CV Assignment-4 Report

Note: All the links for the code and data are provided at the end of the report.

Task: To give solutions and approaches to the given problems.

Objectives:

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.**

Procedure:

- Checking the available GPU using !nvidia-smi.
- Importing the required libraries: torch for PyTorch, nn for neural networks, torchvision for computer vision tasks, matplotlib for plotting graphs, numpy for numerical computations, and torchmetrics and torchinfo for evaluating the model.
- Creating a variable device to store the device used for computations. The variable is set to "cuda" if a GPU is available, or "cpu" otherwise.
- Downloading the MNIST dataset and transforming it into tensors using PyTorch's torchvision module.
- Visualizing some samples from the train dataset using matplotlib. The images are selected randomly and plotted along with their corresponding labels.
- Converting the datasets into PyTorch dataloaders to facilitate batch processing.
- Defining the CNN architecture using a custom MNISTModel class that extends nn.Module. The architecture consists of three convolutional blocks followed by a classifier.

- Initializing the model with the chosen hyperparameters: input_shape (the number of input channels), hidden_units (the number of hidden units in the convolutional layers), and output_shape (the number of output classes).
- Defining the loss function (CrossEntropyLoss) and optimizer (Adam) used to train the model.
- Defining the training and validation loops using PyTorch's built-in nn.Module functionality. The training loop includes backpropagation and gradient descent steps, while the validation loop only evaluates the model's performance on the validation set.
- Running the training and validation loops for a specified number of epochs and storing the results in two lists: train_losses and valid_losses.
- Evaluating the model's accuracy using the Accuracy class from torchmetrics and printing the result.
- Visualizing the loss curves using matplotlib.
- Training custom and ResNet-18 models on the MNIST dataset.
- They use different combinations of hyperparameters such as learning rate, beta, epsilon, weight decay, and number of epochs.
- They use the Adam optimizer for training.
- They compute the loss and accuracy curves for each experiment.
- They time the training process for each experiment using the timeit library.
- They save the best-performing model based on the lowest loss and highest accuracy values.
- They load the best-performing model and compute the confusion matrix, class-wise accuracy, and overall accuracy for the test dataset.
- They plot the confusion matrix using the seaborn library.
- Results:

