif gluon_count < subset_len // 2:
                for key in keys:
                    p[key][idx] = f[key][idx]
                gluon_count += 1
                idx += 1
            elif idx >= subset_len:
                break
```

In [ ]:
```
uncompressed_data_path = "../Data/hdf5/processed/quark-gluon-dataset.hdf5"
subset_data_path = "../Data/hdf5/processed/processed.hdf5"
CHECKPOINT_PATH = "saved_models/"
```

In [ ]:
```
if not os.path.exists(subset_data_path):
    subset_dataset(uncompressed_data_path, subset_data_path, subset_len=30000)
```

In [ ]:
```
class QuarkGluonDataset(Dataset):
    def __init__(self, path, transform = None) -> None:
        super().__init__()
        self.path = path
        self.transform = transform
        with h5py.File(self.path, 'r') as f:
            self.keys = list(f.keys())
    def __len__(self):
        with h5py.File(self.path, 'r') as f:
            return len(f[self.keys[1]])
    def __getitem__(self, index):
        with h5py.File(self.path, 'r') as f:
            x = f[self.keys[0]][index]
            y = np.array(f['y'][index])
            y = torch.from_numpy(y)
            # y = torch.nn.functional.one_hot(y.long(), 2)
            x = torch.from_numpy(x)
            x = torch.permute(x, (2, 0, 1)) # convert (n, n, 3) -> (3, n, n)
            if self.transform is not None:
                x = self.transform(x)
                return x, y
            return x, y
```

In [ ]:
```
class ContrastiveTransformations(object):

    def __init__(self, base_transforms, n_views=2):
        self.base_transforms = base_transforms
        self.n_views = n_views
```

```python
    def __call__(self, x):
        return [self.base_transforms(x) for i in range(self.n_views)]

contrast_transforms = transforms.Compose([
                                          transforms.ToPILImage(),
                                          transforms.RandomResizedCrop(size=96),
                                          transforms.RandomApply([
                                              transforms.ColorJitter(brightness=0.5,
                                                                     contrast=0.5,
                                                                     saturation=0.5,
                                                                     hue=0.1)
                                          ], p=0.8),
                                          transforms.ToTensor(),
                                          transforms.Normalize((0.5,), (0.5,)),
                                         ])
```

In [ ]:
```python
def train_val_test_split(dataset, train = 0.6, val = 0.2, test = 0.2):
    train_data, val_data, test_data = random_split(dataset, [train, val, test])
    datasets = {}
    datasets['train'] = train_data
    datasets['val'] = val_data
    datasets['test'] = test_data
    return datasets

cpu_count = mp.cpu_count()

class QuarkGluonDataModule(pl.LightningDataModule):
    def __init__(self,dataset, batch_size = 64) -> None:
        super().__init__()
        self.batch_size = batch_size
        self.dataset = dataset
    def setup(self, stage:str):
        self.train_data = self.dataset['train']
        self.val_data = self.dataset['val']
        self.test_data = self.dataset['test']
    def get_train(self, idx):
        return self.train_data[idx][0]
    def train_dataloader(self):
        return DataLoader(self.train_data, batch_size=self.batch_size, shuffle=True,
                          num_workers=cpu_count, prefetch_factor=2* cpu_count)
    def val_dataloader(self):
        return DataLoader(self.val_data, batch_size=self.batch_size, shuffle=False,
                           num_workers=cpu_count, prefetch_factor=2* cpu_count)
    def test_dataloader(self):
        return DataLoader(self.test_data, batch_size=self.batch_size, shuffle=False,
                          num_workers=cpu_count, prefetch_factor=2* cpu_count)
```

In [ ]:
```python
class Model(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 f(.)
        self.convnet = torchvision.models.resnet18(num_classes=4*hidden_dim)  # Output of last linear layer
        # The MLP for g(.) consists of Linear->ReLU->Linear
        self.convnet.fc = nn.Sequential(
            self.convnet.fc,  # Linear(ResNet output, 4*hidden_dim)
            nn.ReLU(inplace=True),
            nn.Linear(4*hidden_dim, hidden_dim),
            nn.Softmax()

