Assignment- 2 Report

Group- K

Khushboo Singhania(19CS30023)
Hardik Soni (20CS30023)

Problem Statement

- In this assignment, two problems with different datasets were given and we were to use different models and different loss functions to perform certain operations.
- The first problem is an image classification problem in which we use two different models, check their percentage accuracy and compare their performance.
- The second problem is an image segmentation problem in which we use a U-Net architecture and Negative log likelihood loss along with DICE loss to compute DICE score and mean average precision.

1. Image Classification Problem

A. Data Download: CIFAR10
B. Data Preparation
   i. There are 10 classes and six thousand images are given in each class.
   ii. Divide the train data into train-val splits. 80% for training and 20% for validation. The split should be in such a way that for each class 80% images should be in the train set and 20% images in the validation set.
   iii. Use 5 different data augmentation technique and show that data augmentation improves the performance.
C. Deep Learning Model
   i. Use pre-trained Resnet-50 from Pytorch Models for training.
   ii. Use pre-trained EfficientNet-b0 from Pytorch Models for training.
D. Loss Function
   i. Negative log likelihood (NLL) Loss
E. Test Metric
   i. Percentage Accuracy on test set.
   ii. Compare the performance of these 2 model architectures.

2. Image Segmentation Problem

A. Data Download: Cityscapes
B. Data Preparation:
   a. The dataset consists of around 5000 fine annotated images and 20000 coarse annotated ones.
   b. Randomly select 3500 images for training and 1500 images for test from the fine annotated images.
C. Deep Learning Model Use U-Net architecture for segmentation

D. Loss Function Negative log likelihood (NLL) Loss along with DICE loss

E. Test Metric DICE Score on test set. Mean Average Precision on test set.

Image Classification Problem

1. Importing Libraries: Libraries required for Classification

[1]:
```
from torch.utils.data import Dataset, DataLoader
import torchvision.transforms as transforms
from IPython.display import clear_output
import torchvision.models as models
import torch.optim as optim
import torch.nn as nn
import matplotlib.pyplot as plt
import seaborn as sns
from tqdm import tqdm
from PIL import Image
import pandas as pd
import torchvision
import numpy as np
import torch
import time
import random
```

2. Importing and transforming data: CIFAR10

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-py
thon.tar.gz to ./data/cifar-10-python.tar.gz

100%|██████████| 170498071/170498071 [00:04<00:00, 402489
46.45it/s]

Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
Number of classes: 10
```

2. Data Preparation using StratifiedShuffleSplit: We split the dataset into train and validation sets and then print the distribution of labels in both train_dataset and val_dataset.

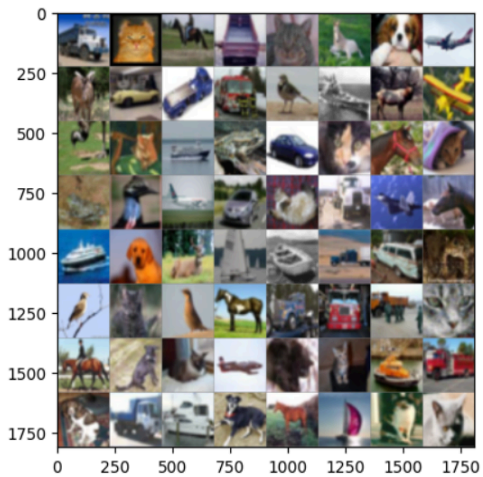
[8]:
```
from sklearn.model_selection import StratifiedShuffleSplit

# Get the labels of the CIFAR-10 dataset
labels = train_dataset.targets

# Use StratifiedShuffleSplit to split the dataset into train and validation sets
sss = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)

# Get the indices for the train and validation sets
train_idx, val_idx = next(sss.split(labels, labels))

# Create Subset objects using the indices
train_subset = torch.utils.data.Subset(train_dataset, train_idx)
val_subset = torch.utils.data.Subset(train_dataset, val_idx)
```

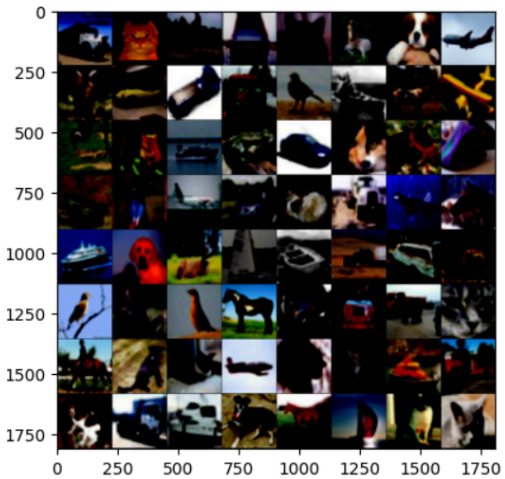
```
Label distribution in train_subset:
Label 0: 4000 samples
Label 1: 4000 samples
Label 2: 4000 samples
Label 3: 4000 samples
Label 4: 4000 samples
Label 5: 4000 samples
Label 6: 4000 samples
Label 7: 4000 samples
Label 8: 4000 samples
Label 9: 4000 samples

Label distribution in val_subset:
Label 0: 1000 samples
Label 1: 1000 samples
Label 2: 1000 samples
Label 3: 1000 samples
Label 4: 1000 samples
Label 5: 1000 samples
Label 6: 1000 samples
Label 7: 1000 samples
Label 8: 1000 samples
Label 9: 1000 samples
```

As we can observe the split is done in such a way that for each class 80% images are in the train set and 20% images in the validation set.

3. Data Visualization:

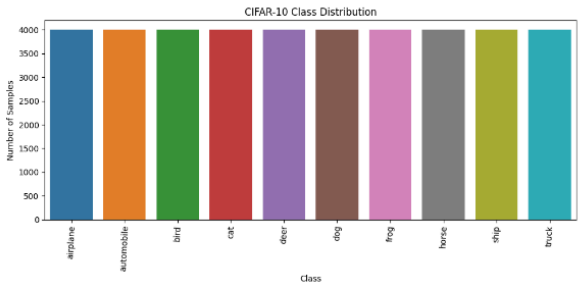
I. **Normalized vs non-normalized:** We use the ***imshow*** function which normalizes the image using numpy.





II. **Class distribution:** We plot the class or label distribution using matplotlib for all the datapoints.

```
/opt/conda/lib/python3.10/site-packages/seaborn/_o
ldcore.py:1765: FutureWarning: unique with argumen
t that is not not a Series, Index, ExtensionArray,
or np.ndarray is deprecated and will raise in a fu
ture version.
  order = pd.unique(vector)
```



4. Downloading pretrained models: RESNET-50 and EfficientNet-50

Defining the Resnet 50 Model with the Sequential definitions of the layers to be used with configuration to use cpu or gpu.

In [7]:

```
model_setup_res50 = models.resnet50(pretrained=True)

model_setup_res50 = model_setup_res50.cuda() if use_cuda else model

num_ftrs = model_setup_res50.fc.in_features
model_setup_res50.fc = torch.nn.Linear(num_ftrs, 1024)
model_setup_res50.fc = nn.Sequential(
    torch.nn.Dropout(0.5),
    torch.nn.Linear(num_ftrs, 1024),
    torch.nn.Dropout(0.2),
    torch.nn.Linear(1024, 512),
    torch.nn.Dropout(0.2),
    torch.nn.Linear(512, 256),
    torch.nn.Dropout(0.2),
    torch.nn.Linear(256, 128),
    torch.nn.Dropout(0.2),
    torch.nn.Linear(128, len(train_dataset.classes))
)

model_setup_res50.fc = model_setup_res50.fc.cuda() if use_cuda else model.fc
```

:

```
import torch.nn as nn
import torchvision.models as models

model_setup_effb50 = models.efficientnet_b0(pretrained=True)
model_setup_effb50.classifier = nn.Linear(1280, 100)

# Determine the total number of layers in the model
total_layers = len(list(model_setup_effb50.children()))

num_layers_to_freeze = total_layers // 2

frozen_layers = 0

for name, param in model_setup_effb50.named_parameters():
    if frozen_layers < num_layers_to_freeze:
        param.requires_grad = False
        frozen_layers += 1
    else:
        break
```

The above are the models resenet and effnet-b50 defined.

5. *def train:* In train function, we define the model, data, loss function, optimizer and predictions. This function returns the average loss and average correct predictions.

In [9]:
```
def train(model, dataloader, loss_fn, optimizer, device, length_dataloader, length_dataset):
    model.train()
    total_loss = 0

    total_correct = 0
    for batch in tqdm(dataloader):
        inputs, labels = batch[0].to(device), batch[1].to(device)
        optimizer.zero_grad()
        outputs = model(inputs)

        loss = loss_fn(outputs, labels)

        loss.backward()
        optimizer.step()

        total_loss += loss.item()

        predictions = outputs.argmax(dim=1)

        correct = (predictions == labels).sum().item()
        total_correct += correct

    return total_loss / length_dataloader, total_correct / length_dataset
```

6. *def compute_accuracy:* This function evaluates the number of correct predictions by total number of predictions giving us the accuracy of the model.

In [10]:
```
def compute_accuracy(dataloader, model):
    model.eval()

    total_correct = 0
    total_count = 0
    with torch.no_grad():
        for batch in tqdm(dataloader):
            inputs, labels = batch[0].to(device), batch[1].to(device)
            outputs = model(inputs)
            predictions = outputs.argmax(dim=1)
            correct = (predictions == labels).sum().item()
            total_correct += correct
            total_count += len(labels)

    accuracy = total_correct / total_count
    return accuracy
```

7. Model initialization and configuration: class **LogSoftmax Loss** is used to calculate the loss function.

In [11]:

```
class LogSoftmaxNLLLoss(nn.Module):
    def __init__(self):
        super(LogSoftmaxNLLLoss, self).__init__()
        self.log_softmax = nn.LogSoftmax(dim=1)
        self.nll_loss = nn.NLLLoss()

    def forward(self, input, target):
        log_probs = self.log_softmax(input)
        return self.nll_loss(log_probs, target)
```

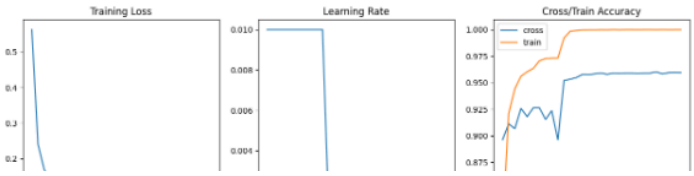
8. Main Training:

I. RESNET-50 Training and Evaluation: We use 30 epoches and the optimizer used is-

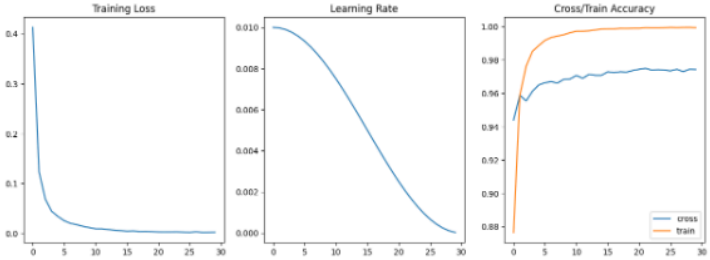
optimizer = optim.SGD(model_resnet.parameters(), lr=0.01, momentum=0.9, weight_decay=5e-4)

```
Epoch 30/30, Train Loss: 0.0012, lr: 0.0001000
Cross Acc: 0.9594, Train Acc: 0.9999
Patience REM :  4

100%|██████████| 157/157 [00:18<00:00,  8.57it/s]

Test Acc: 0.9594
```

II. EfficientNet_B0 Training and Evaluation: We use 30 epoches and the same optimizer as the previous model.