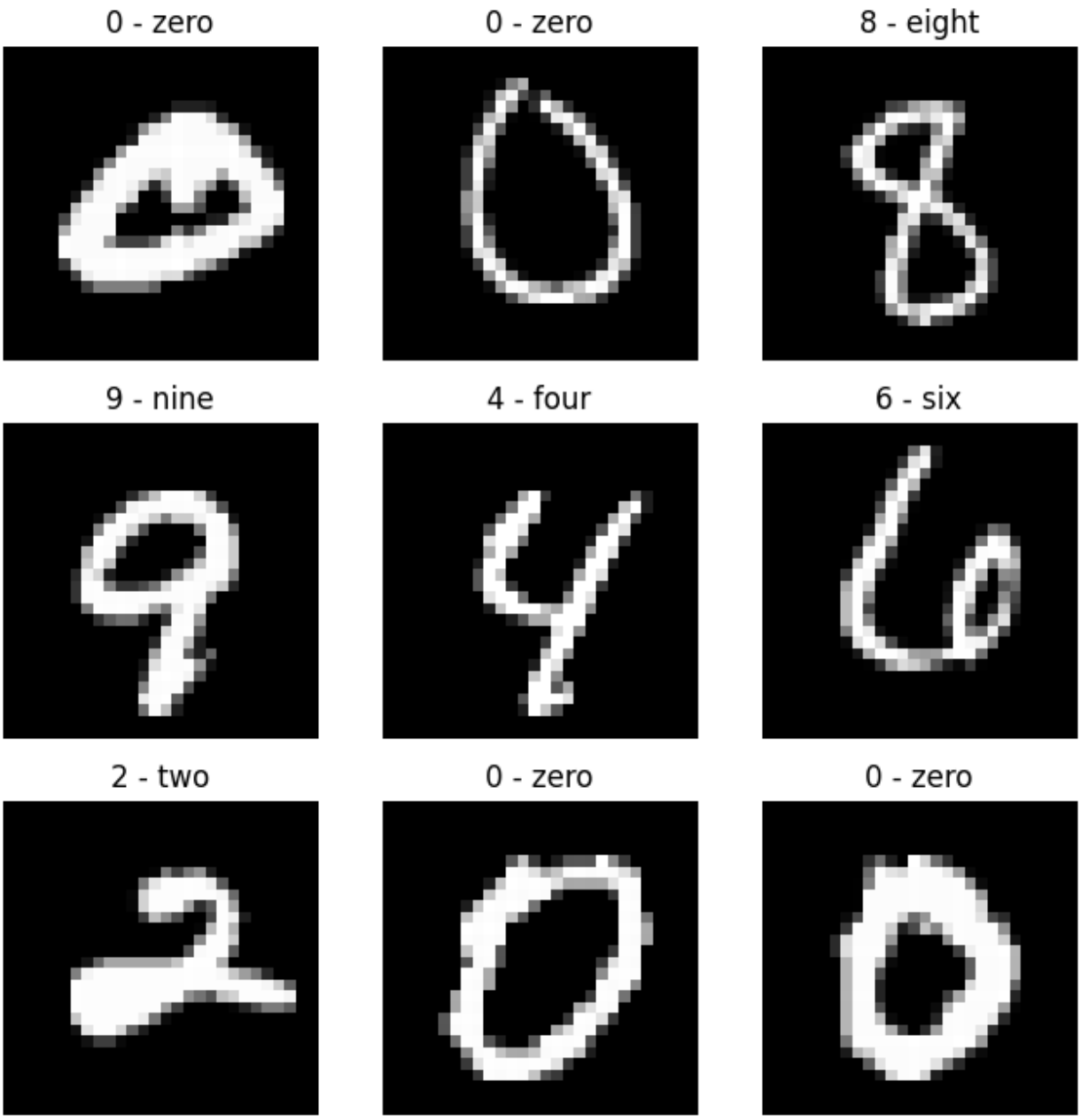
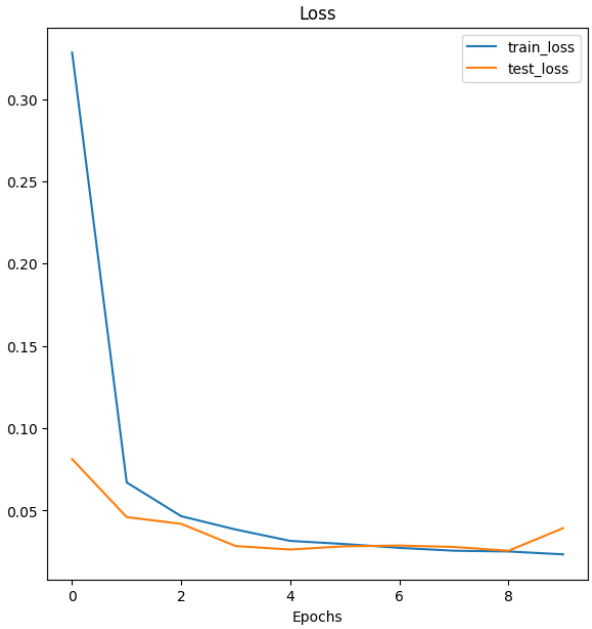


