IMAGE SEGMENTATION USING U-NET

Submitted By:
J Janeika

ABSTRACT

This project applies convolutional neural networks (CNNs) for semantic image segmentation, specifically targeting the Cityscapes dataset, which includes complex urban environments. Preprocessing steps involve resizing images to 256x256 pixels, normalizing them using ImageNet statistics, and applying data augmentation techniques such as random cropping and flipping to enhance model generalization across diverse visual inputs.

A CNN architecture is trained over 15 epochs, utilizing mixed precision to optimize memory usage and computational efficiency. The model outputs pixel-wise classification maps, assigning each pixel a semantic label corresponding to predefined categories like roads, buildings, and vehicles. Key computer vision techniques, such as feature extraction and down sampling, are employed within the model architecture.

Performance evaluation is conducted using standard metrics in computer vision, including Dice loss and Intersection over Union (IoU), which assess the accuracy and overlap between the predicted segmentation masks and ground truth labels. These metrics demonstrate the model's capacity to accurately segment various elements in the urban landscape.

By combining CNN-based feature extraction with data augmentation and mixed precision training, this approach shows high potential for effective semantic segmentation. The workflow is adaptable for other computer vision tasks involving scene understanding and object classification across different datasets.

INTRODUCTION

Semantic segmentation is a computer vision task that involves assigning a label to each pixel in an image, with the goal of segmenting the image into meaningful regions. It is an essential problem for many applications, including autonomous driving, medical imaging, robotics, and scene understanding. By dividing an image into labeled parts, semantic segmentation helps machines interpret complex visual information and make decisions based on it.

In this project, we focus on applying semantic segmentation to the **Cityscapes dataset instead of medical datasets due to large computational time**, which contains images of urban environments with annotated classes like roads, buildings, cars, and pedestrians. The Cityscapes dataset provides an excellent testbed for segmentation tasks due to the complexity of urban scenes, where multiple objects overlap and the boundaries between classes can be difficult to distinguish.

Due to limitations in memory and GPU time, the input image size was reduced from **(256, 256, 3)** to **(128, 128, 3)**. This scaling helps the model process data faster and ensures training can be performed efficiently without exceeding available computational resources. Despite the lower resolution, the model retains enough detail to perform accurate segmentation.

The core of the project revolves around leveraging **convolutional neural networks (CNNs)** to perform pixel-wise classification, categorizing each pixel into predefined classes. Key techniques like **data augmentation** and **mixed precision training** are applied to improve both the accuracy and efficiency of the model. The goal is to build a robust segmentation system capable of handling the complexity of urban scenes and generalizing well to unseen data.

ARCHITECHTURE

The architecture used in this project is based on a convolutional neural network (CNN), specifically designed for the task of semantic segmentation. CNNs are well-suited for image segmentation due to their ability to extract spatial features from images and learn complex relationships between objects at different scales. The network follows a typical encoder-decoder structure, which allows the model to learn both high-level semantics and fine-grained details necessary for pixel-wise classification.

- **Encoder:**

The encoder is responsible for capturing the spatial context of the input images. It consists of several convolutional layers followed by max-pooling layers, which progressively down sample the input while increasing the depth of the feature maps. Each convolutional layer applies a set of filters to the image, extracting important features such as edges, textures, and object boundaries. The max-pooling layers reduce the spatial resolution of the feature maps, allowing the model to capture more global, high-level information about the scene.
The Cityscapes dataset contains highly detailed urban images, so the encoder's task is crucial for learning the spatial relationships between objects like roads, buildings, and vehicles. By downscaling the image, the encoder focuses on extracting meaningful representations that can later be refined by the decoder.

- **Decoder:**

The decoder is designed to upsample the encoded feature maps and generate pixel-level predictions for the segmentation task. This process involves upsampling the feature maps through layers like bilinear upsampling or transposed convolutions, which restore the original image resolution. The decoder learns to assign each pixel a semantic label from predefined categories such as roads, sidewalks, and cars.

In this project, the decoder is particularly effective at handling the complexity of urban scenes, where multiple objects may overlap, and boundaries between classes can be ambiguous. The upsampling process enables the network to recover spatial details that were lost during the encoding phase while preserving the learned high-level semantics.

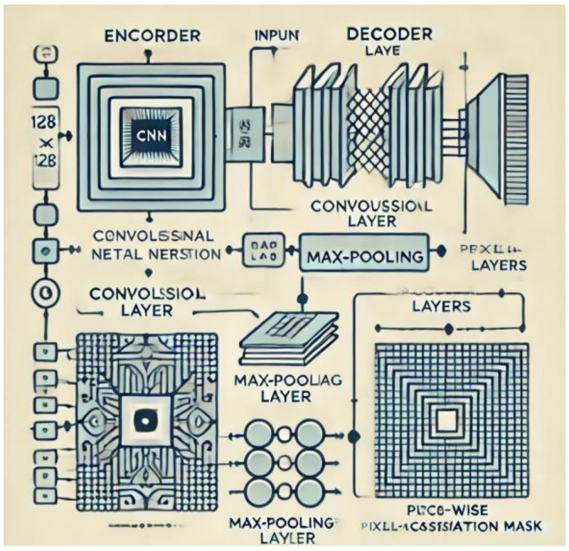
- **Feature Mapping and Preprocessing:**

The network uses the Cityscapes dataset, which includes pixel-level annotations. As part of the data preprocessing pipeline, images are resized to 128x128 pixels due to memory constraints. The input images undergo normalization using the mean and standard deviation values from the ImageNet dataset. This helps stabilize the training process by ensuring that the inputs have similar intensity distributions.

Additionally, data augmentation techniques, such as random cropping and flipping, are applied to improve generalization. These techniques introduce variations in the input data, allowing the network to become more robust to changes in the image's orientation or lighting conditions.

- **Training:**

To optimize the training process, mixed precision training is employed. This allows the model to operate on both 16-bit and 32-bit floating-point numbers, significantly reducing memory usage while speeding up computations. This is particularly beneficial when working with large datasets like Cityscapes, as it allows for faster training without compromising accuracy. The HuggingFace Accelerate library is used to handle the mixed precision training and parallel processing across available hardware resources.



- **Optimization and Loss Function**:

The model is optimized using Dice loss, a popular choice for segmentation tasks because it is effective in handling class imbalance. Dice loss measures the overlap between the predicted segmentation and the ground truth by calculating the ratio of the intersection to the union of the predicted and true regions. This ensures that the model focuses on correctly classifying both common and rare classes, such as vehicles or pedestrians.

Additionally, Intersection over Union (IoU), another common metric in segmentation tasks, is used to evaluate the model's performance during validation. IoU measures the extent of overlap between the predicted and ground-truth masks, making it a reliable metric for assessing how well the model can segment objects with clear boundaries.

