DATA PREPARATION

- Considering the CIFAR - 10 dataset
- Converting this dataset into binary dataset [which have label as 0 or 1]
- Converting this dataset into unbalalanced dataset by converting labels '0' as '0' and rest eveything label as '1'

```
from torchvision.datasets import CIFAR10
import torchvision.transforms as transforms
import os
import pandas as pd
from torchvision.transforms import ToPILImage

transform = transforms.ToTensor()

train_set = CIFAR10(root='./data', train=True, download=Tru
test_set = CIFAR10(root='./data', train=False, download=Tru

```

```
100%|██████████| 170M/170M [00:15<00:00, 11.2MB/s]
```

```
def label_adjustment(dataset):
    results = []
    for img, label in dataset:
        if label == 0:
            results.append([img, label])
        if label == 1 or label == 2:
             results.append([img, 1])
    return results
```

```
train_data = label_adjustment(train_set)
test_data  = label_adjustment(test_set)
```

```
save_root = '/content/drive/MyDrive/cifar10_binary'
os.makedirs(save_root, exist_ok = True)
to_pil = ToPILImage()

def save_images_and_make_csv(data, split_name):
    dir = os.path.join(save_root, split_name)
    os.makedirs(dir, exist_ok = True)
    rows = []
    for idx, (img_tensor, label) in enumerate(data):
        img_path = os.path.join(dir, f'{idx}.png')
        to_pil(img_tensor).save(img_path)
        rows.append([img_path, label])

    df = pd.DataFrame(rows, columns=["MD5HASH", "LABEL"])
    df.to_csv(os.path.join(save_root, f"{split_name}.csv"),
    print(f"{split_name}.csv saved with {len(rows)} entries

# Save both splits
save_images_and_make_csv(train_data, "train")
save_images_and_make_csv(test_data, "test")

```

```
train.csv saved with 15000 entries.
test.csv saved with 3000 entries.
```

```
!ls /content/drive/MyDrive/cifar10_binary
```

```
test  test.csv  train  train.csv
```

IMPORTING LIBRARIES AND SETTING CONGIURATION PARAMETERS

```
from torch.utils.data import Dataset, DataLoader
import torch
import torch.nn as nn
import torch.nn.functional as F
from sklearn.metrics import classification_report
from transformers import AutoImageProcessor, AutoModelForIm
from torch.optim.lr_scheduler import StepLR
# note: PIL stands for pillow; to install type "pip3 instal
from PIL import Image
from datetime import datetime
from torchvision.transforms import Compose, Resize, RandomR
import pandas as pd
import numpy as np
import os, gc
```

```
2025-06-19 07:17:03.011678: E external/local_xla/xla/stream_executor/cuda/cud
WARNING: All log messages before absl::InitializeLog() is called are written
E0000 00:00:1750317423.209937      70 cuda_dnn.cc:8310] Unable to register cu
E0000 00:00:1750317423.267046      70 cuda_blas.cc:1418] Unable to register c
```

```
# Check for CUDA and MPS availability, set the device accor
if torch.backends.mps.is_available():
    device = torch.device("mps")
    # setting environment variables, need to run training i
    os.environ['PYTORCH_MPS_HIGH_WATERMARK_RATIO'] = '0.0'
    os.environ['PYDEVD_DISABLE_FILE_VALIDATION'] = '1'
    print("Using MPS as the device.")
else:
    if torch.cuda.is_available():
    # the syntax 'cuda:3' used to point a specific GPU from
    # 'cuda' points to first GPU from the cluster of GPUs
        device = torch.device("cuda")
        print("Using CUDA as the device.")
    else:
        device = torch.device("cpu")
        print("Using CPU as the device.")
```

```
Using CUDA as the device.
```

```
TRAINING_DATA                         = '/content/drive/MyD
TESTING_DATA                          = '/content/drive/MyD
BATCH_SIZE                            = 32#256
WORKERS                               = 4
PIN_MEMORY                            = True
MIXING                                = True
MODEL_NAME                            = "facebook/dinov2-ba
RESULTS                               = 'results'
EPOCHS                                = 4
BEST_MODEL                            = None
PRETRAINING                           = False
LEARNING_RATE                         = 5e-5
L2_PENALTY                            = 1e-5
GAMMA                                 = 0.1
STEPSIZE                              = 3
SAVE_CHECKPOINTS                      = True
MIN_LOSS                              = float('inf')
MODEL_SAVED                           = f'{RESULTS}/bestmod
THRESHOLD                             = 0.5
OUTPUT_DIM                            = 1

```

PERFORMANCE METRIC [PRECISION & RECALL]

```
def calculate_classification_accuracy(loader, model):
    model.eval()  # Set the model in evaluation mode
    LABELS = []
    PREDICTIONS = []

    with torch.no_grad():
        for images, labels in loader:
            # Move to device and cast to float32
            images, labels = images.to(device), labels.to(d
            probabilities = model(images).squeeze()
            # Predictions based on the threshold
            prediction = torch.where(probabilities > THRESH
            LABELS.extend(labels.tolist())
            PREDICTIONS.extend(prediction.tolist())
    return classification_report(LABELS, PREDICTIONS)

```

DINO MODEL [WITH ADDITIONAL CLASSIFICATION HEAD]

```
class Network(nn.Module):
    def __init__(self):
        super(Network, self).__init__()

        # taking processor for necessary substitions, if ne
        self.processor = AutoImageProcessor.from_pretrained
        self.model = Dinov2Model.from_pretrained(MODEL_NAM
        self.pretrained_model_last_dim = self.model.layern

        # Additional classification head used for downstre
        self.cls_head = nn.Sequential(
            nn.Linear(self.pretrained_model_last_dim, OUTPU
            nn.Sigmoid(),
        )

        self.initialize_weights() # weights initialization

    def initialize_weights(self):
        torch.manual_seed(444)
        for layer in self.cls_head:
            if isinstance(layer, nn.Linear):
                nn.init.kaiming_uniform_(layer.weight, non
                nn.init.zeros_(layer.bias)
                nn.init.zeros_(layer.bias)
                print(f"kaiming_uniform_ Initialization: {


    def forward(self, x):
        x = self.model(x).last_hidden_state[:, 0]
        x = self.cls_head(x)
        return x


```

```
model = Network()

model.to(device)


```

```
preprocessor_config.json:   0%|          | 0.00/436 [00:00<?, ?B/s]
Using a slow image processor as `use_fast` is unset and a slow processor was
config.json:   0%|          | 0.00/548 [00:00<?, ?B/s]
model.safetensors:   0%|          | 0.00/346M [00:00<?, ?B/s]
kaiming_uniform_ Initialization: Linear
Network(
  (model): Dinov2Model(
    (embeddings): Dinov2Embeddings(
      (patch_embeddings): Dinov2PatchEmbeddings(
        (projection): Conv2d(3, 768, kernel_size=(14, 14), stride=(14, 14))
      )
      (dropout): Dropout(p=0.0, inplace=False)
    )
    (encoder): Dinov2Encoder(
      (layer): ModuleList(
        (0-11): 12 x Dinov2Layer(
          (norm1): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
          (attention): Dinov2Attention(
```

DATA LOADER WITH UPSAMPLING

```python

class MD5HASHDataset(Dataset):
    def __init__(self, dataframe):
        self.dataframe = dataframe
        self.images = self.dataframe['MD5HASH'].values
        self.labels = self.dataframe['LABEL'].values
        self.processor = model.processor
        self.mean = self.processor.image_mean
        self.std = self.processor.image_std
        self.interpolation = self.processor.resample

        self.train_transform = Compose([
            Resize(size = (32, 32)),
            #Resize(size = (85, 550)),
            #RandomResizedCrop(size = (224, 224),
            #                  scale = (0.08, 1.0),
            #                  ratio = (0.75, 1.3333),
            #                  interpolation = self.interpo
            #RandomHorizontalFlip(p = 0.5),
            #ColorJitter(brightness = (0.6, 1.4),
            #            contrast = (0.6, 1.4),
            #            saturation = (0.6, 1.4)),
            ToTensor(),
            Normalize(mean = self.mean, std = self.std),
        ])


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

    def __getitem__(self, idx):
        # Load the image from the file path
        image_path = self.images[idx]
        image = self.train_transform(Image.open(image_path)
        # Get the label
        label = torch.tensor(self.labels[idx], dtype=torch.

        return image, label

```

```
def create_training_loader(data_csv = TRAINING_DATA, upsamp
    # Load data
    training_data = pd.read_csv(data_csv)
    print('::: TRAINING DATA DETAILS :::')
    print('- Number of Samples:', training_data.shape[0])
    print('- LABEL DISTRIBUTION: \n',training_data['LABEL']

    # Create dataset and dataloader
    md5hash_dataset = MD5HASHDataset(training_data)
    if upsampling:
        # References:
        # https://pytorch.org/docs/stable/data.html
        # https://towardsdatascience.com/demystifying-pytor
        from torch.utils.data import WeightedRandomSampler
        classes_count = dict(training_data['LABEL'].value_c
        sample_weights = [ 1 / classes_count[i] for i in tr
        sampler = WeightedRandomSampler(weights = sample_we
                                        num_samples = len(t
                                        replacement = True)
        data_loader = DataLoader(md5hash_dataset,
                                 batch_size = BATCH_SIZE,
                                 num_workers = WORKERS,
                                 pin_memory = PIN_MEMORY,
                                 shuffle = False,
                                 sampler = sampler)
    else:
        data_loader = DataLoader(md5hash_dataset,
                                 batch_size = BATCH_SIZE,
                                 num_workers = WORKERS,
                                 pin_memory = PIN_MEMORY,
                                 shuffle = MIXING)

    # Clean memory, :)
    del training_data

    return data_loader


```

```
data_loader = create_training_loader(upsampling = True)

```

```
::: TRAINING DATA DETAILS :::
- Number of Samples: 15000
- LABEL DISTRIBUTION:
 LABEL
1    10000
0     5000
Name: count, dtype: int64
```

OPTIMIZER, SCHEDULER AND LOSS FUNCTION

```
# ## MODEL TRAINING

# Define the optimizer
# Idea borrowed from Research paper titled as "Improving Ge
if PRETRAINING:
    optimizer = torch.optim.SGD(model.parameters(), lr = LE
else:
    optimizer = torch.optim.Adam(model.parameters(), lr = L

# Define a learning rate scheduler
scheduler = StepLR(optimizer, step_size = STEPSIZE, gamma =
# Define the loss function: BCE
criterion = nn.BCELoss()


```

```
# directory creation
os.makedirs(RESULTS, exist_ok = True)
if SAVE_CHECKPOINTS:
    CHECKPOINTDIR = f'{RESULTS}/checkpoints'
    os.makedirs(CHECKPOINTDIR, exist_ok = True)

```

MODEL FINE-TUNING TRAINING

```
# TRAINING LOOP
for epoch in range(EPOCHS):
    print('-'*70)
    # Define the total number of batches in the loader
    total_loss = 0.0

    # setting model stage to training
    model.train()

    for batch_idx, (images, labels) in enumerate(data_loade
        # shifting to MPS
        # Shift to MPS and then cast to float32
        images, labels = images.to(device), labels.to(devic

        # Forward pass
        optimizer.zero_grad()  # Moved this line here to av

        with torch.set_grad_enabled(True):
            # Forward pass
            outputs = model(images).squeeze()  # Squeeze to
            loss = criterion(outputs, labels)

            loss.backward()
            optimizer.step()

        total_loss += loss.item()
        # Explicitly free up GPU memory
        if torch.backends.mps.is_available():
            torch.backends.mps.is_macos13_or_newer.cache_cl
        if torch.cuda.is_available():
            torch.cuda.empty_cache()
        # Run garbage collector to free up CPU memory
        gc.collect()

    print(f"Epoch {epoch + 1}/{EPOCHS}, Loss: {total_loss /
    print('TRAINING DATA')
    print(f'- Performance: \n{calculate_classification_accu
    # Update the learning rate
    scheduler.step()


    if SAVE_CHECKPOINTS:
        timestamp = datetime.now().strftime("%Y%m%d%H%M%S")
        checkpointmodel = '{}/epoch_{}_{}.pth'.format(CHECK
        print('Saving checkpoint: ', checkpointmodel)
        torch.save(model.state_dict(), checkpointmodel)

    # Check if this epoch had the minimum loss
    if total_loss < MIN_LOSS:
        MIN_LOSS = total_loss
        best_model = model.state_dict()
        # Save the best model
        if best_model is not None:
            print('Saving Best Model: ', MODEL_SAVED)
            torch.save(best_model, MODEL_SAVED)

################################

```

```
----------------------------------------------------------------------
Epoch 1/4, Loss: 0.4652398204101301
TRAINING DATA
- Performance:
              precision    recall  f1-score   support

         0.0       0.88      0.90      0.89      7477
         1.0       0.90      0.87      0.89      7523

    accuracy                           0.89     15000
   macro avg       0.89      0.89      0.89     15000
weighted avg       0.89      0.89      0.89     15000

Saving checkpoint:  results/checkpoints/epoch_1_20250619072045.pth
Saving Best Model:  results/bestmodel.pth
----------------------------------------------------------------------
Epoch 2/4, Loss: 0.18003429818366254
TRAINING DATA
- Performance:
              precision    recall  f1-score   support

         0.0       0.92      0.98      0.95      7466
         1.0       0.98      0.91      0.94      7534

    accuracy                           0.94     15000
   macro avg       0.95      0.94      0.94     15000
weighted avg       0.95      0.94      0.94     15000

Saving checkpoint:  results/checkpoints/epoch_2_20250619072407.pth
Saving Best Model:  results/bestmodel.pth
----------------------------------------------------------------------
Epoch 3/4, Loss: 0.12192414922000312
TRAINING DATA
- Performance:
              precision    recall  f1-score   support

         0.0       0.97      0.95      0.96      7550
         1.0       0.95      0.97      0.96      7450

    accuracy                           0.96     15000
   macro avg       0.96      0.96      0.96     15000
```

```
weighted avg       0.96      0.96      0.96     15000

Saving checkpoint:  results/checkpoints/epoch_3_20250619072732.pth
Saving Best Model:  results/bestmodel.pth
----------------------------------------------------------------------
Epoch 4/4, Loss: 0.05339157602428071
TRAINING DATA
- Performance:
              precision    recall  f1-score   support

         0.0       0.99      1.00      0.99      7483
         1.0       1.00      0.99      0.99      7517

    accuracy                           0.99     15000
   macro avg       0.99      0.99      0.99     15000
weighted avg       0.99      0.99      0.99     15000
```

```
model.load_state_dict(torch.load(MODEL_SAVED, weights_only
```

```
<All keys matched successfully>
```

RESULTS ON ORIGINAL TRAINING AND TEST DATA

```
data_loader = create_training_loader() # without upsampling

# getting Best model performance on training, validation an
print('TRAINING DATA')
print(f'- Performance: \n{calculate_classification_accuracy
```

```
::: TRAINING DATA DETAILS :::
- Number of Samples: 15000
- LABEL DISTRIBUTION:
 LABEL
1    10000
0     5000
Name: count, dtype: int64
TRAINING DATA
- Performance:
              precision    recall  f1-score   support

         0.0       0.98      1.00      0.99      5000
         1.0       1.00      0.99      0.99     10000

    accuracy                           0.99     15000
   macro avg       0.99      0.99      0.99     15000
weighted avg       0.99      0.99      0.99     15000
```

```
testset_loader = create_training_loader(data_csv = TESTING_

print('TESTING DATA')
print(f'- Performance: \n{calculate_classification_accuracy

```

```
::: TRAINING DATA DETAILS :::
- Number of Samples: 3000
- LABEL DISTRIBUTION:
 LABEL
1    2000
0    1000
Name: count, dtype: int64
TESTING DATA
- Performance:
              precision    recall  f1-score   support

         0.0       0.90      0.88      0.89      1000
         1.0       0.94      0.95      0.95      2000

    accuracy                           0.93      3000
   macro avg       0.92      0.92      0.92      3000
weighted avg       0.93      0.93      0.93      3000
```