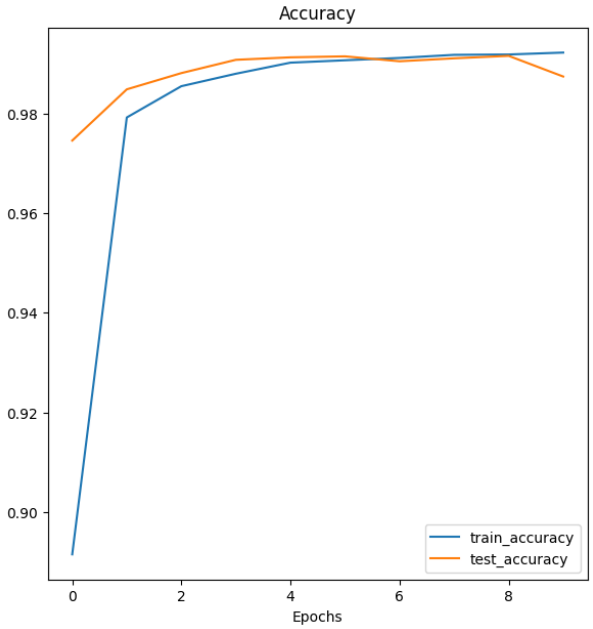
Fig: Random Samples from the Dataset

Training Curves for Custom CNN Architecture:

current exp / total: 1 / 8
Training with: lr: 0.001, betas: (0.8, 0.888), eps: 1e-08, weight_decay: 0.001

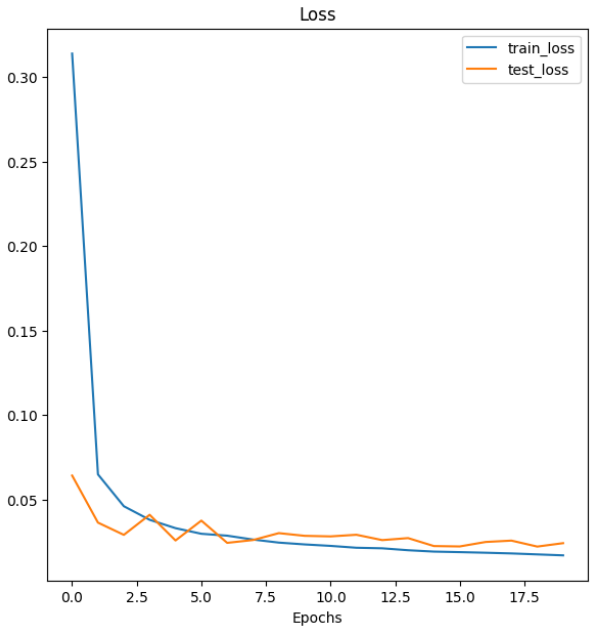
LOSS & Accuracy Curves

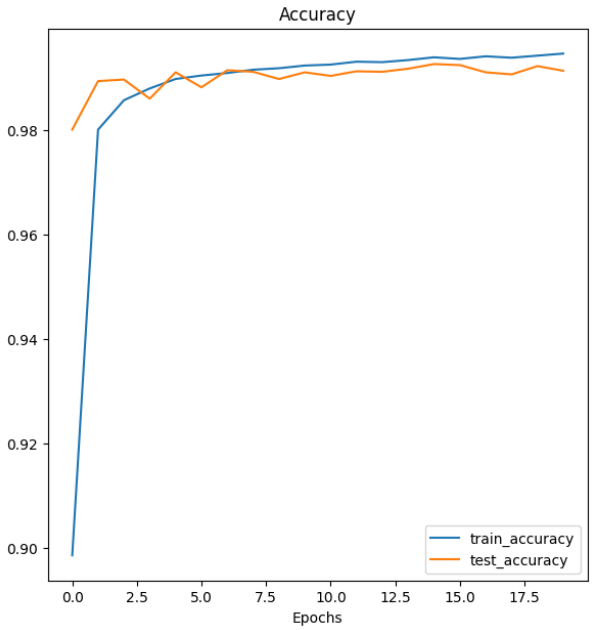




current exp / total: 2 / 8
Training with: lr: 0.001, betas: (0.8, 0.888), eps: 1e-08, weight_decay: 0.001

