CSCI 6515 MACHINE LEARNING FOR BIG DATA

Assignment – 2 / Project

Submitted by SRISAICHAND SINGAMANENI Banner Id B00792835

Task: Image classification to detect the type of the car.

Data – Images of different cars and a csv file, which represents the images and its specific class.

Loading the Data To load the data into the Google Colab notebook. Images and CSV file are loaded into the colab notebook using "google.colab import files".

- Total number of classes are determined from the imported CSV.

Splitting the data: As mentioned in the instructions provided for the Assignment, CSV file is split into train, valid, test.
Test – 10% of the whole Dataset
Valid – 9% of the whole Dataset
Train – 81% of the whole Dataset

Reading the Data: In this step, I are reading the images names from the CSV file column "image_name" and using the names I read the actual Images from the Images Folder. This is done with the help of a helper class named as "CarsDataSet".

This class will take csv file, image folder, subset details, and transform parameters as the Input parameters and joins image with the label and returns the image and the label. But the image here is transformed according to the transform parameters we provided in the function call.

Transform Parameters:

Resize: Resize the input PIL Image to the given size.

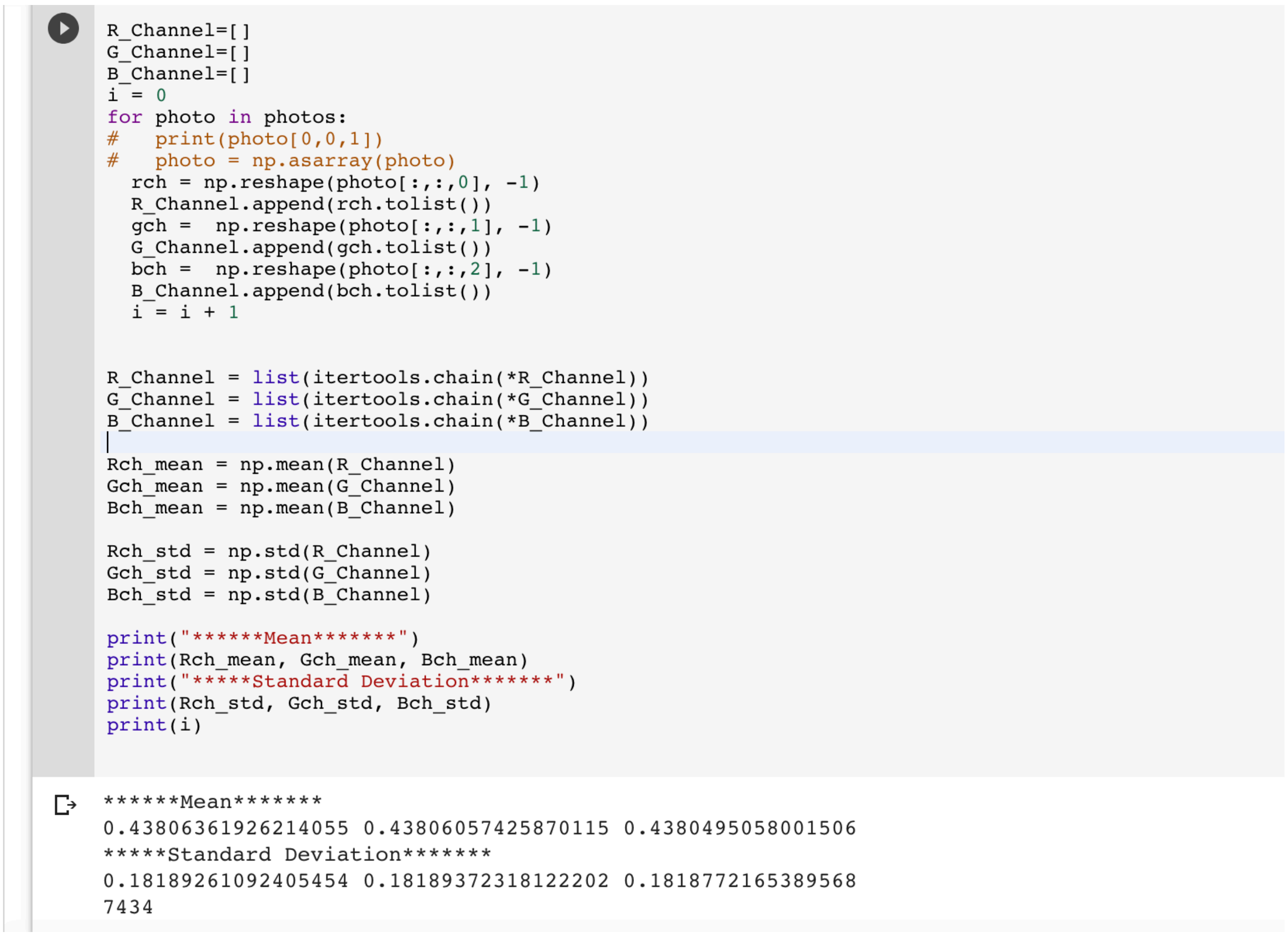
ToTensor: Convert a PIL Image to tensor. Tensor is a multidimensional array, which is generalization of vectors and matrices.
The above function Converts a PIL Image or numpy.ndarray (H x W x C) in the range [0, 255] to a torch.FloatTensor of shape (C x H x W) in the range [0.0, 1.0].

