Assignment 4

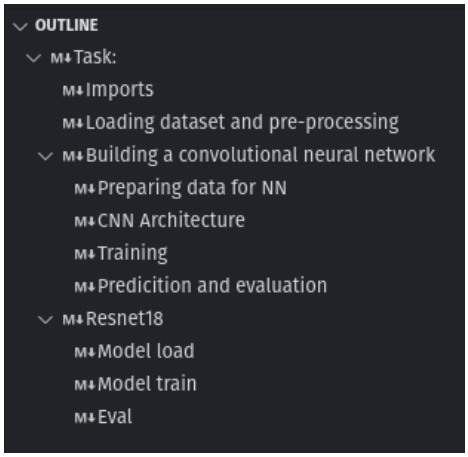
Abu Shahid- B20CS003

▼ `Note`

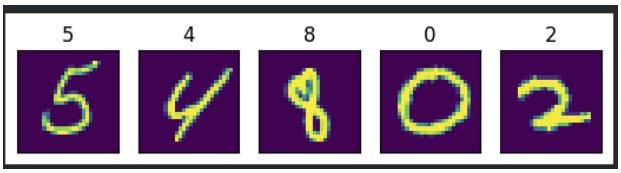
- Q2 was brainstormed collaboratively alongside `Aaditya Baranwal (B20EE001)`. This majorly involved getting the data loader ready and debugging the errors as we grasped the concepts of LSTM on the way. The model architecture was proposed by me and we both agreed to work on it. The analysis and evaluation were, however, done independently.
- In question 3, I used YOLOv5 and followed the official repository for implementation. Here

▼ MNIST with DL

▼ Output filter size



- The shape of dataset: `(70000, 784)`, (original image is 28x28)
- 80:20 split for train and test.



CNN Architecture

```
CNN(
  (layer1): Sequential(
```

```
    (0): Conv2d(1, 4, kernel_size=(7, 7), stride=(1, 1))
    (1): ReLU()
  )
  (layer2): Sequential(
    (0): Conv2d(4, 8, kernel_size=(5, 5), stride=(1, 1))
    (1): ReLU()
  )
  (layer3): Sequential(
    (0): Conv2d(8, 16, kernel_size=(5, 5), stride=(1, 1))
    (1): ReLU()
  )
  (layer4): Sequential(
    (0): Conv2d(16, 32, kernel_size=(5, 5), stride=(1, 1))
    (1): ReLU()
  )
  (layer5): Sequential(
    (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
    (1): ReLU()
  )
  (layer6): Sequential(
    (0): Conv2d(64, 96, kernel_size=(3, 3), stride=(1, 1))
    (1): ReLU()
  )
  (layer7): Sequential(
    (0): Conv2d(96, 96, kernel_size=(3, 3), stride=(1, 1))
    (1): ReLU()
  )
  (layer8): Sequential(
    (0): Conv2d(96, 128, kernel_size=(3, 3), stride=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (fc1): Linear(in_features=128, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=10, bias=True)
)
```

The formula to determine the output size is of filter is **[(W−K+2P)/S]+1, where**

- W: width of the input filter
- K: kernel size
- P: Padding size
- S: Stride

```
Layer 1:

Input size: 28x28
Convolution operation: (28 - 7 + 2*0) / 1 + 1 = 22
Output size: 22x22
Layer 2:

Input size: 22x22
Convolution operation: (22 - 5 + 2*0) / 1 + 1 = 18
Output size: 18x18
Layer 3:

Input size: 18x18
Convolution operation: (18 - 5 + 2*0) / 1 + 1 = 14
Output size: 14x14
Layer 4:

Input size: 14x14
```

```
Convolution operation: (14 - 5 + 2*0) / 1 + 1 = 10
Output size: 10x10
Layer 5:

Input size: 10x10
Convolution operation: (10 - 3 + 2*0) / 1 + 1 = 8
Output size: 8x8
Layer 6:

Input size: 8x8
Convolution operation: (8 - 3 + 2*0) / 1 + 1 = 6
Output size: 6x6
Layer 7:

Input size: 6x6
Convolution operation: (6 - 3 + 2*0) / 1 + 1 = 4
Output size: 4x4
Layer 8:

Input size: 4x4
Convolution operation: (4 - 3 + 2*0) / 1 + 1 = 2
Output size: 2x2
MaxPooling operation: 2 / 2 = 1
Output size: 1x1
```

The output size of the last convolutional layer (layer 8) is 2x2, which is then reduced to 1x1 through max pooling. The output of the last layer is then flattened and passed to the fully connected layers (fc1 and fc2) for classification.