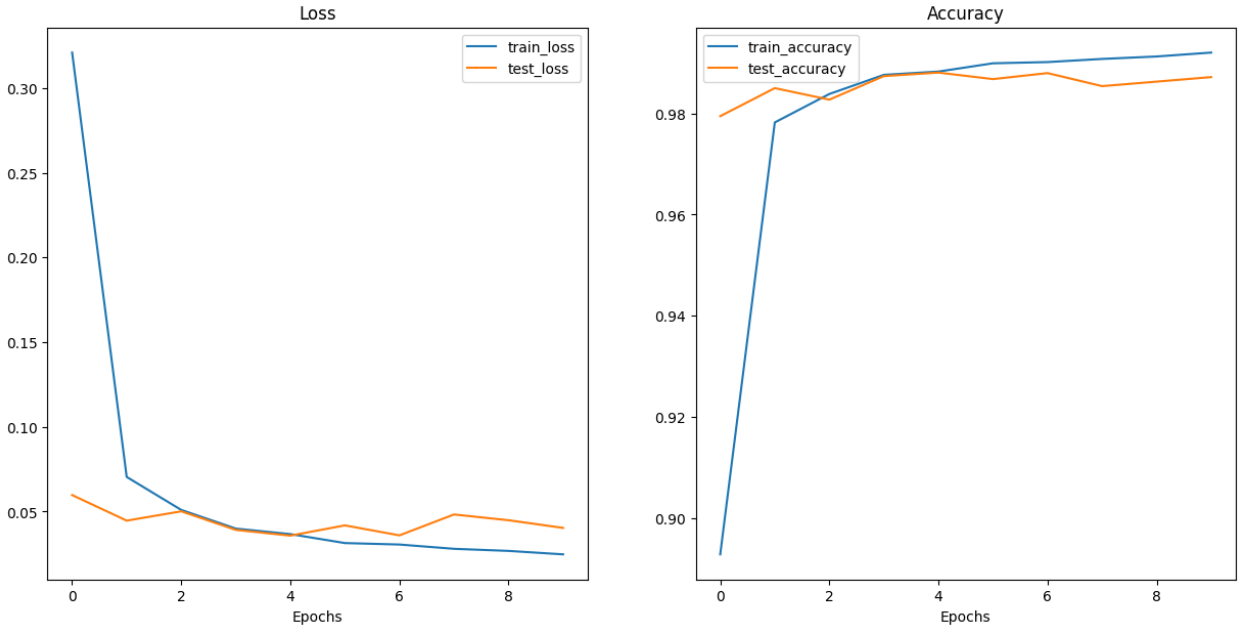
LOSS & Accuracy Curves





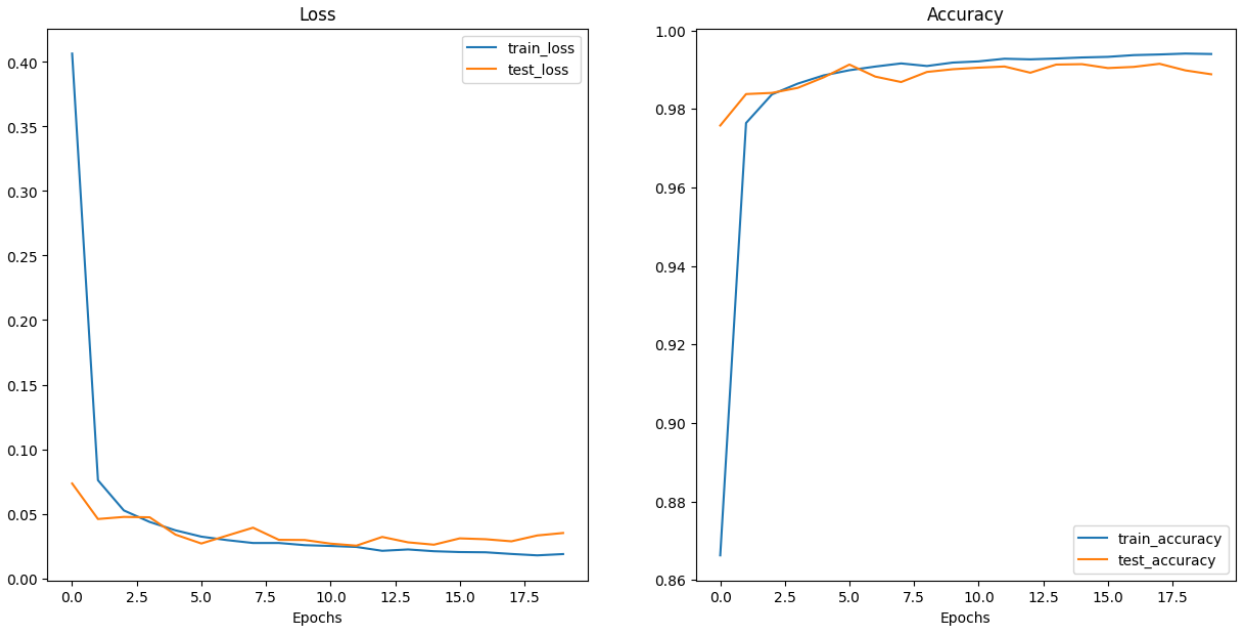
current exp / total: 3 / 8
Training with: lr: 0.001, betas: (0.9, 0.999), eps: 1e-08, weight_decay: 0.001