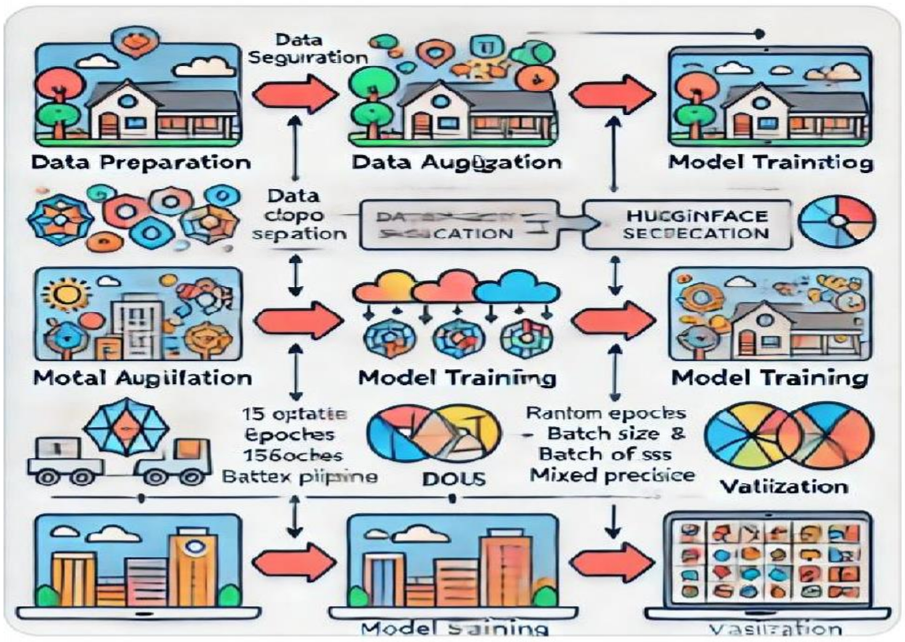
- **Scalability and Efficiency:**

The final architecture is highly efficient and scalable, capable of processing large urban scenes while maintaining high accuracy. The use of mixed precision training ensures that the model can be trained within limited memory constraints, while the encoder-decoder structure provides a flexible approach to handling complex scene segmentation. The network is adaptable and can be fine-tuned for other segmentation tasks or extended to higher-resolution images as needed.

WORKFLOW

The workflow for this semantic segmentation project is structured into several stages, starting from data preparation to model evaluation. Each step is carefully designed to ensure that the model can efficiently process and segment urban scenes from the Cityscapes dataset.

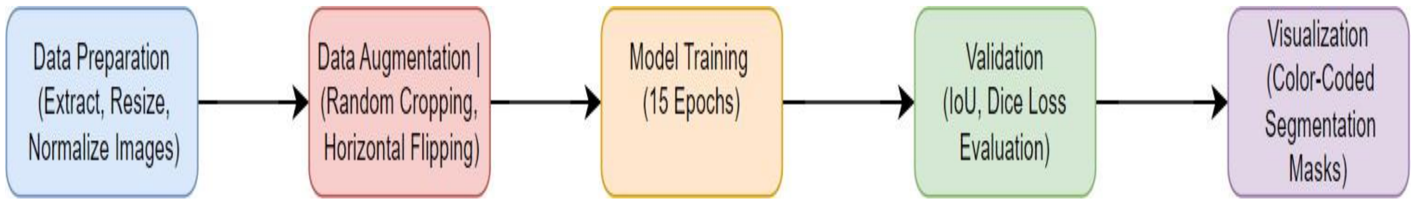
1. Data Preparation:
The Cityscapes dataset is first extracted and divided into training (2,975 images) and validation (500 images) sets. Each image is preprocessed by resizing it to a uniform size of 256x256 pixels to standardize input dimensions. Furthermore, the images are normalized using the mean and standard deviation values from the ImageNet dataset to stabilize the learning process.



2. Data Augmentation:
To improve the generalization of the model, data augmentation techniques are applied. These include:
- Random Cropping: Crops random portions of the image, helping the model learn better from different parts of the scene.
- Horizontal Flipping: Mirrors the image horizontally, which is especially useful for urban scenes where symmetry in object placement (like roads and buildings) is common.

3. Model Training:
The CNN model is trained for 15 epochs using a batch size of 8. Each batch is passed through the network, and the predictions are compared to the ground truth masks using Dice loss. The training is optimized using the Adam optimizer, which is known for its fast convergence.

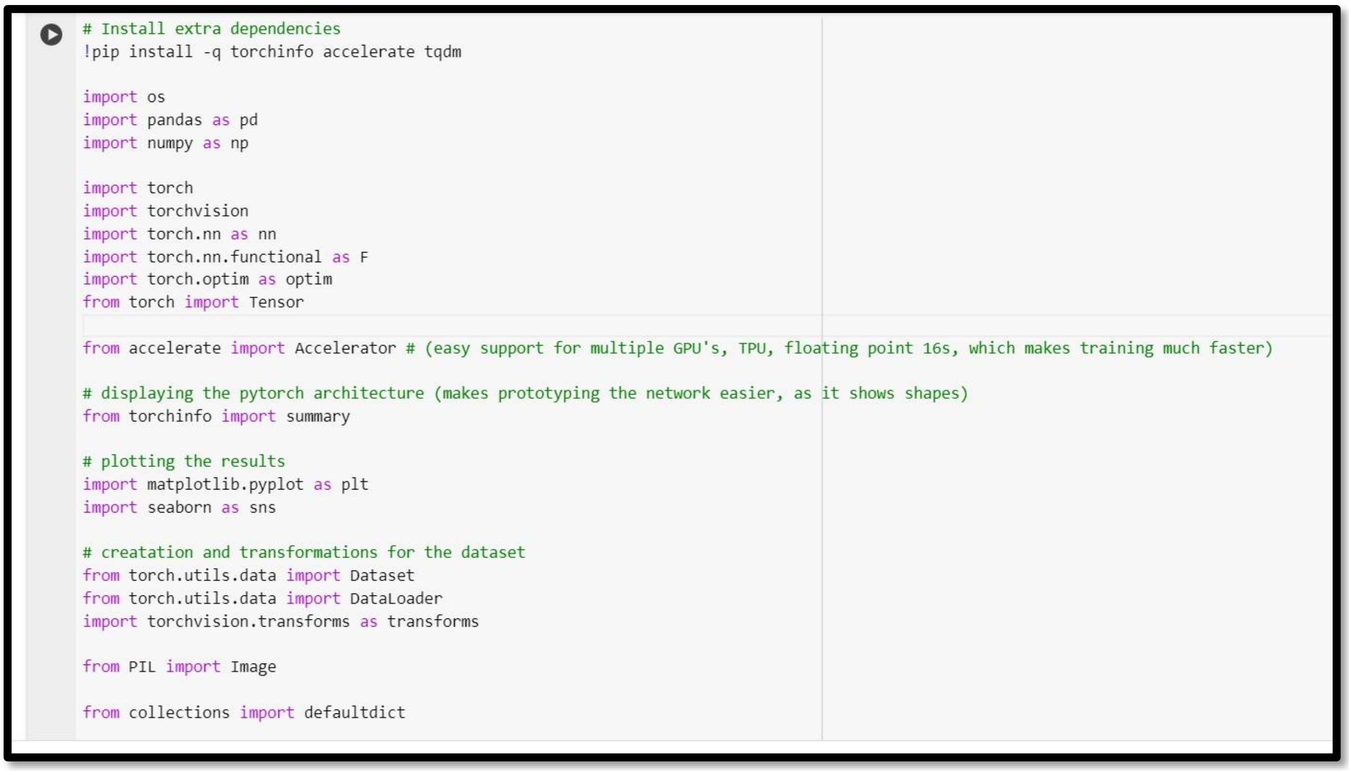
The HuggingFace Accelerate library is used to support mixed precision training and ensure efficient multi-GPU or TPU usage when available. This allows the model to leverage hardware accelerations to speed up the learning process.


