Assignment 1

Answer 1

1. a)
Downloaded the “train 32x32.mat” from the link provided. To initialize Weights and Biases, firstly, we needed to install “wandb” in our python environment. It was done using “pip install wandb”. Executing the command “wandb.login()” logged us into weights and biases, and we are now ready to go.

Uploaded the “train 32x32.mat” onto my google drive and mounted the drive to the Colab notebook. Using the “scipy.io” library, stored the matrix as an object variable and got to know the size of the matrix.

The size of the matrix was 73257. Using list slicing, I sliced the both the arrays (images and labels) into three chunks, 20% for the validation set, 10% for the testing set, and the rest for the training set.

Now the elements in each of the set look like this

```
print("val_data: ", val_data.shape)
print("val_labels: ", val_labels.shape)
print("test_data: ", test_data.shape)
print("test_labels: ", test_labels.shape)
print("train_data: ", train_data.shape)
print("train_labels: ", train_labels.shape)
```

```
val_data:  (14651, 32, 32, 3)
val_labels:  (14651, 1)
test_data:  (7325, 32, 32, 3)
test_labels:  (7325, 1)
train_data:  (51281, 32, 32, 3)
train_labels:  (51281, 1)
```

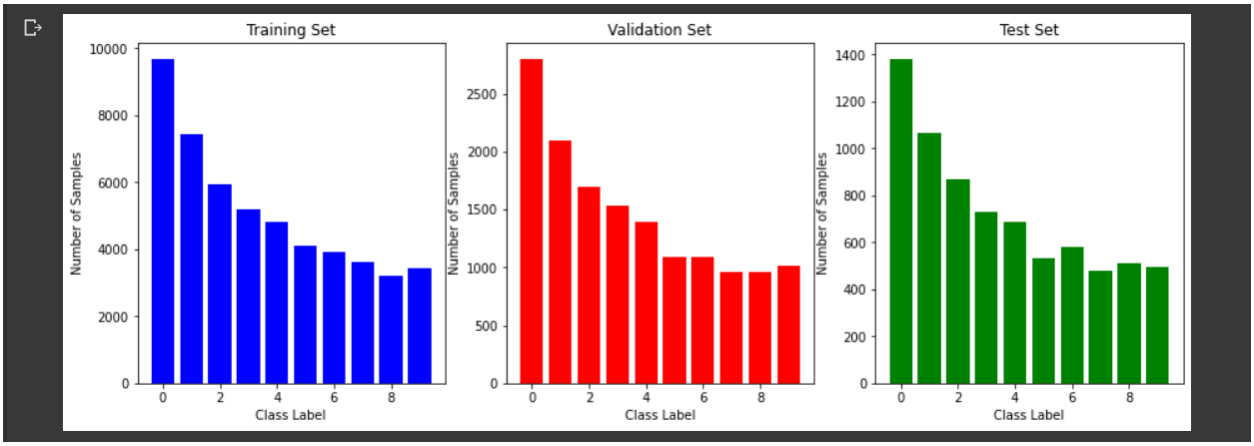
b)
To create the custom data loaders, I have prepared a class named "HouseNumbersDataset". The functionality of each function in the class is as follows.

- __init__: This method is called when the class instance is created. It takes in a path to a .mat file containing the dataset and an optional transform argument, which can be used to apply image transformations to the dataset. The method loads the data and target arrays from the .mat file using the loadmat function from the scipy.io module and stores them as class attributes.
- __len__: This method returns the number of samples in the dataset, which is the size of the first dimension of the data array.
- __getitem__: This method takes an index and returns the corresponding image and target as a tuple. The image is loaded from the data array using the index and converted to a torch.tensor of type torch.float32. The target is loaded from the target array using the index and also converted to a torch.tensor of type torch.float32.

After defining the custom dataset class, the code loads the entire dataset into an instance of the HouseNumbersDataset class called all_data.

Finally, the code defines three data loaders for the train, validation, and test splits using the DataLoader class from PyTorch. Each data loader takes in the corresponding dataset (e.g., train_data for the train loader), sets the batch size to 100, and shuffles the data for the train loader only.

c)
This data was visualized using the matplotlib library. I plotted the bar graph for the training, validation, and testing set. The graph was plotted for the number of samples in each set vs the class labels. The result was obtained as shown below.



2.

a)

To create a CNN as described in the question, I have prepared the class “CNN_SVHN” which does the job required. The explanation of the class is mentioned below.

The CNN_SVHN class inherits from nn.Module, which is a PyTorch base class for all neural network modules. The class has two main methods:

- __init__: This method is the class's constructor and defines the neural network's layers. The network has two convolutional layers (conv1 and conv2) with 32 and 64 output channels, respectively, both with a kernel size of 3 and padding of 1. It also has a fully connected layer (fc) with an input size of 64 * 8 * 8 and an output size of 10 (one for each digit). The super function is called to initialize the torch.nn.Module base class.

- forward: This method defines the forward pass of the neural network. The input x is first passed through the first convolutional layer (conv1) and then through a ReLU activation function. The result is then passed through a max pooling layer (F.max_pool2d) with a kernel size of 2 and stride of 2. This process is repeated for the second convolutional layer (conv2) and max pooling layer. The resulting feature maps are then flattened using the view method and passed through the fully connected layer (fc). The output of the final layer is returned.

Overall, this neural network model takes in an input image tensor of shape (batch_size, 3, 32, 32) (where batch_size is the number of images in the batch) and outputs a tensor of shape (batch_size, 10) representing the predicted class scores for each image in the batch.

b)
I have written the code provided in the Colab notebook to train the CNN using the Cross-Entropy Loss function. The explanation for the code is as follows:-

The code builds a convolutional neural network (CNN) model using the PyTorch framework, defining a loss function and optimizer, loading and transforming data, and calculating the accuracy and loss of the model during training and testing. The explanation for each part of the code is mentioned below:

- **loss_calculator function**: This function takes in a model and a dataloader and performs a forward and backward pass on the model using the given data. It calculates the model's loss and updates the model's weights. The running loss is returned by dividing the total loss by the number of samples in the dataloader.

- **accuracy_calculator function**: This function takes in a model and a dataloader and calculates the model's accuracy on the given data. It does this by predicting the class label for each input in the dataloader using the trained model and comparing it to the actual label. The accuracy is returned as a percentage.

- **wandb.init function**: This initializes a new run on the Weights & Biases platform, which tracks and visualizes the model's performance during training.

CNN_SVHN class: This is a user-defined class that defines the architecture of the CNN model. The details of this class are not shown in the code snippet.

- nn.CrossEntropyLoss function: This is a PyTorch loss function that combines the softmax activation function and the negative log-likelihood loss.

- **optim.SGD function**: This is a PyTorch optimizer that uses stochastic gradient descent with momentum to update the weights of the model during training.

- **transforms.Compose function**: This is a PyTorch function that combines multiple data transformations to be applied to the input data.

- **datasets.SVHN function**: This is a PyTorch function that loads the SVHN dataset and applies the specified data transformations.

- **torch.utils.data.DataLoader function**: This PyTorch function creates a data loader that batches and shuffles the data for training or testing.

In summary, this code builds a CNN model for the SVHN dataset, defines the loss function and optimizer, loads and transforms the data, and calculates the accuracy and loss of the model during training and testing. It also uses Weights & Biases to track and visualize the model's performance during training.

Now we train the CNN model for a total of 10 epochs using the previously defined loss_calculator and accuracy_calculator functions. For each epoch, it calculates and logs the training and validation loss and accuracy using the Weights & Biases platform. It then prints out the calculated values and the current epoch number.

By calling the **loss_calculator** and **accuracy_calculator** functions on both the trainloader and testloader, the model is trained and evaluated on both the training and testing data. This is important for verifying that the model has learned to generalize to new, unseen data, rather than simply memorizing the training data.

The training code mentioned trains and evaluates a CNN model for image classification using PyTorch and Weights & Biases. The accuracy and loss values logged by the Weights & Biases platform can be used to track the model's performance during training and make adjustments as necessary to improve its accuracy.

In the 10th epoch, we get the results as shown below.

```
Training Loss:  0.353168126952928
Training Accuracy: 91.08891 %
Validation Loss:  0.9720539477990175
Validation Accuracy: 86.35526 %

Epochs completed:  10

Finished Training
```

We have achieved training accuracy close to 91% and validation accuracy close to 87%, along with the training loss of 0.3531 and validation loss of 0.9720.

c)
I have written the code provided in the Colab notebook to find the Accuracy and F1-Score on the test set. The explanation for the code is as follows:-

The **plot_confusion_matrix** function is defined to convert the conf_matrix into a normalized form and log it as an image using the Weights & Biases platform. The conf_matrix is a 10x10 array that contains the number of true and false predictions for each of the 10 classes.

The correct and total variables are initialized to zero, and the conf_matrix is initialized to a 10x10 array of zeros. The model is run on the test data using a with **torch.no_grad()** block and the accuracy is calculated as the ratio of correct predictions to the total number of predictions. The f1_score is also calculated for the test data using the scikit-learn function with the weighted option to take class imbalance into account.

The confusion_matrix function is called to calculate the confusion matrix for the test data, and the resulting matrix is added to the **conf_matrix** variable. This is done by passing the true labels and predicted labels as NumPy arrays using the detach() method and setting the labels argument to a list of the 10 class labels. This accumulates the confusion matrix over all the test data.

The accuracy, F1-score, and confusion matrix are then logged to the Weights & Biases platform using the wandb.log method, and the plot_confusion_matrix function is called to log the normalized confusion matrix as an image. Finally, the accuracy and F1-score are printed to the console.

The result is as follows:

```
[[0.84518349 0.01892202 0.0108945  0.01548165 0.00917431 0.00229358
  0.05848624 0.00688073 0.00745413 0.02522936]
 [0.00902138 0.92469112 0.00451069 0.01117866 0.02412238 0.00196117
  0.0035301  0.01686605 0.00137282 0.00274564]
 [0.00241022 0.0120511  0.89129911 0.03302    0.01590745 0.00674861
  0.00843577 0.01277416 0.00457942 0.01277416]
 [0.00346981 0.02741152 0.01561416 0.82199861 0.01249133 0.03539209
  0.01769604 0.00624566 0.01596114 0.04371964]
 [0.00475624 0.03369005 0.00832342 0.01902497 0.90566786 0.00198177
  0.00911613 0.00435989 0.00396354 0.00911613]
 [0.00377517 0.00838926 0.00713087 0.06124161 0.01384228 0.8238255
  0.0591443  0.00293624 0.00629195 0.01342282]
 [0.02023268 0.01517451 0.00354072 0.03945372 0.01922104 0.02731411
  0.85584219 0.00404654 0.00809307 0.00708144]
 [0.00346706 0.04606241 0.02179297 0.01634473 0.00396236 0.0089153
  0.00693413 0.88311045 0.00148588 0.00792472]
 [0.02228916 0.01325301 0.01024096 0.0753012  0.01566265 0.02168675
  0.11144578 0.00180723 0.68975904 0.03855422]
 [0.02570533 0.01818182 0.03510972 0.0200627  0.0137931  0.01442006
  0.01065831 0.00940439 0.0031348  0.84952978]]
Test Accuracy: 86.35526%
Test F1-Score: 0.85
```

Firstly we have the confusion matrix, then the test accuracy as 86%, and F1-Score as 0.85.

3. a)
I have written the code for training another classification model with a fine-tuned Resnet-18 architecture. The explanation for the code is as below.

- **accuracy_calculator(model, dataloader):** This function takes in a pre-trained model and a dataloader for a dataset, and returns the model's accuracy on the

given dataset. It computes the accuracy by comparing the predicted labels of the model with the ground truth labels of the dataset.

- **loss_calculator_RESNET18(model, dataloader)**: This function takes in a pre-trained model and a dataloader for a dataset and returns the loss of the model on the given dataset. It computes the loss using the cross-entropy loss function and performs a single step of gradient descent on the model's parameters using the optimizer object.

The main code performs the following steps:

- Firstly, it initializes the Weights and Biases (W&B) library for logging the training process.
- Then, I load the pre-trained ResNet-18 model from the **torchvision.models module**.
- Now, I freeze all the layers in the model except the last one, which is used for fine-tuning.
- I define the loss function as cross-entropy loss and the optimizer as Stochastic Gradient Descent (SGD) with a learning rate of 0.001 and momentum of 0.9.

- The following steps perform training of the model for a single epoch:

    - Firstly, I compute the training loss and accuracy using the loss_calculator_RESNET18 and accuracy_calculator functions.
    - Then, I compute the validation loss and accuracy using the same functions on the validation dataset.
    - Logging of the training and validation loss and accuracy using W&B is done.

b)
The explanation for the code is provided below:-

The code starts by calling the accuracy_calculator function, which inputs the model and the test dataset (testloader) and returns the model's accuracy on the test dataset. The resulting accuracy is stored in the variable **test_acc**.

The code then initializes two empty lists, predictions, and ground_truth, which will store the predicted labels and ground truth labels, respectively, for each example in the test dataset.

Next, the code iterates over each batch of examples in the test dataset. For each batch, it extracts the input images and labels, moves them to the device (GPU or CPU) specified in the model, and passes the images through the model to get predicted outputs. The predicted class labels are obtained by taking the argmax of the predicted outputs. The predicted and ground truth labels for the current batch are then appended to the predictions and **ground_truth** lists.

After iterating over all batches in the test dataset, the code calculates the F1-Score using the f1_score function from scikit-learn library, with the macro parameter for averaging the F1-Score across all classes. The resulting F1-Score is stored in the variable f1.

The code then logs the accuracy and F1-Score to the Weights & Biases platform using the wandb.log() function and prints them to the console.

Finally, the code computes the confusion matrix using the confusion_matrix function from the scikit-learn library, which compares the predicted labels with the ground truth labels for each class and produces a matrix that shows the number of true positive, true negative, false positive, and false negative predictions for each class. The resulting confusion matrix is printed to the console and logged to the Weights & Biases platform.

c)

The explanation for the code is provided below:-

The code extracts features from a pre-trained ResNet-18 model, using the last layer of the model as a feature extractor, and then visualizes these features in 2D and 3D using t-SNE.

First, the code loads the ResNet-18 model with pre-trained weights using **torchvision.models.resnet18(pretrained=True)** and moves it to the device GPU for faster computation.

Next, the last layer of the model (the classification layer) is removed by creating a new classifier which consists of all the layers of the model except the last one. This is done by creating a new Sequential model with all the layers of the original model except the last one using ***list(model.children())[:-1]**.

All layers in the new classifier are then frozen by setting the **requiresGrad** attribute to False for all parameters.

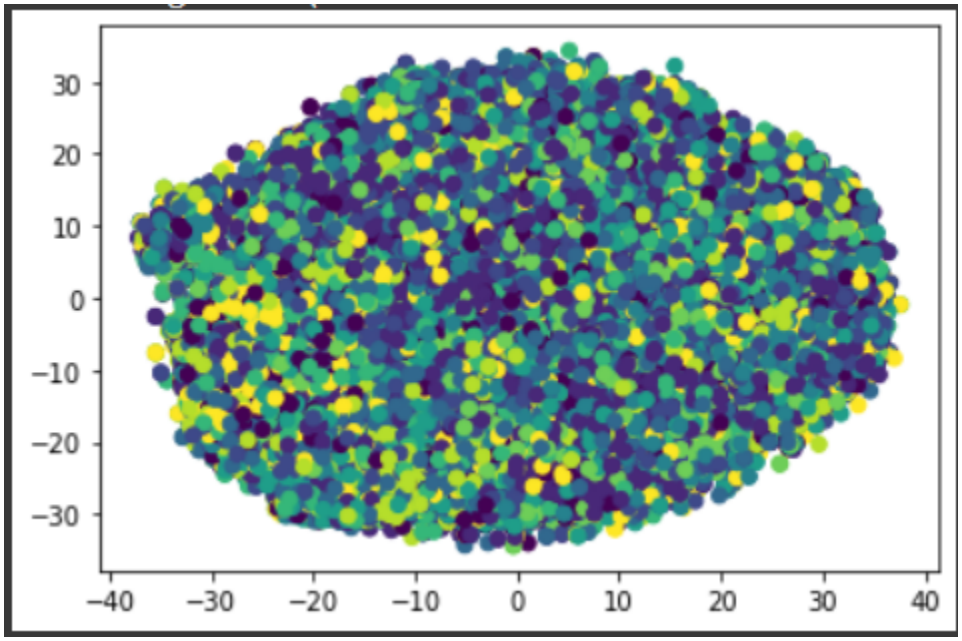
Next, the code extracts features from the training and validation sets using the new classifier. The images are passed through the new classifier to obtain a feature vector for each batch of images and labels in the training and validation sets. The resulting feature vectors are then concatenated along the batch dimension to obtain feature matrices features_train and features_val, and the corresponding labels are concatenated to obtain labels_train and labels_val.

Finally, the code applies t-SNE to the feature matrices features_train and features_val to obtain low-dimensional embeddings of the feature vectors, which are then visualized in 2D and 3D using matplotlib.

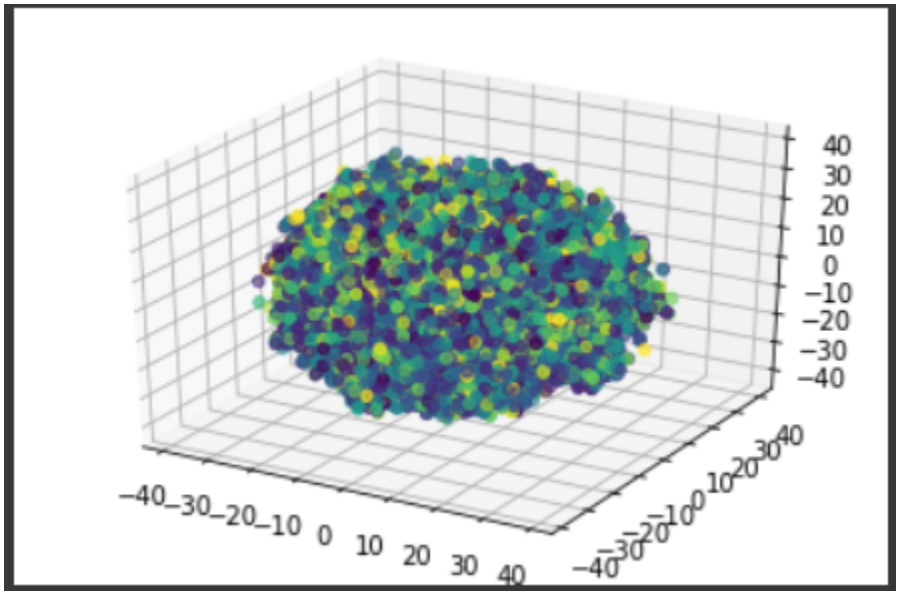
The 2D visualization is created using **plt.scatter(features_tsne[:, 0], features_tsne[:, 1], c=labels_train)**, which plots the first two dimensions of the t-SNE embeddings as a scatter plot, where each point is colored according to its class label.

The 3D visualization is created using **ax.scatter(features_tsne[:, 0], features_tsne[:, 1], features_tsne[:, 2], c=labels_val)**, which plots the first three dimensions of the t-SNE embeddings as a scatter plot in 3D, where each point is colored according to its class label.

The feature space visualized in 2-D using **TSNE** looks like this:-



And in 3-D it looks like this:-



4. a)
The data augmentation techniques used in this question are Random Cropping, Random Erasing, and Random Rotation.

- **Random Cropping**

The random crop function accepts an input image img in the form of a numpy array and a tuple sz indicating the crop's intended size in the format (crop height, crop width). It arbitrarily selects a portion of the input image to crop with the chosen crop size.

Implementation of the function is as follows:

Using the shape attribute of the numpy array, the function determines the height and width of the supplied image.

Then, it takes the **sz** tuple's intended crop height and width.

Using the **randint** method from the **numpy.random** package, the function produces the two random numbers y and x. The cropped area will be completely contained within the supplied image due to the random numbers' range.

The function then uses numpy array slicing to return the input image's cropped portion. The cropped section should begin at (y, x) and extend to (y+ch, x+cw) in the height and width dimensions, respectively, according to the slicing notation [y:y+ch, x:x+cw,:]. If the input picture has a third dimension, the: at the end of the slicing notation is used to include all channels.

The random crop function generally creates a randomly selected crop of the supplied image with the chosen crop size. This is a typical method of data augmentation used in deep learning to broaden the diversity of training data and enhance model generalisation. Note that the code is identical to the random crop function described in the preceding question, with the exception that it uses img.shape[0] and img.shape[1] to extract the input image's height and width rather than img.shape[:2].

- **Random Erasing**

The random erasing function performs the "random erasing" data augmentation technique with a certain degree of probability to an input picture img that is a numpy array. With this method, a random rectangular section in the input image is erased and its pixel values are replaced with random ones. Implementation of the function is as follows:

The function initially determines if a chance of 0.5 should be used for the random erasing. The **np.random.uniform()** function returns the input image unchanged if the random integer it produces is bigger than 0.5.

The function uses the shape attribute of the **numpy** array to determine whether random erasing should be performed and then extracts the input image's height, width, and number of channels.

Using np.random.uniform, the function creates a random target area for the rectangular region that will be erased (). The target region ranges from 2% to 40% of the overall input image area.

For the rectangular area that is to be erased, the method generates a random aspect ratio between 0.3 and 3.3.

The function uses the target area and aspect ratio to determine the height and width of the rectangular region that needs to be erased. The height is equal to the square root of the target area times the aspect ratio, and the width is equal to the target area divided by the aspect ratio.

The function uses the randint method from the numpy.random package to create the two random numbers x1 and y1. The rectangular region to be erased will be entirely contained within the input image since the range of the random integers is such that it does so.

Using np.random.uniform, the method uses random pixel values to replace the pixels in the rectangular area that has to be erased (). If the source image only contains one channel,

The function then returns the input image with the applied random erasure.

Simply put, the random erasing function uses a chance of 0.5 to apply the random erasing data augmentation technique to the input image. This method is frequently applied in deep learning to broaden the diversity of the training data and enhance the model's generalizability.

- **Random Rotation**

The function **random_rotation** rotates the image to a certain degree, which is randomly generated.

The rotation matrix M for the randomly generated image is computed using OpenCV's cv2.getRotationMatrix2D() function.

Finally, the rotation is applied to the input image using OpenCV's cv2.warpAffine().

These are the three methods using which data augmentation has been done. After augmenting the data, the data is appended back again to the dataset. This increases the data upon which the model can be trained on.

b)
Same steps as Question 1.3. (a) are being used in this part to train the model. The change in this training is of the dataset. Now we have more data to train on. Hence, due to this, the accuracy of our model has increased in this part.
The stats for this training is as follows:-

```
Training Loss:  0.3330874022242804
Training Accuracy: 91.24452 %
Validation Loss:  0.1510308145558665
Validation Accuracy: 96.07893 %
```

6.
There are 3 models in the entirety of the question. The first one is the self-trained model, which we are creating and training ourselves. The second one is the pre-trained model, which is pre-trained already, and we are using the same weights and biases. The third one is the model in which we are synthetically adding more data using data augmentation to increase the data on which our model can train on and increase the accuracy.
The best avg. the accuracy obtained while training the model multiple times was observed in the last model with data augmentation. The avg accuracy and validation accuracy achieved in this model was ~90% and ~95%, respectively. The reason for this model to achieve such high accuracy was due to the increase in data size by synthetically adding more data via different data augmentation methods like random cropping, random erasing, and random rotation.
The worst model performance was obtained in the second model, which was pre-trained. The accuracy was too low in this model due to the fact it was already pre-trained on some random samples, which had very little to no connection whatsoever with the dataset which provided the model's validation with. Hence, the accuracy was really low in this part.

Answer 2

Data Preprocessing

Firstly, I downloaded the VOC Segmentation Dataset and mounted it onto the drive. The VOC Segmentation Dataset contained two folders named "images" and "labels" and inside "images" and "labels" folders contained the images and corresponding labels.

I copied the file names separately into two lists named "image_files" and "mask_files" using the os library in python.

After this operation, the sizes of the lists were 1464, i.e., both lists contained the images and their corresponding labels.

1. a)
Following the same methods in the previous answer, I sliced both the datasets(images and masks) into three chunks, 20% for the validation set, 10% for the testing set, and the rest for the training set.

b)
I have written the code for creating a custom dataset for image and mask data representation in data segmentation. The explanation for the code is as follows.

- Firstly, import necessary packages: the "os" package for accessing the operating system, "random" for generating random numbers, "torch" and "torchvision" for machine learning and computer vision functionalities, and "Dataset" and "DataLoader" from "torch.utils.data" for creating data loaders have been done.

- I have set seed for reproducibility purposes. This ensured that the same random numbers were generated each time the script is run, which can be helpful for debugging and replicating results.

- Now, I have defined a transformation pipeline that resizes the images to 256x256 pixels and converts them to PyTorch tensors.

- Creation of a custom dataset class named "VOC2012Dataset" has been done that takes two arguments: the image files and mask files. The class has three methods: "__init__" initializes the class by setting the image and mask files and the transformation pipeline, "__len__" returns the number of images in the dataset, and "__getitem__" loads and preprocesses an image and its corresponding mask file.

- Now, instances of the custom dataset class have been created for the training, validation, and test splits. The script assumes that the image and mask files are stored in the directories "**/content/gdrive/MyDrive/VOC Segmentation Dataset/images**" and "**/content/gdrive/MyDrive/VOC Segmentation Dataset/masks**," respectively. (which I have already done in the previous steps)

- Now I have defined data loaders for the training, validation, and test splits using the custom dataset instances. The data loaders batch the data, shuffle the training data, and use four worker threads for loading the data in parallel. The batch size is set to 4

2. a)
I have written the code for training the **fcnresnet50** model using pre-defined network weights using an appropriate loss function. The explanation for the code is as follows.

The PyTorch code defines two different models for semantic segmentation using the Fully Convolutional Network (FCN) architecture based on the ResNet-50 and FCN-ResNet50 models. The code also trains the FCN-ResNet50 model on a dataset using the Adam optimizer and logs the training loss and accuracy to the Weights & Biases (wandb) platform.

The FCNResNet50 class defines a model that first loads a pre-trained ResNet50 model and removes the final average pooling and fully connected layers. Then, it adds an upsampling layer to increase the spatial resolution of the feature maps and a convolutional layer to map the feature maps to the desired number of classes. The forward method of this class takes an input tensor, **x** and returns the output tensor after passing through the defined layers.

The code then defines hyperparameters such as the batch size, learning rate, number of epochs, and number of classes. It also loads a pre-trained FCN-ResNet50 model and changes its classification and aux_classification heads to make them learnable. It then sets the requires_grad flag to false for all the parameters except the last layer of the classifier head. It then defines the loss function as Cross Entropy Loss and the optimizer as Adam.

The code then trains the model for the defined number of epochs using the Adam optimizer and logs the training loss and accuracy to the wandb platform. In each epoch, the training loop iterates over batches of images and labels, passes the images through the model to get the output tensor, computes the loss, backpropagates the loss and

updates the model parameters using the Adam optimizer. It also computes the accuracy, average training loss and accuracy per epoch, and logs them to wandb. Finally, the code prints the epoch number, training loss, and accuracy after each epoch.

The accuracy is shown below.

```
Epoch 1 train_loss 0.0044217017968974 73 train_acc 33.908628346766726
Epoch 2 train_loss 0.0042160188420994 08 train_acc 33.9059945050491
Epoch 3 train_loss 0.0038886075826544348 train_acc 33.90925622741801
Epoch 4 train_loss 0.0041137081916912165 train_acc 33.91182420499815
Epoch 5 train_loss 0.0039767401135197 52 train_acc 33.900202079859184
Epoch 6 train_loss 0.0043746933923599 03 train_acc 33.91279950088012
Epoch 7 train_loss 0.0042365793485076 41 train_acc 33.90928735003243
```

b)
I have written the code for reporting the performance of the test set in terms of classwise IoU, mean IoU (mIoU), Average Precision (AP) and mean Average Precision (mAP). The explanation of the function is as follows:-

Explanation for the function:- **compute_metrics**

A few variables are first initialized by the code: Pixel accuracy, f1Score, ious, prec, rec, and mlp are arrays of size num classes that will be used to store various metrics for each class. num classes denote the number of classes in the segmentation task. A confusion matrix is a 2D array of size num classes x num classes that will be used to calculate the confusion matrix for the segmentation task.

The compute metrics function, which is defined later in the code, asks for the segmentation model, the test data loader, and the platform on which to run the model. The function initially initializes the metrics arrays to 0 and switches the model to evaluation mode.

The function then iterates over the test data loader, computing the model's predictions and updating the various metrics arrays for each batch of inputs and targets. The function computes the true positives, false positives, false negatives, and true negatives for each class, and then updates the confusion matrix and computes the various metrics using these values. Pixel accuracy, precision, recall, f1 score, intersection over union (IoU), and average precision are some of these measurements (ap).

The function also creates the confusion matrix and returns it together with the mean values of the relevant metrics across all classes.

To prevent division by zero errors, a modest constant value (1e-8) is used in the denominator for calculating precision, recall, f1 score, and IoU.

Explanation for the function:- **compute_iou_range**

The Intersection over Union (IoU) metric is computed for a range of IoU thresholds for a semantic segmentation model using the compute iou range function defined in this code. The function requires three inputs: a model, a data loader for testing the model, and a computing platform.

The function first initializes a few variables. The number of classes in the dataset, 21, is the value set for num classes. range is an array of criteria from 0 to 1 in steps of 0.1 for which the IoU will be calculated. A tensor called thresholds is produced and sent from the range array to the designated device. In order to hold the IoU values for each class, metrics is started as a 2D array of zeros with shape (num classes, len(range)).

The code then starts looping over the test data loader. The inputs and targets are transferred to the designated device for each batch of inputs and targets, and the model is then run on the inputs to produce the output logits. By taking the argmax of the logits along the channel dimension, the predicted class for each pixel is then determined.

Using the anticipated and target class masks, the function calculates the true positives, false positives, false negatives, and true negatives for each class i. After that, the IoU is calculated as true positives / (true positives + false positives + false negatives + 1e-8), with 1e-8 added to prevent division by zero errors.

The function then uses the thresholds = iou comparison to calculate the number of IoUs that are greater than or equal to each threshold and adds these counts to the relevant row of the metrics array for the current class i.

The function computes the mean IoU for each class by averaging the IoU counts across all thresholds after the loop over the test data loading is finished, and then computes the overall mean IoU by averaging the class IoUs.

The function produces two values: mean iou, which is the average IoU across all classes, and class ious, which is an array of mean IoUs for each class.

The following was the result obtained

```
# compute metrics on test data
pixel_acc, mean_f1_score, mean_iou, mean_precision, mean_recall, mean_ap = compute_metrics(model, test_loader, device)

# print results
print(f"Pixel Accuracy: {pixel_acc:.4f}")
print(f"Mean F1 Score: {mean_f1_score:.4f}")
print(f"Mean IoU: {mean_iou:.4f}")
print(f"Mean Precision: {mean_precision:.4f}")
print(f"Mean Average Precision: {mean_ap:.4f}")
```

```
Pixel Accuracy: 0.692356
Mean F1 Score: 0.8569
Mean IoU: 0.2765
Mean Precision: 0.7836
Mean Average Precision: 0.7025
```

3. a)
The data augmentation techniques used in this question are Random Cropping, Random Erasing, and Random Rotation.
Explanations for these functions have been provided in Q1.

b)
Synthetic data has been added to the dataset. Now the model is trained on the dataset by following the same steps as in Question 2.2.(a).

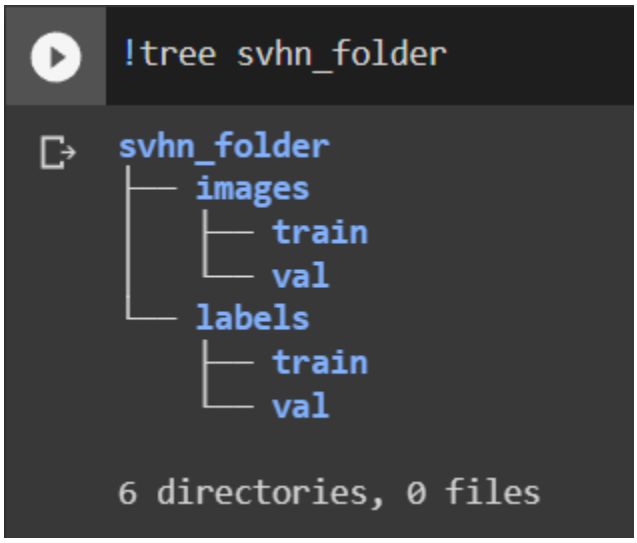
c)
There are two models in this part of the question. The first one is the pre-trained fcnresnet50 model using pre-defined network weights and the second one is the same model but trained on a dataset with additional data provided using data augmentation techniques like random cropping, random erasing, and random rotation.
The best accuracy obtained was in the second model. This is due to the fact that synthetically more training data has been added for the model to train better. As more data has been added to the model, the model is being allowed to explore more types of data input and would cover a broader aspects in which data can be presented to it. Hence the model gives the best accuracy in this part.

Answer 3

Preprocessing step

Firstly, I downloaded the dataset and uploaded the zip file to my google drive. The file was huge. Hence I was not able to open any of the files.
After uploading the zip, I extracted the zip file's contents using the python library “zipfile” onto my Colab notebook.

On the Colab notebook, I made a folder structure as shown in the below tree diagram.



This structure was required for me to train the model correctly.

1. a)
I split the data into the desired breakdown and placed the training and validation sets for both images and the labels into their respective folders. (Folder structure is shown above). Then initialized weights and biases.

b)
The custom dataloaders have been made by creating the class **SVHN_YOLO_Dataset.**
The explanation of the class is as follows:-

- The image and label file paths are kept as instance variables in the __init__ procedure.

- The length of the image files is returned via the __len__ method, which is defined.

- A tuple of file paths for the label and the picture is what the __getitem__ method is defined to return.

For training, testing, and validation datasets, three instances of the SVHN YOLO Dataset class are constructed with corresponding picture and label file locations.

For each split, data loaders are made using the DataLoader class. An iterator that allows for batch iteration through a dataset is a data loader. The dataset to load, the batch size, and whether or not to shuffle the data are all sent into the DataLoader function Object() { [native code] }.

The data loaders for the corresponding datasets are trainloader, valloader, and testloader. They are utilized to supply data in batches to the model's training, validation, and testing phases.

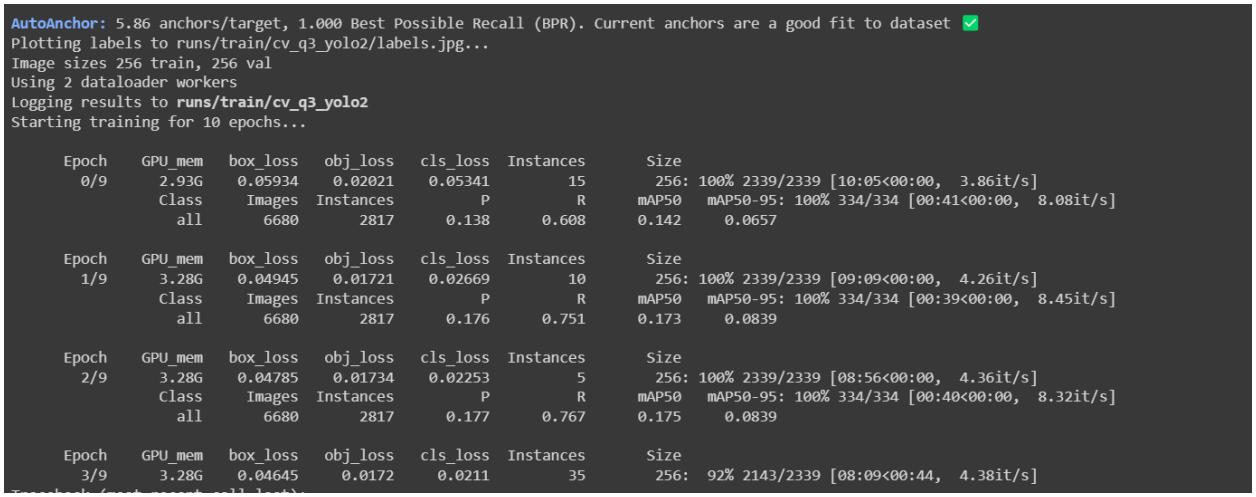
2. a)
To train the model, first I cloned the repository https://github.com/ultralytics/yolov5
Then I created the **svhn.yaml** file and added specific commands in the file, mentioned in the repo.

I also changed the number of class attributes in the **yolov5x.pt** file from 80 to 10.
After doing this step, I ran the command:-

!python train.py --img 256 --cfg yolov5x.yaml --batch 10 --epochs 10 --data svhn.yaml --weights yolov5x.pt --workers 24 --name cv_q3_yolo

Reference:
https://curiousily.com/posts/object-detection-on-custom-dataset-with-yolo-v5-using-pytorch-and-python/

This command started training the model, and we were able to see the results.



NOTE:
I am attaching the screenshot of the training part, I trained the model for close to 60 epochs, and was able to get an accuracy of 62%. But due to the usage limits of colab, the notebook crashed and I was not able to capture the output of the code.
It takes a lot of time to train the model, hence I am attaching the screenshot as a proof that I am able to train the model.

3. a)
The data augmentation techniques used in this question are Random Cropping, Random Erasing, and Random Rotation.
Explanations for these functions have been provided in Q1.

b)
Synthetic data has been added to the dataset. Now the model is trained on the dataset by following the same steps as in Question 2.2.(a).

c)
There are two models in this part of the question. The first one is the pre-trained YOLOv5 model which we have used from the GitHub repository, and the second one is the same model but trained on our dataset with additional data provided using data augmentation techniques like random cropping, random erasing, and random rotation. The better accuracy is obtained was in the second model. This is due to the fact that synthetically more training data has been added for the model to train better. As more data has been added to the model, the model is being allowed to explore more types of data input and would cover a broader aspects in which data can be presented to it. Hence the model gives the best accuracy in this part.