LOSS & Accuracy Curves

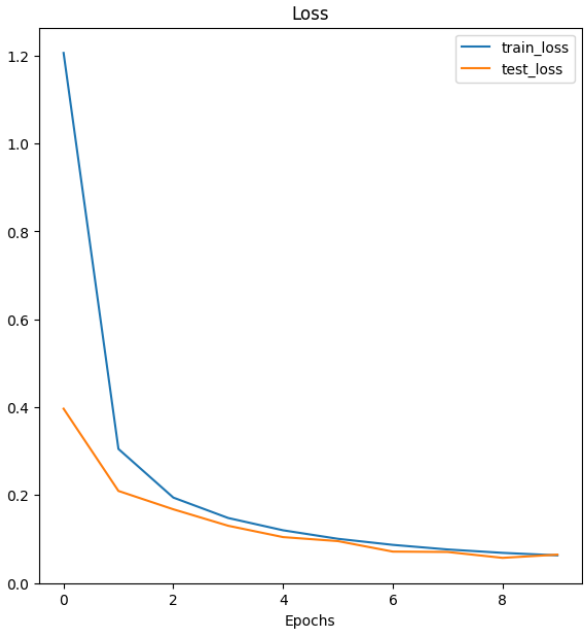


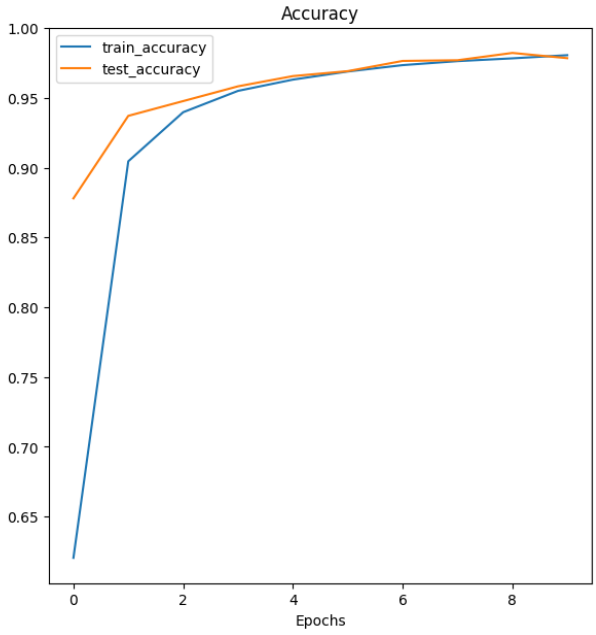
current exp / total: 4 / 8
Training with: lr: 0.001, betas: (0.9, 0.999), eps: 1e-08, weight_decay: 0.001

LOSS & Accuracy Curves

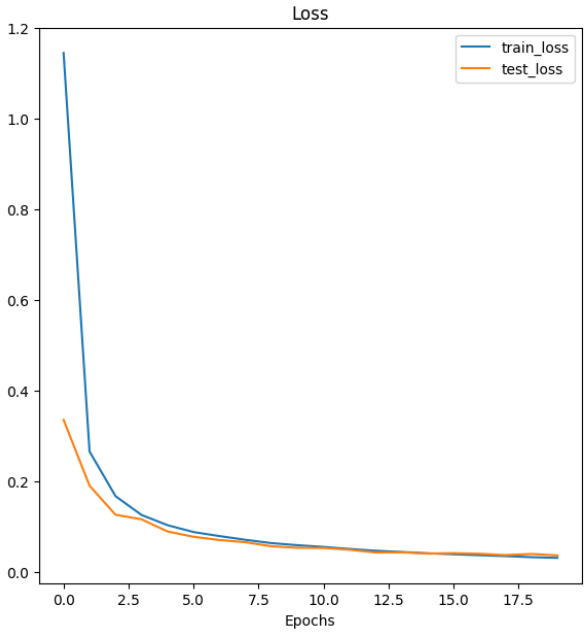


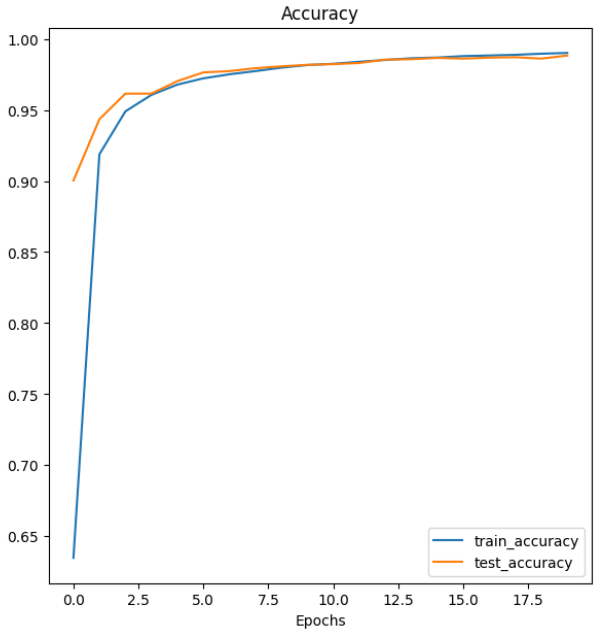
current exp / total: 5 / 8
Training with: lr: 0.0001, betas: (0.8, 0.888), eps: 1e-08, weight_decay: 0.001
LOSS & Accuracy Curves