Data Preparation (Extract, Resize, Normalize Images)
Data Augmentation | (Random Cropping, Horizontal Flipping)
Model Training (15 Epochs)
Validation (IoU, Dice Loss Evaluation)
Visualization (Color-Coded Segmentation Masks)

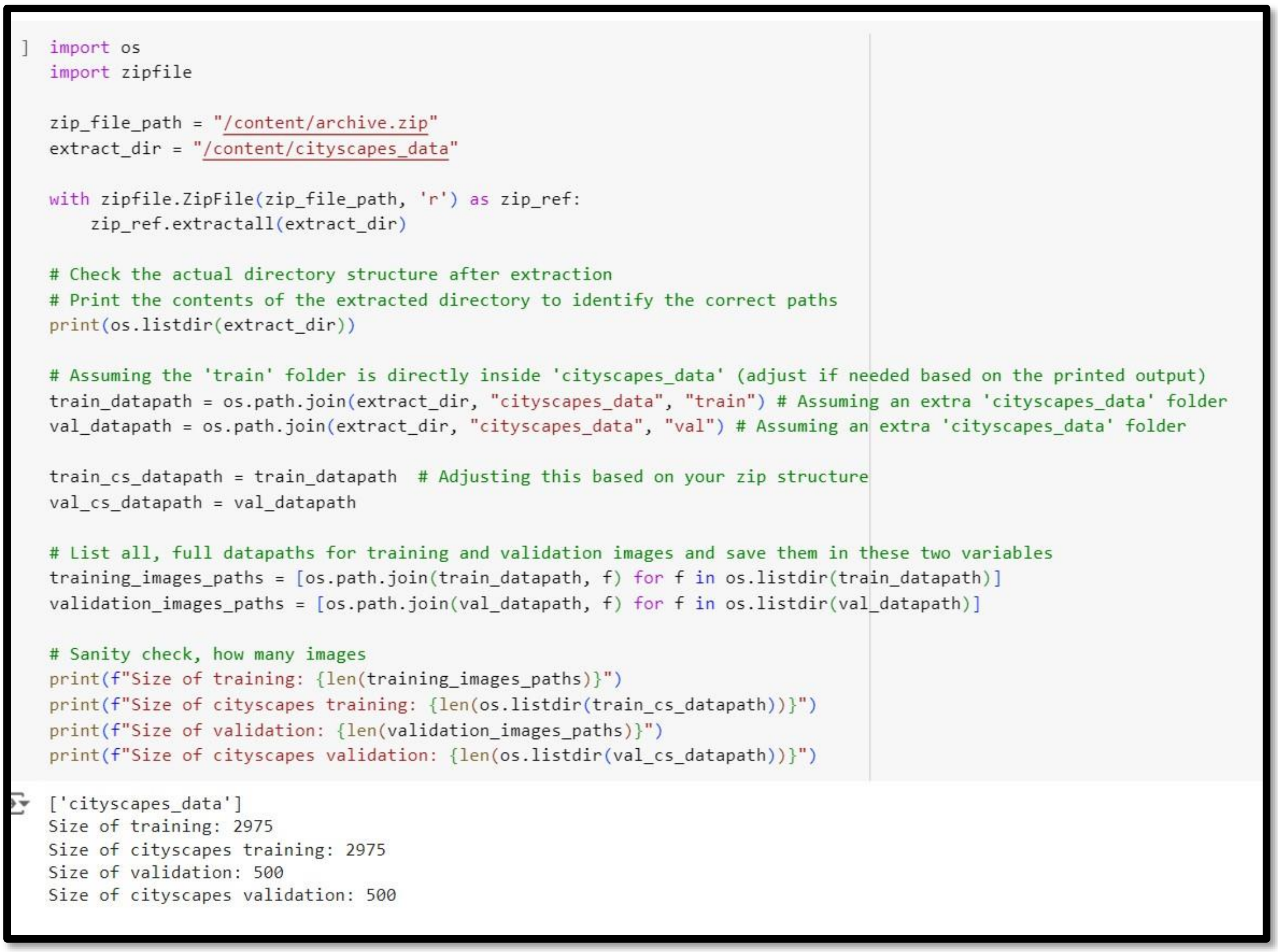
4. **Validation**
   After each training epoch, the model's performance is evaluated on the validation set. The evaluation is based on metrics like IoU and Dice loss, which are calculated to monitor the overlap between predicted segmentation maps and ground-truth labels.

5. **Visualization**
   The results of the segmentation are visualized at regular intervals. Color-coded segmentation masks are displayed side-by-side with the original images and ground-truth annotations, providing visual feedback on how well the model is performing.