```
Epoch 30/30, Train Loss: 0.0023, lr: 0.0000274
Cross Acc: 0.9743, Train Acc: 0.9994
Patience REM :  8

100%|██████████| 157/157 [00:11<00:00, 13.17it/s]

Test Acc: 0.9743
```



9. *def getPreds*: We use numpy to get model predictions for the architecture.

In [19]:
```
def getPreds(dataloader, model):
    preds = []
    with torch.no_grad():
        for batch in dataloader:
            inputs = batch[0].to(device)
            outputs = model(inputs)
            predictions = outputs.argmax(dim=1)
            preds.extend(predictions.cpu().numpy())

    return preds
```

10. Inference time: We calculate the inference time of the two models in the architecture used.
For RESNET-50, it is 0.06244039535522461
For EfficientNET-50, it is 0.0491490364074707.

Observations: As it can be readily inferred from the above result, test accuracy for model RESNET50 is 95.54% and for model EfficientNet50 is 97.43%.
The performance of model EfficientNet50 is better as it has smaller inference time and higher test accuracy.

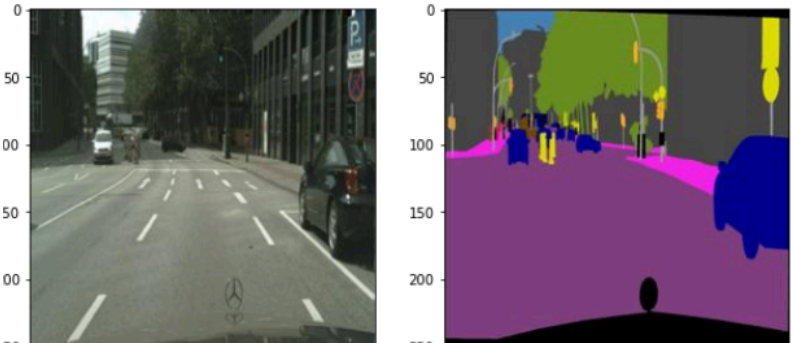
Packages Required: Python3, NumPy, torch, time, torchvision, pandas, matplotlib, seaborn, tqdm, PIL, sklearn.

Image Segmentation Problem

1. Importing and transforming data: The dataset is stored in the following link-
https://www.kaggle.com/dansbecker/cityscapes-image-pairs

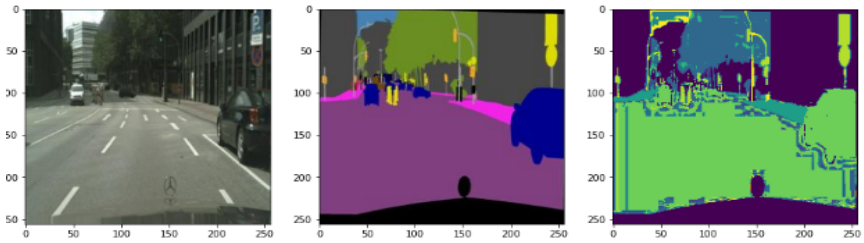
2. Analyze Image: The image is processed such that axis[0] shows cityscape and axis[1] shows labels.

```
<matplotlib.image.AxesImage at 0x7c00943051d0>
```



3. Define labels: We define the label.class as -
label_class = label_model.predict(label.reshape(-1, 3)).reshape(256, 256)
where label.model uses Kmeans on number of classes
The axis[2] shows the label class-

```
<matplotlib.image.AxesImage at 0x7c0093d16dd0>
```



4. Define model: The model used is U-Net architecture for segmentation.

5. Train model: The model is trained using 30 epoches.
I. def mean_average_precision: This function returns the mean avg. precision score.

In [24]:
```
# Define a function to calculate Mean Average Precision (mAP)
def mean_average_precision(y_pred, y_true):
    y_pred = y_pred.detach().cpu().numpy()
    y_true = y_true.detach().cpu().numpy()
    return average_precision_score(y_true, y_pred_proba, average='macro')
```

II. def dice_loss: This function returns the dice loss value.

In [25]:
```
def dice_loss(output, target):
    smooth = 1e-6
    output_flat = output.view(-1)
    target_flat = target.view(-1)
    intersection = (output_flat * target_flat).sum()
    dice = (2. * intersection + smooth) / (output_flat.sum() + target_flat.sum() + smooth)
    return 1 - dice
```

III. def calculate_ap: This function returns the avg. precision.