        )
    def configure_optimizers(self):
        optimizer = optim.AdamW(self.parameters(),
                                lr=self.hparams.lr,
                                weight_decay=self.hparams.weight_decay)
        lr_scheduler = 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'):
        # print(batch.shape)
        imgs, _ = batch

        # Encode all images
        feats = self.convnet(imgs)
        # Calculate cosine similarity
        cos_sim = torch.nn.functional.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()

        # Logging loss
        self.log(mode+'_loss', nll)
        # Get ranking position of positive example
        comb_sim = torch.cat([cos_sim[pos_mask][:,None],  # First position positive example
                              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 nll

    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')
```

In [ ]:
```
def train_model(batch_size, max_epochs=500, **kwargs):
    trainer = pl.Trainer(default_root_dir=os.path.join(CHECKPOINT_PATH, 'CLossCNN'),
                         accelerator="gpu",
                         devices=1,
                         max_epochs=max_epochs,
                         callbacks=[ModelCheckpoint(save_weights_only=True, mode='max', monitor='val_acc_top5'),
                                    LearningRateMonitor('epoch')],
                        enable_progress_bar=False)
    trainer.logger._default_hp_metric = None # Optional logging argument that we don't need

    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, 'CLossCNN.ckpt')
    if os.path.isfile(pretrained_filename):
        print(f'Found pretrained model at {pretrained_filename}, loading...')
        model = Model.load_from_checkpoint(pretrained_filename) # Automatically loads the model
        # with the saved hyperparameters
    else:
        pl.seed_everything(42) # To be reproducable
        dataset = QuarkGluonDataset(subset_data_path, transform=contrast_transforms)
        dataset = train_val_test_split(dataset)
        dataset = QuarkGluonDataModule(dataset, batch_size=batch_size)
        model = Model(max_epochs=max_epochs, **kwargs)
        trainer.fit(model, datamodule=dataset)
        # Load best checkpoint after training
        model = model.load_from_checkpoint(trainer.checkpoint_callback.best_model_path)

    return model
```

In [ ]:
```
CLossCNN = train_model(batch_size=64,
                         hidden_dim = 2,
                         lr=5e-4,
                         temperature=0.07,
                         weight_decay=1e-4,
                         max_epochs=500)
```

```
GPU available: True (cuda), used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs
Global seed set to 42
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

  | Name    | Type   | Params
-----------------------------------
0 | convnet | ResNet | 11.2 M
-----------------------------------
11.2 M    Trainable params
0         Non-trainable params
11.2 M    Total params
44.723    Total estimated model params size (MB)
/home/guru/miniconda3/envs/gnn/lib/python3.9/site-packages/torch/nn/modules/container.py:204: UserWarning: Implicit dimension c
hoice for softmax has been deprecated. Change the call to include dim=X as an argument.
  input = module(input)
`Trainer.fit` stopped: `max_epochs=500` reached.
```

In [ ]:
```
dataset = QuarkGluonDataset(subset_data_path, transform=contrast_transforms)
dataset = train_val_test_split(dataset)
test_loader = DataLoader(dataset['test'], batch_size=64, shuffle=False)
```

In [ ]:
```
with torch.no_grad():
    all_labels = []
    all_preds = []
    for batch in test_loader:
        pred = CLossCNN.convnet(batch[0])
        all_preds.append(pred)
        all_labels.append(batch[1])
```

In [ ]:
```
all_labels = torch.cat(all_labels)
all_preds = torch.cat(all_preds)
```

In [ ]:
```
# %tensorboard --logdir ./saved_models/CLossCNN
```

In [ ]:
```
accuracy = tm.Accuracy(task="binary", num_classes=2)
```

In [ ]:
```
accuracy(all_preds.argmax(dim=1), all_labels)
```

Out[ ]:
```
tensor(0.5192)
```

References:

model Architecture

In [ ]: