# Task 2: Deep Learning based Quark-Gluon Classification

---

● **Data Preparation**: Please train your model on 80% of the data and evaluate on the remaining 20%. Please make sure not to overfit on the test dataset - it will be checked with an independent sample.

● **Model Training**: Train a **VGG13** model and another model of your choice.

---

In [1]:
```
import os
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```
import gc
import pyarrow.parquet as pq
from random import shuffle
```

In [3]:
```
import torch
from torch import nn
from torch import optim

from torchvision import transforms as T
from torch.utils.data import Dataset, DataLoader, TensorDataset, random_spli

from torchvision import models
```

```
/home/harkhymadhe/miniforge3/lib/python3.11/site-packages/torchvision/io/ima
ge.py:13: UserWarning: Failed to load image Python extension: '/home/harkhym
adhe/miniforge3/lib/python3.11/site-packages/torchvision/image.so: undefined
symbol: _ZN3c106detail23torchInternalAssertFailEPKcS2_jS2_RKSs'If you don't
plan on using image functionality from `torchvision.io`, you can ignore this
warning. Otherwise, there might be something wrong with your environment. Di
d you have `libjpeg` or `libpng` installed before building `torchvision` fro
m source?
  warn(
```

---

## Phase I: Data Preparation

**Aim**: Please train your model on 80% of the data and evaluate on the remaining 20%. Please make sure not to overfit on the test dataset - it will be checked with an independent sample.

First, the parquet files are downloaded and stored in the **./dataset/** folder.

In [4]:
```
# File paths
file1 = "dataset/QCDToGGQQ_IMGjet_RH1all_jet0_run0_n36272.test.snappy.parque
file2 = "dataset/QCDToGGQQ_IMGjet_RH1all_jet0_run1_n47540.test.snappy.parque
file3 = "dataset/QCDToGGQQ_IMGjet_RH1all_jet0_run2_n55494.test.snappy.parque
```

A litle bit of experimentation showed that loading the Parquet files via **Pandas** or basic **PyArrow** was very inefficient and resulted in OOM errors. I attempt to bypass this via the more specialized **parquet** subpackage in **PyArrow**.

In [5]:
```
# Load data files
class ParquetDataset(Dataset):
    def __init__(self, filename):
        self.parquet = pq.ParquetFile(filename)
        self.cols = None

    def __getitem__(self, index):
        data = self.parquet.read_row_group(index, columns=self.cols).to_pydi
        data['X_jets']= 1.*np.float32(data['X_jets'][0])#/data['mGG']
        data['X_jets']=data['X_jets'][0][:80000]

        data = dict(data)
        return torch.as_tensor(np.expand_dims(data["X_jets"], axis = 0)), ir

    def __len__(self):
        return self.parquet.num_row_groups

    @classmethod
    def from_files(cls, filenames):
        return ConcatDataset([cls(fname) for fname in filenames])
```

In [ ]:
```

```

Loading the Parquet data using the **ParquetDataset** class defined above is quite OK, but it makes actual file loading for multiple data points more cumbersome. A more efficient **BatchedParquetDataset** is implemented below:

In [6]:
```
# Load data files
class BatchedParquetDataset(Dataset):
    def __init__(self, filename, batch_size):
        super().__init__()

        self.batch_size = batch_size
        self.parquet = pq.ParquetFile(filename)
        self.cols = None

        self.size = self.parquet.num_row_groups

        self.remainder = self.size % self.batch_size
```

```python
        self.batch_indices = list(range(0, self.size, self.batch_size))
        self.batch_indices = list(
            zip(
                self.batch_indices,
                self.batch_indices[1:] + [self.batch_indices[-1] + (self.rem
            )
        )

    def __getitem__(self, index):
        indexes = range(*self.batch_indices[index])
        data = self.parquet.read_row_groups(indexes, columns=self.cols).colu

        image = torch.as_tensor(data[0].to_pylist())
        targets = torch.as_tensor(data[-1].to_pylist(), dtype = torch.long)

        return image, targets

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

    @classmethod
    def from_files(cls, filenames, batch_size):
        return ConcatDataset([cls(filename = fname, batch_size = batch_size)
```

In [7]:
```python
batch_size = 64
```

In [8]:
```python
batched_data = BatchedParquetDataset.from_files(batch_size = batch_size, fil
```

In [9]:
```python
sample = batched_data[0]
```

In [10]:
```python
sample[0].shape
```

Out[10]:
```
torch.Size([64, 3, 125, 125])
```

In [11]:
```python
sample[1].shape
```

Out[11]:
```
torch.Size([64])
```

In [12]:
```python
len(batched_data) * batch_size
```

Out[12]:
```
139392
```

In [ ]:

In [13]:
```python
train_data, test_data = random_split(batched_data, lengths = [.8, .2])
```

class BatchedDataLoader(DataLoader): def **init**(self, *args, **kwargs):*
*super().**init**(*args, **kwargs)

```python
    def __iter__(self):
        return iter(super())
```

```
    def __next__(self):
        return
```

In [14]:
```
train_dl = DataLoader(
    dataset = train_data,
    batch_size = 1,
    shuffle = True,
    num_workers = 4,
    pin_memory = True
)

test_dl = DataLoader(
    dataset = test_data,
    batch_size = 1,
    shuffle = True,
    num_workers = 4,
    pin_memory = True
)
```

In [ ]:

In [15]:
```
# Set device
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

In [16]:
```
gc.collect()
```

Out[16]:
```
20
```

---

# Phase II: Model Training

**Aim**: Train a VGG13 model and another model of your choice.

---

In [ ]:

In [17]:
```
model = models.vgg13(pretrained=True)
```

```
/home/harkhymadhe/miniforge3/lib/python3.11/site-packages/torchvision/model
s/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since
0.13 and may be removed in the future, please use 'weights' instead.
  warnings.warn(
/home/harkhymadhe/miniforge3/lib/python3.11/site-packages/torchvision/model
s/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` f
or 'weights' are deprecated since 0.13 and may be removed in the future. The
current behavior is equivalent to passing `weights=VGG13_Weights.IMAGENET1K_
V1`. You can also use `weights=VGG13_Weights.DEFAULT` to get the most up-to-
date weights.
  warnings.warn(msg)
```

In [18]:
```
model
```

Out[18]:
```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mod
e=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mod
e=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (13): ReLU(inplace=True)
    (14): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mo
de=False)
    (15): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (16): ReLU(inplace=True)
    (17): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (18): ReLU(inplace=True)
    (19): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mo
de=False)
    (20): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (21): ReLU(inplace=True)
    (22): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (23): ReLU(inplace=True)
    (24): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mo
de=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

In [ ]:

In [19]:
```
class ParticleModel(nn.Module):
    def __init__(self, model, freeze = False, out_features = 2, channels = 2
        super().__init__()
        self.backbone = model
        self.freeze = freeze

        self.channels = channels
        self.height = height
        self.width = width

        self.out_features = out_features
        self.layer_norm = nn.LayerNorm([self.channels, self.height, self.wic

        if self.freeze:
            for param in self.backbone.parameters():
                param.requires_grad_(False)

        in_ = self.backbone.classifier[-1].in_features
        self.backbone.classifier[-1] = nn.Linear(in_features = in_, out_feat

    def forward(self, x):
        x = self.layer_norm(x)
        return self.backbone(x)
```

In [20]:
```
def initialize_weights(model):
    for (name, weights) in filter(lambda x: x[1].requires_grad, model.named_
        if name.split(".")[1] not in ["fc", "conv1"]:
            continue
        try:
            nn.init.kaiming_normal_(weights)
        except:
            nn.init.normal_(weights, 0., 0.05)

    return model
```

In this notebook, the pretrained weights will be finetuned. This is in contrast to the previous one, where the weights were kept frozen. Also, the learning rate is increased from 1e-4 to 1e-3.

**Update 1**: A learning rate of 1e-3 may be too small for non-frozen weights. I will now attempt to freeze the weights and leave the learning rate as is. Freezing the weights might even speed up training.

**Update 2**: Freezing all the weights seem to have reduced performance. This might be due to the fact that the data we have here is not actualy a set of images, even though they seem so. It appears I might have to unfreeze the weights and increase the learning rate a bit.

**Update 3**: Applying the ideas from **Update 2** led to even worse performance! Returning the state of training to **Update 1**...

In [21]:
```
EPOCHS = 20
l2_lambda = 1e-4

criterion = nn.CrossEntropyLoss().to(DEVICE)

# Optimizer hyperparameters
LR = 1e-3
FACTOR = 100
AMSGRAD = False
BETAS = (.9, .999)
FREEZE = False
```

In [22]:
```
model = ParticleModel(
    model = model,
    freeze = FREEZE,
    channels = 3,
    height = 125,
    width = 125
).to(DEVICE)
```

In [23]:
```
model = initialize_weights(model)
```

In [24]:
```
# Instantiate optimizer
opt = optim.AdamW(
    params = [{
        "params" : model.backbone.parameters(),
        "lr": LR
    }],
    lr=LR/FACTOR,
    amsgrad = AMSGRAD,
    betas = BETAS,
    weight_decay = l2_lambda,
    fused = True
)

# scheduler = optim.lt_scheduler.Cos
```

In [25]:
```
from sklearn.metrics import accuracy_score
```

In [26]:
```
def training_loop(epochs, model, optimizer):
    TRAIN_LOSSES, TEST_LOSSES = [], []
    TRAIN_ACCS, TEST_ACCS = [], []

    for epoch in range(1, epochs + 1):
        train_losses, test_losses = [], []
        train_accs, test_accs = [], []

        model.train() # Set up training mode

        for batch in iter(train_dl):
            # X, y = collate_function(batch)
            X, y = batch
            X, y = X.squeeze().to(DEVICE), y.view(-1).to(DEVICE)
```

```
            y_pred = model(X)

            train_loss = criterion(y_pred, y.to(torch.long)) # Compare actua
            train_loss.backward() # Backpropagate the loss

            optimizer.step()
            optimizer.zero_grad()

            train_losses.append(train_loss.detach().item())

            train_acc = accuracy_score(y.cpu().numpy(), y_pred.max(dim = -1)
            train_accs.append(train_acc)

        # Persist model architecture
        torch.save(model.state_dict(), f"epoch_{epoch}_vgg_model.pt")

        with torch.no_grad(): # Turn off computational graph
            model.eval() # Set model to evaluation mode
            for batch in iter(test_dl):
                # X_, y_ = collate_function(batch)
                X_, y_ = batch
                X_, y_ = X_.squeeze().to(DEVICE), y_.view(-1).to(DEVICE)

                y_pred_ = model(X_)

                test_loss = criterion(y_pred_, y_.to(torch.long)) # Compare
                test_losses.append(test_loss.item())

                test_acc = accuracy_score(y_.cpu().numpy(), y_pred_.max(dim
                test_accs.append(test_acc)

        avg_train_loss = sum(train_losses) / len(train_losses)
        avg_test_loss = sum(test_losses) / len(test_losses)

        avg_train_acc = sum(train_accs) / len(train_accs)
        avg_test_acc = sum(test_accs) / len(test_accs)

        print(
            f"Epoch: {epoch} | Train loss: {avg_train_loss: .3f} | Test loss
            f"Train accuracy: {avg_train_acc: .3f} | Test accuracy: {avg_tes
        )

        TRAIN_LOSSES.append(avg_train_loss)
        TEST_LOSSES.append(avg_test_loss)

        TRAIN_ACCS.append(avg_train_acc)
        TEST_ACCS.append(avg_test_acc)

    # Clear CUDA cache
    torch.cuda.empty_cache()
    torch.clear_autocast_cache()

    return {
        "loss": [TRAIN_LOSSES, TEST_LOSSES],
        "accuracy": [TRAIN_ACCS, TEST_ACCS],
```

```
        "model": model
    }
```