▼ Parameters in CNN

The number of parameters associated with different layers are calculated as:

- `Conv2d`: (input_channels * output_channels * kernel_size + output_channels)
- `ReLU`: 0
- `MaxPool2d`: 0
- `Linear`: (input_features * output_features + output_features)

```
Layer 1:
Conv2d: (1 * 4 * 7 * 7) + 4 = 200 (input channels * output channels * kernel size + output channels)
ReLU: 0 (no parameters)
Total: 200

Layer 2:
Conv2d: (4 * 8 * 5 * 5) + 8 = 808
ReLU: 0
Total: 808

Layer 3:
Conv2d: (8 * 16 * 5 * 5) + 16 = 3,216
ReLU: 0
Total: 3,216

Layer 4:
Conv2d: (16 * 32 * 5 * 5) + 32 = 12,832
ReLU: 0
Total: 12,832

Layer 5:
Conv2d: (32 * 64 * 3 * 3) + 64 = 18,496
```

```
ReLU: 0
Total: 18,496

Layer 6:
Conv2d: (64 * 96 * 3 * 3) + 96 = 55,392
ReLU: 0
Total: 55,392

Layer 7:
Conv2d: (96 * 96 * 3 * 3) + 96 = 83,040
ReLU: 0
Total: 83,040

Layer 8:
Conv2d: (96 * 128 * 3 * 3) + 128 = 110,720
ReLU: 0
MaxPool2d: 0
Total: 110,720

Fully connected layer 1 (fc1):
Linear: (128 * 64) + 64 = 8,256
Total: 8,256

Fully connected layer 2 (fc2):
Linear: (64 * 10) + 10 = 650
Total: 650
```

The total number of parameters: 293,610

▼ Reporting on test data

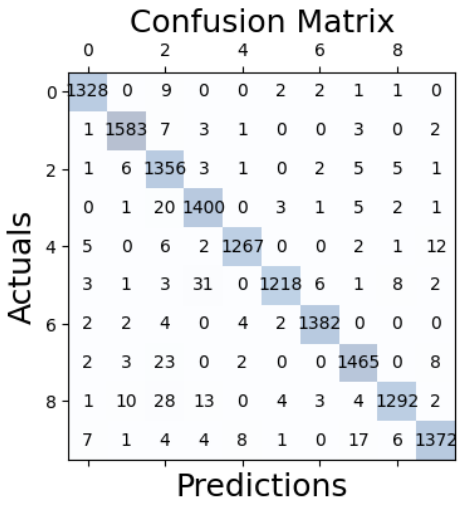
`Confusion Matrix`, `Overall & Classwise Accuracy` and `ROC curve` were implemented from scratch. You may refer to the attached files for reference.

Reference for ROC curve- `here`

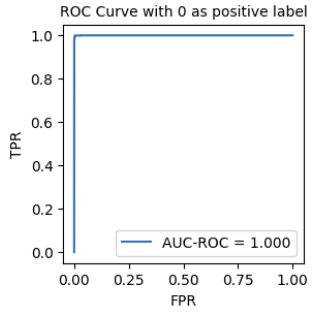
Reference for neat plotting- `here`

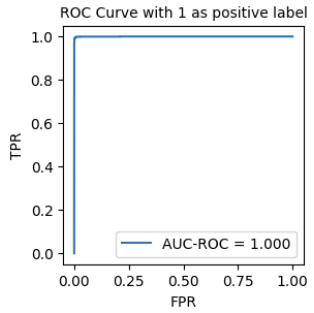
```
num_epochs = 15
num_classes = 10
learning_rate = 0.0003

valid_size = 0.1
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(),lr=learning_rate)
```




```
Overall Accuracy : 0.9759
Accuracy of 0 : 0.989
Accuracy of 1 : 0.989
Accuracy of 2 : 0.983
Accuracy of 3 : 0.977
Accuracy of 4 : 0.978
Accuracy of 5 : 0.957
Accuracy of 6 : 0.99
Accuracy of 7 : 0.975
Accuracy of 8 : 0.952
Accuracy of 9 : 0.966
```