```python
def calculate_ap(Y_pred, Y):
    total_ap = 0.0
    total_samples = 0

    for i in range(Y_pred.shape[0]):
        # Flatten the 256x256 predictions and labels for each image in the batch
        pred = Y_pred[i].flatten()
        label = Y[i].flatten()
        # Calculate true positive, false positive, and total positive pixels
        tp = torch.sum((pred == label) & (pred != 0)).item()
        fp = torch.sum((pred != label) & (pred != 0)).item()
        total_positive = torch.sum(label != 0).item()
        # Calculate precision for this image
        if tp + fp == 0:
            precision = 0.0
        else:
            precision = tp / (tp + fp)

        # Update total AP and total samples
        total_ap += precision
        total_samples += 1

    # Calculate Average Precision
    ap = total_ap / total_samples
    return ap
```

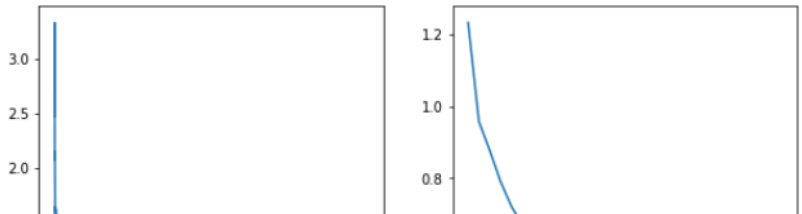
V. The criterion and optimizer used are- Negative Likelihood loss (Cross Entropy) along with Adam Optimiser.

```python
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=lr)
```

```
Epoch 1/30, Loss: 1.0134930610656738, DICE Score: -4.271
98600769043, Mean Average Precision: 0.6520913901136888

Epoch 2/30, Loss: 0.9117664098739624, DICE Score: -4.301
535129547119, Mean Average Precision: 0.6814406261079075

Epoch 3/30, Loss: 0.8191751837730408, DICE Score: -4.406
458377838135, Mean Average Precision: 0.6950820351693058

Epoch 4/30, Loss: 0.7572449445724487, DICE Score: -4.501
906871795654, Mean Average Precision: 0.712199454476812
```

```
Epoch 5/30, Loss: 0.7152444124221802, DICE Score: -4.512
967586517334, Mean Average Precision: 0.7293512612911834

:
:
:
:

Epoch 26/30, Loss: 0.2875829339027405, DICE Score: -4.63
1735801696777, Mean Average Precision: 0.803244760501132
4

Epoch 27/30, Loss: 0.2979801297187805, DICE Score: -4.63
3327484130859, Mean Average Precision: 0.798101258793036
5

Epoch 28/30, Loss: 0.2739851176738739, DICE Score: -4.63
0856990814209, Mean Average Precision: 0.796512835118586
5

Epoch 29/30, Loss: 0.28970977663993835, DICE Score: -4.6
45493984222412, Mean Average Precision: 0.79225483708667
73

Epoch 30/30, Loss: 0.3043934404850006, DICE Score: -4.64
4899845123291, Mean Average Precision: 0.785442338838612
5
```

The following figure shows the plots for step losses and epoch losses.



Observation: As shown by the above results, the loss value keeps decreasing and the DICE score and mean average precision keeps increasing as the number of epoch increases except for a slight variation as epoch approaches 30.

Packages Required: Python3, NumPy, torch, os, torchvision, pandas, matplotlib, random, tqdm, PIL, sklearn.