RESULTS

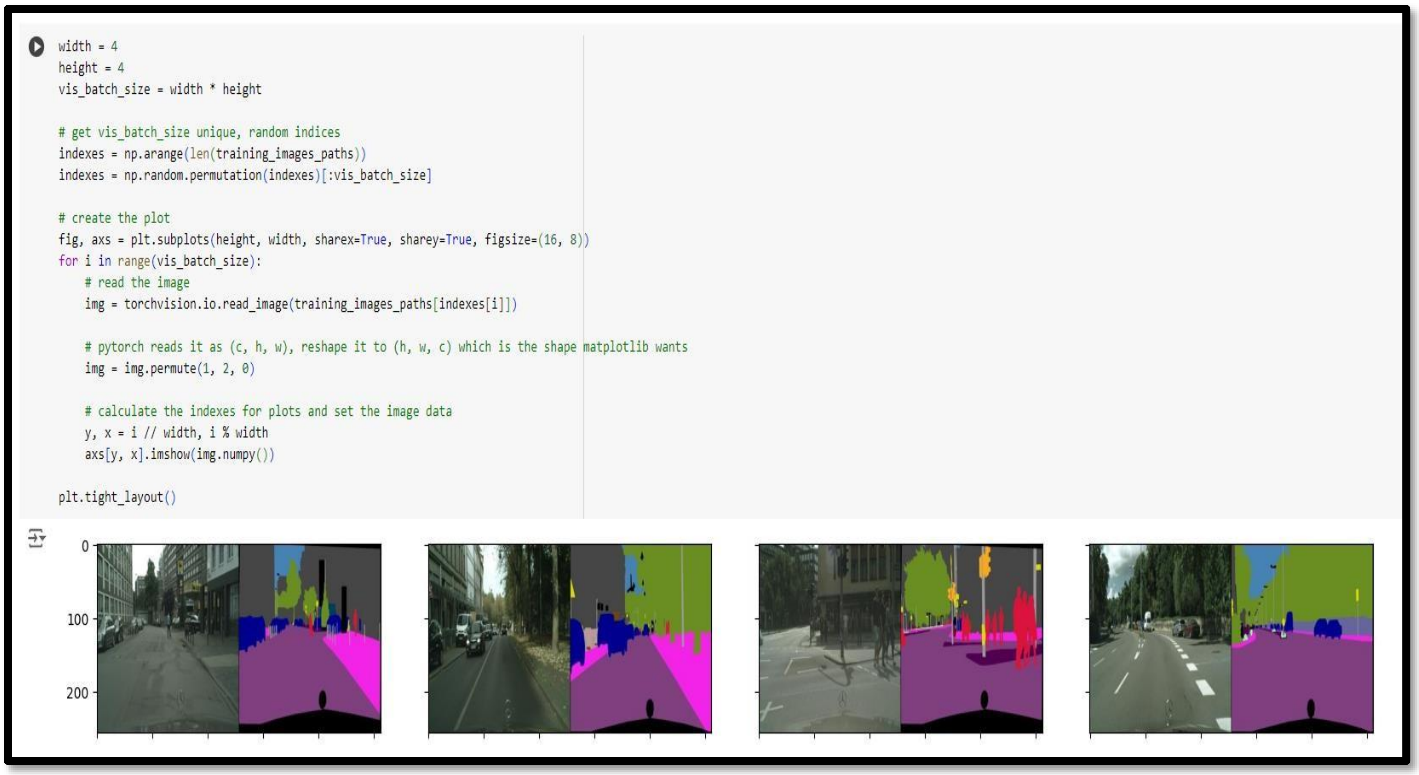


```
# Install extra dependencies
!pip install -q torchinfo accelerate tqdm

import os
import pandas as pd
import numpy as np

import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch import Tensor

from accelerate import Accelerator # (easy support for multiple GPU's, TPU, floating point 16s, which makes training much faster)

# displaying the pytorch architecture (makes prototyping the network easier, as it shows shapes)
from torchinfo import summary

# plotting the results
import matplotlib.pyplot as plt
import seaborn as sns

# creatation and transformations for the dataset
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
import torchvision.transforms as transforms

from PIL import Image

from collections import defaultdict
```

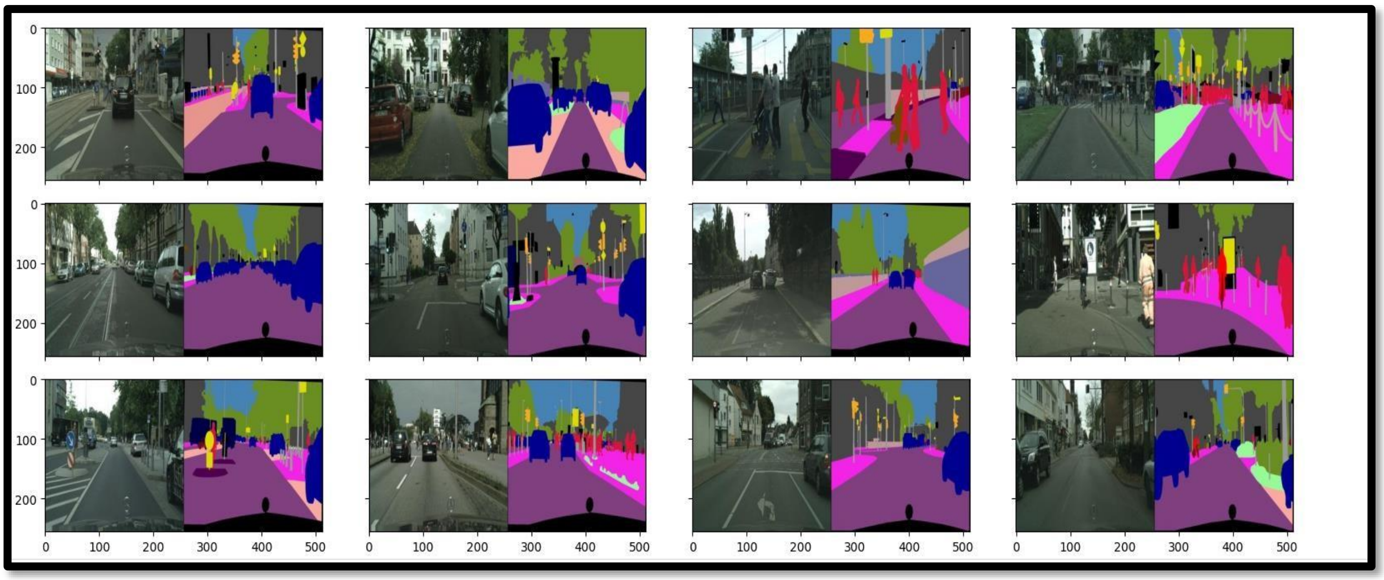


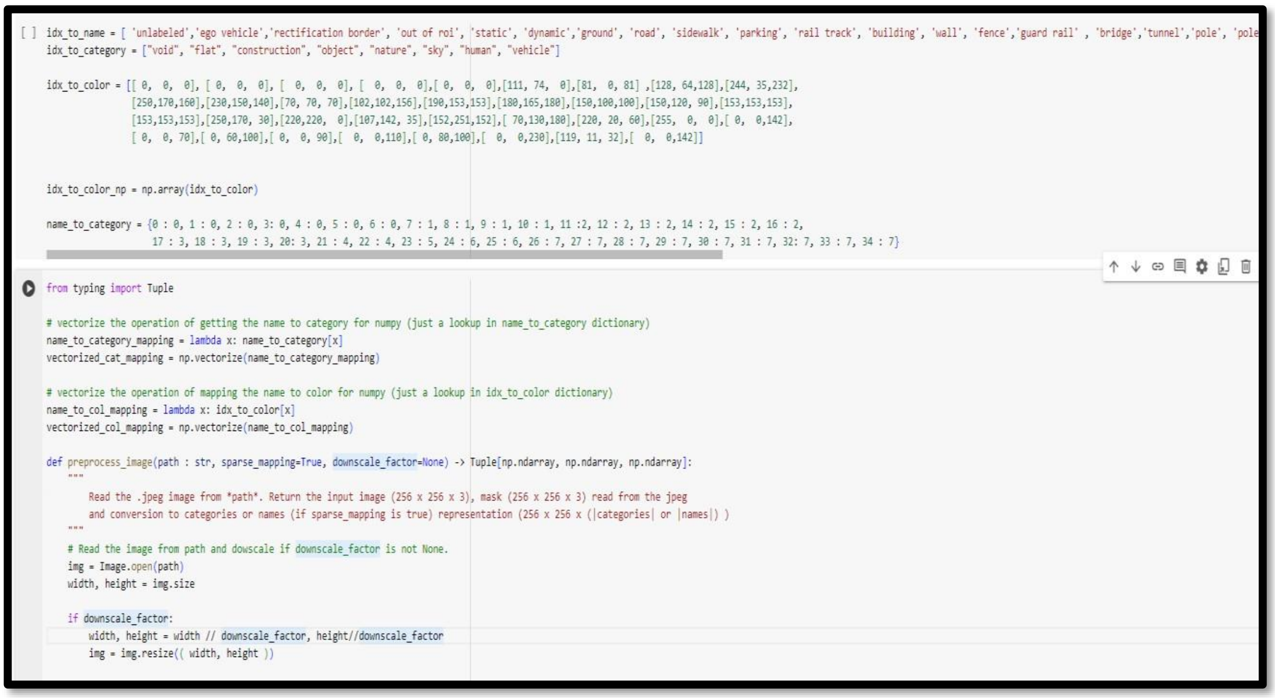
```
] import os
  import zipfile

  zip_file_path = "/content/archive.zip"
  extract_dir = "/content/cityscapes_data"

  with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
      zip_ref.extractall(extract_dir)

  # Check the actual directory structure after extraction
  # Print the contents of the extracted directory to identify the correct paths
  print(os.listdir(extract_dir))

  # Assuming the 'train' folder is directly inside 'cityscapes_data' (adjust if needed based on the printed output)
  train_datapath = os.path.join(extract_dir, "cityscapes_data", "train") # Assuming an extra 'cityscapes_data' folder
  val_datapath = os.path.join(extract_dir, "cityscapes_data", "val") # Assuming an extra 'cityscapes_data' folder

  train_cs_datapath = train_datapath  # Adjusting this based on your zip structure
  val_cs_datapath = val_datapath

  # List all, full datapaths for training and validation images and save them in these two variables
  training_images_paths = [os.path.join(train_datapath, f) for f in os.listdir(train_datapath)]
  validation_images_paths = [os.path.join(val_datapath, f) for f in os.listdir(val_datapath)]

  # Sanity check, how many images
  print(f"Size of training: {len(training_images_paths)}")
  print(f"Size of cityscapes training: {len(os.listdir(train_cs_datapath))}")
  print(f"Size of validation: {len(validation_images_paths)}")
  print(f"Size of cityscapes validation: {len(os.listdir(val_cs_datapath))}")
```

```
['cityscapes_data']
Size of training: 2975
Size of cityscapes training: 2975
Size of validation: 500
Size of cityscapes validation: 500
```

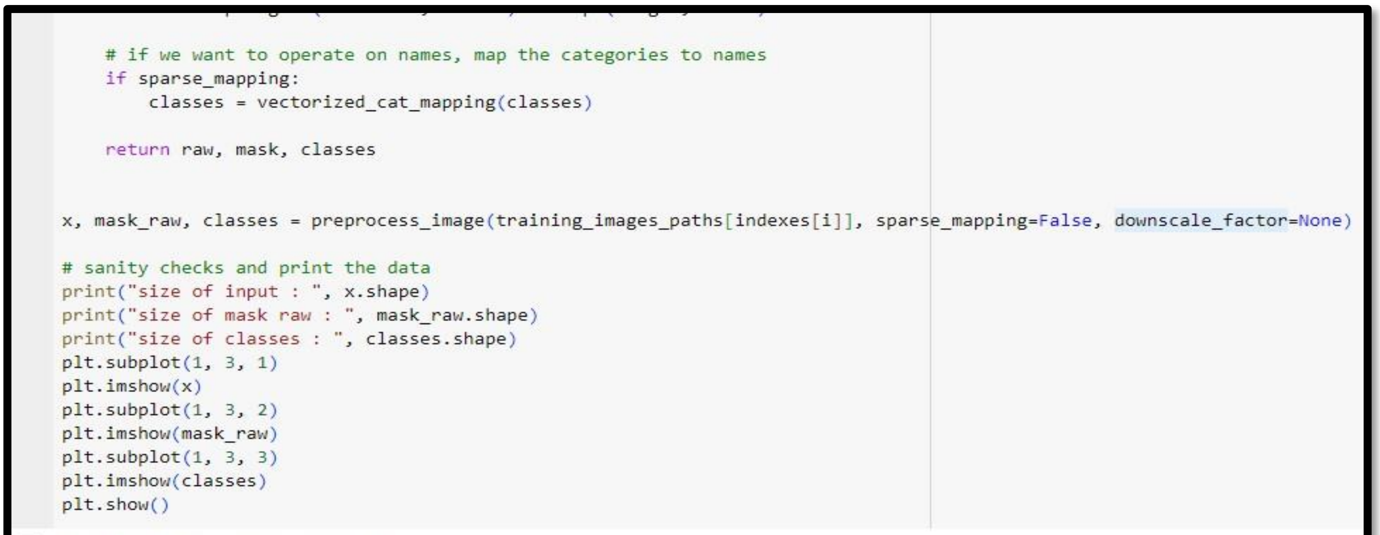


```
width = 4
height = 4
vis_batch_size = width * height

# get vis_batch_size unique, random indices
indexes = np.arange(len(training_images_paths))
indexes = np.random.permutation(indexes)[:vis_batch_size]

# create the plot
fig, axs = plt.subplots(height, width, sharex=True, sharey=True, figsize=(16, 8))
for i in range(vis_batch_size):
    # read the image
    img = torchvision.io.read_image(training_images_paths[indexes[i]])

    # pytorch reads it as (c, h, w), reshape it to (h, w, c) which is the shape matplotlib wants
    img = img.permute(1, 2, 0)

    # calculate the indexes for plots and set the image data
    y, x = i // width, i % width
    axs[y, x].imshow(img.numpy())

plt.tight_layout()
```



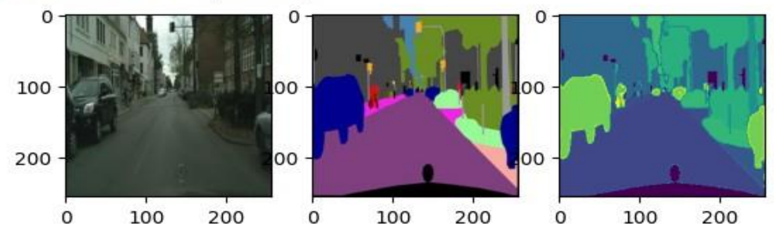


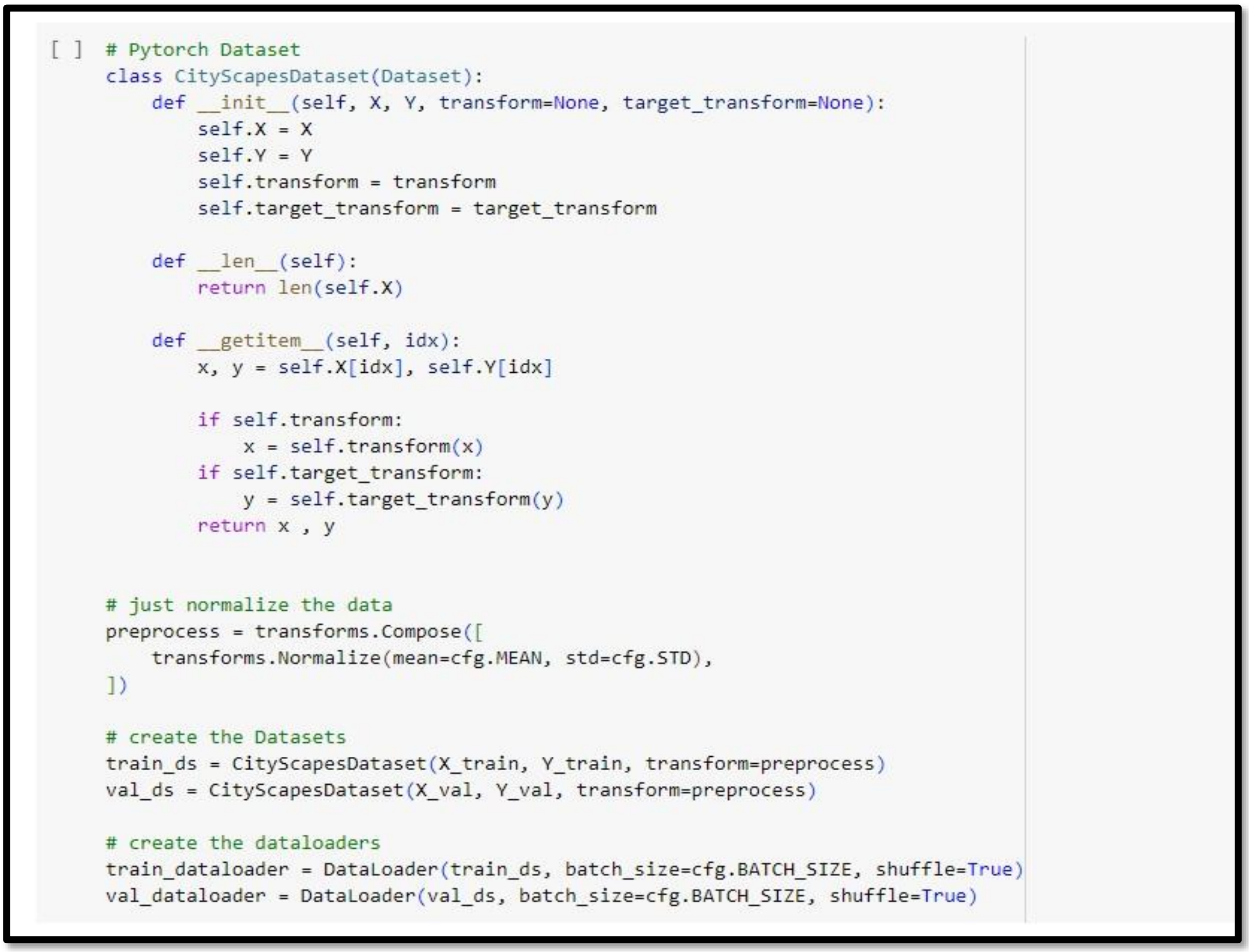
```
[ ] idx_to_name = [ 'unlabeled','ego vehicle','rectification border', 'out of roi', 'static', 'dynamic','ground', 'road', 'sidewalk', 'parking', 'rail track', 'building', 'wall', 'fence','guard rail' , 'bridge','tunnel','pole', 'pole
    idx_to_category = ["void", "flat", "construction", "object", "nature", "sky", "human", "vehicle"]

    idx_to_color = [[ 0,  0,  0], [ 0,  0,  0], [  0,  0,  0], [  0,  0,  0],[ 0,  0,  0],[111, 74,  0],[81,  0, 81] ,[128, 64,128],[244, 35,232],
                   [250,170,160],[230,150,140],[70, 70, 70],[102,102,156],[190,153,153],[180,165,180],[150,100,100],[150,120, 90],[153,153,153],
                   [153,153,153],[250,170, 30],[220,220,  0],[107,142, 35],[152,251,152],[ 70,130,180],[220, 20, 60],[255,  0,  0],[ 0,  0,142],
                   [ 0,  0, 70],[ 0, 60,100],[ 0,  0, 90],[  0,  0,110],[ 0, 80,100],[  0,  0,230],[119, 11, 32],[  0,  0,142]]


    idx_to_color_np = np.array(idx_to_color)

    name_to_category = {0 : 0, 1 : 0, 2 : 0, 3: 0, 4 : 0, 5 : 0, 6 : 0, 7 : 1, 8 : 1, 9 : 1, 10 : 1, 11 :2, 12 : 2, 13 : 2, 14 : 2, 15 : 2, 16 : 2,
                        17 : 3, 18 : 3, 19 : 3, 20: 3, 21 : 4, 22 : 4, 23 : 5, 24 : 6, 25 : 6, 26 : 7, 27 : 7, 28 : 7, 29 : 7, 30 : 7, 31 : 7, 32: 7, 33 : 7, 34 : 7}

    from typing import Tuple

    # vectorize the operation of getting the name to category for numpy (just a lookup in name_to_category dictionary)
    name_to_category_mapping = lambda x: name_to_category[x]
    vectorized_cat_mapping = np.vectorize(name_to_category_mapping)

    # vectorize the operation of mapping the name to color for numpy (just a lookup in idx_to_color dictionary)
    name_to_col_mapping = lambda x: idx_to_color[x]
    vectorized_col_mapping = np.vectorize(name_to_col_mapping)

    def preprocess_image(path : str, sparse_mapping=True, downscale_factor=None) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
        """
            Read the .jpeg image from *path*. Return the input image (256 x 256 x 3), mask (256 x 256 x 3) read from the jpeg
            and conversion to categories or names (if sparse_mapping is true) representation (256 x 256 x (|categories| or |names|) )
        """
        # Read the image from path and dowscale if downscale_factor is not None.
        img = Image.open(path)
        width, height = img.size

        if downscale_factor:
            width, height = width // downscale_factor, height//downscale_factor
            img = img.resize(( width, height ))
```



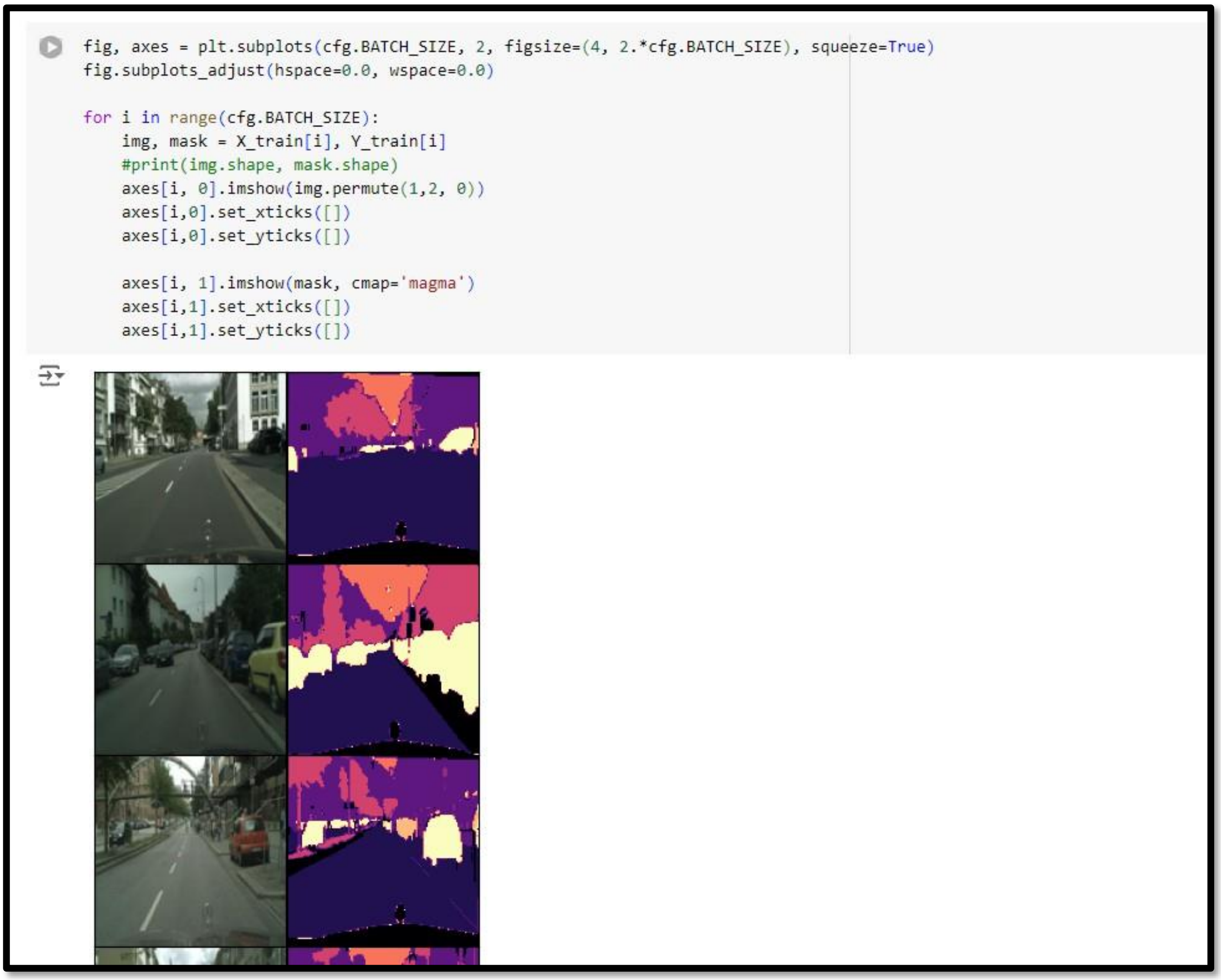
```python
    # if we want to operate on names, map the categories to names
    if sparse_mapping:
        classes = vectorized_cat_mapping(classes)

    return raw, mask, classes


x, mask_raw, classes = preprocess_image(training_images_paths[indexes[i]], sparse_mapping=False, downscale_factor=None)

# sanity checks and print the data
print("size of input : ", x.shape)
print("size of mask raw : ", mask_raw.shape)
print("size of classes : ", classes.shape)
plt.subplot(1, 3, 1)
plt.imshow(x)
plt.subplot(1, 3, 2)
plt.imshow(mask_raw)
plt.subplot(1, 3, 3)
plt.imshow(classes)
plt.show()
```

```
size of input :  (256, 256, 3)
size of mask raw :  (256, 256, 3)
size of classes :  (256, 256)
```

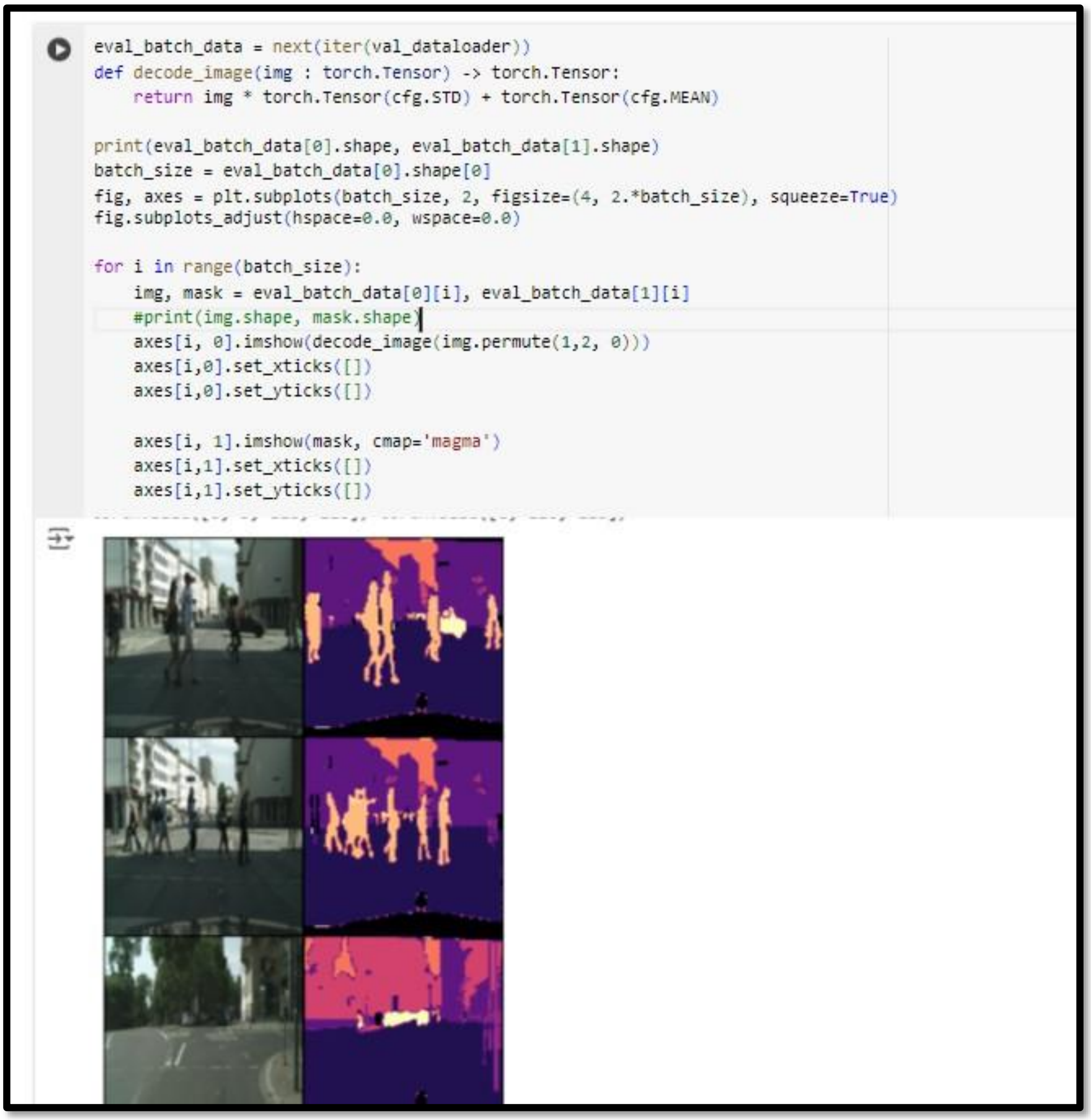




```
[ ] # Pytorch Dataset
    class CityScapesDataset(Dataset):
        def __init__(self, X, Y, transform=None, target_transform=None):
            self.X = X
            self.Y = Y
            self.transform = transform
            self.target_transform = target_transform

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

        def __getitem__(self, idx):
            x, y = self.X[idx], self.Y[idx]

            if self.transform:
                x = self.transform(x)
            if self.target_transform:
                y = self.target_transform(y)
            return x , y


    # just normalize the data
    preprocess = transforms.Compose([
        transforms.Normalize(mean=cfg.MEAN, std=cfg.STD),
    ])

    # create the Datasets
    train_ds = CityScapesDataset(X_train, Y_train, transform=preprocess)
    val_ds = CityScapesDataset(X_val, Y_val, transform=preprocess)

    # create the dataloaders
    train_dataloader = DataLoader(train_ds, batch_size=cfg.BATCH_SIZE, shuffle=True)
    val_dataloader = DataLoader(val_ds, batch_size=cfg.BATCH_SIZE, shuffle=True)
```



```
fig, axes = plt.subplots(cfg.BATCH_SIZE, 2, figsize=(4, 2.*cfg.BATCH_SIZE), squeeze=True)
fig.subplots_adjust(hspace=0.0, wspace=0.0)

for i in range(cfg.BATCH_SIZE):
    img, mask = X_train[i], Y_train[i]
    #print(img.shape, mask.shape)
    axes[i, 0].imshow(img.permute(1,2, 0))
    axes[i,0].set_xticks([])
    axes[i,0].set_yticks([])

    axes[i, 1].imshow(mask, cmap='magma')
    axes[i,1].set_xticks([])
    axes[i,1].set_yticks([])
```



```
eval_batch_data = next(iter(val_dataloader))
def decode_image(img : torch.Tensor) -> torch.Tensor:
    return img * torch.Tensor(cfg.STD) + torch.Tensor(cfg.MEAN)

print(eval_batch_data[0].shape, eval_batch_data[1].shape)
batch_size = eval_batch_data[0].shape[0]
fig, axes = plt.subplots(batch_size, 2, figsize=(4, 2.*batch_size), squeeze=True)
fig.subplots_adjust(hspace=0.0, wspace=0.0)

for i in range(batch_size):
    img, mask = eval_batch_data[0][i], eval_batch_data[1][i]
    #print(img.shape, mask.shape)
    axes[i, 0].imshow(decode_image(img.permute(1,2, 0)))
    axes[i,0].set_xticks([])
    axes[i,0].set_yticks([])

    axes[i, 1].imshow(mask, cmap='magma')
    axes[i,1].set_xticks([])
    axes[i,1].set_yticks([])
```

CONCLUSION AND FUTURE ENHANCEMENT

Conclusion:

This project successfully demonstrates the use of **convolutional neural networks (CNNs)** for semantic segmentation of urban scenes. By leveraging the Cityscapes dataset, the model was able to accurately classify different objects such as roads, vehicles, and buildings at the pixel level. The use of advanced techniques like **mixed precision training** and **data augmentation** proved to be crucial in achieving high performance while maintaining computational efficiency.

The project shows that CNNs, when combined with proper data preprocessing and evaluation techniques, can effectively handle the complexities of urban scenes, making them ideal for applications such as autonomous driving, smart city surveillance, and geographic information systems (GIS).

Future Enhancements:

While the model has shown promising results, several enhancements can be made to further improve its performance:

1. **Advanced Architectures**: Implementing architectures such as **U-Net** or **DeepLabV3+**, which are specifically designed for segmentation tasks, could improve the accuracy of the model, especially in capturing fine details of small objects.

2. **Transfer Learning**: Pretraining the model on a larger and more diverse dataset, such as COCO, and fine-tuning it on Cityscapes could lead to faster convergence and improved generalization.

3. **Real-time Segmentation**: Optimizing the model for real-time segmentation by reducing the computational complexity could make it suitable for applications like autonomous driving, where fast processing is critical.

4. **Multi-scale Feature Extraction**: Incorporating multi-scale feature extraction techniques could improve the model’s ability to segment objects of varying sizes, enhancing its performance in scenes with both large and small objects.

5. **Attention Mechanisms**: Integrating attention mechanisms such as **self-attention** or **spatial attention** could help the model focus on important regions of the image, leading to better segmentation accuracy, especially in crowded or complex scenes.

REFERENCES

- https://ieeexplore.ieee.org/document/10643318 - Q1 Journal
- https://www.ijournalse.org/index.php/ESJ/article/view/1444
- https://ieeexplore.ieee.org/document/10098596
- https://medium.com/analytics-vidhya/image-processing-using-opencv-cnn-and-keras-backed-by-tensor-flow-c9adf22bb271