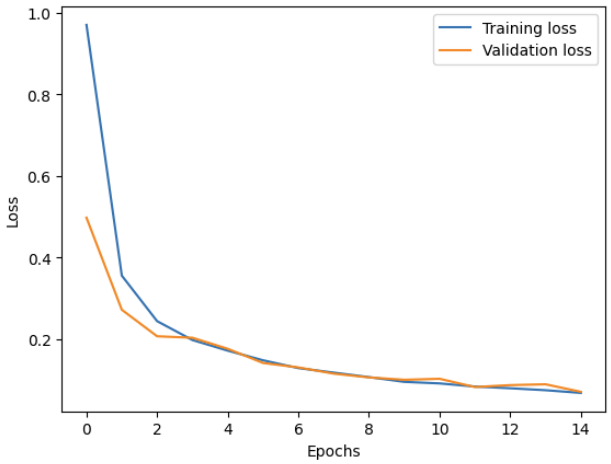


ROC curves for other digits can be found in the .ipynb file

▼ Loss curve during training

Model was trained for 10 epochs

```
Training Loss @1 epoch = 0.970
Training Loss @2 epoch = 0.355
Training Loss @3 epoch = 0.244
Training Loss @4 epoch = 0.197
Training Loss @5 epoch = 0.171
Training Loss @6 epoch = 0.148
Training Loss @7 epoch = 0.128
Training Loss @8 epoch = 0.117
Training Loss @9 epoch = 0.106
Training Loss @10 epoch = 0.095
Training Loss @11 epoch = 0.091
Training Loss @12 epoch = 0.083
Training Loss @13 epoch = 0.079
Training Loss @14 epoch = 0.074
Training Loss @15 epoch = 0.068
```



▼ ResNet18

For this part, we imported un-trained model from PyTorch using

```
import torch.optim as optim
import torchvision.models as models
resnet = models.resnet18(pretrained=False)
```

To this model, we just added an input and output layer to deal with our input images and give desired output shape.

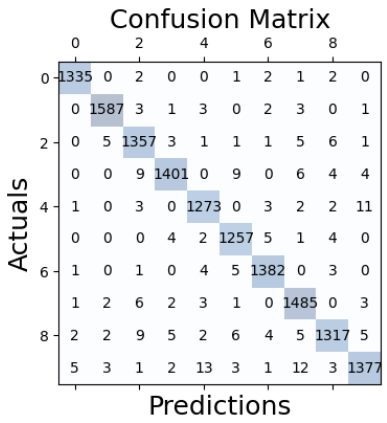
```
resnet.conv1 = nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3, bias=False)
```

```
resnet.fc = nn.Linear(resnet.fc.in_features, 10)
```

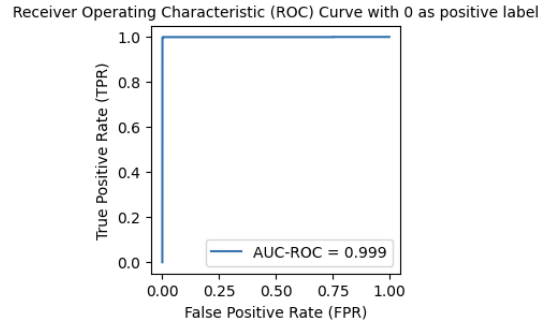
Model parameters

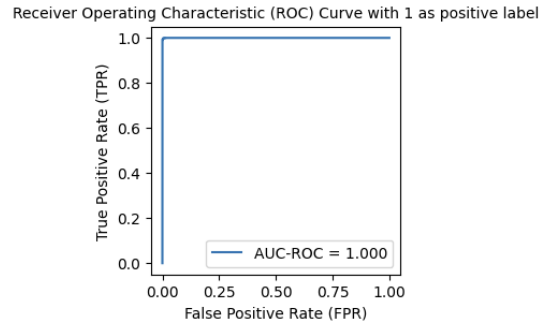
```
num_epochs = 10
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(resnet.parameters(), lr=0.001, momentum=0.9)
```

▼ Metrics and evaluation




```
Overall Accuracy : 0.9836
Accuracy of 0 : 0.994
Accuracy of 1 : 0.992
Accuracy of 2 : 0.983
Accuracy of 3 : 0.978
Accuracy of 4 : 0.983
Accuracy of 5 : 0.987
Accuracy of 6 : 0.99
Accuracy of 7 : 0.988
Accuracy of 8 : 0.971
Accuracy of 9 : 0.97
```





▼ Comment

The overall accuracy of both models is relatively high, which indicates that both models are performing well on the given task. However, the accuracy of the ResNet model is slightly higher than ours.

One possible reason for the difference in accuracy could be the architecture of the models. ResNet is a more profound architecture with skip connections, which helps to address the vanishing gradient problem and improves the flow of information throughout the network. On the other hand, our model is a shallower architecture and may not have the same level of information flow.

Another reason could be the number of epochs trained. The accuracy of our model could be improved with more training epochs.

▼ Image Captioning

- Data was bifurcated into standard train-test-val.
- For the encoder and decoder, we used ResNet18 and LSTM, respectively.

▼ Model

- We calculated our embeddings from Resnet50 and stored it. This translates to freezing our encoder when we train our decoder. Used weights of the Resnet50 model trained on `imagenet` dataset.
- Our vocab size is `8254`

- Our dataset consists of images and a text file in the following format

```
1026685415_0431cbf574.jpg#4	A wet black dog is carrying a green toy through the grass .
1028205764_7e8df9a2ea.jpg#0	A man and a baby are in a yellow kayak on water .
1028205764_7e8df9a2ea.jpg#1	A man and a little boy in blue life jackets are rowing a yellow canoe .
1028205764_7e8df9a2ea.jpg#2	A man and child kayak through gentle waters .
1028205764_7e8df9a2ea.jpg#3	A man and young boy ride in a yellow kayak .
1028205764_7e8df9a2ea.jpg#4	Man and child in yellow kayak
1030985833_b0902ea560.jpg#0	A black dog and a brown dog are jumping up to catch a red toy .
1030985833_b0902ea560.jpg#1	A black dog and a brown dog play with a red toy on a courtyard .
1030985833_b0902ea560.jpg#2	A brown and black lab are outside and the black lab is catching a toy in its mouth .
1030985833_b0902ea560.jpg#3	Black dog snaps at red and black object as brown dog lunges .
1030985833_b0902ea560.jpg#4	The Chocolate Lab jumps too late to get the toy as the Black Lab captures it in the driveway .
103106960_e8a41d64f8.jpg#0	A boy with a stick kneeling in front of a goalie net
103106960_e8a41d64f8.jpg#1	A child in a red jacket playing street hockey guarding a goal .
```

- Here each line consists of image name and a caption description, separated by `#` and a number. Based on this we define our `CaptionDataset()` class.
- `class CaptionDataset():`
- `__init__(self, img_caption_file, transforms=None, img_dir='', max_len=40, encoding_model=models.resnet50(weights=ResNet50_Weights.IMAGENET1K_V1))`: Initializes the class instance with the input parameters. This method also initializes various instance variables and calls other methods to populate these variables.

- `get_data(self)`: Parses the input file to extract image paths and their corresponding captions. It also creates a vocabulary of unique words present in the captions.
- `preprocessing(self, img_path)`: Preprocesses an image from its file path by applying transformations and returns the processed image as a numpy array.
- `pad(self, seq)`: Pads the input sequence with zeros to make it of maximum length `max_len`.
- `make_dataset(self)`: Creates the dataset by splitting each caption sequence into multiple partial sequences of increasing length and their corresponding subsequent words. These partial sequences are padded to the maximum length, and the subsequent words are one-hot encoded.
- `get_encoded_images(self)`: Encodes each image in the dataset using the specified encoding model (default is ResNet50), and returns the encoded features as a numpy array.
- `__len__(self)`: Returns the length of the dataset.
- `__getitem__(self, idx)`: Returns the image and caption sequence at the specified index `idx`.

```
embedding_size = 128
hidden_size = 256
max_len = 40
batch_size=256
epochs=200
LEARNING_RATE = 0.001
MOMENTUM = 0.9
WEIGHT_DECAY = 0.0005
```

- We create an intermediate image processing model called `image_model` in between our ResNet-based image encoder and LSTM-based text decoder. The "image_model" consists of two layers:

1. A fully connected layer that reduces the dimensionality of the image representation generated by the ResNet-based encoder and generates a condensed feature vector of fixed length using the ReLU activation function.
2. A layer that repeats the condensed feature vector generated by the first layer "max_len" number of times to create a fixed-length input sequence that can be fed into the LSTM-based text decoder to generate a sequence of words.

- We then define our Decoder `language model`.

The model consists of three layers:

1. An embedding layer that takes an input sequence of integers representing words in the input text and generates a sequence of word embeddings of fixed size "embedding_size". Each input integer is mapped to a dense vector representation of the corresponding word.
2. An LSTM layer that takes the sequence of word embeddings generated by the embedding layer and processes them to generate a sequence of hidden state vectors. The LSTM layer has 256 memory units, and the "return_sequences" parameter is set to True, which means that the layer returns the full sequence of hidden state vectors rather than just the final output.
3. A dense layer that takes the sequence of hidden state vectors generated by the LSTM layer and applies a linear transformation to each element in the sequence to generate a sequence of output

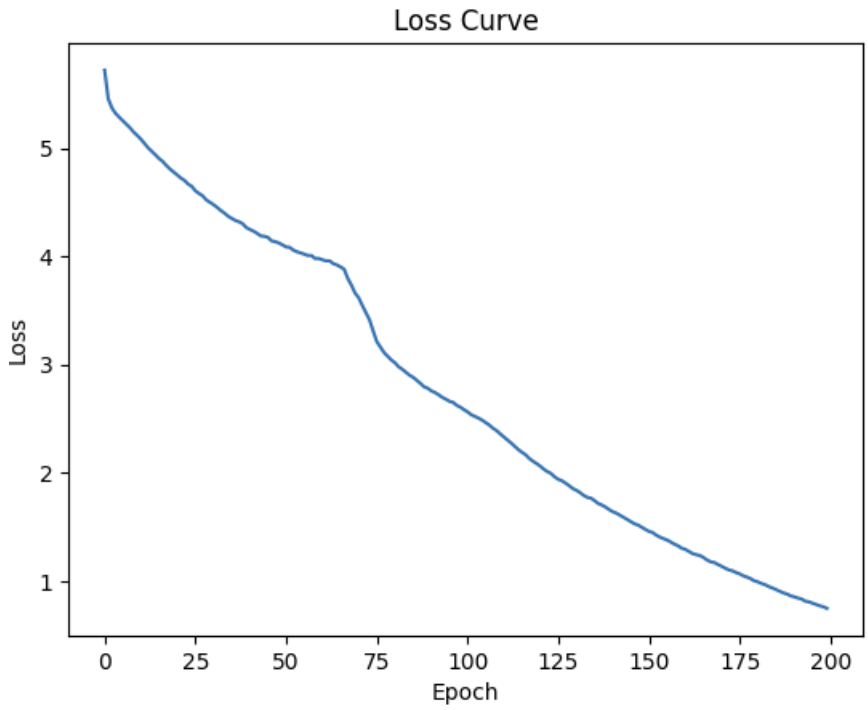
vectors of fixed size "embedding_size". The wrapper is used to apply the dense layer to each element in the sequence independently, rather than applying the same weights to all elements in the sequence.

This architecture was then put to training.

In our dataset, there were `8091` images and `40460` captions.

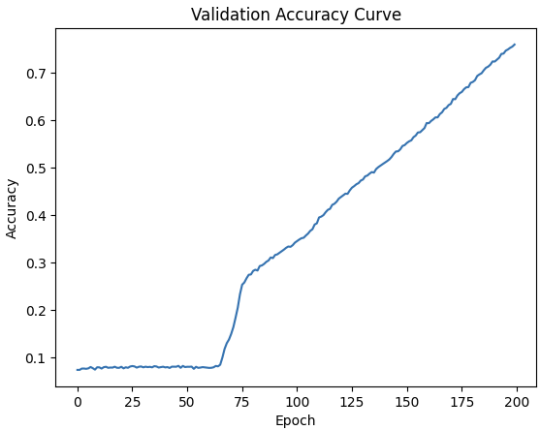
- For tokenization, we used `BERT Toeknizer` and `nltk` for basic text pre-processing.

▼ Training Curve Loss



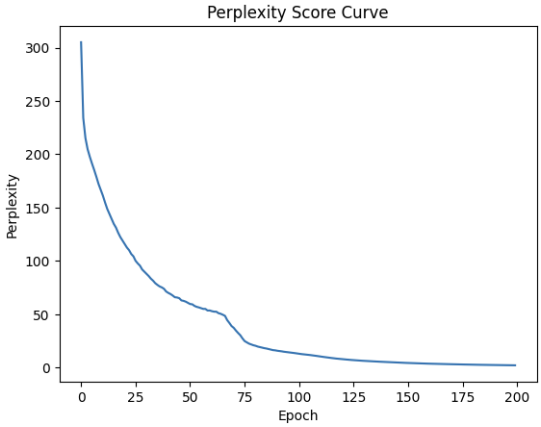
▼ Evaluation

`maximum validation accuracy: 0.7598`



Our metric of choice was `perplexity score`. Perplexity is defined as the exponentiated average negative log-likelihood of a sequence. Moreover, due to its straightforward calculation, perplexity is a favourite choice for NLP tasks. In general, perplexity is a measurement of how well a probability model predicts a sample. In the context of Natural Language Processing, perplexity is one way to evaluate language models.

The lower the `perplexity score` the better the result.

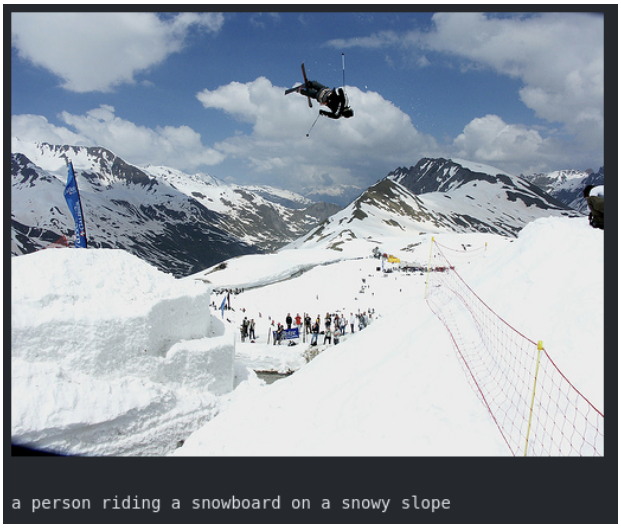


▼ The reason behind the selection of the metric

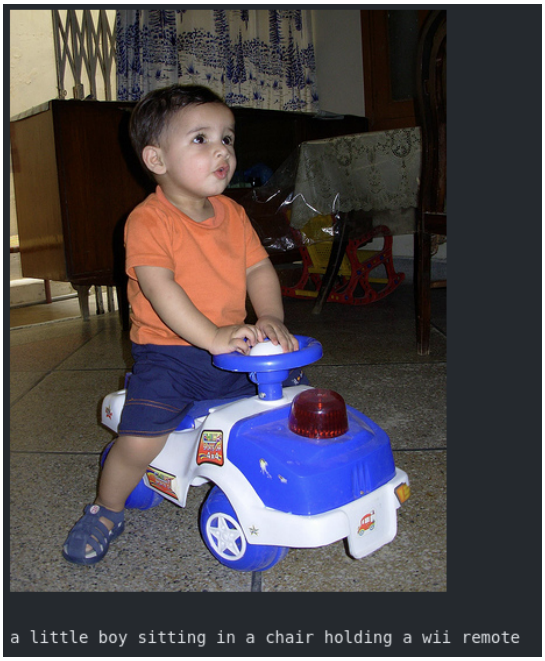
1. It directly measures the model's ability to predict the next word in the sequence, which is the primary goal of the language modeling task.
2. It is a standard metric used to evaluate language models, which makes it easy to compare the performance of different models and approaches.
3. It provides a probabilistic interpretation of the model's performance, which can be useful for understanding the model's behavior and identifying areas for improvement.
4. It is differentiable, which means it can be used as a loss function for training neural network-based language models using gradient-based optimization methods like backpropagation.

▼ Final result

a dog running with a frisbee in its mouth



a person riding a snowboard on a snowy slope



▼ YOLO on Deers

To create the dataset, we used the images from `Assignment-3`

Further, we produced the annotated data in COCO dataset format, using an online tool.

The dataset is available here.

```
@misc{ deerfinder-uuxyf_dataset,
    title = { DeerFinder Dataset },
    type = { Open Source Dataset },
    author = { CV },
    howpublished = { \url{ https://universe.roboflow.com/cv-bvqsm/deerfinder-uuxyf } },
    url = { https://universe.roboflow.com/cv-bvqsm/deerfinder-uuxyf },
    journal = { Roboflow Universe },
    publisher = { Roboflow },
    year = { 2023 },
    month = { apr },
    note = { visited on 2023-04-27 },
}
```

I used YOLOv5 and followed the official repository for implementation. Here

▼ Training

For this part, the official implementation of Yolov5 was used, here. Having had our dataset prepared, we confined ourselves to the following steps.

- `train.py`: Used the following hyperparameters for training
  - We used a pre-trained model and fine-tuned it on our dataset. For this, we froze the first 9 layers of the model and the remaining models were set to train.

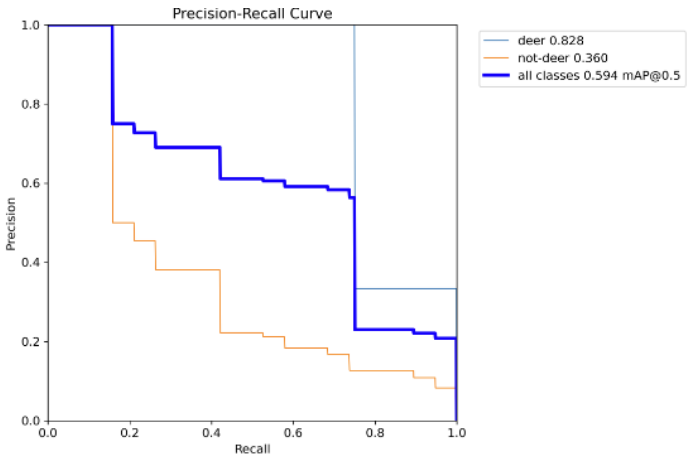
```
--img 448 --batch 1 --epochs 200 --data
```

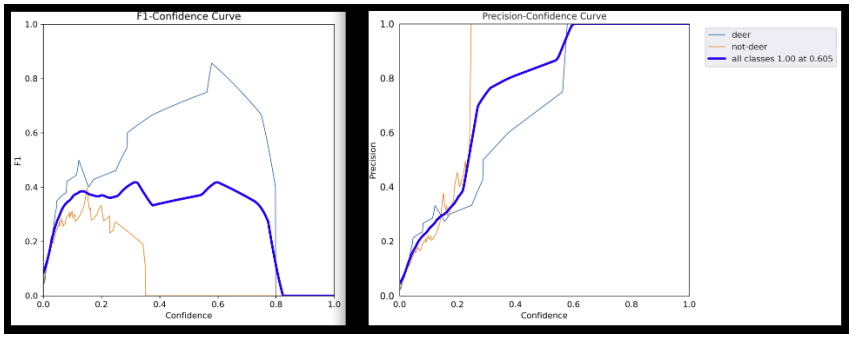
- `val.py`:

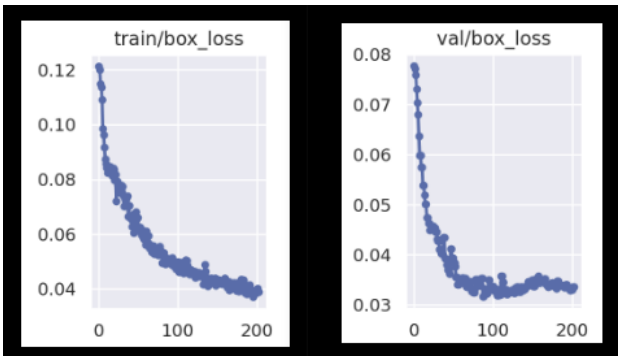
```
--iou-thres 0.3 --single-cls
```

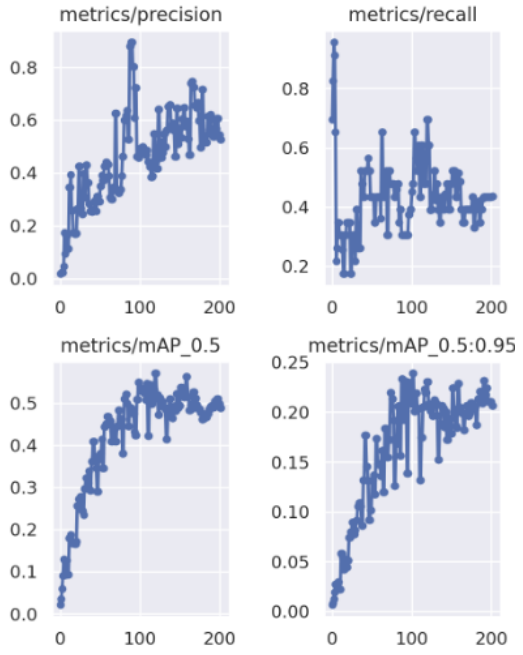
▼ Training results can be summarized as follows:

```
Model summary: 157 layers, 7015519 parameters, 0 gradients, 15.8 GFLOPs
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100%
                   all          3         23          1      0.348      0.594       0.34
                  deer          3          4          1      0.696      0.828      0.559
              not-deer          3         19          1          0       0.36       0.12
```

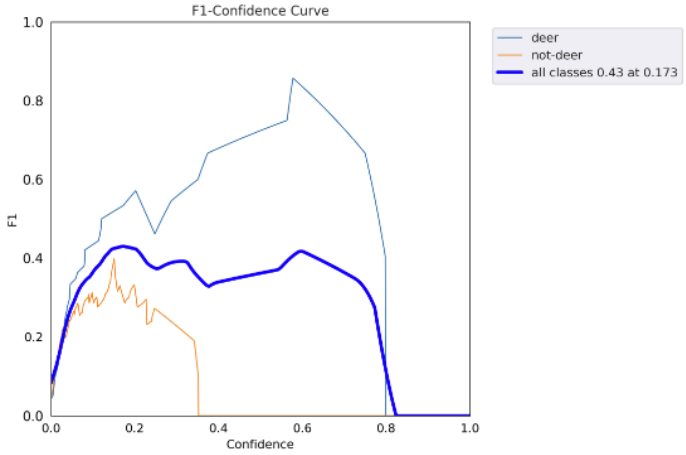


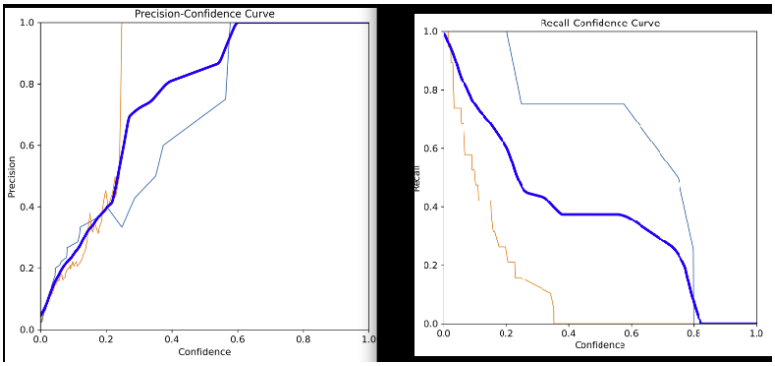






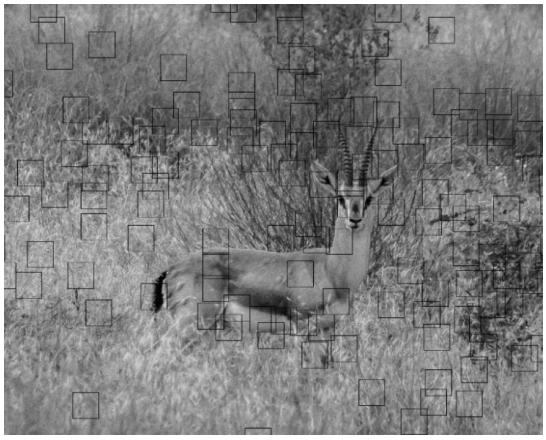
▼ Results on validation data can be summarized as follows





▼ **Comparison with HOG**

My previous implementation of HOG detector (please refer to my submission for `Assignment-3` was giving following results.

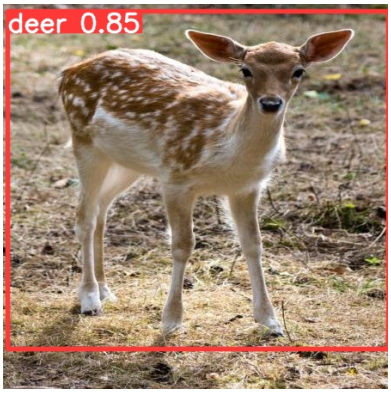


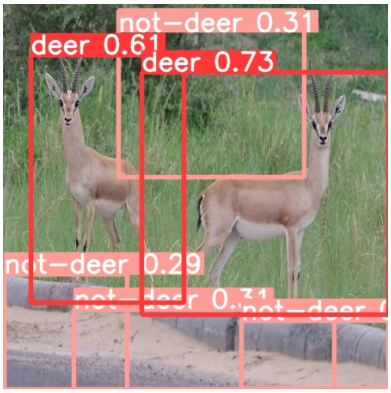
Clearly, the result is nowhere close to YOLO results. (Result was similar with other images as well.

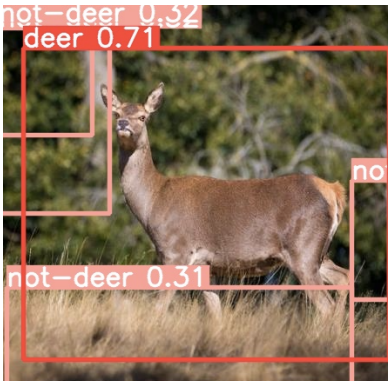
Reasoning:
Training data is very sparse: It might very much be possible that 11 images are not enough to learn the latent features, Although, I tuned the classifier to use a poly kernel with degree 3, enough features could not be learned. Next, test_set is not representative of training data. While all the images in train_data have dimensions not more than (600,500) with the actual deer being localizable in (400,400) , the images in test_set are (4000,2000) on average with deer roughly localizable in (1000,1000) . Therefore the HOG parameters for detecting deer on train_set_population will be different from that on test_set_population
Another reason is the inclusion of non-deer patches in the tight-deer crop. To test this, I made another set of manually cropped deer images, but this time excluding their horns so as to keep the non-deer pixels in the crop to a minimum (refer to the report for assignment-3). Doing as said, gave the following results.

Clearly, the result is better in terms of way fewer false positives. The model however still fails to be as good as YOLOv5.

▼ **Visual Results**






not-deer 0.32
deer 0.71
no
not-deer 0.31