

**Assignment 4**  
**Computer Vision**  
**Padhye Soham Satish**  
**M22RM007**

**Question 1-**

Perform image classification using CNN on the MNIST dataset. Follow the standard train and test split. Design an 8-layer CNN network (choose your architecture, e.g., filter size, number of channels, padding, activations, etc.). Perform the following tasks:

1. Show the calculation of output filter size at each layer of CNN.
  2. Calculate the number of parameters in your CNN. Calculation steps should be clearly shown in the report.
  3. Report the following on test data: (should be implemented from scratch)
    - a. Confusion matrix
    - b. Overall and classwise accuracy.
    - c. ROC curve. (you can choose one class as positive and the rest classes as negative)
  4. Report loss curve during training.
  5. Replace your CNN with resnet18 and compare it with all metrics given in part 3.
- Comment on the final performance of your CNN and resnet18.

Colab Link-

[https://colab.research.google.com/drive/1gNBdoURqhVaFrU2GrzHua84Rn\\_IIdaZO#scrollTo=M  
AP6Rybp0hUI](https://colab.research.google.com/drive/1gNBdoURqhVaFrU2GrzHua84Rn_IIdaZO#scrollTo=MAP6Rybp0hUI)

MNIST dataset is a collection of handwritten digits used for image classification tasks in machine learning. It is a widely used dataset for training and testing image classification models. The dataset consists of 70,000 grayscale images of handwritten digits (28x28 pixels). We are going to use this data for digit classification

I'm using pytorch to implement this task

#### Procedure-

1. Load the MNIST dataset using the PyTorch torchvision library. Make download as true for downloading and train = True to download the training data.
2. Apply the transform and transform the data to tensor. Transforms is used for data augmentation, data preprocessing and it transforms the image to tensor that is useful for doing further calculations
3. Then generate the random number and see corresponding image in data and check whether the dataset is loaded properly or not.
4. Then use the train test split to split the training data into training and validation set. Validation set is important in training the model because it helps to evaluate the performance of a model during the training process.
5. Now load the test data same as training data from pytorch library. This time we should write train=False. And apply same transform as training images.
6. Then use the dataloader to load the data in batches. In my case I have selected the batch size as 32 images.
7. Once again check the data is loaded properly or not by displaying the images.

#### Training the model-

1. Define the class for image classification and inherit the class by nn.Module.
2. Define a method for training that takes a batch of data as input. This method will be used during training to compute the loss on a single batch of data. This method calculates the loss between the predictions and the labels using the cross-entropy loss function.
3. Then define a method for validation that takes a batch of data as input. This method will be used during validation to compute the loss and accuracy on a single batch of data. This also computes the cross entropy loss.
4. Then define a method called for combining the accuracy and losses. This method will be used at the end of each validation epoch to compute the average loss and accuracy across all batches.
5. Next we will define a function to get the predictions. Here Im using torch.max() function to get the index of the predicted class with the highest probability. The output of torch.max() is a tuple which contains two tensors, the first tensor represents the maximum value along a given axis, and the second tensor represents the index location of that maximum value. Im only interested in the prediction that's why I have written like `_, preds`

Define custom class for creating the layers of the neural network.

1. I'm using `nn.sequential` to connect the layers.
2. In this line--`nn.Conv2d(1, 32, kernel_size=3, padding=1)`, convolutional layer with input channel is one as we have black and white images, 32 output channels means the 32 channel feature map will be created, a kernel size of 3 means the kernel used for convolution is 3x3 matrix, and a padding of 1 means the output images size will be same as input size. After convolution the output size is 32 x 28 x 28 after this shape.
3. Detailed calculation for the output filter size is-

To calculate the output size at each layer of the CNN, we need to use the following formula

$$\text{Output size} = \frac{\text{Input size} - \text{Filter size} + 2 \times \text{padding}}{\text{stride}} + 1$$

① For the first layer input size is 28x28 & Filter size is 3x3, padding is one & stride is one.

$$\text{output size} = \frac{28 - 3 + 2 \times 1}{1} + 1 = 28$$

② For the second layer the i/p size is 28x28, Filter size is 3x3 padding=1, stride=1 so the o/p size is

$$\text{output size} = \frac{28 - 3 + 2 \times 1}{1} + 1 = 28$$

③ For the third layer the input size is 28x28, Filter size is 3x3, padding is 1 & stride is 1.

$$\text{output size} = \frac{28 - 3 + 2 \times 1}{1} + 1 = 28$$

④ For the fourth layer that is maxpooling layer (2,2) The input size is 28x28 & the filter size is 2x2 padding is zero, & stride is 2. So the o/p size is

$$\text{output size} = \frac{28 - 2 + 2 \times 0}{2} + 1 = 14$$

⑤ For the fifth layer the input size is 14x14, Filter size is 3x3 padding 1 & stride is 1. So the output size will be

$$\text{output size} = \frac{14 - 3 + 2 \times 1}{1} + 1 = 14$$

⑥ For the sixth layer the input size is 14x14, filter size is 3x3 padding 1 & stride is 1. So the o/p filter will be

$$\text{output size} = \frac{14 - 3 + 2 \times 1}{1} + 1 = 14$$

⑦ For the seventh layer the input size is 14x14, Filter size is 2x2 padding is zero & stride is 2 [Maxpool(2,2)]

$$\text{output size} = \frac{14 - 2 + 2 \times 0}{2} + 1 = 7$$

4. After this we will use rectified linear unit (ReLU) activation function to the output. Activation functions introduce non-linearity into neural networks, allowing them to learn complex relationships between inputs and outputs. ReLU is the most commonly used activation function for classification task.
5. Then again we will do convolution from 32 channels to 64 channels.

Now the model is ready. Give images to the model and check results for first image. Also check whether the GPU is available or not.

Now define new function to check the GPU so that we can use this as many times without repeating same line of codes . Also define the function for giving the model to GPU for calculation.

Define custom data loader. It provides a convenient way to load data and move it to a specific device for training or inference in PyTorch.

Training and validation of the model-

1. Define evaluation function. This function sets the PyTorch model to evaluation mode, which turns off certain operations like dropout that should only be used during training. The important thing to remember this time is we should not track gradients during evaluation, where we don't need to update the model parameters. So I have used the python decorator for not calculating the gradients while evaluation.
2. Now this method then compute the validation metrics based on the list of outputs from the validation step and return it.
3. Now we will define fit function for fitting and training the model.
4. Define the optimizer function. I'm using SGD as the optimizing function. SGD means Stochastic Gradient Descent, which is a popular optimization algorithm used in deep learning for updating the parameters of a model during training. The goal of the optimization algorithm is to minimize the loss function. This optimizer computes the gradient of the loss with respect to the model parameters for a small batch of training examples, instead of the entire training set. That's why we call it as stochastic gradient descent.
5. Now we will loop over the specified number of epochs to train the model.
6. Create an empty list to store the training history and one more to store the training losses for each batch.
7. Then the next step is to train the model by calling `model.train()`. This puts PyTorch model to training mode, which turns on certain operations like dropout that should only be used during training.
8. Now loop over the batches in the training data loader.
9. Calculate the loss and append it to the loss list.

10. Then do the back propagation. Compute the gradients of the loss with respect to the model parameters.
11. Then use optimizer step. This updates the model parameters using the optimizer.
12. Then the important step is calling optimizer zero gradient function. This resets the gradients to zero for the next batch.
13. In this way we will run each epoch and update the weights.
14. At the end store the training history that is needed for plotting the validation curves.
15. Model parameters
  - a. Learning rate 0.001.  
Learning rate is a hyperparameter that determines the step size at which the optimizer updates the parameters of a model during training.
  - b. Optimizer = SGD details are already mentioned.
  - c. Number of epochs 10. During each epoch, the model's parameters are updated based on the optimization algorithm. The training phase consists of multiple epochs, where the number of epochs is a hyperparameter that needs to be chosen by the user based on the complexity of the problem, size of the dataset, and the model architecture. We should run the model for different epochs so that the training and validation loss saturates to low value and model gives good accuracy.

For the given model, the **number of parameters in each layer** can be calculated as follows:

To calculate the total number of parameters and parameters for each layer in the given model, we need to use the following formulas:

- For Convolutional Layers:

number of parameters = (input\_channels x kernel\_size x kernel\_size x output\_channels) + output\_channels

- For Linear Layers:

number of parameters = (input\_features x output\_features) + output\_features

ReLU layer has no parameters.

Conv2d(1, 32, kernel\_size=(3, 3)):  $(1 \times 3 \times 3 \times 32) + 32 = 320$

Conv2d(32, 64, kernel\_size=(3, 3)):  $(32 \times 3 \times 3 \times 64) + 64 = 18496$

Conv2d(64, 64, kernel\_size=(3, 3)):  $(64 \times 3 \times 3 \times 64) + 64 = 36928$

Conv2d(64, 128, kernel\_size=(3, 3)):  $(64 \times 3 \times 3 \times 128) + 128 = 73856$

Conv2d(128, 128, kernel\_size=(3, 3)):  $(128 \times 3 \times 3 \times 128) + 128 = 147584$

Conv2d(128, 256, kernel\_size=(3, 3)):  $(128 \times 256 \times 3) + 256 = 295168$

Conv2d(256, 256, kernel\_size=(3, 3)):  $(256 \times 256 \times 3) + 256 = 590080$

Linear(in\_features=2304, out\_features=1024):  $(2304 \times 1024) + 1024 = 2360320$

Linear(in\_features=1024, out\_features=512):  $(1024 \times 512) + 512 = 524800$

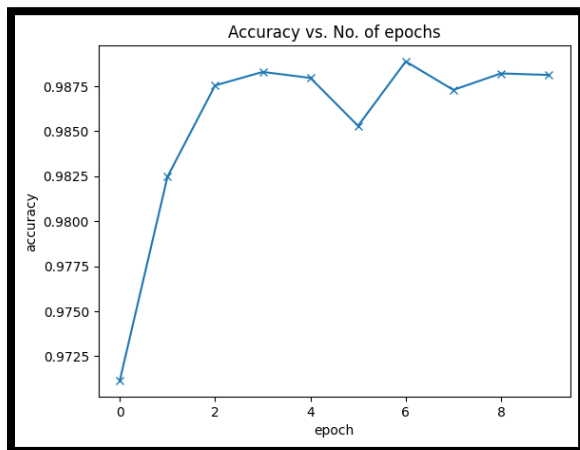
Linear(in\_features=512, out\_features=10):  $(512 \times 10) + 10 = 5130$

Therefore, the total number of parameters in the given CNN model is the sum of the above values:

$320 + 18496 + 36928 + 73856 + 147584 + 295168 + 590080 + 2360320 + 524800 + 5130 = 4052682$

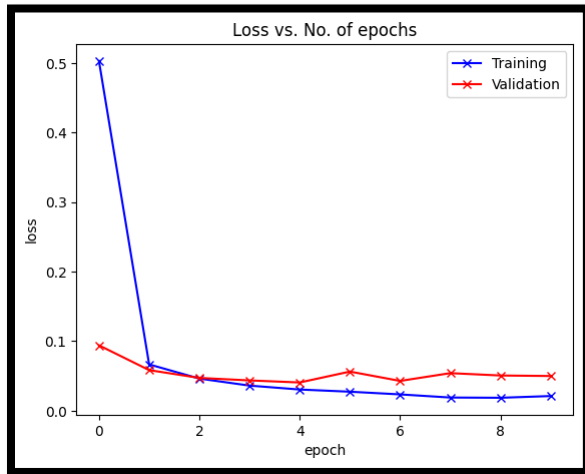
This calculation is also shown in the code layer wise weight and bias.

### Results for MNIST dataset and custom layers to do the digits classification-



We can clearly see that as the number of epochs increases, the accuracy of the model also increasing. Which is considered good for model training. It is important to monitor both the training and validation accuracies to ensure that the model is not overfitting the training data. If it is overfitting then do some modification in training data, do data augmentation to increase the data and reduce the bias.

## The graph of loss vs no. of epochs

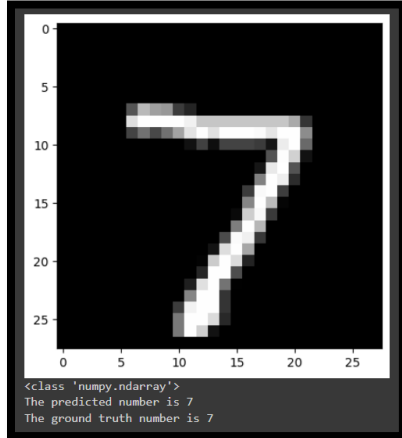


As we can see clearly the shape of the loss vs. number of epochs graph is decreasing curve. The initial value of the loss is high as the model learns and then it learns and gets better predicting output in next epochs the loss value gradually decreasing.

After the training is over, we will do the testing-

- We already have trained model so we pass the test image to it.
- The important step is to get the prediction class probabilities. It can be done by applying the Softmax function to the output of the model. It will give the probability values for the each class.
- As we are classifying the digits from 0 to 9 there will be a vector of 10 elements. Each element will show the probability value for respective digit.
- We will pick up the maximum value of probability as it is most likely prediction.
- Now we have the predicted number, and we already have the groundtruth with us we can plot the confusion matrix.

## Predicted number-



We can see that the model has predicted right number.

## Procedure for finding the confusion matrix

Input:

True labels: a list or array of true labels for the test data that we already have.

Predicted labels: We have obtained it from test set.

Number of classes – 10 in our case

Output:

Confusion matrix of size a num\_classes x num\_classes matrix, where each row represents the true class and each column represents the predicted class.

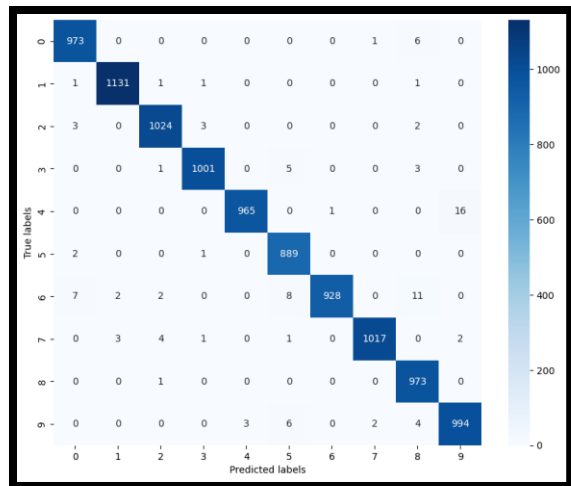
Algorithm for finding the confusion matrix.

Initialize one variable as a num\_classes x num\_classes matrix with all elements as 0 using numpy.

For each pair of true and predicted labels, increment the value of the corresponding cell in the confusion matrix by 1. For example, if the true label is 4 and the predicted label is 2, then we would increment the value in the (4,2) cell of the confusion matrix by 1.

We iterate this for length of confusion matrix and finally we will get the 10 x 10 confusion matrix.



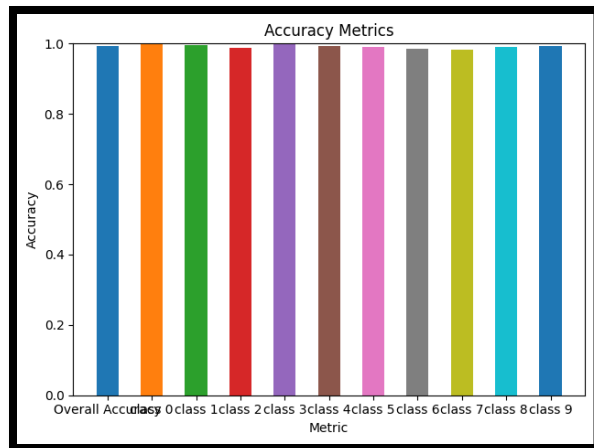


### Observations from the confusion matrix

- The diagonal of the matrix represents the number of correct predictions for each class. It shows which classes are being predicted accurately and which are not. In my case the diagonal entries are high so it indicates that most of the digits are classified correctly.
- Then the non-diagonal elements represent the misclassifications of the digits. They show which classes are being confused with each other. In my case we can see very few examples from classes are misclassified as we are getting very less number in the non-diagonal entries.

### Calculating the class wise accuracy and overall accuracy-

- Define a function that takes a confusion matrix as an input.
- Compute the overall accuracy of the classifier by dividing the sum of the diagonal elements of the confusion matrix by the sum of all elements of the matrix.
- Compute the class-wise accuracy of the classifier by dividing each diagonal element of the confusion matrix by the sum of the corresponding row of the matrix.



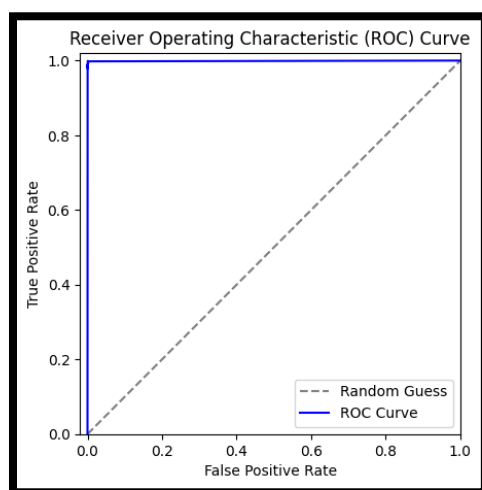
Here the first bar is overall accuracy and other bars are classwise accuracy.

We can clearly see that the classwise and overall accuracy is very high and near to one.

### Plotting the ROC curve -

Calculate the True Positive Rate (TPR) and False Positive Rate (FPR) for each class:

- Loop through each class index  $i$  from 0 to 9.
- Calculate the True Positive (TP), False Negative (FN), False Positive (FP), and True Negative (TN) rates using the confusion matrix.
- Sort the points by increasing FPR.
- Plot a dashed line from (0, 0) to (1, 1) as a reference for random guessing. If the ROC falls below this line then ROC curve is not good and model is not trained well.
- Plot the line connecting TPR and FPR.



### Observations from ROC Curve:

- The ROC curve is a plot of True Positive Rate (TPR) vs False Positive Rate (FPR) for various threshold values. In my case the threshold value is 0.5.
- Then the diagonal line from (0,0) to (1,1) is the ROC curve for a random classifier. I will explain why this is needed.
- Then the important point regarding ROC is classifier is considered as good classifier that have a curve that is as close to the top-left corner (TPR=1, FPR=0) as possible. In my case it is very similar to the ideal curve.
- The area under the ROC curve (AUC) is a measure of the overall performance of the classifier, with higher AUC indicating better performance.

### Role of Random Guess:

- Random classifier has no predictive power and makes random guesses.
- The purpose of including this line on the plot is to provide a baseline for comparison, i.e., any classifier that performs worse than random guessing should be considered as a failure classification is not good.
- A good classifier should have a ROC curve that is above the random guessing line, indicating that it is performing better than random guessing.

Now the next task is to train the model of RESNET18 architecture for digits classification.

About resnet18-

ResNet18 consists of 18 layers, including a convolutional layer, a series of residual blocks, and a fully connected layer. The residual block is the key innovation of ResNet, which addresses the vanishing gradient problem by introducing a shortcut connection between the input and output of each block. In ResNet18, each residual block contains two convolutional layers with a skip connection that bypasses the convolutional layers. The network also includes downsampling layers, which reduce the spatial resolution of the feature maps while increasing their depth. This helps the network learn more abstract features and reduces the number of parameters in the fully connected layer

In the pytorch the RESNET18 architecture is already available.

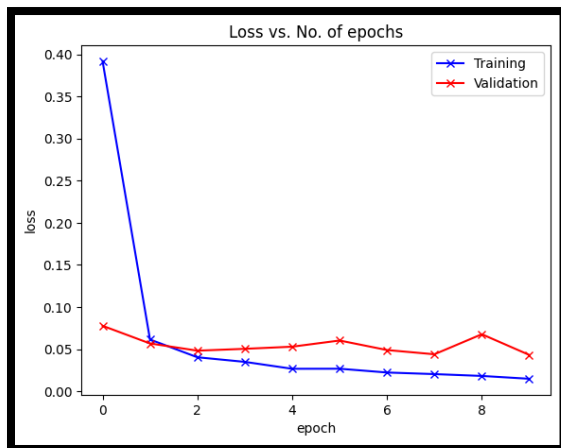
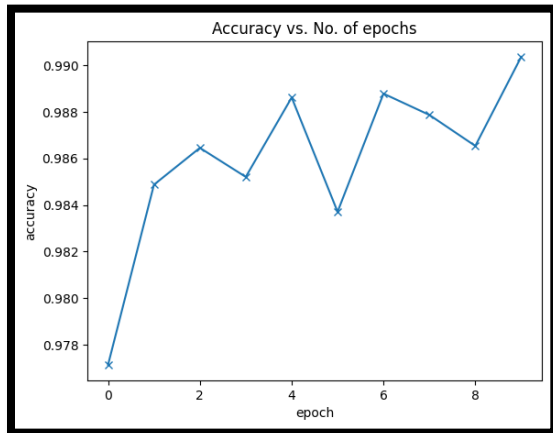
We can download this and train the model on our MNIST dataset.

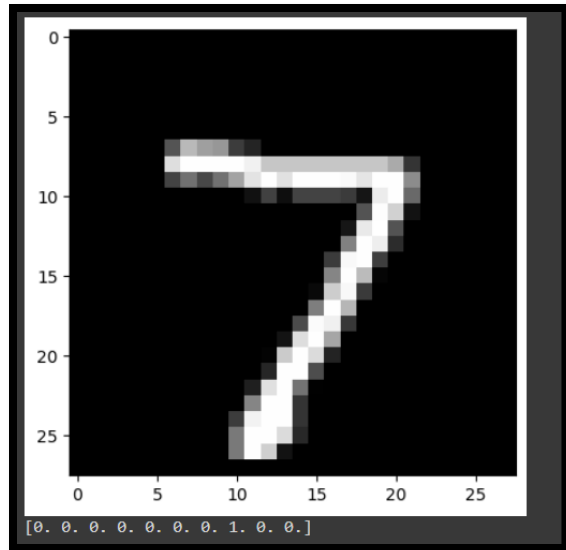
The important step to do this is we should write pretrained = False in this command.

```
self.network = models.resnet18(pretrained=False)
```

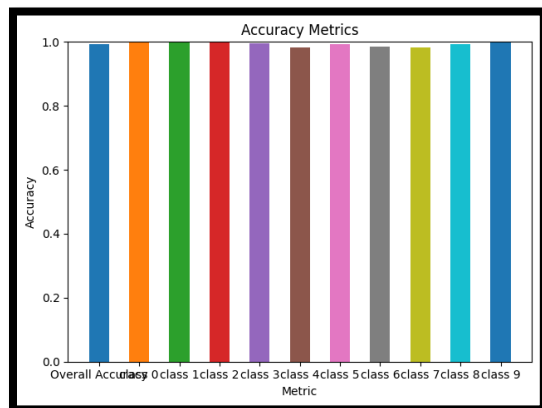
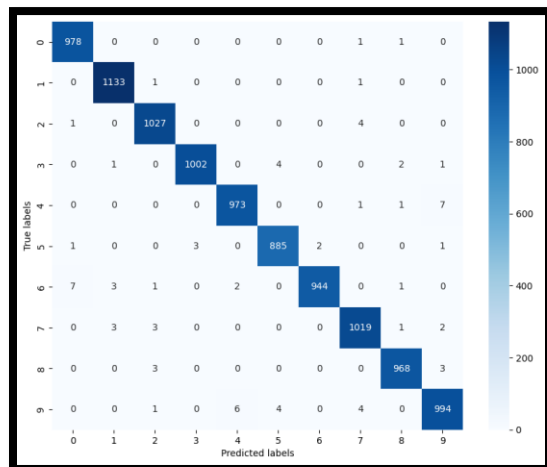
Now we have the resnet18 architecture and we will train the model on this architecture same as we trained the model on custom layers.

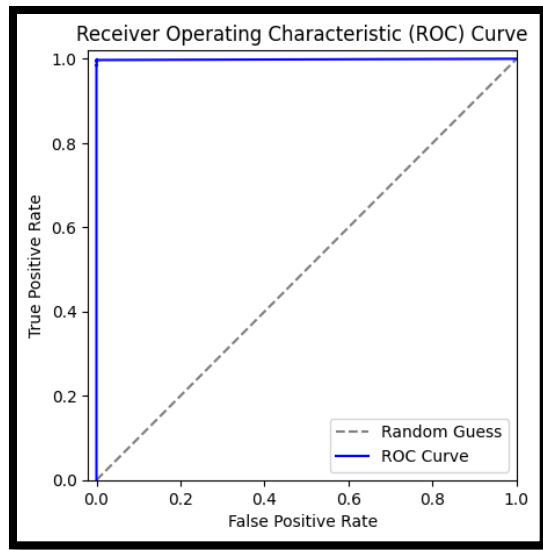
### Results for RESNET18 model-





The output is one vector Of 10 entries. One indicates that at that index the prediction is maximum.(One hot encoding) Therefore the predicted number is seven.





### **Comparison between the resnet18 and custom layers for digit classification-**

ResNet18 is generally a better choice for image classification tasks compared to the custom CNN layers that I have defined. Especially for larger datasets with complex patterns resnet18 is good. However, the specific choice between these architectures would depend on the specific requirements of the task and available resources.

As we can see in the accuracy vs no. Of epoch for both the models is quite similar.

There is no significant change in the accuracy. That's why the layers that I have defined are well classifying the digits. But the resnet is very powerful and its accuracy is also good.

## Question 2-

### Image Captioning

I'm training this model on local device. Please find attached python file for the code.

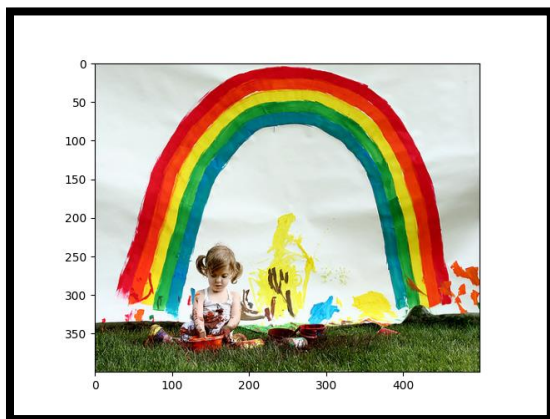
The Flickr8k dataset is a collection of 8,000 images, each with five descriptive captions.

The images in the Flickr8k dataset are a mix of indoor and outdoor scenes, such as people, animals, landscapes, and objects.

Each image in the dataset has five different captions, which were created by human annotators. The captions are intended to describe the content of the image in natural language and often contain rich and detailed descriptions. For example, a caption for an image of a cat might read "A small, grey cat is sitting on a windowsill looking out at the street below."

Data preprocessing and loading the data is the most important step in the image captioning process. Steps for loading the data-

- Read the caption file and print the length of file for checking the length of data.
- Load the image and its corresponding 5 captions so that can get the clear idea about the dataset.



This is the image in the dataset and its corresponding captions are-

```
There are 40455 image to captions
Caption: A little girl is sitting in front of a large painted rainbow .
Caption: A small girl in the grass plays with fingerpaints in front of a white canvas with a rainbow on it .
Caption: There is a girl with pigtails sitting in front of a rainbow painting .
Caption: Young girl with pigtails painting outside in the grass .
Caption: A man lays on a bench while his dog sits by him .
```

- As it is an image captioning task, we need to use some technique to tokenize the sentences. I'm using spacy to do the NLP task.
- Then we will define one class for creating the vocabulary for a given sentences.

- The vocabulary is used to convert the sentences into numerical vectors, where each number corresponds to a particular word in the vocabulary.
- The constructor of the class initializes the vocabulary by defining four special tokens, including <PAD>, <SOS>, <EOS>, and <UNK>. PAD means padding . It is used to make the vectors equal length. SOS means the start of the sentence. EOS means end of the sentence and UNK is unknown vocabulary. These tokens are assigned integer indices from 0 to 3 in the itos dictionary. The stoi (string to image) dictionary is the reverse of the itos(image to string) dictionary, mapping each string token to its integer index. The freq\_threshold parameter specifies the minimum frequency required for a word to be included in the vocabulary. We can set this parameters as per our requirement.
- The \_\_len\_\_ method in vocabulary class will return the length of the vocabulary, which is the number of words in the itos dictionary.
- The build\_vocab method takes a list of sentences sentence\_list as input and iterates over each sentence. For each sentence, it tokenizes the sentence using the tokenize method and counts the frequency of each word using the frequencies Counter object. If the frequency of a word exceeds the freq\_threshold value, the word is added to the vocabulary with a unique integer index starting from 4. The stoi and itos dictionaries are updated accordingly.
- The numericalize method takes a sentence text as input and tokenizes the sentence using the tokenize method. It then converts each token in the sentence into its corresponding integer index in the stoi dictionary. We need to convert the words to certain number as model only understands the numbers. If a token is not found in the dictionary, it is replaced with the <UNK> token. The resulting list of integer indices represents the numerical vector for the sentence.
- Now we have the vocabulary of words that is needed to predict the caption for the image.

Define the new class to load and preprocess the Flickr8k dataset to be used in training a machine learning model

- This is a class inherits from the Dataset class in PyTorch.
- The \_\_init\_\_ method takes four arguments: root\_dir, caption\_file, transform, and freq\_threshold.
- root\_dir is the directory path where the image files are located. In my case I'm running the model on my local laptop so I have saved the files in one folder and I'm giving the path of this folder. This file contained the images.
- caption\_file is the file path for the text file that contains the image names and corresponding captions. Each image has 5 corresponding captions.
- Transform is an optional argument that allows for image transformations to be applied to the images before they are returned. I'm transforming the image to tensor and doing some data preprocessing when I create the instance for this class..



- The `__getitem__` method takes an index and returns a tuple containing the preprocessed image and its corresponding preprocessed caption. This is very important step. When I was running the model there was an error in this method so I rectified it and now it is fine.
- Inside the `__getitem__` method, the image file name and caption are retrieved based on the index . The image is then loaded using the PIL Image library and converted to RGB format.
- Next, the caption is converted to a numerical vector using the `numericalize` method of the Vocabulary class. The numerical vector is then prepended with the start of sequence token (<SOS>) and appended with the end of sequence token (<EOS>) to indicate the start and end of the caption respectively.
- Finally, the preprocessed image and caption are returned as a tuple.

Split the dataset to training and validation set by using standard train test split function from sklearn library. I'm using 80% ,10% ,10% split of the data for training, validation and testing respectively. On

testing dataset we can check the model performance on the test set.

Now we are ready to design the model structure for image captioning.

I have taken the reference for image captioning from this page -

<https://www.kaggle.com/code/mdteach/image-captioning-with-attention-pytorch>

This is an image captioning model based on attention encoder and decoder.

Attention-based encoder-decoder models are a type of neural network architecture used for sequence-to-sequence (seq2seq) tasks, such as machine translation or image captioning.

The encoder is responsible for taking in an input sequence of images and encoding it into a fixed-size vector representation of vocabulary. In an attention-based encoder, each element of the input sequence is weighted differently based on its relevance to the output sequence. These weights are learned during training and allow the model to focus on different parts of the input sequence as needed.

The decoder is responsible for taking the fixed-size vector representation generated by the encoder and using it to generate the output sequence (e.g., a caption for the image). In an attention-based decoder, the decoder generates each element of the output sequence one at a time, attending to different parts of the input sequence as needed. The attention mechanism

helps the model to better capture long-range dependencies and to generate more accurate and fluent output sequences.

Source - <https://www.analyticsvidhya.com/blog/2020/11/attention-mechanism-for-caption-generation/#:~:text=The%20encoder%2Ddecoder%20image%20captioning,LSTM%20and%20generate%20a%20caption.>

Define EncoderCNN class :

1. This class inherits from the PyTorch nn.Module class and is used to encode the images
2. As mentioned in the question I'm using the resnet20 pretrained model for encoder.
3. The for loop then loops through all the parameters in the ResNet50 model and sets their requires\_grad attribute to False. This is because we want to freeze the weights of the pre-trained model and prevent them from being updated during the training of our model.
4. The list(resnet.children())[:-2] code extracts all the layers of the ResNet50 model except the last two layers (i.e., the global average pooling layer and the fully connected layer). These two layers are typically used in the ResNet50 architecture to convert the output of the convolutional layers into a fixed-length feature vector.
5. In the forward method, takes the input image tensor and passes it through the ResNet50 model
6. Reshapes the output tensor to be of shape (batch\_size, 49, 2048) where 49 is the number of regions in the image (7x7) and 2048 is the dimension of each region's feature vector

Define Attention class:

- In the class constructor we are defining the Attention module, which takes in the encoder and decoder dimensions and an attention dimension as input parameters. The attention dimension represents the number of hidden units in the attention layer.
- There are W and U linear layers . The W layer maps the hidden state of the decoder to the attention space, and the U layer maps the encoder output to the attention space.
- The A linear layer is initialized in the \_\_init\_\_ method. This layer maps the concatenated U and W outputs to a scalar value, which is then used to calculate the attention weights.
- In the forward method, the features tensor (the output of the encoder) and the hidden\_state tensor (the hidden state of the decoder) are passed as input parameters.
- The U layer is applied to the features tensor, and the W layer is applied to the hidden\_state tensor. This generates two tensors that represent the encoder and decoder states in the attention space.

- The two tensors are added together and then tanh function is applied to the resulting tensor. This generates a tensor that represents the combined states of the encoder and decoder in the attention space.
- The A linear layer is applied to the combined tensor, generating a tensor with shape (batch\_size, num\_layers, 1).
- The last dimension of the tensor is squeezed out using the squeeze method, resulting in a tensor with shape (batch\_size, num\_layers) that represents the attention scores for each sequence in the batch.
- The softmax function is applied to the tensor along the second dimension, generating a tensor with shape (batch\_size, num\_layers) that represents the attention weights for each sequence in the batch.
- The attention weights are applied to the features tensor using element-wise multiplication, generating a tensor with shape (batch\_size, num\_layers, features\_dim).
- The tensor is then summed along the second dimension, resulting in a tensor with shape (batch\_size, features\_dim) that represents the attention context vector.
- The attention scores and attention context vector are returned as output of the forward method.

#### Define Attention decoder-

- This procedure is very complex so I have not modified this model architecture. I'm implementing the model to get the results.
- The constructor initializes the model and sets up the layers and parameters required for decoding the captions.
- The embedding layer is an embedding matrix that is used to map the index of each word to its corresponding dense vector representation.
- The attention layer is an instance of the Attention class, which uses the encoder output and decoder hidden state to calculate the context vector at each time step.
- As mentioned in the question we are using the LSTM for decoder. Here the init\_h and init\_c layers are linear layers that are used to initialize the LSTM hidden state and cell state with the mean of the encoder output.
- The lstm\_cell layer is an LSTM cell that takes the concatenated input of the embedded word and the attention context vector to generate the decoder hidden state and cell state at each time step.
- The f\_beta layer is a linear layer that is used to calculate the attentional bias, which is used to weight the encoder output in the attention mechanism.
- The fcn layer is a linear layer that is used to generate the final output from the decoder hidden state.
- The drop layer is a dropout layer that is used to prevent overfitting.
- The forward function takes in the image features and the ground truth captions, and generates the output predictions and attention scores for each time step.

- The `generate_caption` function is used to generate captions given an input image feature. It takes in the image features and the maximum length of the generated caption, and returns the generated caption and the attention scores for each time step.
- The `init_hidden_state` function is used to initialize the LSTM hidden state and cell state with the mean of the encoder output. It takes in the encoder output and returns the initialized hidden state and cell state.

Define Encoder decoder class-

- The `EncoderDecoder` class is the main module that contains two submodules: `EncoderCNN` for encoding the image and `DecoderRNN` for generating the caption.
- In the constructor of `EncoderDecoder`, the `EncoderCNN` and `DecoderRNN` are initialized with appropriate parameters.
- In the forward method of `EncoderDecoder`, first, the input image is passed through the `EncoderCNN` to get the encoded features. Then, the encoded features and caption are passed to the `DecoderRNN` to generate the caption. Finally, the outputs are returned.
- The `EncoderDecoder` combines the image encoder and caption decoder to generate a caption for the input image

Hyperparams for the model

`embed_size=300`

`attention_dim=256`

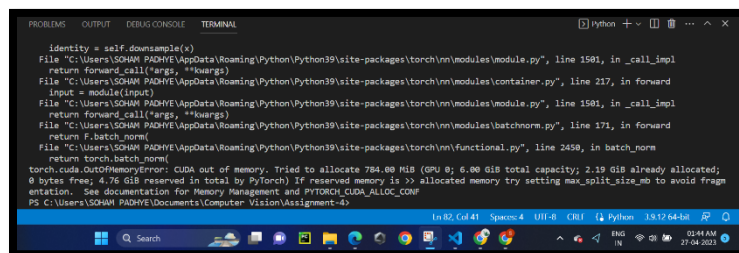
`encoder_dim=2048`

`decoder_dim=512`

`learning_rate = 3e-4`

Then define the optimizer and loss function to calculate the loss as we have defined in the question 1.

I'm running the model for 20 epochs. I was running the model and it failed because of memory error when the number of epoch was 25.



```

identity = self.downsample(x)
File "C:\Users\SOHAM PADHYE\AppData\Roaming\Python\Python39\site-packages\torch\nn\modules\module.py", line 1501, in _call_impl
    return forward_call(*args, **kwargs)
File "C:\Users\SOHAM PADHYE\AppData\Roaming\Python\Python39\site-packages\torch\nn\modules\container.py", line 217, in forward
    input = module(input)
File "C:\Users\SOHAM PADHYE\AppData\Roaming\Python\Python39\site-packages\torch\nn\modules\module.py", line 1501, in _call_impl
    return forward_call(*args, **kwargs)
File "C:\Users\SOHAM PADHYE\AppData\Roaming\Python\Python39\site-packages\torch\nn\modules\batchnorm.py", line 171, in forward
    return F.batch_norm_
File "C:\Users\SOHAM PADHYE\AppData\Roaming\Python\Python39\site-packages\torch\nn\functional.py", line 2450, in batch_norm
    return torch.batch_norm(
torch.cuda.OutOfMemoryError: CUDA out of memory. Tried to allocate 784.00 MiB (GPU 0; 6.00 GiB total capacity; 2.19 GiB already allocated;
0 bytes free; 4.76 GiB reserved in total by PyTorch) If reserved memory is >> allocated memory try setting max_split_size_mb to avoid fragm
entation. See documentation for Memory Management and PYTORCH_CUDA_ALLOC_CONF
PS C:\Users\SOHAM PADHYE\Documents\Computer_Vision\Assignment-4>
  
```

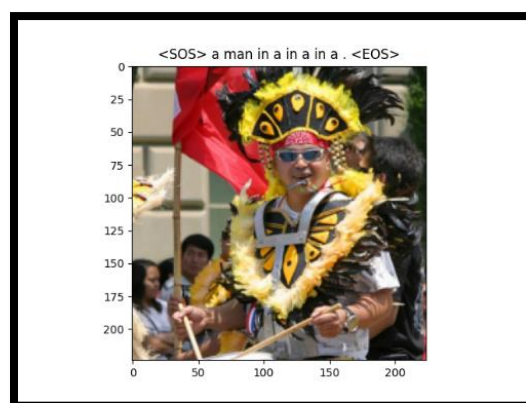
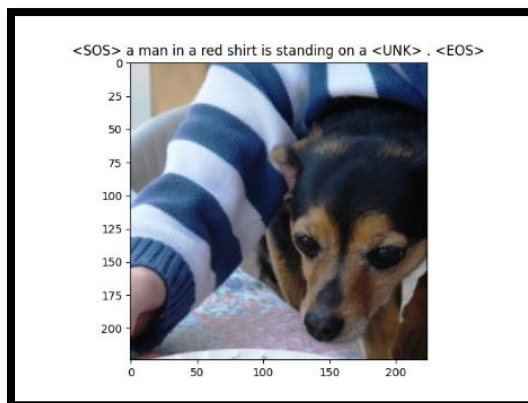
That's why I have reduced the number of epochs to 20.

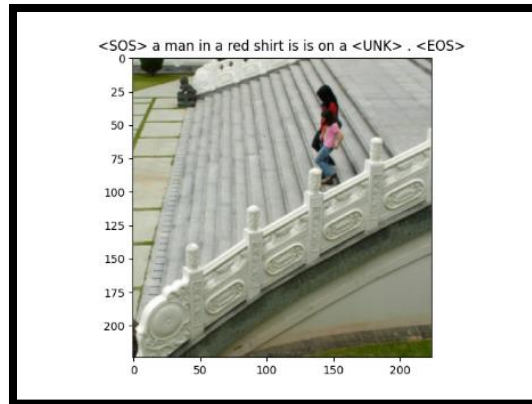
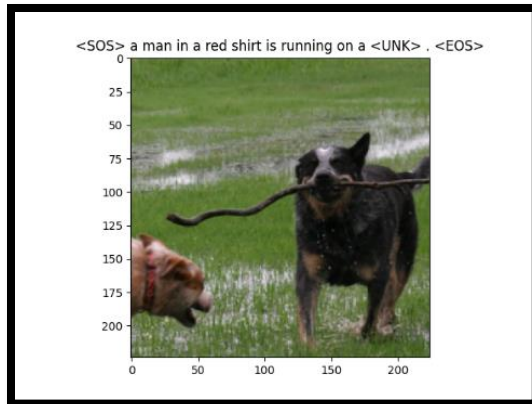
Now we will run the epochs to train the model.

1. The outer loop runs in the code is used for a specified number of epochs, and for each epoch, the inner loop iterates over batches of data.
2. Then the optimizer's gradients are set to zero at the beginning of each batch to avoid accumulation of gradients from previous batches.
3. Then we will call the model with the image and caption inputs to generate predicted outputs and attention weights.
4. The batch loss is computed using loss function by comparing the predicted outputs to the actual targets.
5. The loss is backpropagated through the network to compute gradients for each parameter.
6. The optimizer is updated with these gradients to adjust the model's parameters.
7. Then we will show the image with generated caption for each epoch. It is very necessary because we will get the idea about the progress of the model. If the model is not performing good then we will change the hyperparameters and rerun the model.
8. Then the model is set to evaluation mode before generating the caption and then returned to training mode afterward.

### Important learnings and observations for the problem-

1. Initially I was running the model without attention encoder and decoder with the learning rate of 0.001 and batch size of 64 but it was not producing the good results. Results are shown below-



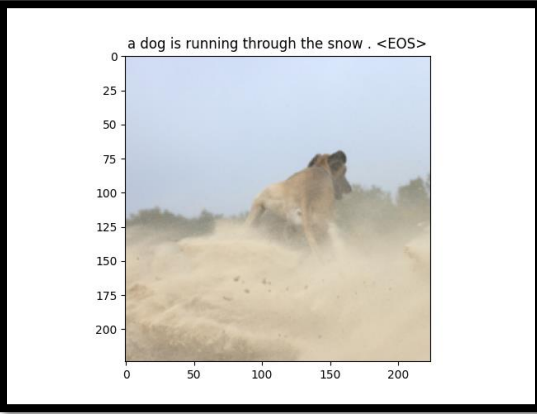
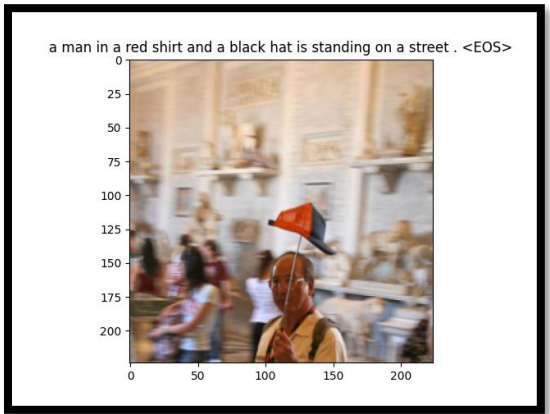
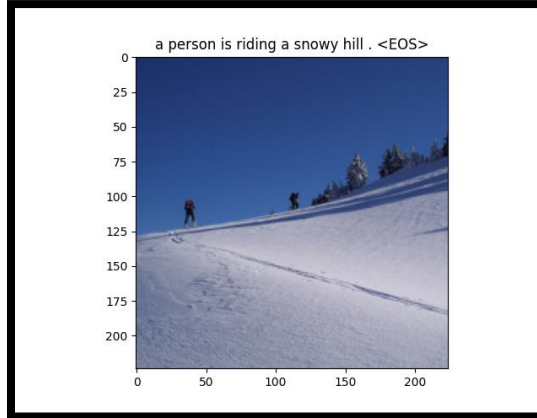
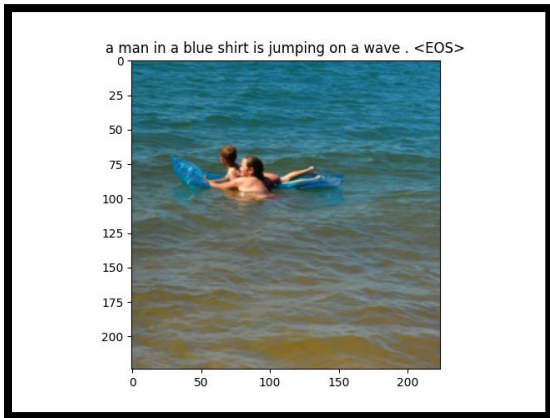
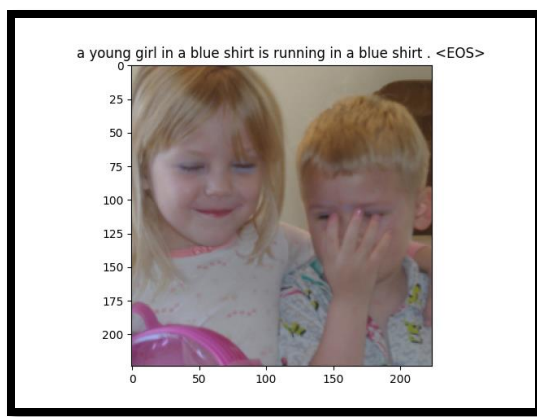
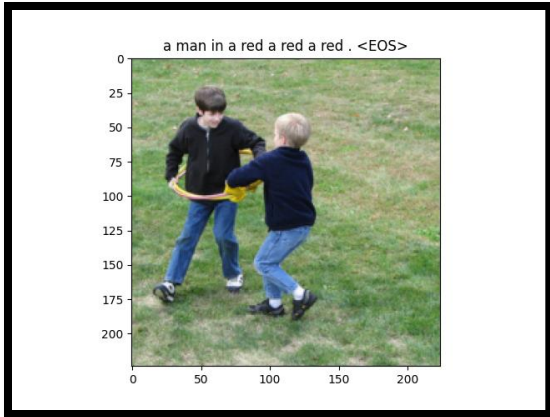


These are the results from the model without attention encoder and decoder. You can see here “the man in red” phrase is repeated in every image. The reason for this may be-

1. As the model does not have an attention mechanism that allows it to focus on different parts of the image while generating the caption, it might generate captions based on only a few salient features of the image, such as the color red. It is lacking the attention mechanism.
2. Overfitting of the model may be happening. If the model is overfitting to the training data, it might memorize certain patterns in the data, such as the association between red and the presence of a man, and apply them to all images, even if they do not follow the same pattern.

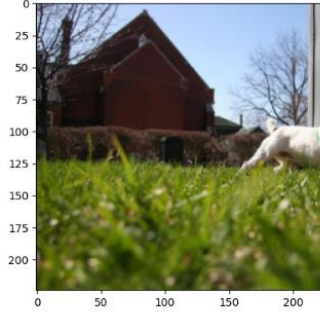
So when I implemented the attention encoder and decoder I get the good results.

**These are the image produced by the model after each epoch.** Again I'm using my laptop to run this model.





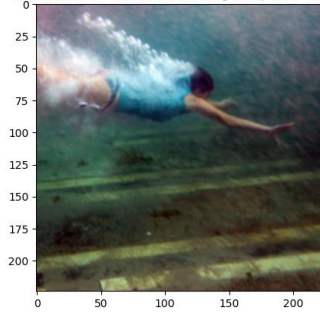
a white dog is running through a grassy field . <EOS>



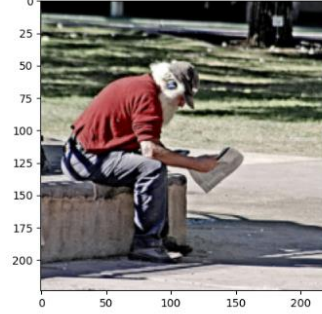
a white dog is jumping over a log . <EOS>



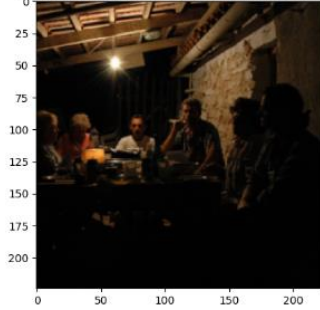
a man in a blue shirt is swimming in a pool . <EOS>



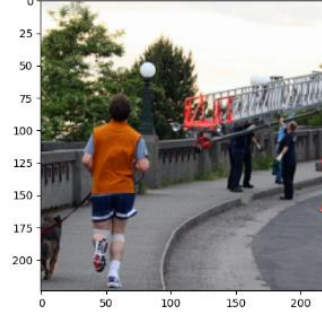
a man in a blue shirt and a white shirt is sitting on a bench . <EOS>



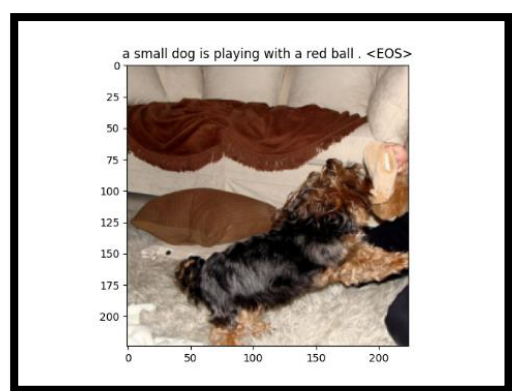
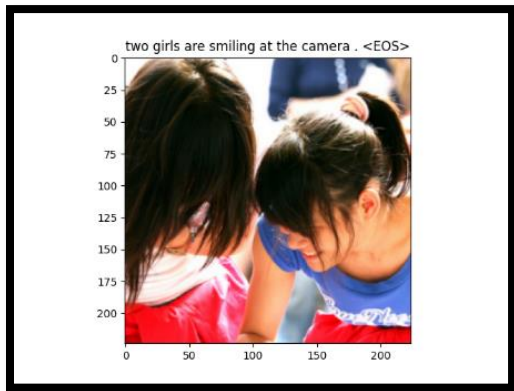
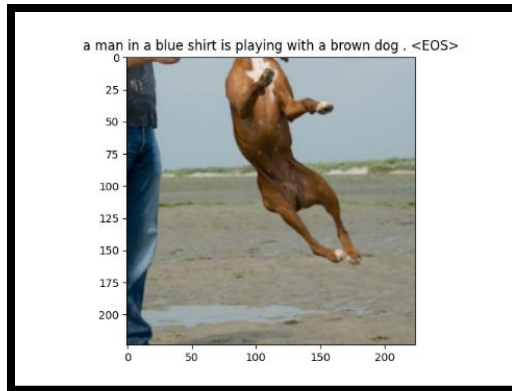
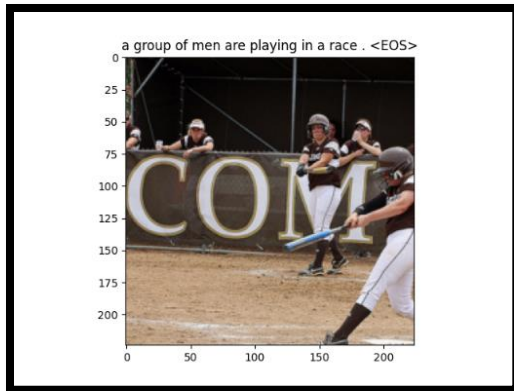
a group of people are sitting on a bench . <EOS>



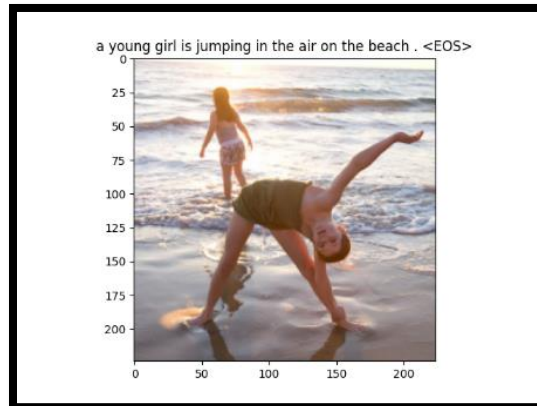
a man in a blue shirt is walking down a street . <EOS>

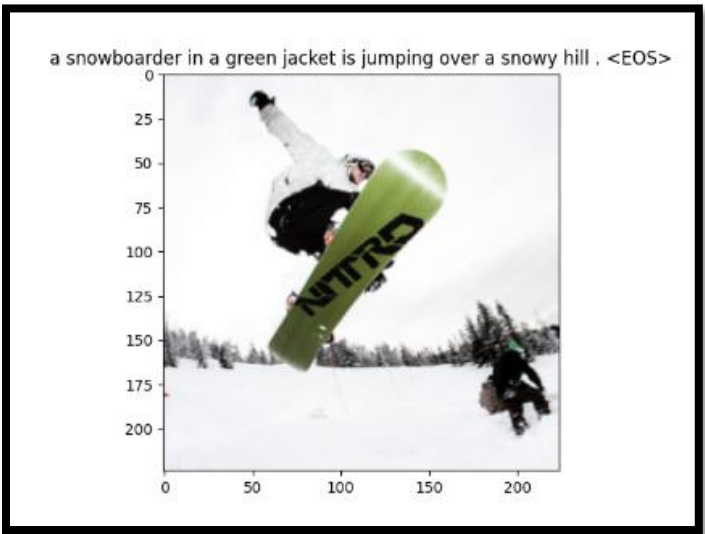
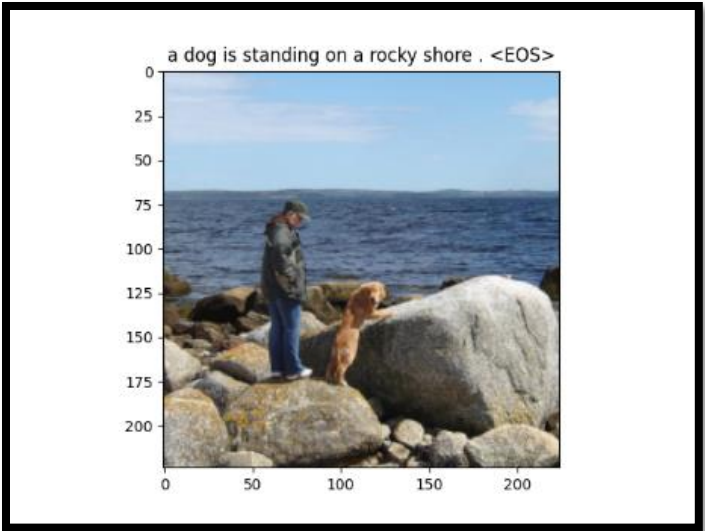




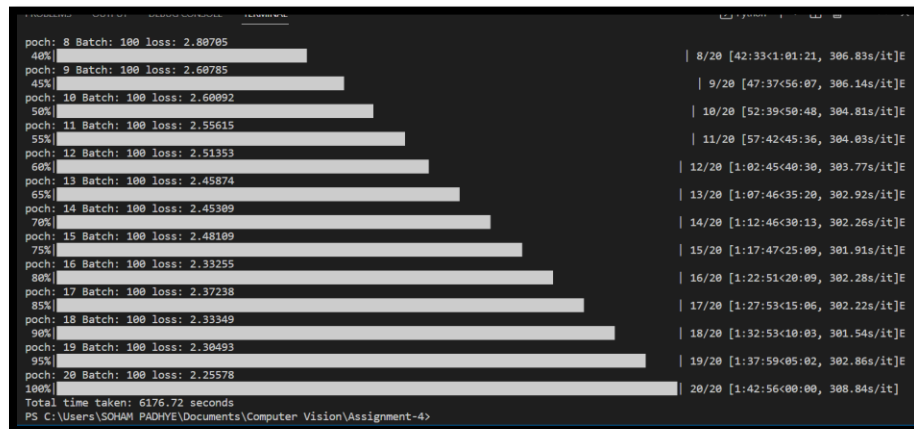
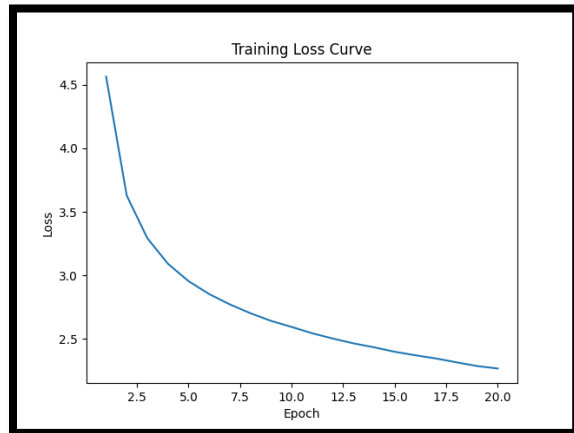


### Test images and its corresponding caption-





## Graph of loss curve during training-



As you can clearly see in the graph that the loss is decreasing continuously for each epoch. So, the model trained on this dataset is very good and it is producing the good results.

I have also shown the time required to run this model on my laptop. It is around **1.6 hours**.

## Evaluation metric for the performance of image captioning-

One of the most commonly used evaluation metrics for image captioning is the BLEU (Bilingual Evaluation Understudy) score. BLEU is a metric that measures how well the generated captions match the human reference captions. It works by comparing the n-gram (contiguous sequence of n words) overlap between the generated caption and the reference captions.

I would use a modified version of the CIDEr (Consensus-based Image Description Evaluation) score. CIDEr is a metric that evaluates the quality of the generated captions based on how well they describe the visual content of the image. It takes into account the word frequency, diversity, and importance of the generated captions.

The reason behind using a modified version of the CIDEr score is that it has been shown to be more correlated with human evaluation than other metrics such as BLEU. Additionally, CIDEr score takes into account not only the n-gram overlap but also the semantic similarity between the generated and reference captions.

Reference for CIDEr - <https://arxiv.org/abs/1411.5726>

Reference- <https://youtu.be/y2BaTt1fxJU>

### Question 3-

Use the dataset from Assignment 3 (Q. 4). Train YOLO object detection model (any version) on the train set. Compute the AP for the test set and compare the result with the HOG detector. Show some visual results and compare both of the methods.

**Colab link-** <https://colab.research.google.com/drive/1Cd4A9PAkc4xo7FGuiyHciwsDDmquG-Yo#scrollTo=odKEqYtTgbRc>

In this question I have trained YOLOV5 model on custom dataset that contains the deer images.

There are different variants of YOLOV5 like yolov5s, yolov5m, yolov5l, yolov5x. I have used yolov5s.

- I. YOLOv5s: The smallest and fastest version of YOLOv5, with a model size of 27 MB
- II. YOLOv5m: A medium-sized version of YOLOv5, with a model size of 54 MB
- III. YOLOv5l: A larger version of YOLOv5, with a model size of 113 MB
- IV. YOLOv5x: The largest and most computationally expensive version of YOLOv5, with a model size of 177 MB

Because the GPU capacity of colab is less I have chosen the small version of yolov5 that is yolov5s.

### About YOLOv5-

In YOLOv5, the neural network architecture is divided into three main components: backbone, neck, and head.

**Backbone:** The backbone is responsible for extracting features from the input image. It usually consists of several convolutional layers and is often pre-trained on a large dataset (e.g.,

ImageNet) to learn generic features that can be useful for a wide range of computer vision tasks.

**Neck:** The neck connects the backbone to the head and is responsible for further processing the features extracted by the backbone. In YOLOv5, the neck is a single convolutional layer that reduces the spatial resolution of the feature maps and increases the receptive field of the network.

**Head:** The head is responsible for predicting the bounding boxes and class probabilities for the objects in the image. It usually consists of several fully connected layers and is specific to the task at hand. In YOLOv5, the head is a combination of convolutional, upsampling, and prediction layers. The head predicts the bounding box coordinates, objectness scores, and class probabilities for each anchor box.

**Anchors** are pre-defined bounding boxes of different sizes and aspect ratios that are used by the model to predict the location and size of objects in the image. The anchor boxes are defined based on the size and aspect ratio of the objects in the dataset.

The backbone, neck, and head are typically trained end-to-end using a large annotated dataset to learn the optimal parameters for the task of object detection

Ref- <https://iq.opengenus.org/yolov5/>

### **Procedure for deer detection using YOLOv5s**

1. First clone the repository of yolov5 from github.
2. Then we will install all the requirements for running yolov5
3. Setting inbuilt Colab GPU to run the model
4. Then the next important step is collecting the deer dataset for yolo object. detection model training.
5. Then the load the data and check whether the data is loaded correctly or not. In our case there is only one class that is deer.
6. The YAML file specifies the hyperparameters of the model, the dataset and its properties, and the paths to the necessary files, such as images and annotations.
7. I have considered around 120+ images for creating the deer dataset.
8. For training the model in the first iteration I'm giving pretrained weights to the model. As we have very less number of images to train it is not possible to train the model on these small training data that's why im using pretrained weights. We will then finetune the model on deer dataset for detecting the deer in the image.

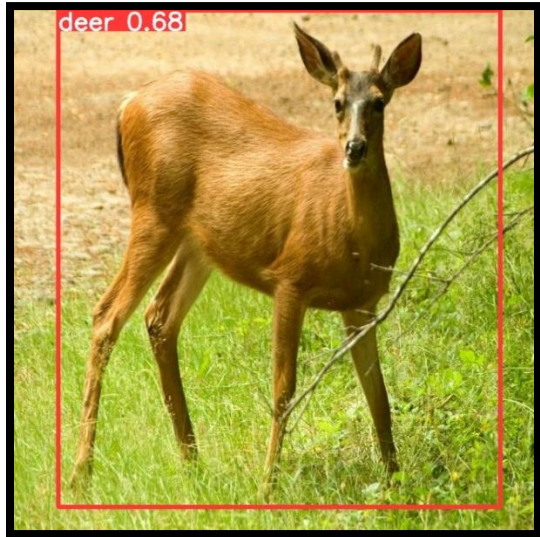
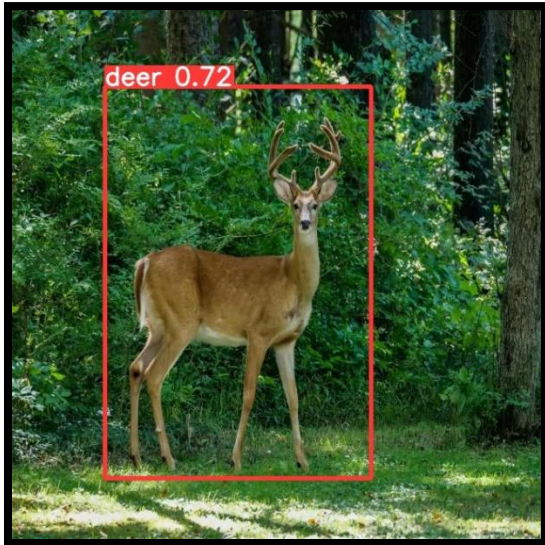
9. For training the model we have to give image size, batch size, epochs, dataset location and the main thing is pretrained weights.
10. Once the model is trained we will see the detector performance in various graphs like training loss, validation loss, recall, precision etc.
11. If the results are satisfying then we will go with the same model parameters.
12. In my case I tried different batch size and running the model without pretrained weights. It was giving very poor results on this model so I gave pretrained weights.
13. Then we visualize the validation batch so that we can get the idea about detection performance.

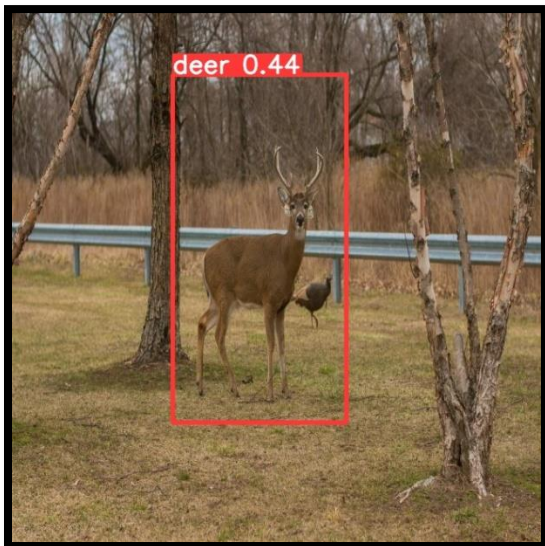
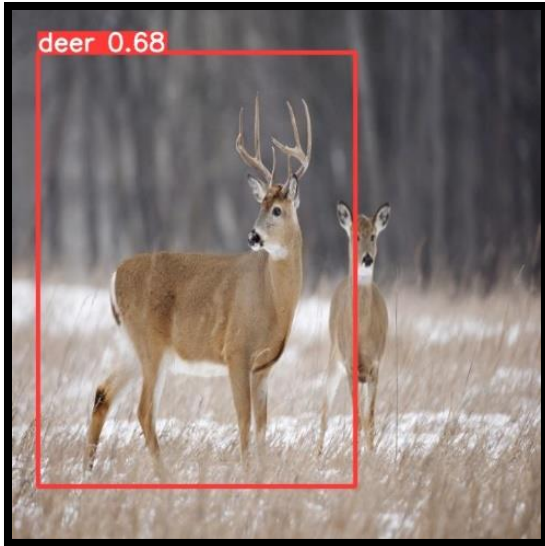


14. Once the model is trained two different weights files are generated named best.pt and last.pt. This means best corresponds to best weights that are obtained in epoch. And last.pt denotes training weights in last epoch.
15. Now we will use best weights to get the results on the testing data.
16. Here are the outputs from the code.

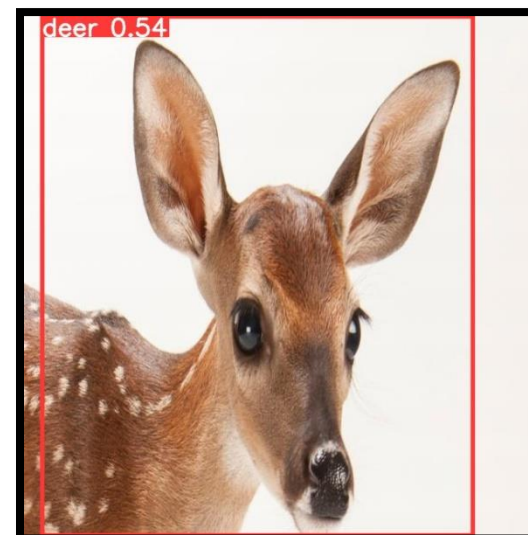
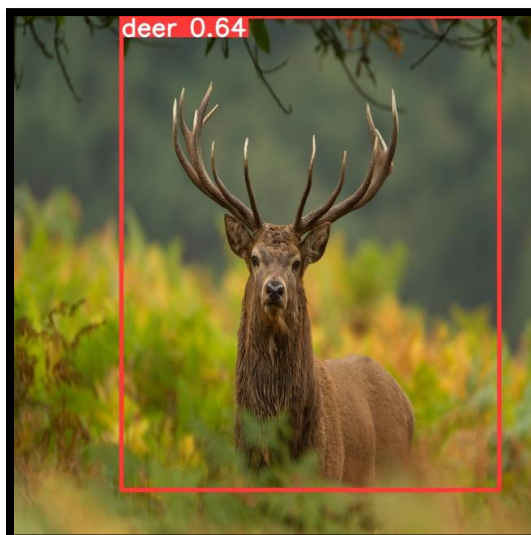
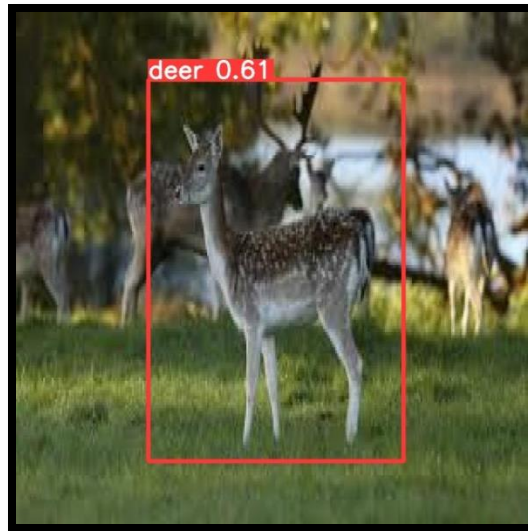
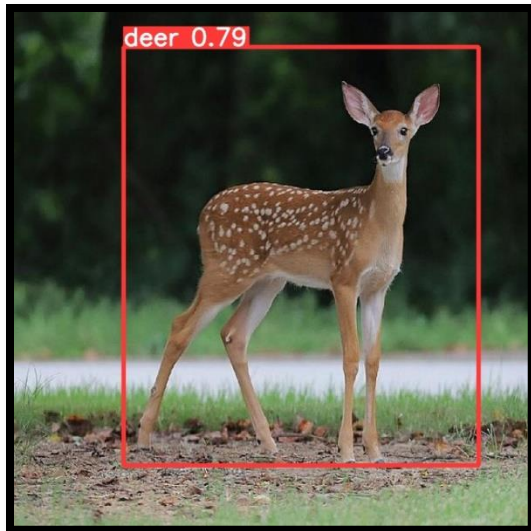
## Outputs-





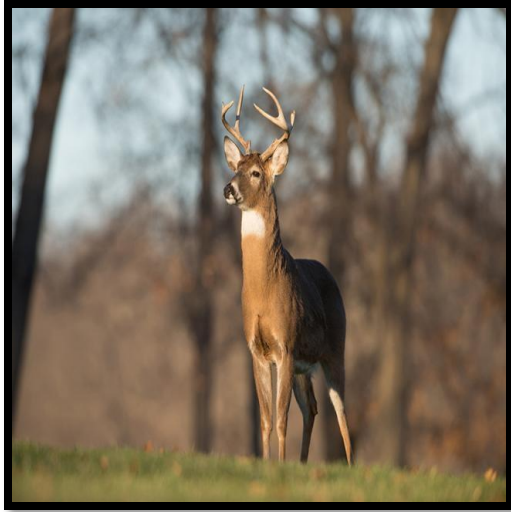






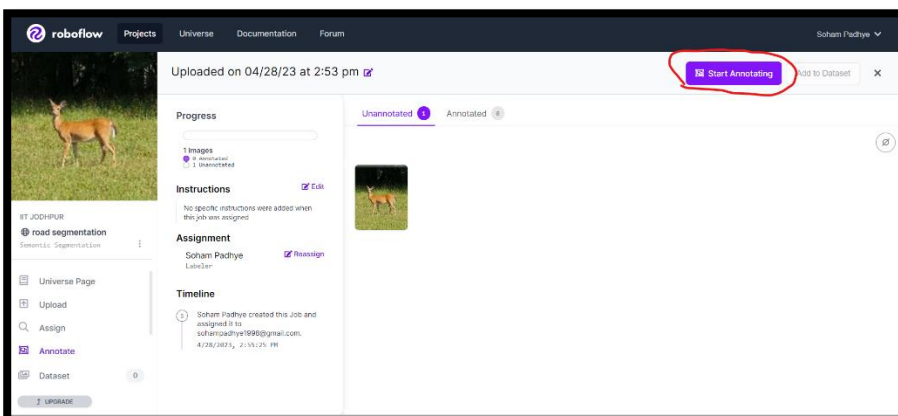
There are some test cases in which the deer is not detected –

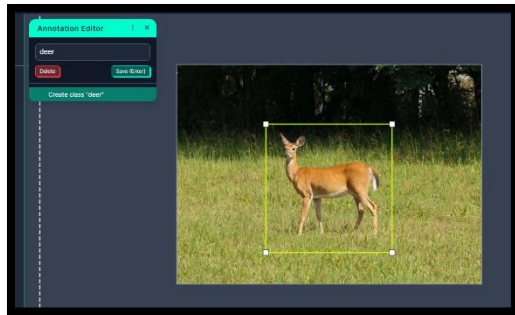
We can improve the performance of yolo object detection by using bigger version of yolov5 that is yolov5x Surely this will produce the good results.



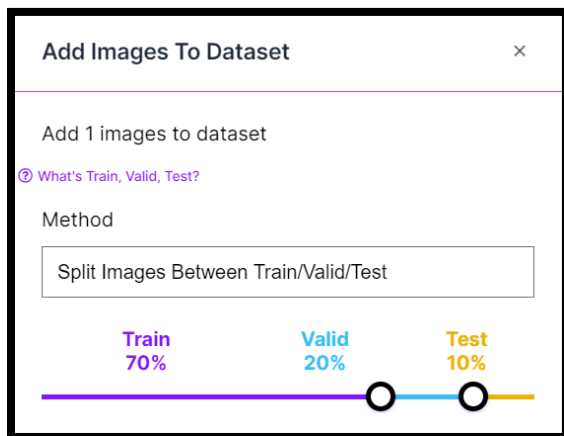
## Procedure for creating the custom dataset-

1. First download the deer images from the internet
2. I have used one extension for downloading the multiple images in one go. In this way I have downloaded the deer images from the internet also I have included the images from the last assignment also that were used for detecting the HoG features.
3. I have used Roboflow software to create the bounding box around the deer.
4. The output from the software is image with bounding box and one text file corresponding to the image. Text file contains the class of the image and coordinates of the bounding box. For example- 0 -class 0.507500 0.550117 -Centre 0.365000 0.503497 - height and width
5. Deer bounding box finding procedure-



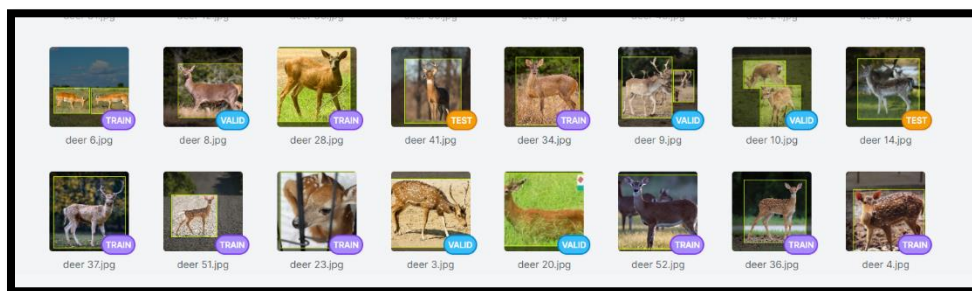


Then I have splitted the dataset for training validation and testing in the software itself

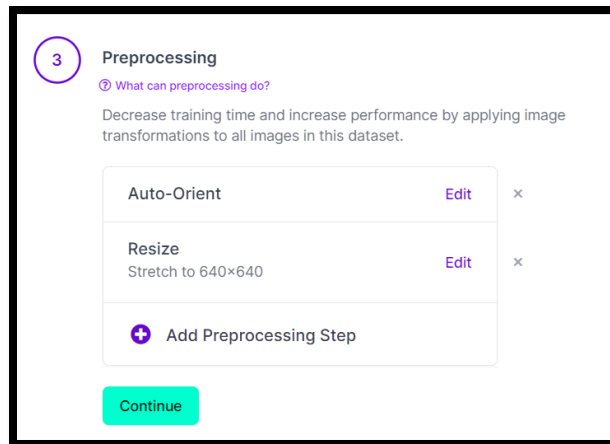


After splitting the images the overview of the dataset is like this

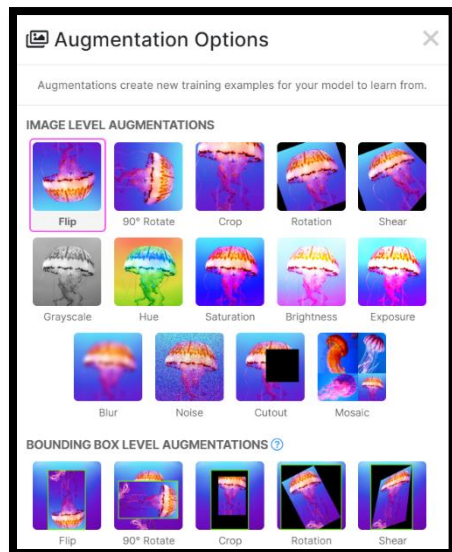
Images are marked with their group(training, validation,testing)



Then I have done the preprocessing on the images in same software-



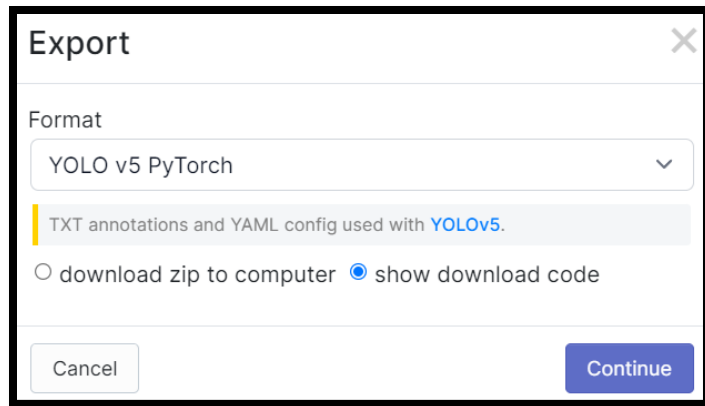
Then I have done the data augmentation and used some of the methods from below like flip, mirror saturate etc.



And finally the dataset is ready-



Now we can export this data in the code by pasting the snippet provided by Roboflow which is private to me. In this step we can select the export type for the different versions of yolo. I have selected yolov5 as export type.

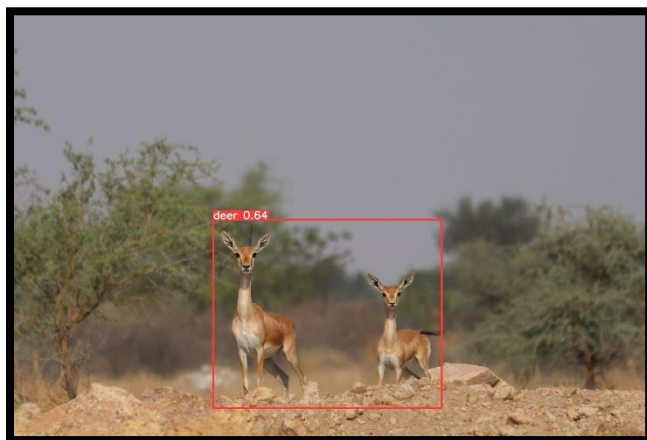


So finally we are ready with our dataset

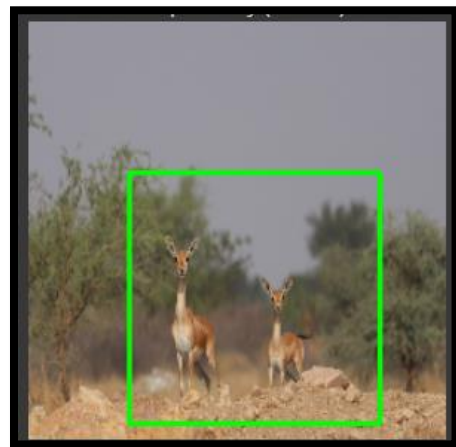
#### Points that needs to be considered while training the model-

1. First, we should change the number of classes to one as we are detecting the deer only in the image. That can be done by opening the yolov5.yml file.
2. Get the pretrained weights of yolov5s from github for training the model.
3. Finetune the model for deer dataset.

#### Comparing the results of HOG and YOLOv5 results-

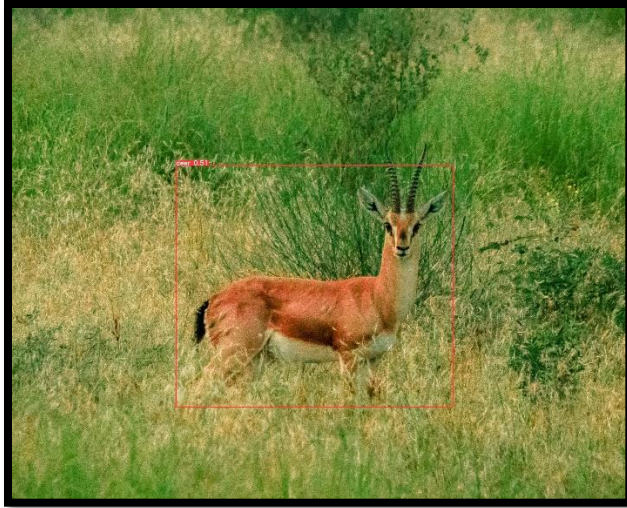


YOLOV5



HOG

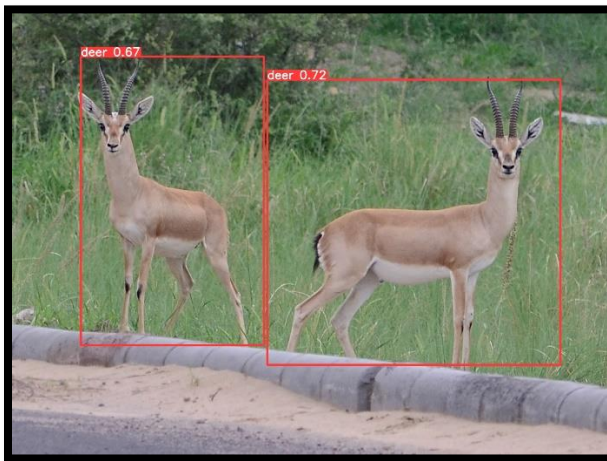




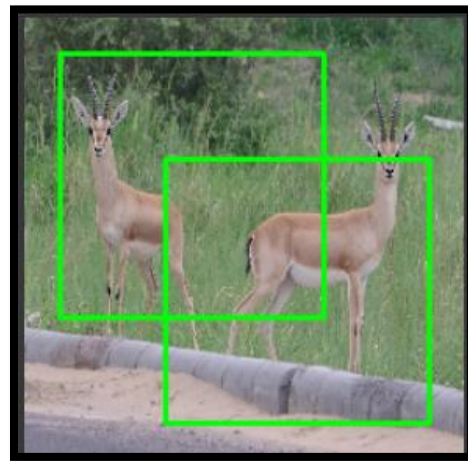
**YOLOV5**



**HOG**



**YOLOV5**



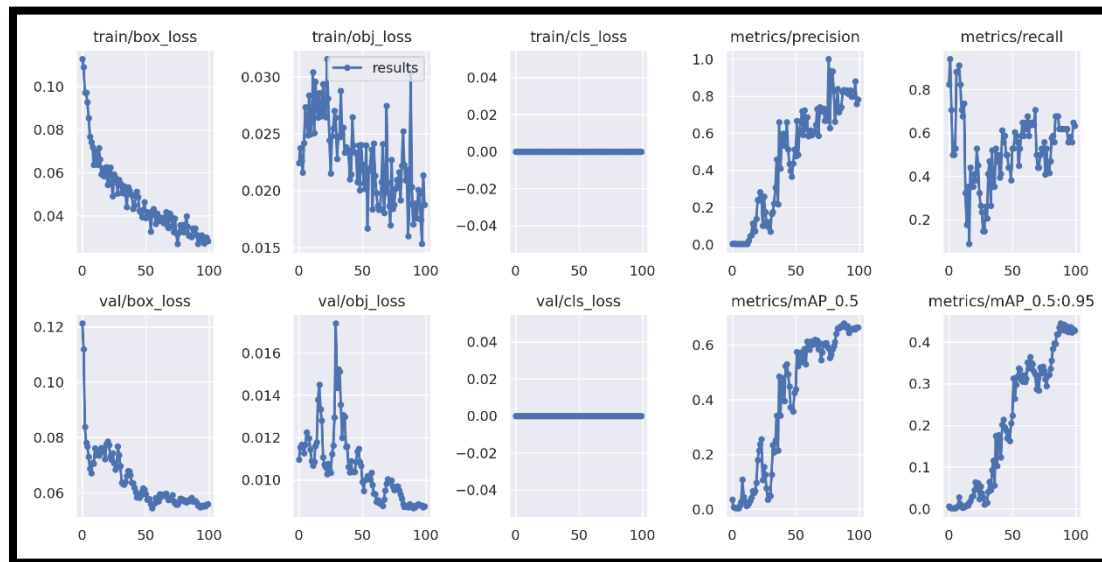
**HOG**

**This is the comparison between the HOG and YOLOV5 object detection-**

1. Object detection for HOG is not good as compared with the YOLOv5 as we can see the results.
2. In the HOG every time we should select the matching window size this is not the case for YOLOv5
3. HOG is a traditional computer vision approach which focuses on extracting features from images and using them to train classifiers for object detection. YOLOv5 is a deep learning-based approach so the results are good for YOLOv5
4. YOLOv5 is faster than HOG for object detection when processing large datasets.

- YOLOv5 has shown higher accuracy than HOG for object detection, particularly for complex objects and scenes as you can see it has perfectly detected 2 deer images.

### Performance characteristics of model-



In the above graph we can see the mAP50 and mAP50-95 values after each epoch.

mAP50 stands for mean Average Precision at 50% IoU threshold. It is a common evaluation metric used in object detection tasks to measure the accuracy of the model in detecting objects in an image. In our case we are detecting the deer by creating the bounding box around it.

mAP50 is calculated by first computing the Average Precision (AP) for each class at different IoU thresholds (e.g., 0.5, 0.75, 0.95), and then taking the mean of the AP scores across all classes. The "50" in mAP50 indicates that the AP is calculated at an IoU threshold of 50%. Higher mAP50 scores indicate better performance of the object detection model in terms of accuracy.

And we can see clearly in the graph that the mAP is increasing, which means the model is running fine. In my case I found mAP50 value as 0.678 which is good.

Precision is 0.832 and recall is 0.618

Generally, higher values of precision and recall are desirable, but In practice, the precision and recall values for object detection are often evaluated and compared using a metric such as the average precision AP that I have already calculated. If we use other versions of yolov5 like yolov5m, yolov5x the we will get better mAP for sure. Because the small model is built for the faster performance. That's why its mAP is less as compared with the tolov5x and yolov5m.

The advantage of yolov5s is that the time required to run the model and detect the test result is very fast at the cost of relatively less mAP. But still it is producing the good results.

```
Validating runs/train/yolov5s_results/weights/best.pt...
Fusing layers...
YOLOv5s summary: 157 layers, 7012822 parameters, 0 gradients, 15.8 GFLOPs

```

Class	Images	Instances	P	R	mAP50	mAP50-95
all	26	34	0.832	0.618	0.678	0.445

```
Results saved to runs/train/yolov5s_results
CPU times: user 3.41 s, sys: 247 ms, total: 3.65 s
```

### Learnings from the question-

- How to train the already defined model on custom dataset.
- How to finetune the model by giving the pretrained weights.
- Making the custom dataset for object detection
- Performance matrix for objection detection, Mean average precision and IoU loss.
- Different variants of YOLOv5.

Reference - <https://youtu.be/x0ThXHbtqCQ>



**Thank you**