This "CarsDataset" class returns images which are converted to tensors and assign them a label according to their class. Tensor takes only integer inputs; **hence I have assigned every class in the dataset a number. The below table represents that assigned numbers for each class.**

Bus	Cement Mixer	Crane Truck	Dedicated Agriculture vehicle	Hatchback	Jeep	Light truck	minibus	Minivan	pickup	Prime Mover	sedan	Tanker	Truck	Van
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Deriving Mean and Standard Deviation to normalize the Data:

Image in tensors are represented as RGB Values. Each channel has multiple values ranging from 0-255.
Mean is to derive the average of these values and the standard deviation can be thought of measuring how far the data values lie from the mean.
Mean and Standard Deviation is calculated to Normalize the Images.



```
R_Channel=[]
G_Channel=[]
B_Channel=[]
i = 0
for photo in photos:
#   print(photo[0,0,1])
#   photo = np.asarray(photo)
  rch = np.reshape(photo[:,:,0], -1)
  R_Channel.append(rch.tolist())
  gch =  np.reshape(photo[:,:,1], -1)
  G_Channel.append(gch.tolist())
  bch =  np.reshape(photo[:,:,2], -1)
  B_Channel.append(bch.tolist())
  i = i + 1


R_Channel = list(itertools.chain(*R_Channel))
G_Channel = list(itertools.chain(*G_Channel))
B_Channel = list(itertools.chain(*B_Channel))

Rch_mean = np.mean(R_Channel)
Gch_mean = np.mean(G_Channel)
Bch_mean = np.mean(B_Channel)

Rch_std = np.std(R_Channel)
Gch_std = np.std(G_Channel)
Bch_std = np.std(B_Channel)

print("******Mean*******")
print(Rch_mean, Gch_mean, Bch_mean)
print("*****Standard Deviation*******")
print(Rch_std, Gch_std, Bch_std)
print(i)
```

```
******Mean*******
0.43806361926214055 0.43806057425870115 0.4380495058001506
*****Standard Deviation*******
0.18189261092405454 0.18189372318122202 0.1818772165389568
7434
```

Normalizing the Images:

In image processing, **normalization** is a process that changes the range of pixel intensity values.

Normalize does the following for each channel:
image = (image - mean) / std
The parameters mean, std (Standard deviation) are passed, which will normalize the image in the range [-1,1].

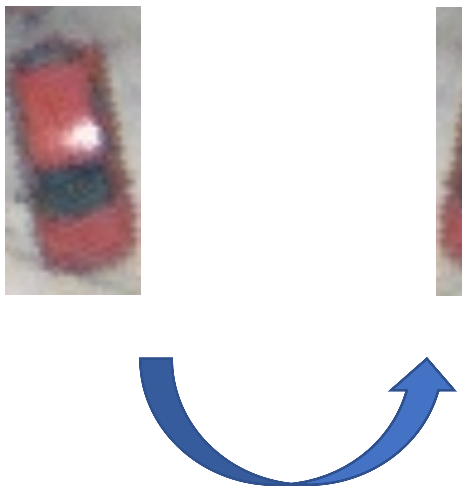
For example, the minimum value 0 will be converted to (0-0.5)/0.5=-1, the maximum value of 1 will be converted to (1-0.5)/0.5=1.

If to get the image back in [0,1] range, the formula is image = ((image * std) + mean)

Reference: https://discuss.pytorch.org/t/understanding-transform-normalize/21730/2

<u>**Augmentation**</u>: **Data augmentation means increasing the number of data points**. In terms of images
But if we increase the data it will result in complex model, hence instead of increasing the dataset size or increasing the number of images given to the model, we will rotate the existing images, change lighting conditions, crop it differently, so for one image we can generate different sub-samples.

Illustration:



In this scenario I have performed "transforms.RandomHorizontalFlip ()" to flip the images horizontally and feed to the hypothesis.

Data Loader:

Data loader. Combines a dataset and a sampler and provides single- or multi-process iterators over the dataset.

- Data loader also divides the data into batches which means how many samples per batch to load.
- Shuffle is set to True to have the data reshuffled at every epoch
- Number of workers - how many subprocesses to use for data loading. 0 means that the data will be loaded in the main process.

<u>Model Selection:</u>

Resnet 18: Resnet was introduced in the paper [Deep Residual Learning for Image Recognition]. There are several variants of different sizes, including Resnet18, Resnet34, Resnet50, Resnet101, and Resnet152, all of which are available from torch vision models. Here I used Resnet18, as our dataset is small and only has 15 classes.

ResNet-18 is a convolutional neural network that is trained on more than a million images from the ImageNet database. The network is 18 layers deep and can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224.

Reference: https://www.mathworks.com/help/deeplearning/ref/resnet18.html

Resnet18 comes with pretrained. When we are downloading the model, we can choose pretrained to be True or False. Instancing a pre-trained model will download its weights to a cache directory. When pretrained is set to false, we have to train the model from the scratch.

In this scenario I have done both, i.e, resnet 18 pretrained and the scratch model are trained and the validation accuracy for both of both have been compared.

Before training the model, few other parameters of the model have been tuned according to the requirement like:

- Optimizer – SGD -Stochastic gradient descent: "lr" is the learning rate – it is determined as the rate of reducing the loss function. When it is 0.001 the loss function is reducing fast.

  learning rate lr=0.001 and momentum=0.9

- No of input features are selected by default of the model which is 512. No of Output features are set to 15 as number of classes that we have in our data are 15. Thus I am reinitializing the model with

```
Linear(in_features=512, out_features=15, bias=True)
```

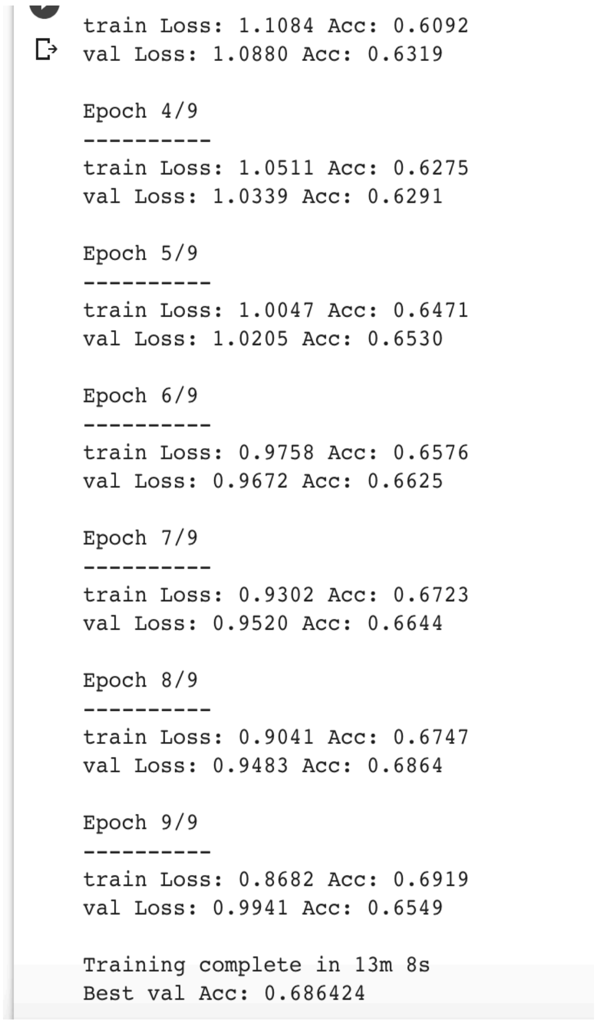
- feature_extract is a boolean that defines if we are finetuning or feature extracting. If feature_extract = False, the model is finetuned and all model parameters are updated. **If feature_extract =True**, only the last layer parameters are updated, the others remain fixed.

  I am selecting it to be True.

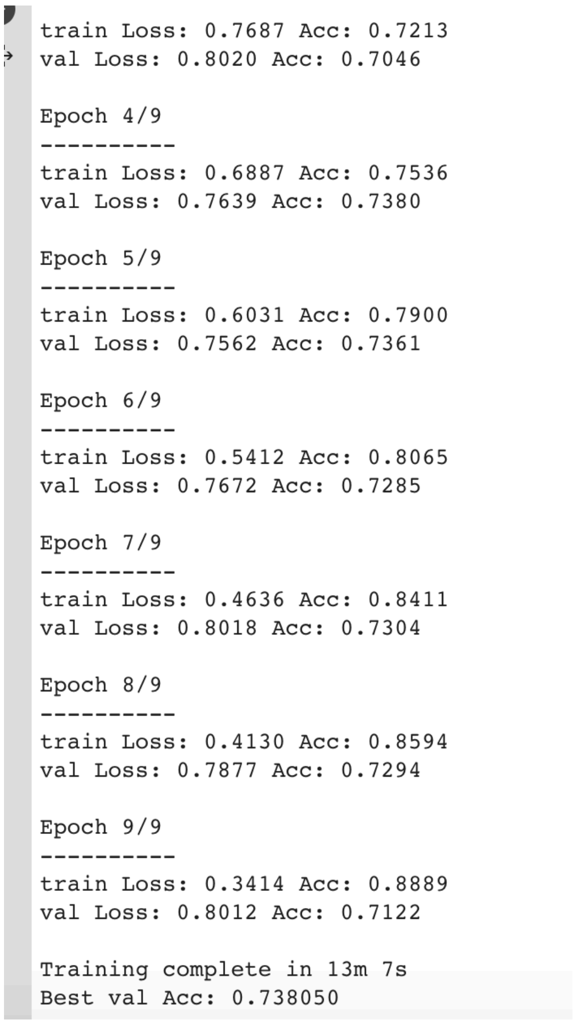
- The train_model function handles the training and validation of a given model. As inputs, it takes a PyTorch model, a dictionary of dataloaders, a loss function, an optimizer, and a specified number of epochs to train and validate. This function returns Accuracy of Train and Validation data and as well as Loss.

- **Validation accuracy is used to select the best model with highest validation accuracy and is stored. Function returns the Epoch of the model with the highest validation accuracy.**
- After Training the Model, **test_model** is a helper function to test the model with the Testing data.
- This function returns test accuracy of the model.

The above-mentioned process is done for the resnet18 model with the Pretrained parameter set as true and also false.
Below images compare the accuracy for each epoch for the PreTrained parameter True and False.



```
train Loss: 1.1084 Acc: 0.6092
val Loss: 1.0880 Acc: 0.6319

Epoch 4/9
----------
train Loss: 1.0511 Acc: 0.6275
val Loss: 1.0339 Acc: 0.6291

Epoch 5/9
----------
train Loss: 1.0047 Acc: 0.6471
val Loss: 1.0205 Acc: 0.6530

Epoch 6/9
----------
train Loss: 0.9758 Acc: 0.6576
val Loss: 0.9672 Acc: 0.6625

Epoch 7/9
----------
train Loss: 0.9302 Acc: 0.6723
val Loss: 0.9520 Acc: 0.6644

Epoch 8/9
----------
train Loss: 0.9041 Acc: 0.6747
val Loss: 0.9483 Acc: 0.6864

Epoch 9/9
----------
train Loss: 0.8682 Acc: 0.6919
val Loss: 0.9941 Acc: 0.6549

Training complete in 13m 8s
Best val Acc: 0.686424
```



```
train Loss: 0.7687 Acc: 0.7213
val Loss: 0.8020 Acc: 0.7046

Epoch 4/9
----------
train Loss: 0.6887 Acc: 0.7536
val Loss: 0.7639 Acc: 0.7380

Epoch 5/9
----------
train Loss: 0.6031 Acc: 0.7900
val Loss: 0.7562 Acc: 0.7361

Epoch 6/9
----------
train Loss: 0.5412 Acc: 0.8065
val Loss: 0.7672 Acc: 0.7285

Epoch 7/9
----------
train Loss: 0.4636 Acc: 0.8411
val Loss: 0.8018 Acc: 0.7304

Epoch 8/9
----------
train Loss: 0.4130 Acc: 0.8594
val Loss: 0.7877 Acc: 0.7294

Epoch 9/9
----------
train Loss: 0.3414 Acc: 0.8889
val Loss: 0.8012 Acc: 0.7122

Training complete in 13m 7s
Best val Acc: 0.738050
```

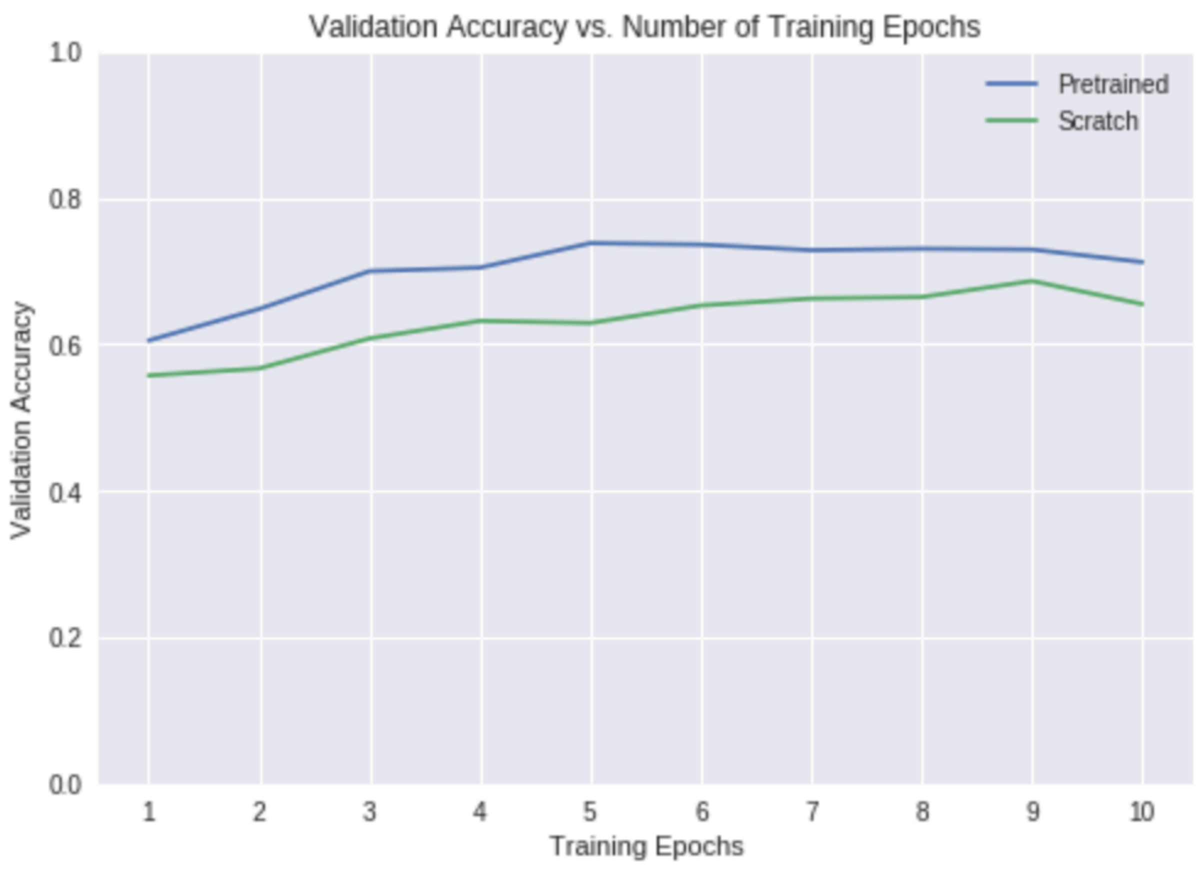
Pretrained = False **Pretrained = True**

Training and Validation accuracy

It can be observed from the above Images that the accuracy of the pretrained model is a little high when compare with the model which is trained from the scratch.
To know the reason behind this difference, I started knowing what exactly is pretrained model. I came to know about a new technique called “**Transfer Learning**”.

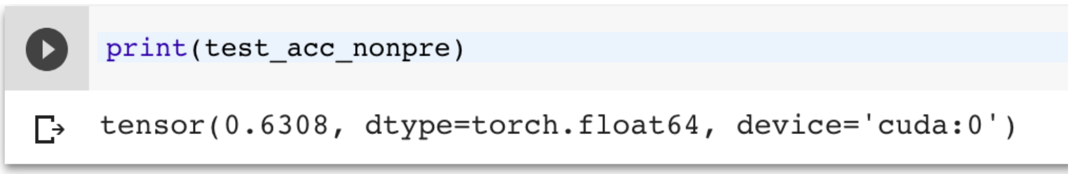
Transfer Learning is a popular method in computer vision which allow us to build accurate models in timesaving way. With this technique, instead of starting the learning process from the scratch, we start patterns that have been learned when solving a similar problem. This will save us from training a model from scratch instead. By observing the training and validation accuracy of our dataset in both the cases, it has been again proved that Transfer learning is more easy and quick to train.

The following Graph Illustrates the Accuracy over Number of Epochs.



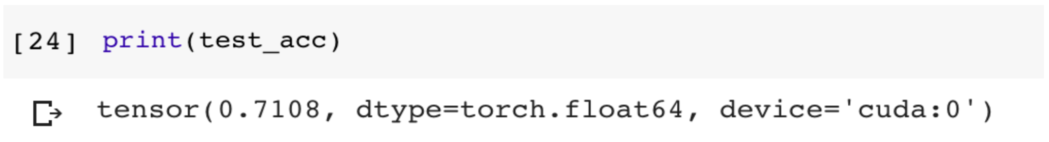
The following screen shots will display the Test accuracy of the Models.

Accuracy for Model trained from Scratch



```
print(test_acc_nonpre)

tensor(0.6308, dtype=torch.float64, device='cuda:0')
```

Accuracy for Model pretrained



```
[24] print(test_acc)

tensor(0.7108, dtype=torch.float64, device='cuda:0')
```

Final Verdict: Pretrained Model is more accurate in test data, but one key point to observe in the accuracies of that model is, in the training data it is overfitting. Overfitting is a modeling error which occurs when a function is too closely fit to a limited set of data points. This results in a much complex model. To overcome this overfitting, we have to early stop training the model. The pretrained model is also said to be overfitting by observing the training and validation loss.
In this case of the model (pretrained = true) training loss is much less than validation loss, which means that our model is fitting very nicely the training data, but not at all the validation data, in other words it's not generalizing correctly to unseen data which is called as "Overfitting".

On the contrary, the model which is trained from scratch is learning and improving its accuracy over epochs. It can also be justified by observing the training and validation losses which are almost similar. Hence, if data is more transformed and clear, model without transfer learning will be more perfect for this scenario.

References:

[1] Understanding transform.Normalize( ). (2018). Retrieved from https://discuss.pytorch.org/t/understanding-transform-normalize/21730/2

[2] Pretrained ResNet-18 convolutional neural network - MATLAB resnet18. (2018). Retrieved from https://www.mathworks.com/help/deeplearning/ref/resnet18.html

[3] Finetuning Torchvision Models — PyTorch Tutorials 1.0.0.dev20181216 documentation. (2018). Retrieved from https://pytorch.org/tutorials/beginner/finetuning_torchvision_models_tutorial.html

[4] Overfitting. (2018). Retrieved from https://www.investopedia.com/terms/o/overfitting.asp