In [ ]:
```python
# Train VGG-13 with finetuning
model_results = training_loop(epochs = EPOCHS, optimizer = opt, model = mode
```

```
Epoch: 1 | Train loss:  0.605 | Test loss:  0.560 | Train accuracy:  0.699 |
Test accuracy:  0.719 |
Epoch: 2 | Train loss:  0.573 | Test loss:  0.558 | Train accuracy:  0.715 |
Test accuracy:  0.724 |
Epoch: 3 | Train loss:  0.578 | Test loss:  0.558 | Train accuracy:  0.716 |
Test accuracy:  0.725 |
Epoch: 4 | Train loss:  0.569 | Test loss:  0.557 | Train accuracy:  0.720 |
Test accuracy:  0.727 |
Epoch: 5 | Train loss:  0.563 | Test loss:  0.558 | Train accuracy:  0.725 |
Test accuracy:  0.727 |
Epoch: 6 | Train loss:  0.564 | Test loss:  0.556 | Train accuracy:  0.723 |
Test accuracy:  0.724 |
Epoch: 7 | Train loss:  0.559 | Test loss:  0.558 | Train accuracy:  0.727 |
Test accuracy:  0.725 |
Epoch: 8 | Train loss:  0.559 | Test loss:  0.552 | Train accuracy:  0.728 |
Test accuracy:  0.728 |
Epoch: 9 | Train loss:  0.556 | Test loss:  0.552 | Train accuracy:  0.729 |
Test accuracy:  0.729 |
Epoch: 10 | Train loss:  0.554 | Test loss:  0.551 | Train accuracy:  0.731
| Test accuracy:  0.728 |
Epoch: 11 | Train loss:  0.554 | Test loss:  0.556 | Train accuracy:  0.732
| Test accuracy:  0.728 |
Epoch: 12 | Train loss:  0.553 | Test loss:  0.555 | Train accuracy:  0.732
| Test accuracy:  0.733 |
Epoch: 13 | Train loss:  0.556 | Test loss:  0.552 | Train accuracy:  0.730
| Test accuracy:  0.727 |
Epoch: 14 | Train loss:  0.556 | Test loss:  0.569 | Train accuracy:  0.731
| Test accuracy:  0.723 |
Epoch: 15 | Train loss:  0.550 | Test loss:  0.550 | Train accuracy:  0.734
| Test accuracy:  0.730 |
```

In [ ]:
```python
# Persist model
torch.save(model_results["model"].state_dict(), "final_epoch_vgg_model.pt")
```

In [ ]:
```python
def visualize_results(history, key = None):
    if key is not None:
        TRAIN_RESULTS, TEST_RESULTS = history[key]

        plt.figure(figsize = (10, 3))

        plt.plot(range(EPOCHS), TRAIN_RESULTS, label = f"Training {key.capit
        plt.plot(range(EPOCHS), TEST_RESULTS, label = f"Test {key.capitalize

        plt.xlabel("Epochs")
        plt.ylabel(key.capitalize())

        plt.title(key.capitalize() + " Evolution for Train and Test Splits",

        plt.legend()
        plt.show(); plt.close("all")
```

```python
    else:
        TRAIN_LOSSES, TEST_LOSSES = history["loss"]
        TRAIN_ACCS, TEST_ACCS = history["accuracy"]

        fig, ax = plt.subplots(1, 2, figsize = (15, 4))

        ax[0].plot(range(EPOCHS), TRAIN_LOSSES, label = "Training Loss")
        ax[0].plot(range(EPOCHS), TEST_LOSSES, label = "Test Loss")

        ax[0].set_xlabel("Epochs")
        ax[0].set_ylabel("Loss")

        ax[0].set_title("Loss Evolution for Train and Test Splits", fontsize

        ax[1].plot(range(EPOCHS), TRAIN_ACCS, label = "Training Accuracy")
        ax[1].plot(range(EPOCHS), TEST_ACCS, label = "Test Accuracy")

        ax[1].set_xlabel("Epochs")
        ax[1].set_ylabel("Accuracy")

        ax[1].set_title("Accuracy Evolution for Train and Test Splits", font

        plt.legend()
        plt.show(); plt.close("all")

    return
```

In [ ]:

In [ ]:

```python
# VGG-13 with finetuning
visualize_results(model_results)
```

In [ ]:

In [ ]: