DOCUMENTATION

Anjaneya Bajaj
2022A7PS0164P

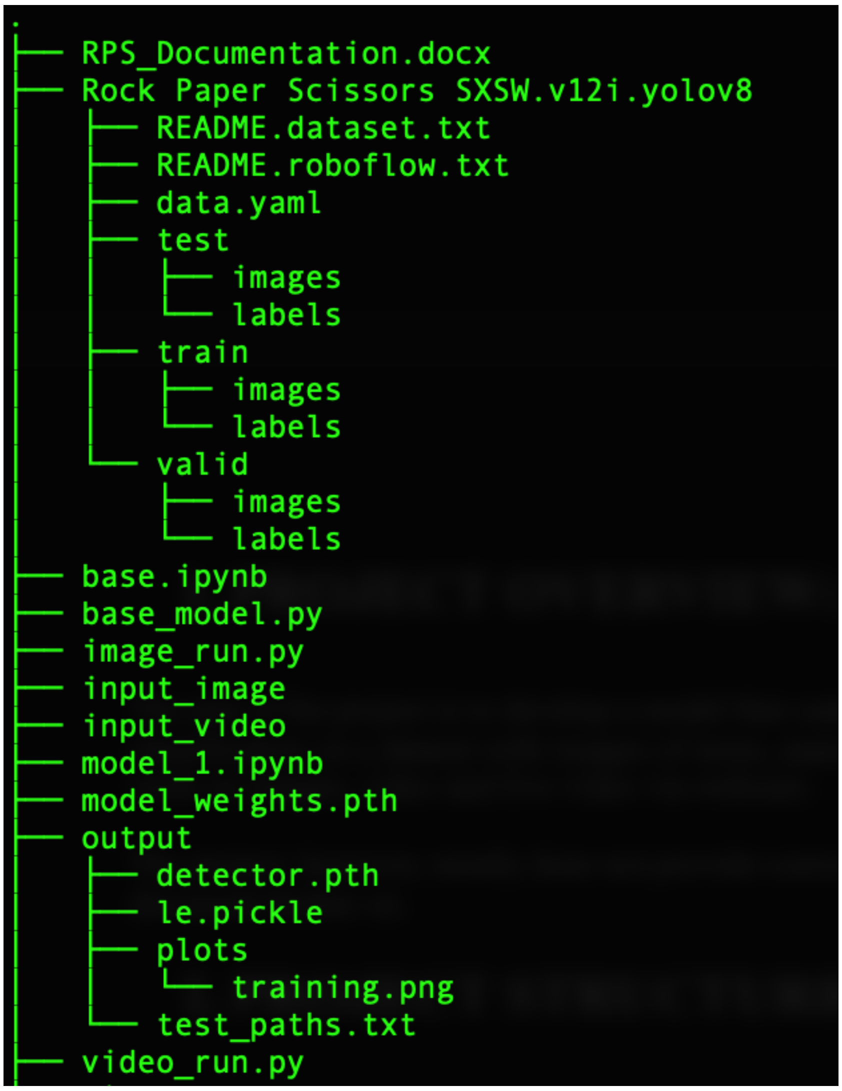
TABLE OF CONTENTS:

1. PROJECT OVERVIEW:

The aim of the project is to develop a model that can perform object detection and classification on a dataset with images of stone, paper and scissors. The project is designed to work on images, video and live video via webcam.

The project, however, mostly does not provide correct outputs – reasons and corrections discussed further on.

2. PROJECT STRUCTURE:

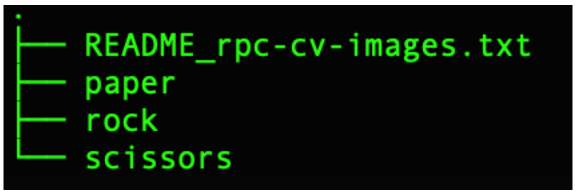


```
.
├── RPS_Documentation.docx
├── Rock Paper Scissors SXSW.v12i.yolov8
│   ├── README.dataset.txt
│   ├── README.roboflow.txt
│   ├── data.yaml
│   ├── test
│   │   ├── images
│   │   └── labels
│   ├── train
│   │   ├── images
│   │   └── labels
│   └── valid
│       ├── images
│       └── labels
├── base.ipynb
├── base_model.py
├── image_run.py
├── input_image
├── input_video
├── model_1.ipynb
├── model_weights.pth
├── output
│   ├── detector.pth
│   ├── le.pickle
│   ├── plots
│   │   └── training.png
│   └── test_paths.txt
├── video_run.py
```

This is the structure of the project.
‘Rock Paper Scissors SXSW.v12i.yolov8’ is the dataset containing the images and labels.
‘base.ipynb’ and ‘base_model.py’ are files related to the base model.
‘model_1.ipynb’ contains the object detection model.
‘output’ is a folder which contains the output files of ‘model_1.ipynb’

‘image_run.py’ and ‘video_run.py’ are the files that will be run to execute the model
‘model_weights.pth’ is used to load in model weights of the base model into ‘model_1.ipynb’

In addition to this there is the ‘archive’ folder that needs to be placed in the same folder as the rest of the files.



```
.
├── README_rpc-cv-images.txt
├── paper
├── rock
└── scissors
```

The folders ‘input_image’, ‘input_video’ and ‘output’ will need to be created by the user in the same folder where the other files are downloaded. Place the files ‘detector.pth’, ‘le.pickle’ and ‘test_paths.txt’ in the output folder.
The ‘Rock Paper Scissors SXSW.v12i.yolov8’ and ‘archive’ folders will need to be downloaded from these links :

https://www.kaggle.com/datasets/drgfreeman/rockpaperscissors?resource=download

https://universe.roboflow.com/roboflow-58fyf/rock-paper-scissors-sxsw/dataset/12

3. INSTALLATION:

This project is built to work with Python 3.x. You need the following libraries to run it :

```
# Install PyTorch (CPU version)
pip install torch

# Install PyTorch (GPU version, if available)
pip install torch torchvision

# Install scikit-learn
pip install scikit-learn

# Install OpenCV
pip install opencv-python

# Install NumPy
pip install numpy

# Install pandas
pip install pandas

# Install Matplotlib
pip install matplotlib

# Install Seaborn
pip install seaborn
```

```
# Install tqdm (for progress bars)
pip install tqdm

# Install imutils
pip install imutils

# Install Pillow (PIL)
pip install Pillow
```

Now, download all the files and folders present in the Github repository. Ensure that all the files and folders are in one directory only.

4. DATA DESCRIPTION AND PRE-PROCESSING:

This project has utilized two datasets.

The first is the Rock Paper Scissor SXSW dataset from Roboflow. It consists of 11813 images – the train set, test set and validation set have 10980, 329 and 604 images respectively. Each of these 3 categories has a directory of its own. Within each directory there are 2 directories – 'images' and 'labels'. The 'images' directory containa the image themselves while the 'labels' directory contains .txt files with the same name as the images. These contain the class and bounding box data. It is of the form -
<class> <startX> <startY> <width> <height>

Link : https://universe.roboflow.com/roboflow-58fyf/rock-paper-scissors-sxsw/dataset/12

The dataset has already been preprocessed and augmented. These are the details for both :

Pre-processing -
Auto-Orient: Applied
Resize: Stretch to 640x640

Augmentations
Outputs per training example: 5
Flip: Horizontal
Crop: 0% Minimum Zoom, 30% Maximum Zoom
Grayscale: Apply to 10% of images
Hue: Between -10° and +10°
Saturation: Between -25% and +25%
Brightness: Between -25% and +25%
Exposure: Between -25% and +25%
Blur: Up to 1px
Noise: Up to 1% of pixels
Cutout: 7 boxes with 3% size each

Furthermore, in the code itself we apply the following processes-

Image Colour Conversion and Resizing

- **cv2.cvtColor**: Converts the colour space of the input image. In this case, it converts the image from the BGR (Blue-Green-Red) colour space to the RGB (Red-Green-Blue) colour space. This conversion ensures that the image is in the correct colour format expected by many machine learning models, which typically use RGB.

- **cv2.resize**: Resizes the image to a specified size. In this case, it resizes the image to a square shape with dimensions (224, 224). Image resizing is often necessary to standardize input dimensions for machine learning models.

These pre-processing steps are applied to each input image to ensure that it is in the correct colour space (RGB) and has a consistent size (224x224) before being used in training or evaluation.

Image Transformations –

ToPILImage: Converts PyTorch tensors to PIL images.
ToTensor: Converts PIL images to PyTorch tensors.
Normalize: Applies pixel-wise normalization to the images.

Normalization Mean Values: Calculated in code.
Normalization Standard Deviation Values: Calculated in code.

These transformations are applied to input images to ensure they are in the correct format and normalized before being fed into the model.

The second dataset is again a set of images of hand gestures of rock, paper and scissors from Kaggle. It has 2188 images – with rock (726 images), paper (710 images) and scissors (752 images). All images are RGB images of 300 pixels wide by 200 pixels high in .png format. The images are separated in three sub-folders named 'rock', 'paper' and 'scissors' according to their respective class.

Link : https://www.kaggle.com/datasets/drgfreeman/rockpaperscissors?resource=download

The following pre-processing is applied to the dataset :

Image Transformation

- **transforms.Resize**: Resizes the input image to a specified size. In this case, it resizes the image to dimensions (224, 224). Image resizing is performed to ensure that all input images have a consistent size, which is important for training and inference with machine learning models.

- **transforms.ToTensor**: Converts the input image into a PyTorch tensor. This transformation changes the image representation from a NumPy array or PIL Image to a tensor, which is the expected format for PyTorch models.

- **transforms.Normalize**: Applies a normalization operation to the image. This normalization process standardizes the pixel values in the image by subtracting the mean values and dividing by the standard deviations. The specific mean and standard deviation values used here are [0.485, 0.456, 0.406] and [0.229, 0.224, 0.225], respectively. Normalization is a common preprocessing step in deep learning to ensure that pixel values have a consistent scale.

These transformations are applied to each input image to standardize their dimensions, convert them into tensors, and normalize their pixel values, making them suitable for feeding into the model.

5. MODEL ARCHITECTURE:

This project utilizes 2 separate models such that the output features from the base model (class CNN) are used as input for the object detector and classifier (class ObjectDetector).

First we will describe the base model.

Convolutional Neural Network (CNN) Model

Overview

The Convolutional Neural Network (CNN) is a deep learning model designed for image classification tasks. It is composed of convolutional and fully connected layers that learn to extract features and make predictions from input images.
The model draws inspiration from several existing models like AlexNet, LeNet and VGGNet. Main features adapted from each model:

AlexNet – Use of ReLU activation function ,max pooling, local response normalisation.
VGGNet – Use of multiple 3*3 kernels instead of fewer kernels of varying sizes.
LeNet and VGGNet – Usage of average pooling at the last layer of the Convolution layers.

Instead of local response normalization like AlexNet this model utilizes batch normalization instead.

Architecture

The CNN model consists of two main parts: the Convolutional Layers and the Fully Connected Layers.

Layer	Feature Maps	Size (Output)	Kernel Size	Stride	Activation
Image Input	1	3*224*224			
Conv2d(1)	16	16*111*111	3*3	2	ReLU
BatchNorm2d	16	16*111*111			

MaxPool	16	16*55*55	2*2	2	
Conv2d(2)	32	32*53*53	3*3	1	ReLU
Conv2d(3)	128	128*51*51	3*3	1	ReLU
BatchNorm2d	128	128*51*51			
MaxPool	128	128*25*25	2*2	2	
Conv2d(4)	256	256*23*23	3*3	1	ReLU
Conv2d(5)	256	256*21*21	3*3	1	ReLU
AvgPool	256	256*10*10	2*2	2	
Dropout(p=0.5)					
Linear(6)		1024			ReLU
Linear(7)		1024			ReLU
Linear(8)		3			Softmax

Convolutional Layers

- **Input Channels:** 3 (corresponding to RGB color channels)
- **Output Channels:** 16, 32, 128, and 256 in different layers
- **Kernel Size:** 3x3
- **Activation Function:** ReLU (Rectified Linear Unit)
- **Batch Normalization:** Applied in selected layers
- **Max Pooling:** Applied to downsample feature maps
- **Average Pooling:** Applied before the fully connected layers

The convolutional layers extract hierarchical features from the input images. They use ReLU activation functions for non-linearity and apply batch normalization for improved training stability. Max pooling and average pooling layers downsample the feature maps to reduce spatial dimensions.

Fully Connected Layers

- **Dropout:** Applied with a probability of 0.5 for regularization
- **Number of Neurons:** 1024 in each of the two hidden layers
- **Activation Function:** ReLU for hidden layers
- **Output Neurons:** 3 (corresponding to the number of output classes)
- **Final Activation:** Softmax

The fully connected layers take the flattened output from the convolutional layers and perform classification. Dropout is applied to reduce overfitting. ReLU activation functions introduce non-linearity, and softmax is used for multi-class classification.

Forward Pass

In the forward pass, input images are processed through the convolutional layers to extract features. The feature maps are then flattened and passed through the fully connected layers to produce class probabilities using the softmax activation function.

This CNN model is designed for image classification tasks with three output classes. \

Now we come to the model for object detection and classification.

Object Detector Model

Overview:

This is a custom object detection and classification model designed for tasks involving detecting and classifying objects within images. The model consists of three main components: a base model, a bounding box regressor, and a classifier.

Architecture:

regressor :

Layer	Size of Output	Activation
baseModel(input)	1024	
Linear(1)	128	ReLU
Linear(2)	64	ReLU
Linear(3)	32	ReLU
Linear(4)	4	Sigmoid

classifier :

Layer	Size of Output	Activation
baseModel(input)	1024	
Linear(1)	512	ReLU
Dropout(p=0.5)		
Linear(2)	512	ReLU
Dropout(p=0.5)		
Linear(3)	numClasses (4)	Softmax

The nn.Identity function is used to ensure that instead of the softmax activated size 3 output that the baseModel normally gives, we get a size 1024 output which then act as input features for the regressor and classifier.

Components:

1. Base Model:
   - The base model is a pretrained convolutional neural network (CNN) that was described above. It serves as the feature extractor for the input images.

2. Bounding Box Regressor:

- The bounding box regressor is responsible for predicting the coordinates of bounding boxes around detected objects.
- It consists of several fully connected (linear) layers followed by ReLU activation functions.
- The final layer outputs four values representing the bounding box's (x, y) coordinates of the top-left corner and its width and height.
- A sigmoid activation function is applied to ensure that the coordinates are within the range [0, 1] – the input tensors are all normalized.

3. Classifier:
   - The classifier is responsible for classifying the detected objects into one of the predefined classes.
   - It consists of fully connected layers with ReLU activation functions and dropout layers for regularization.
   - The final layer outputs class logits, and a softmax activation function is applied to obtain class probabilities.
   - The number of output nodes in the final layer matches the number of classes (numClasses).

Input:

- Input images should have the shape (batch_size, channels, height, width), where 'batch_size' is the number of images per batch, 'channels' is the number of color channels (usually 3 for RGB), 'height' is the image height, and 'width' is the image width.

Output:

- The model outputs a tuple containing two elements: bounding box predictions and class logits.
  - Bounding Box Predictions: A tensor containing the predicted bounding box coordinates for each detected object.
  - Class Logits: A tensor containing the raw class scores for each class. Apply a softmax function to get class probabilities.

6. TRAINING AND EVALUATION:

First we will describe the training procedure of the base model.
The model is trained on the Kaggle dataset that was described above.

Here's an overview of the training process:

1. **Model Initialization**:

A CNN model (an instance of the **CNN** class) is initialized and we select the device we wish to use (cpu/gpu).

2. **Loss Function and Optimizer Definition**:

We used Cross Entropy loss and the Adam optimizer.

Cross entropy loss is one of the most commonly used loss functions for multi-class classification problems. Cross-entropy loss measures how well the predicted probabilities match the true probabilities. When the predicted probabilities are close to the true probabilities, the loss is low (closer to 0), indicating a good model fit. Conversely, when they diverge, the loss is high, indicating a poor fit. The loss is higher when the predicted probability diverges from the actual target class. It heavily penalizes confident incorrect predictions (i.e., high probability for the wrong class).

Adam (short for Adaptive Moment Estimation) is an optimization algorithm commonly used for training deep neural networks. It combines ideas from two other popular optimization techniques: RMSprop (Root Mean Square Propagation) and momentum. Adam is known for its fast convergence, especially when dealing with large-scale deep learning models. It is robust to noisy gradients and works well in practice with a wide range of learning rates.

We use Adam to update the weights of the model during training.

3. **Training Loop**:

The training process is divided into several epochs, where each epoch represents a complete pass through the entire training dataset.
For each epoch, a running total for training loss (train_loss) is initialized to zero, and counters for correct predictions (correct) and total examples (**total**) are set to zero.
For each batch of images and labels in the training data:

- Gradients are reset to zero with optimizer.zero_grad() to avoid accumulation from previous batches.
- Forward pass: The model computes predictions (outputs) for the input images.
- The loss between predicted and actual labels is calculated using the Cross-Entropy Loss.
- Backpropagation: Gradients are computed and propagated back through the network using loss.backward().
- Optimization: The optimizer updates the model's weights using computed gradients via optimizer.step().
- Training loss and accuracy statistics are updated. Training accuracy is calculated as the percentage of correct predictions for the current epoch. Average training loss (avg_train_loss) is calculated by dividing the accumulated training loss by the number of batches.

4. **Validation Loop**:
    - After each epoch of training, the model is set to evaluation mode using model.eval().
    - A running total for validation loss (val_loss) is initialized to zero, and counters for correct predictions and total examples are reset.
    - For each batch of images and labels in the validation data, we do a forward pass. The model computes predictions for the input images. The validation loss is calculated using the same Cross-Entropy Loss as in training. Validation accuracy statistics are updated.
    - Validation accuracy is calculated as the percentage of correct predictions on the validation dataset.
    - Average validation loss (avg_val_loss) is calculated by dividing the accumulated validation loss by the number of validation batches.

We then plot training/validation loss and accuracy vs epochs.

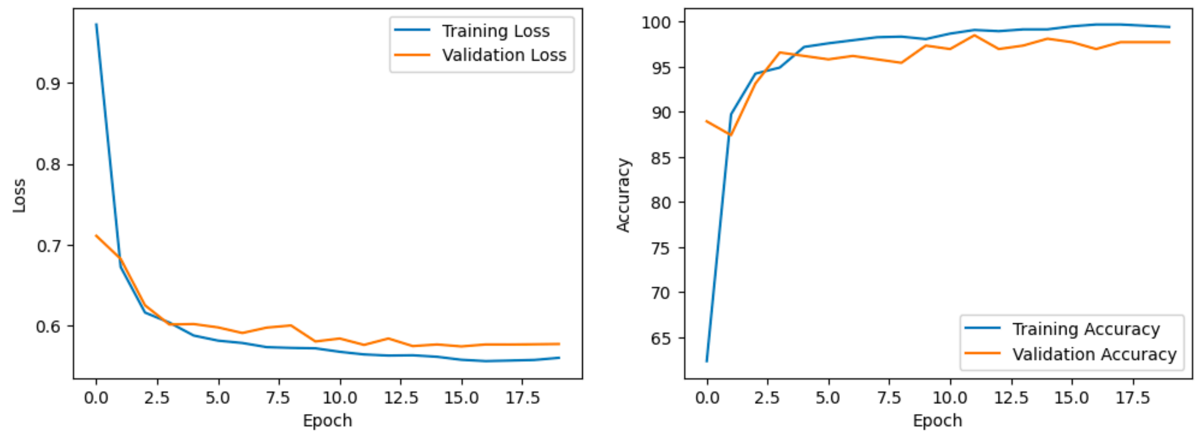
To train the model we mainly experiment with 3 hyperparameters. These are learning rate (LR), number of epochs (EPOCHS), and batch size (BATCH_SIZE).

Run Number	LR	EPOCHS	BATCH_SIZE
1	1e-5	20	16
2	1e-4	20	16
3	1e-5	30	16
4	1e-5	20	64
5	1e-5	10	16
6	1e-5	20	32
7	1e-6	20	16
8	1e-5	40	16

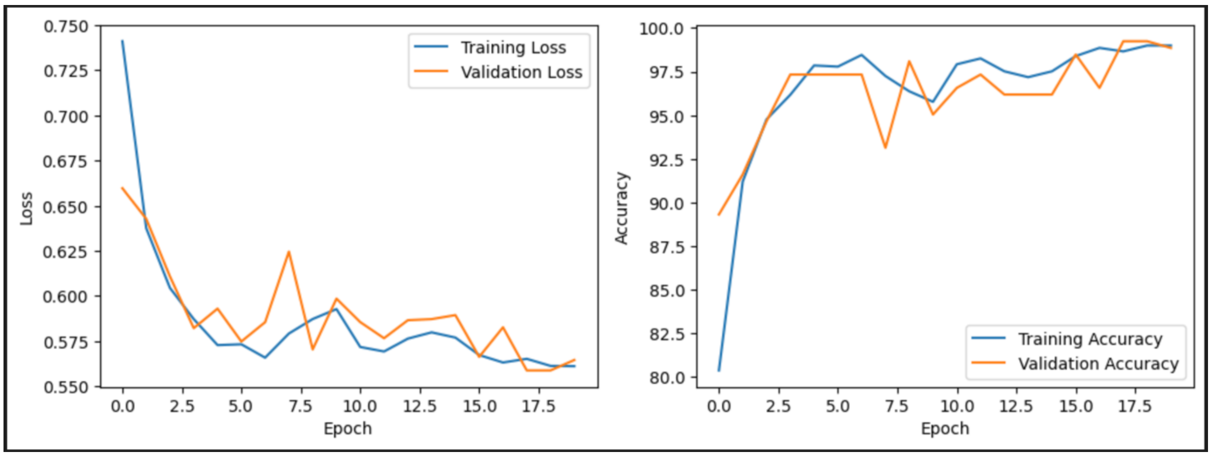
We will now sequentially list the performance of the model in every run. The graphs for the same are attached after this.

Run Number	Train Loss	Train Accuracy (%)	Valid Loss	Valid Accuracy (%)	Test Loss	Test Accuracy (%)	Avg. F1 Score
1	0.5603	99.4	0.5774	97.71	0.5660	98.4	0.9841
2	0.5611	98.99	0.5646	98.85	0.5714	97.95	0.9794
3	0.5538	99.87	0.5794	97.47	0.5671	98.09	0.9908
4	0.5642	98.82	0.5810	96.95	0.5732	97.95	0.9793
5	0.5693	98.52	0.5709	98.09	0.59	96.58	0.9655
6	0.5569	99.66	0.5580	99.24	0.5809	97.49	0.9748
7	0.6941	88.91	0.6978	90.46	0.7007	87.67	0.8757
8	0.5545	99.8	0.5597	98.85	0.5740	97.72	0.977

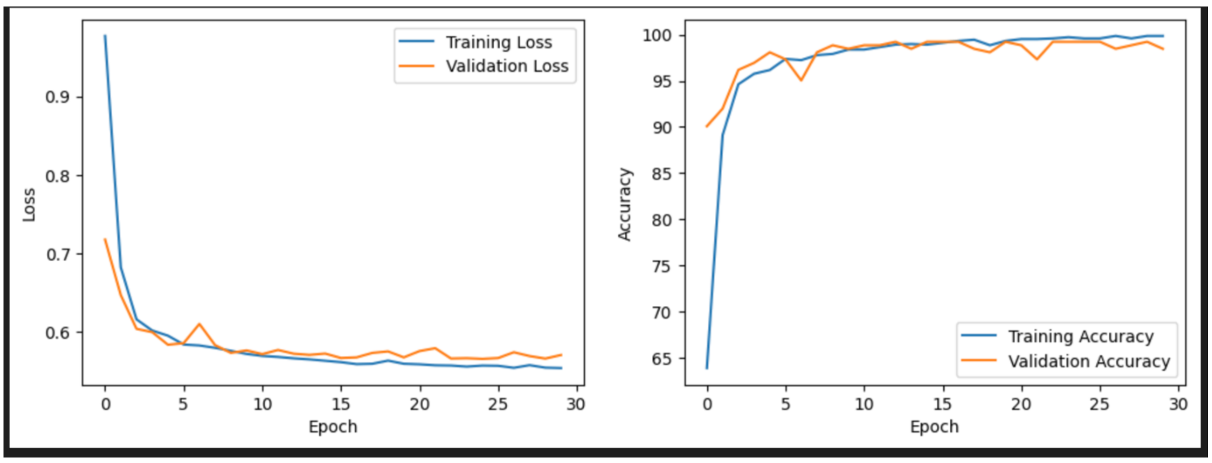
RUN 1:



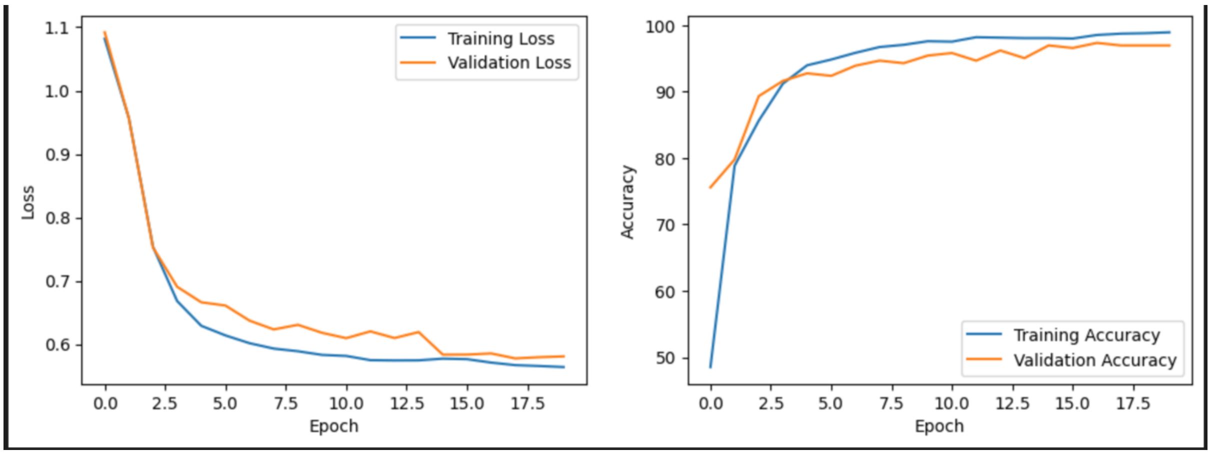
RUN 2:



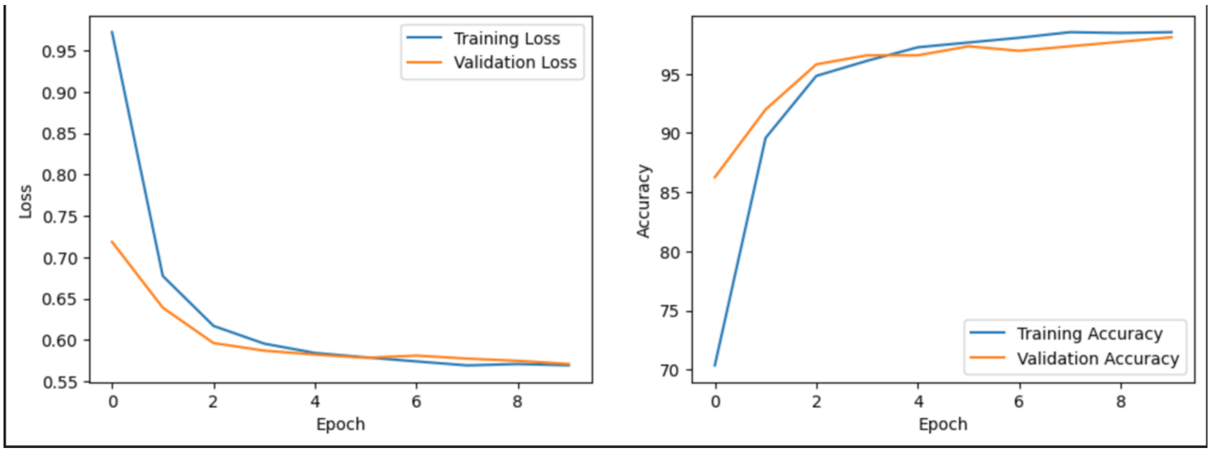
RUN 3:



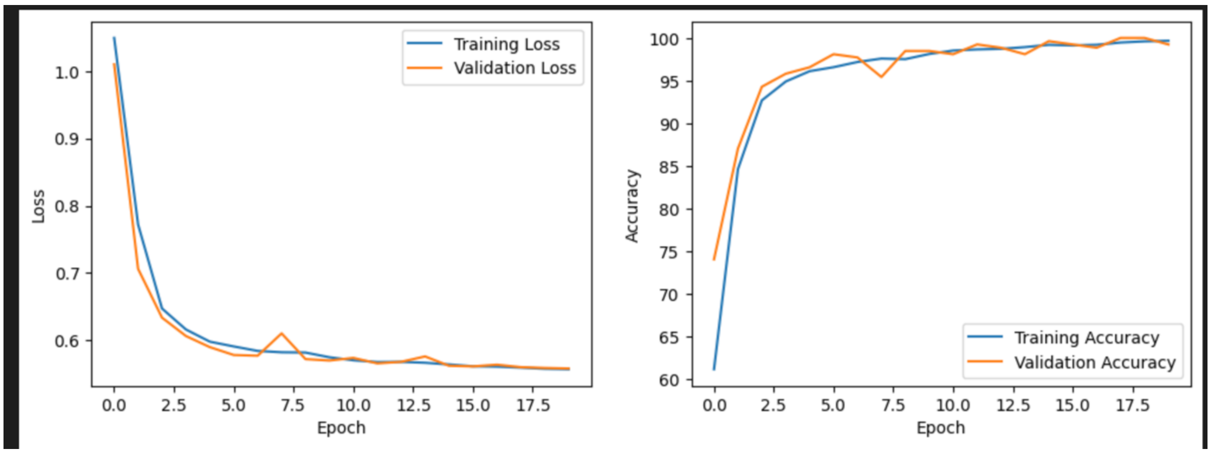
RUN 4:



RUN 5:

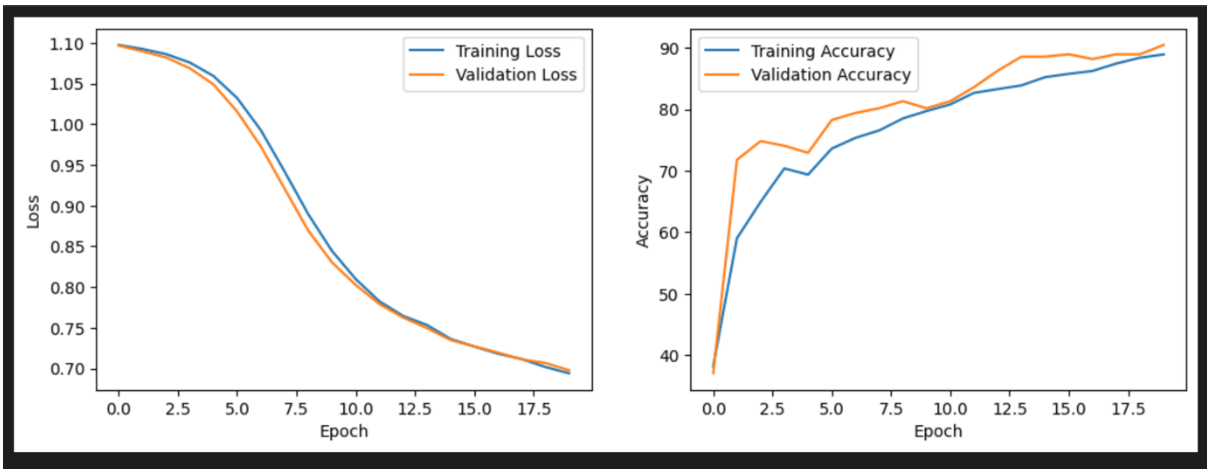


RUN 6:

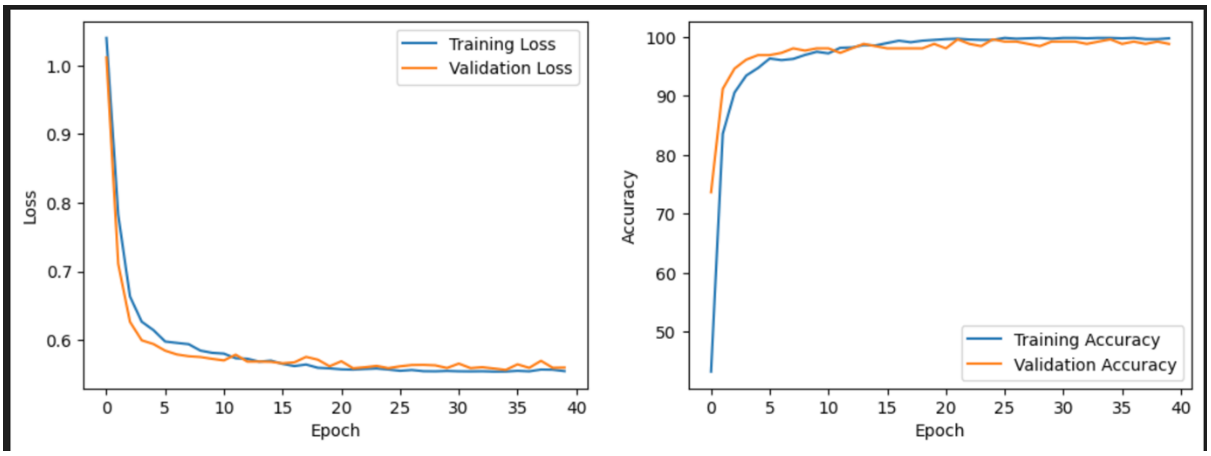


RUN 7:

Possible indications of underfitting.



RUN 8:



Taking into account all these factors, it appears suitable to choose the hyperparameters of run 1 to train the model.

We now describe the training process of the object detector model.

Here's an overview of the training process.

Loading Pre-trained Base Model
First, the code initializes the base model, which serves as the feature extractor. The parameters of the base model are frozen, preventing them from being updated during training. The pre-trained weights of the object detector model are loaded from a file using **torch.load()** and applied to the model.

Loss Functions

Two loss functions are defined:

classLossFunc: Cross-Entropy Loss for classification.

bboxLossFunc: Mean Squared Error (MSE) Loss for bounding box regression.

Optimizer
The Adam optimizer is initialized to update the weights of the object detector during training. The learning rate is set according to your specific configuration.

Training Loop
The code enters a loop that iterates over a specified number of epochs. Each epoch represents one pass through the entire training dataset. Inside the training loop, the model is set to training mode using objectDetector.train(). Training Loss and Accuracy are initialized to keep track of the model's performance during training.

Training Set Loop
The code loops through the training dataset (**trainLoader**) in batches. For each batch:
The input images, labels, and bounding boxes are sent to the chosen device , preferably a GPU for faster training (if present). A forward pass through the model is performed, generating predictions for both bounding boxes and class labels. The losses for bounding box regression and classification are computed. The gradients are zeroed using opt.zero_grad() to avoid gradient accumulation. Backpropagation is performed by calling totalLoss.backward(). The optimizer (opt) updates the model's weights using opt.step(). The training loss and accuracy are updated.
After processing all batches in the training dataset, the average training loss and accuracy for the epoch are calculated.

Validation Set Loop
The code switches off gradient computation using with torch.no_grad() and sets the model to evaluation mode with objectDetector.eval(). Similar to the training set loop, the code loops through the validation dataset to compute the validation loss and accuracy.
The average validation loss and accuracy for the epoch are calculated.

Training History
Training history is recorded in the dictionary **H**, which stores the following metrics for each epoch:

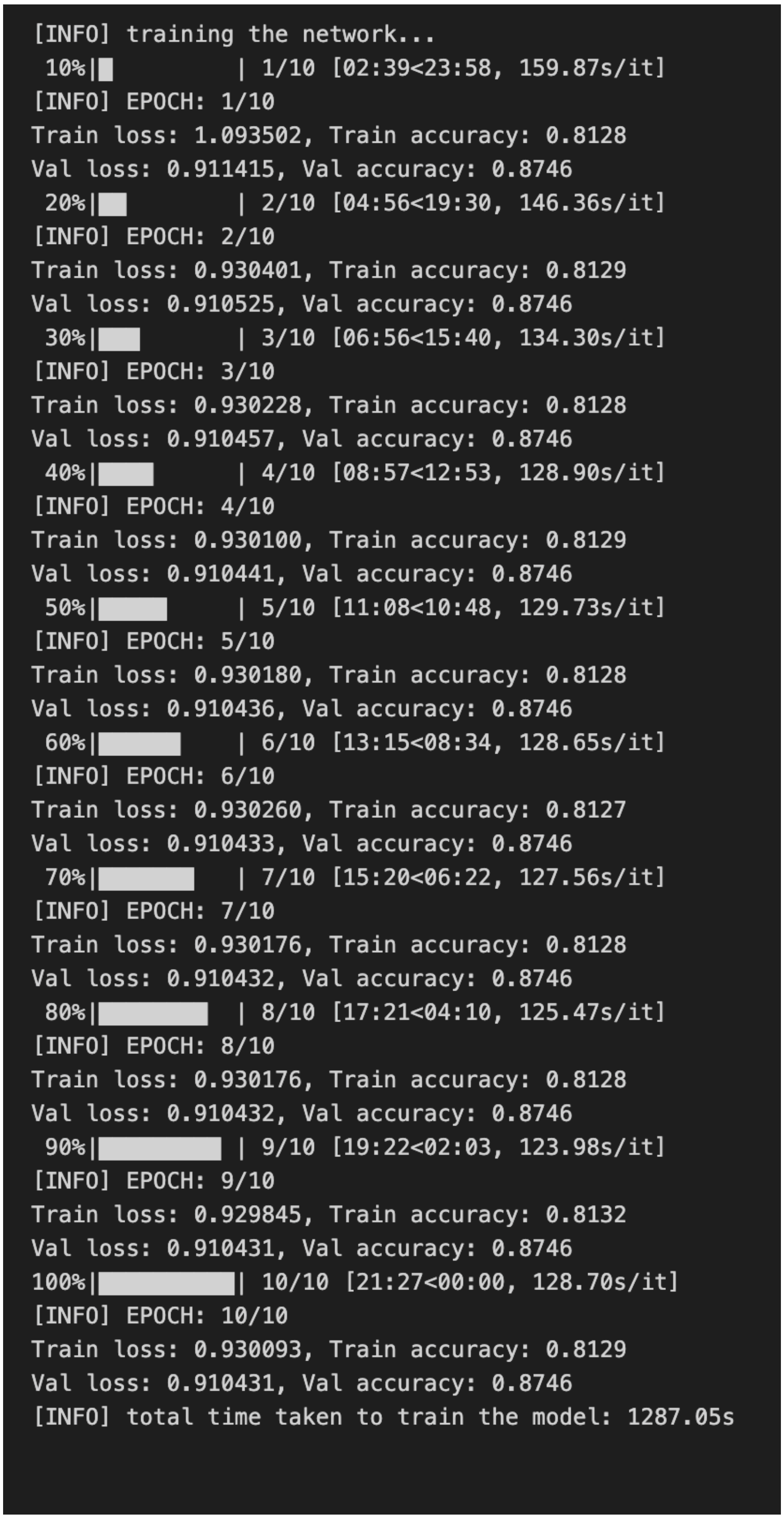
- total_train_loss: Average training loss.
- train_class_acc: Training accuracy.
- total_val_loss: Average validation loss.
- val_class_acc: Validation accuracy.

The training and validation loss and accuracy are printed for each epoch. After completing all epochs, the total time taken for training is reported.

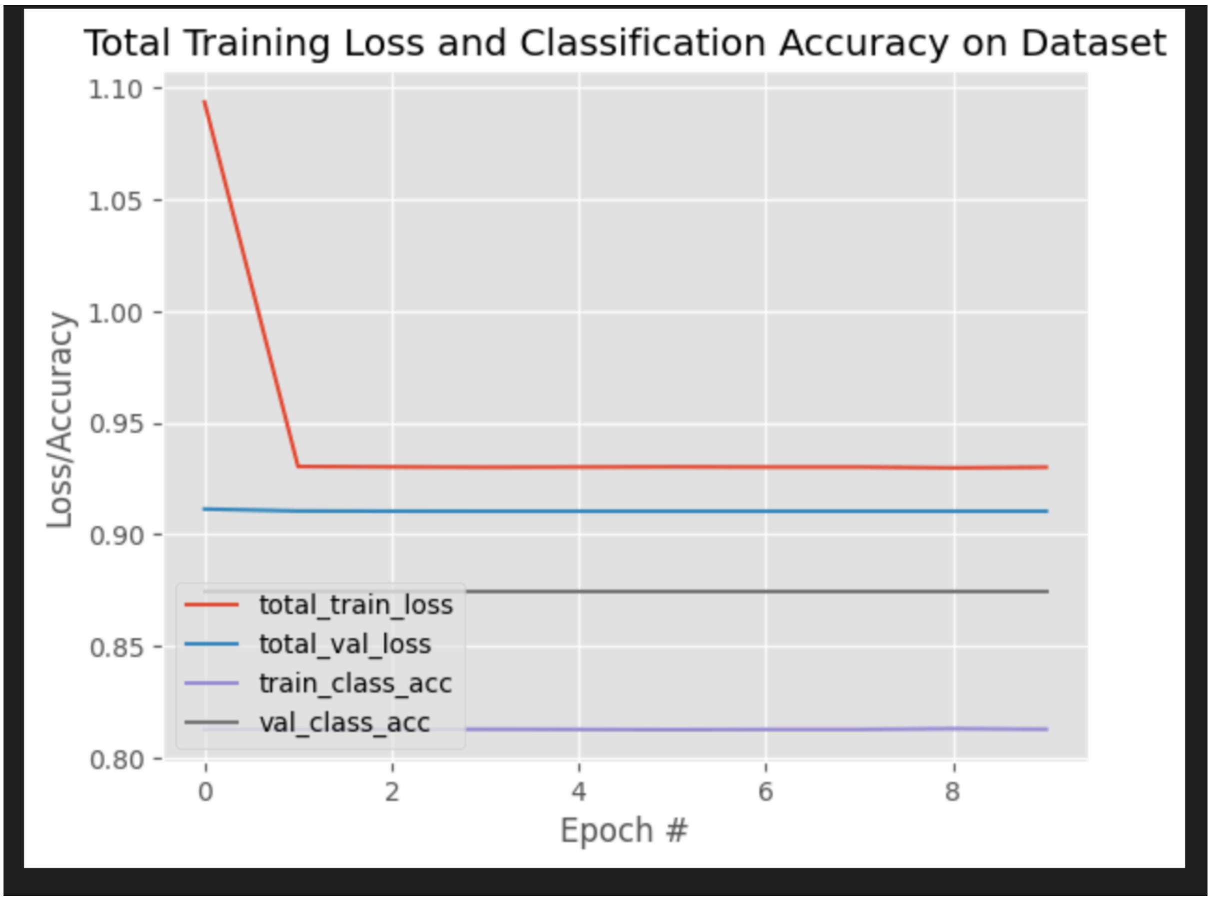
Again, to train the model we mainly experiment with 3 hyperparameters. These are learning rate (LR), number of epochs (EPOCHS), and batch size (BATCH_SIZE).

Run Number	LR	EPOCHS	BATCH_SIZE
1	1e-4	10	16
2	1e-4	20	16
3	1e-2	10	16
4	1e-4	10	64

RUN 1:

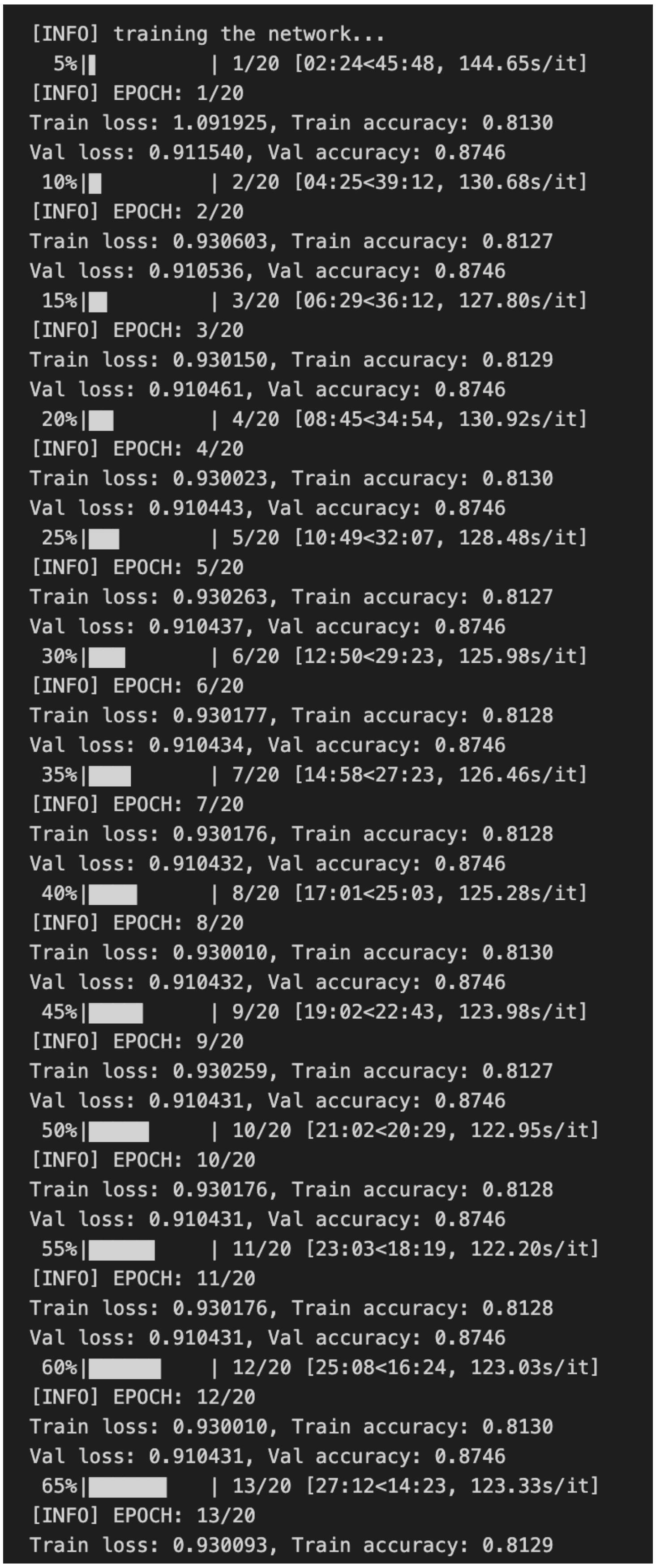


```
[INFO] training the network...
 10%|█         | 1/10 [02:39<23:58, 159.87s/it]
[INFO] EPOCH: 1/10
Train loss: 1.093502, Train accuracy: 0.8128
Val loss: 0.911415, Val accuracy: 0.8746
 20%|██        | 2/10 [04:56<19:30, 146.36s/it]
[INFO] EPOCH: 2/10
Train loss: 0.930401, Train accuracy: 0.8129
Val loss: 0.910525, Val accuracy: 0.8746
 30%|███       | 3/10 [06:56<15:40, 134.30s/it]
[INFO] EPOCH: 3/10
Train loss: 0.930228, Train accuracy: 0.8128
Val loss: 0.910457, Val accuracy: 0.8746
 40%|████      | 4/10 [08:57<12:53, 128.90s/it]
[INFO] EPOCH: 4/10
Train loss: 0.930100, Train accuracy: 0.8129
Val loss: 0.910441, Val accuracy: 0.8746
 50%|█████     | 5/10 [11:08<10:48, 129.73s/it]
[INFO] EPOCH: 5/10
Train loss: 0.930180, Train accuracy: 0.8128
Val loss: 0.910436, Val accuracy: 0.8746
 60%|██████    | 6/10 [13:15<08:34, 128.65s/it]
[INFO] EPOCH: 6/10
Train loss: 0.930260, Train accuracy: 0.8127
Val loss: 0.910433, Val accuracy: 0.8746
 70%|███████   | 7/10 [15:20<06:22, 127.56s/it]
[INFO] EPOCH: 7/10
Train loss: 0.930176, Train accuracy: 0.8128
Val loss: 0.910432, Val accuracy: 0.8746
 80%|████████  | 8/10 [17:21<04:10, 125.47s/it]
[INFO] EPOCH: 8/10
Train loss: 0.930176, Train accuracy: 0.8128
Val loss: 0.910432, Val accuracy: 0.8746
 90%|█████████ | 9/10 [19:22<02:03, 123.98s/it]
[INFO] EPOCH: 9/10
Train loss: 0.929845, Train accuracy: 0.8132
Val loss: 0.910431, Val accuracy: 0.8746
100%|██████████| 10/10 [21:27<00:00, 128.70s/it]
[INFO] EPOCH: 10/10
Train loss: 0.930093, Train accuracy: 0.8129
Val loss: 0.910431, Val accuracy: 0.8746
[INFO] total time taken to train the model: 1287.05s
```

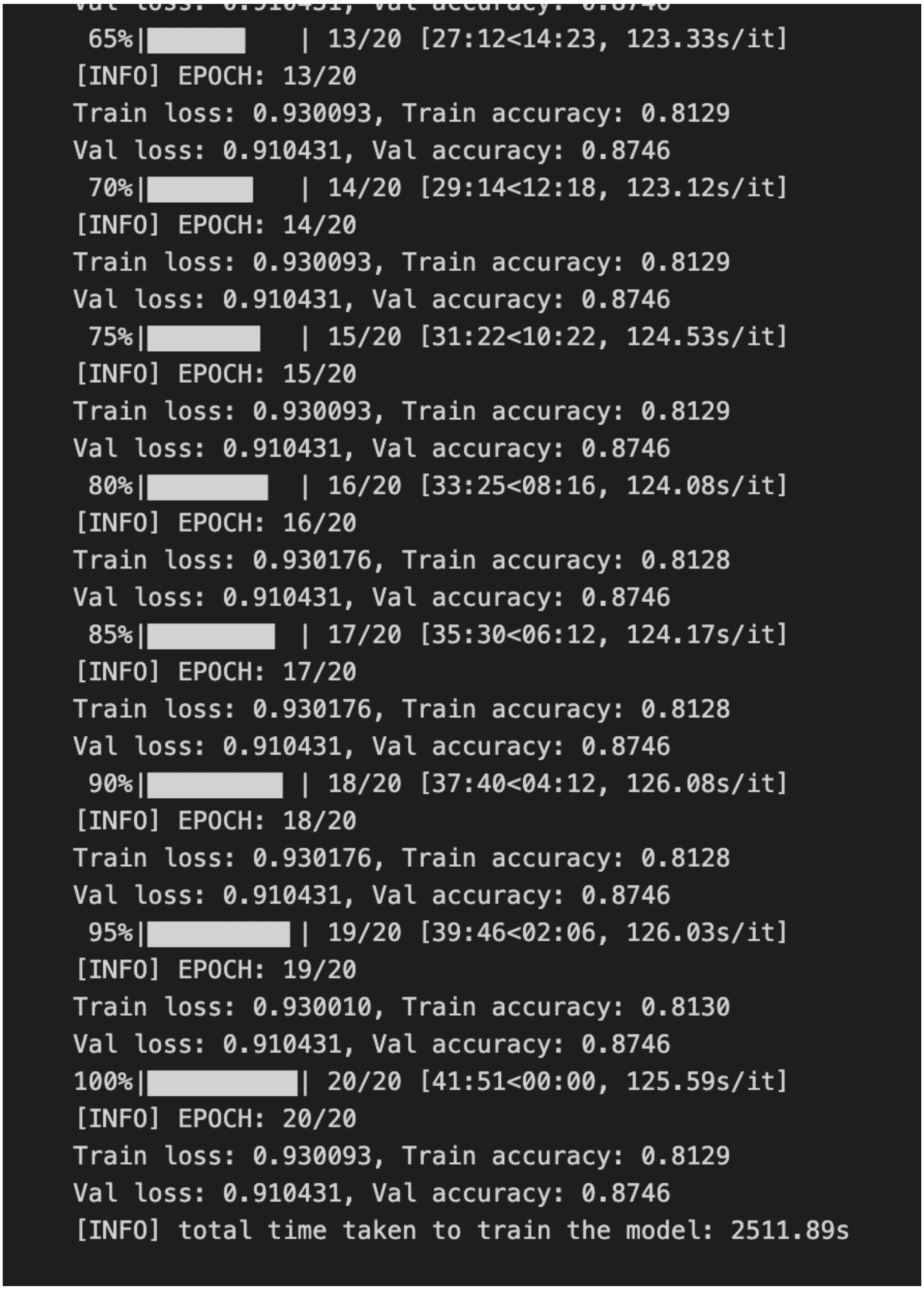


Total Training Loss and Classification Accuracy on Dataset
Loss/Accuracy
Epoch #
1.10
1.05
1.00
0.95
0.90
0.85
0.80
0
2
4
6
8
total_train_loss
total_val_loss
train_class_acc
val_class_acc

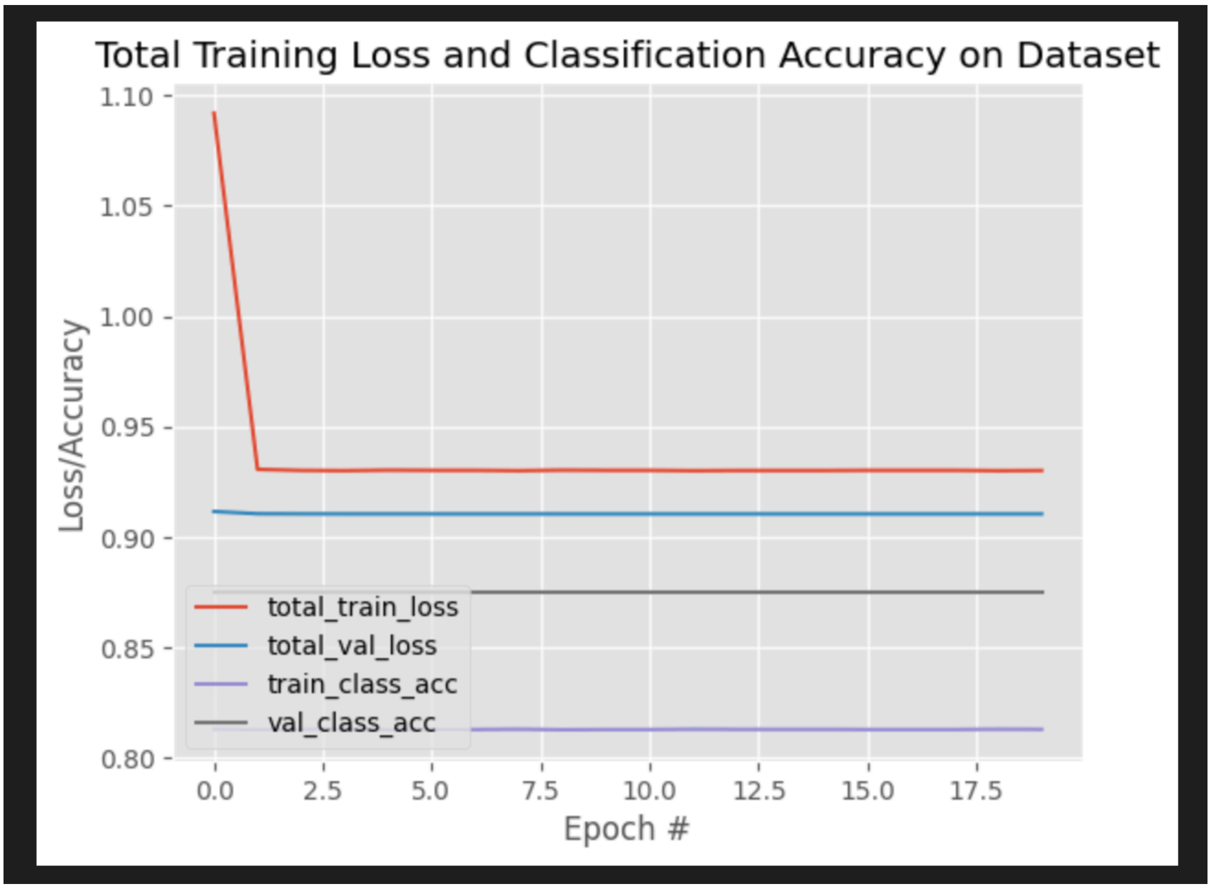
RUN 2:



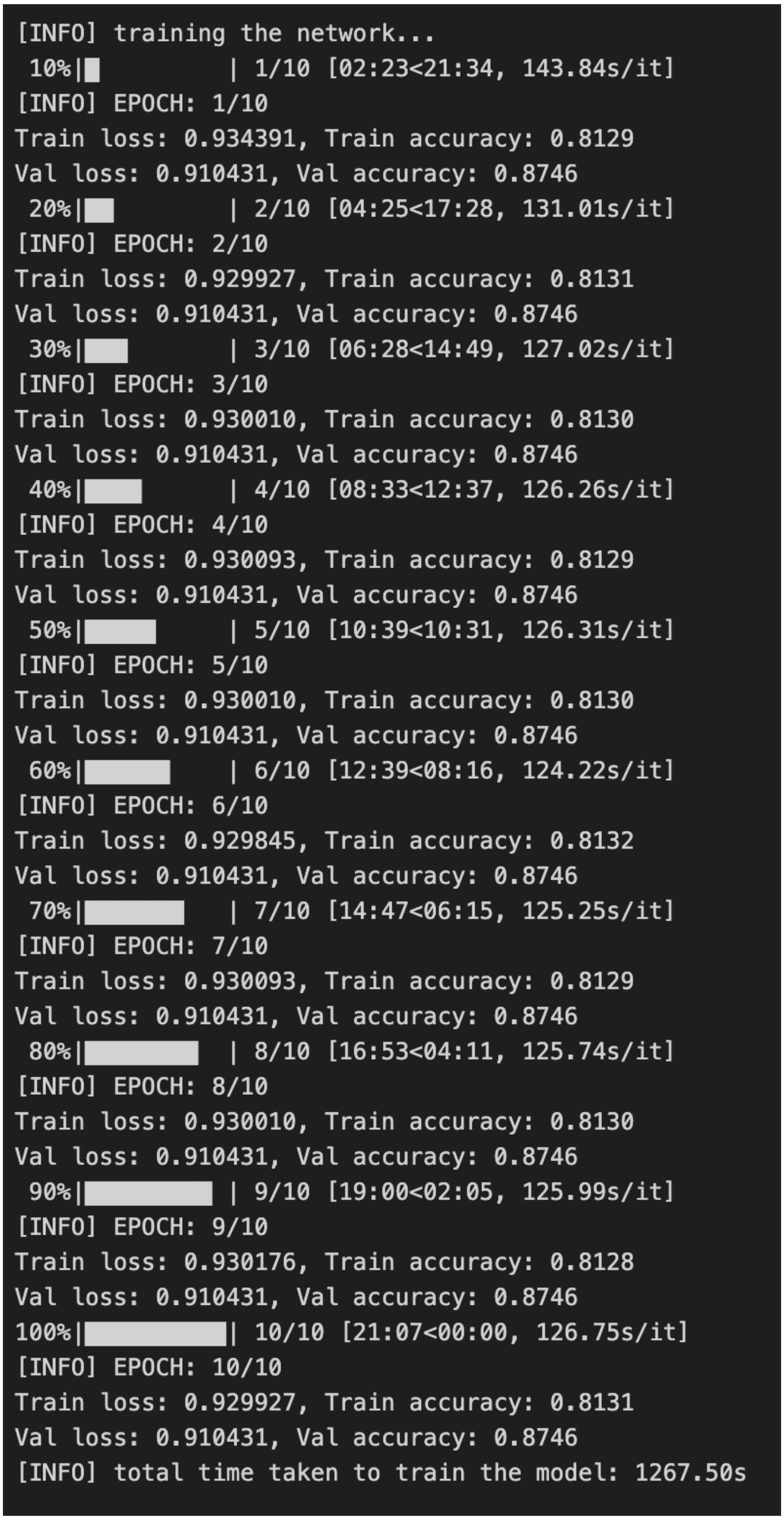
```
[INFO] training the network...
  5%|▌         | 1/20 [02:24<45:48, 144.65s/it]
[INFO] EPOCH: 1/20
Train loss: 1.091925, Train accuracy: 0.8130
Val loss: 0.911540, Val accuracy: 0.8746
 10%|█         | 2/20 [04:25<39:12, 130.68s/it]
[INFO] EPOCH: 2/20
Train loss: 0.930603, Train accuracy: 0.8127
Val loss: 0.910536, Val accuracy: 0.8746
 15%|█▌        | 3/20 [06:29<36:12, 127.80s/it]
[INFO] EPOCH: 3/20
Train loss: 0.930150, Train accuracy: 0.8129
Val loss: 0.910461, Val accuracy: 0.8746
 20%|██        | 4/20 [08:45<34:54, 130.92s/it]
[INFO] EPOCH: 4/20
Train loss: 0.930023, Train accuracy: 0.8130
Val loss: 0.910443, Val accuracy: 0.8746
 25%|██▌       | 5/20 [10:49<32:07, 128.48s/it]
[INFO] EPOCH: 5/20
Train loss: 0.930263, Train accuracy: 0.8127
Val loss: 0.910437, Val accuracy: 0.8746
 30%|███       | 6/20 [12:50<29:23, 125.98s/it]
[INFO] EPOCH: 6/20
Train loss: 0.930177, Train accuracy: 0.8128
Val loss: 0.910434, Val accuracy: 0.8746
 35%|███▌      | 7/20 [14:58<27:23, 126.46s/it]
[INFO] EPOCH: 7/20
Train loss: 0.930176, Train accuracy: 0.8128
Val loss: 0.910432, Val accuracy: 0.8746
 40%|████      | 8/20 [17:01<25:03, 125.28s/it]
[INFO] EPOCH: 8/20
Train loss: 0.930010, Train accuracy: 0.8130
Val loss: 0.910432, Val accuracy: 0.8746
 45%|████▌     | 9/20 [19:02<22:43, 123.98s/it]
[INFO] EPOCH: 9/20
Train loss: 0.930259, Train accuracy: 0.8127
Val loss: 0.910431, Val accuracy: 0.8746
 50%|█████     | 10/20 [21:02<20:29, 122.95s/it]
[INFO] EPOCH: 10/20
Train loss: 0.930176, Train accuracy: 0.8128
Val loss: 0.910431, Val accuracy: 0.8746
 55%|█████▌    | 11/20 [23:03<18:19, 122.20s/it]
[INFO] EPOCH: 11/20
Train loss: 0.930176, Train accuracy: 0.8128
Val loss: 0.910431, Val accuracy: 0.8746
 60%|██████    | 12/20 [25:08<16:24, 123.03s/it]
[INFO] EPOCH: 12/20
Train loss: 0.930010, Train accuracy: 0.8130
Val loss: 0.910431, Val accuracy: 0.8746
 65%|██████▌   | 13/20 [27:12<14:23, 123.33s/it]
[INFO] EPOCH: 13/20
Train loss: 0.930093, Train accuracy: 0.8129
```



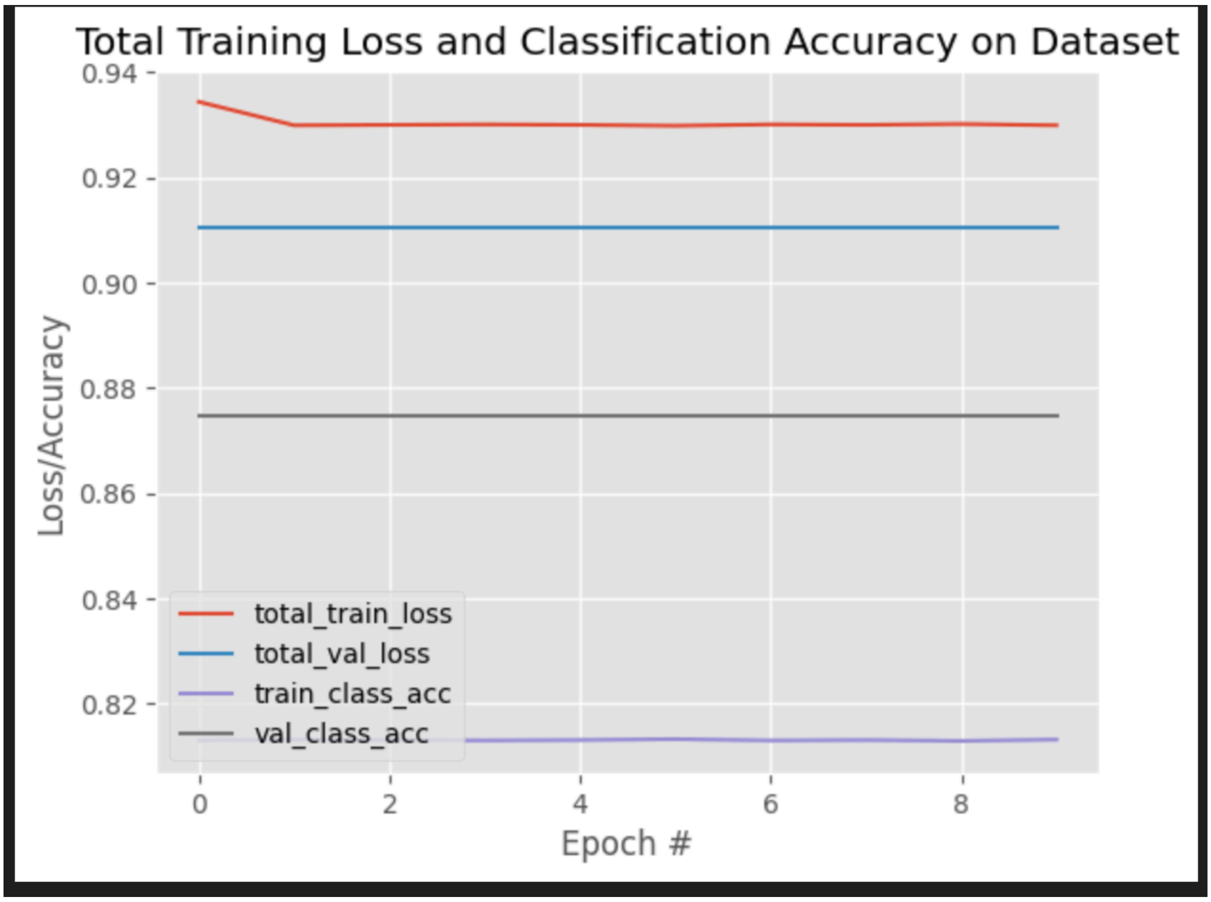
```
 65%|██████▌   | 13/20 [27:12<14:23, 123.33s/it]
[INFO] EPOCH: 13/20
Train loss: 0.930093, Train accuracy: 0.8129
Val loss: 0.910431, Val accuracy: 0.8746
 70%|███████   | 14/20 [29:14<12:18, 123.12s/it]
[INFO] EPOCH: 14/20
Train loss: 0.930093, Train accuracy: 0.8129
Val loss: 0.910431, Val accuracy: 0.8746
 75%|███████▌  | 15/20 [31:22<10:22, 124.53s/it]
[INFO] EPOCH: 15/20
Train loss: 0.930093, Train accuracy: 0.8129
Val loss: 0.910431, Val accuracy: 0.8746
 80%|████████  | 16/20 [33:25<08:16, 124.08s/it]
[INFO] EPOCH: 16/20
Train loss: 0.930176, Train accuracy: 0.8128
Val loss: 0.910431, Val accuracy: 0.8746
 85%|████████▌ | 17/20 [35:30<06:12, 124.17s/it]
[INFO] EPOCH: 17/20
Train loss: 0.930176, Train accuracy: 0.8128
Val loss: 0.910431, Val accuracy: 0.8746
 90%|█████████ | 18/20 [37:40<04:12, 126.08s/it]
[INFO] EPOCH: 18/20
Train loss: 0.930176, Train accuracy: 0.8128
Val loss: 0.910431, Val accuracy: 0.8746
 95%|█████████▌| 19/20 [39:46<02:06, 126.03s/it]
[INFO] EPOCH: 19/20
Train loss: 0.930010, Train accuracy: 0.8130
Val loss: 0.910431, Val accuracy: 0.8746
100%|██████████| 20/20 [41:51<00:00, 125.59s/it]
[INFO] EPOCH: 20/20
Train loss: 0.930093, Train accuracy: 0.8129
Val loss: 0.910431, Val accuracy: 0.8746
[INFO] total time taken to train the model: 2511.89s
```


Total Training Loss and Classification Accuracy on Dataset
Loss/Accuracy
1.10
1.05
1.00
0.95
0.90
0.85
0.80
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epoch #
total_train_loss
total_val_loss
train_class_acc
val_class_acc

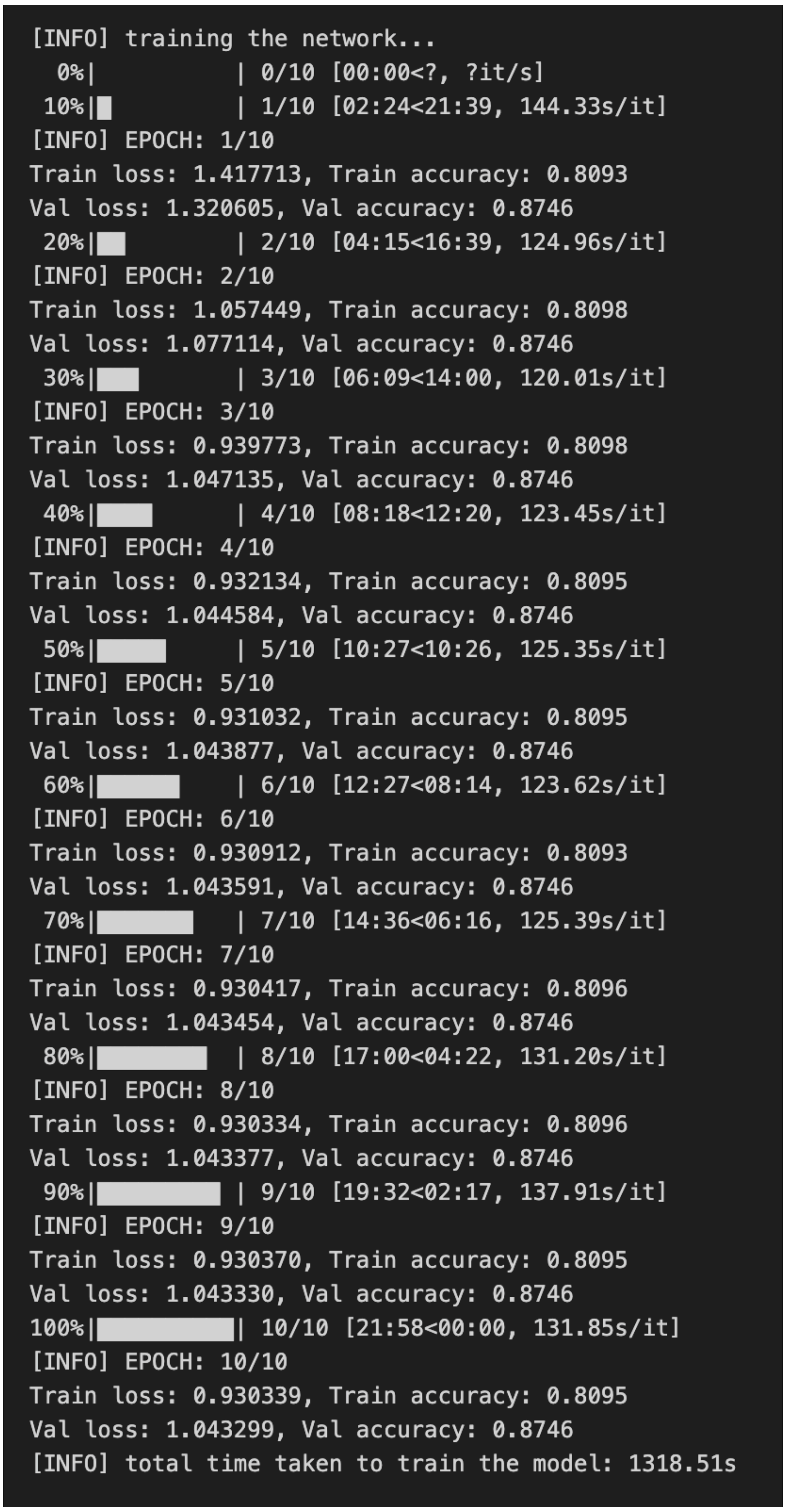
RUN 3:



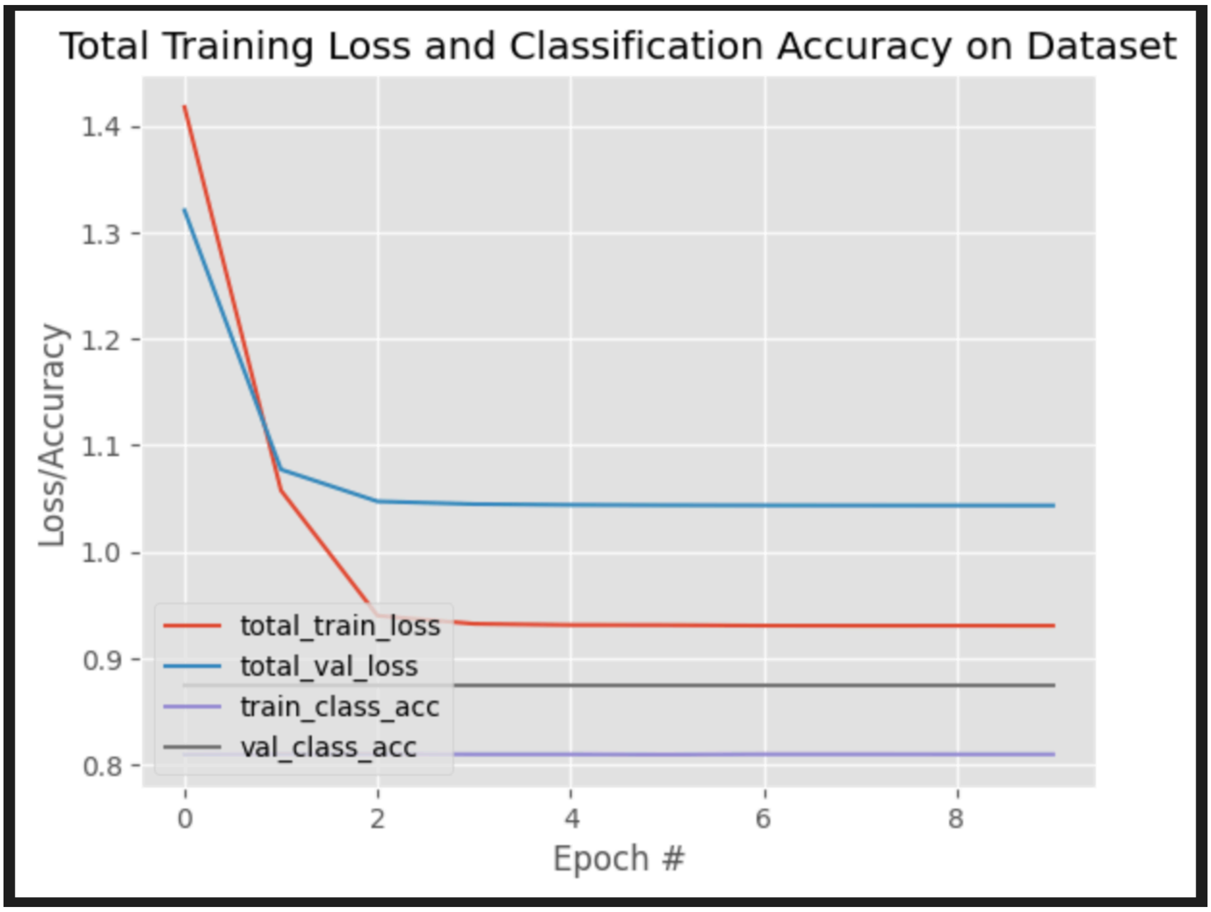
```
[INFO] training the network...
 10%|█         | 1/10 [02:23<21:34, 143.84s/it]
[INFO] EPOCH: 1/10
Train loss: 0.934391, Train accuracy: 0.8129
Val loss: 0.910431, Val accuracy: 0.8746
 20%|██        | 2/10 [04:25<17:28, 131.01s/it]
[INFO] EPOCH: 2/10
Train loss: 0.929927, Train accuracy: 0.8131
Val loss: 0.910431, Val accuracy: 0.8746
 30%|███       | 3/10 [06:28<14:49, 127.02s/it]
[INFO] EPOCH: 3/10
Train loss: 0.930010, Train accuracy: 0.8130
Val loss: 0.910431, Val accuracy: 0.8746
 40%|████      | 4/10 [08:33<12:37, 126.26s/it]
[INFO] EPOCH: 4/10
Train loss: 0.930093, Train accuracy: 0.8129
Val loss: 0.910431, Val accuracy: 0.8746
 50%|█████     | 5/10 [10:39<10:31, 126.31s/it]
[INFO] EPOCH: 5/10
Train loss: 0.930010, Train accuracy: 0.8130
Val loss: 0.910431, Val accuracy: 0.8746
 60%|██████    | 6/10 [12:39<08:16, 124.22s/it]
[INFO] EPOCH: 6/10
Train loss: 0.929845, Train accuracy: 0.8132
Val loss: 0.910431, Val accuracy: 0.8746
 70%|███████   | 7/10 [14:47<06:15, 125.25s/it]
[INFO] EPOCH: 7/10
Train loss: 0.930093, Train accuracy: 0.8129
Val loss: 0.910431, Val accuracy: 0.8746
 80%|████████  | 8/10 [16:53<04:11, 125.74s/it]
[INFO] EPOCH: 8/10
Train loss: 0.930010, Train accuracy: 0.8130
Val loss: 0.910431, Val accuracy: 0.8746
 90%|█████████ | 9/10 [19:00<02:05, 125.99s/it]
[INFO] EPOCH: 9/10
Train loss: 0.930176, Train accuracy: 0.8128
Val loss: 0.910431, Val accuracy: 0.8746
100%|██████████| 10/10 [21:07<00:00, 126.75s/it]
[INFO] EPOCH: 10/10
Train loss: 0.929927, Train accuracy: 0.8131
Val loss: 0.910431, Val accuracy: 0.8746
[INFO] total time taken to train the model: 1267.50s
```


Total Training Loss and Classification Accuracy on Dataset
Loss/Accuracy
Epoch #
0.94
0.92
0.90
0.88
0.86
0.84
0.82
0
2
4
6
8
total_train_loss
total_val_loss
train_class_acc
val_class_acc

RUN 4:



```
[INFO] training the network...
  0%|          | 0/10 [00:00<?, ?it/s]
 10%|█         | 1/10 [02:24<21:39, 144.33s/it]
[INFO] EPOCH: 1/10
Train loss: 1.417713, Train accuracy: 0.8093
Val loss: 1.320605, Val accuracy: 0.8746
 20%|██        | 2/10 [04:15<16:39, 124.96s/it]
[INFO] EPOCH: 2/10
Train loss: 1.057449, Train accuracy: 0.8098
Val loss: 1.077114, Val accuracy: 0.8746
 30%|███       | 3/10 [06:09<14:00, 120.01s/it]
[INFO] EPOCH: 3/10
Train loss: 0.939773, Train accuracy: 0.8098
Val loss: 1.047135, Val accuracy: 0.8746
 40%|████      | 4/10 [08:18<12:20, 123.45s/it]
[INFO] EPOCH: 4/10
Train loss: 0.932134, Train accuracy: 0.8095
Val loss: 1.044584, Val accuracy: 0.8746
 50%|█████     | 5/10 [10:27<10:26, 125.35s/it]
[INFO] EPOCH: 5/10
Train loss: 0.931032, Train accuracy: 0.8095
Val loss: 1.043877, Val accuracy: 0.8746
 60%|██████    | 6/10 [12:27<08:14, 123.62s/it]
[INFO] EPOCH: 6/10
Train loss: 0.930912, Train accuracy: 0.8093
Val loss: 1.043591, Val accuracy: 0.8746
 70%|███████   | 7/10 [14:36<06:16, 125.39s/it]
[INFO] EPOCH: 7/10
Train loss: 0.930417, Train accuracy: 0.8096
Val loss: 1.043454, Val accuracy: 0.8746
 80%|████████  | 8/10 [17:00<04:22, 131.20s/it]
[INFO] EPOCH: 8/10
Train loss: 0.930334, Train accuracy: 0.8096
Val loss: 1.043377, Val accuracy: 0.8746
 90%|█████████ | 9/10 [19:32<02:17, 137.91s/it]
[INFO] EPOCH: 9/10
Train loss: 0.930370, Train accuracy: 0.8095
Val loss: 1.043330, Val accuracy: 0.8746
100%|██████████| 10/10 [21:58<00:00, 131.85s/it]
[INFO] EPOCH: 10/10
Train loss: 0.930339, Train accuracy: 0.8095
Val loss: 1.043299, Val accuracy: 0.8746
[INFO] total time taken to train the model: 1318.51s
```



The straight line like nature of the accuracy vs epoch graph and the high loss indicates there is an issue with the model and that it may be underfitting. We try to diagnose it in subsequent runs by increasing learning rate, number of epochs and batch size. However, none of these leads to any noticeable improvement.

This raises the possibility that the model is simply not complicated enough to process the input dataset.

7. MODEL DEPLOYMENT:

To use the model to perform predictions on images, place the images to test in the ‘input_image’ directory and run the ‘image_run.py’ file.

To use the model to perform predictions on video, place the video to test in the ‘input_video’ directory and run the ‘video_run.py’ file. Ensure that the video file’s name is ‘video_0001.mp4’. If you wish to input a video which doesn’t have this name, go to the ‘input_path’ variable in ‘video_run.py’ and change it the path of your video.

To use the model to perform predictions on live video via webcam, go to the ‘video_run.py’ file and change the ‘use_webcam’ variable to True instead of false.

Ensure that there are no restrictions on camera access while running.

8. SHORTCOMINGS:

This model mostly fails to predict the desired output. This is most likely due to the fact that the model is simply unable to learn enough features of the images as mentioned above in the training process.

Possible solutions :

- Use a more complicated model that can learn and output a greater number of features from the images.
- Modify the code such that the classifier does not focus on the whole image but only on the parts in the bounding boxes. (Currently the classifier runs on the entire image.)