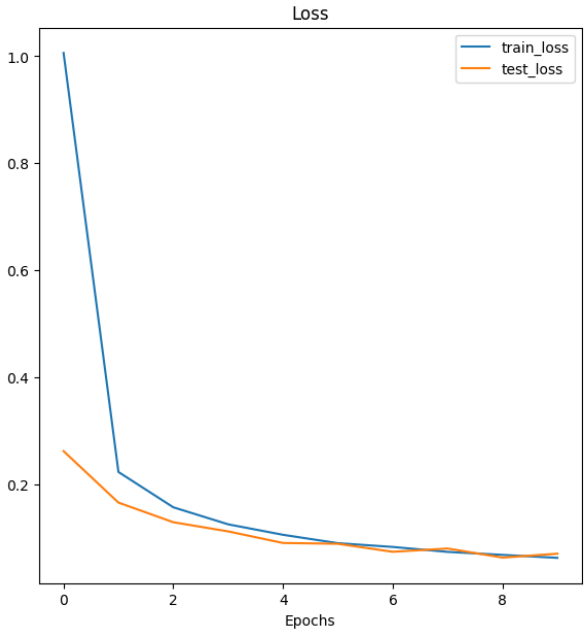


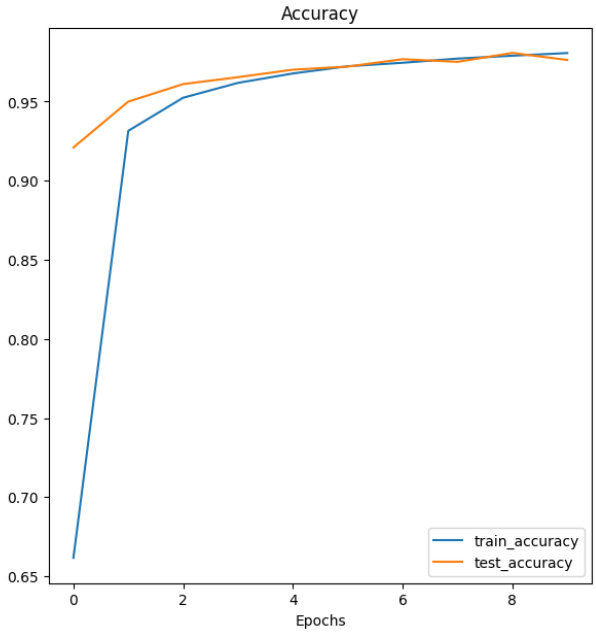
current exp / total: 6 / 8
Training with: lr: 0.0001, betas: (0.8, 0.888), eps: 1e-08, weight_decay: 0.001
LOSS & Accuracy Curves



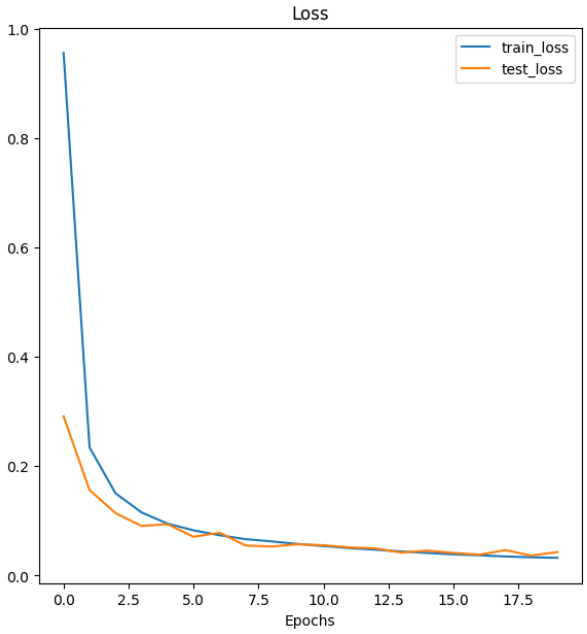


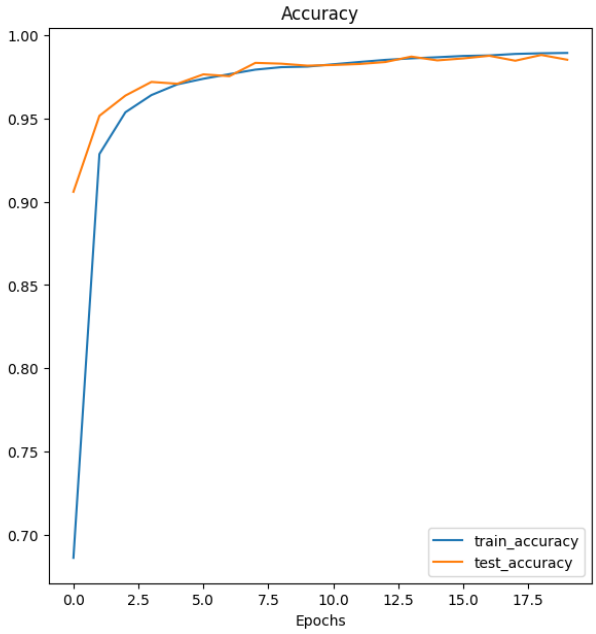
current exp / total: 7 / 8
Training with: lr: 0.0001, betas: (0.9, 0.999), eps: 1e-08, weight_decay: 0.001
LOSS & Accuracy Curves





current exp / total: 8 / 8
Training with: lr: 0.0001, betas: (0.9, 0.999), eps: 1e-08, weight_decay: 0.001
LOSS & Accuracy Curves



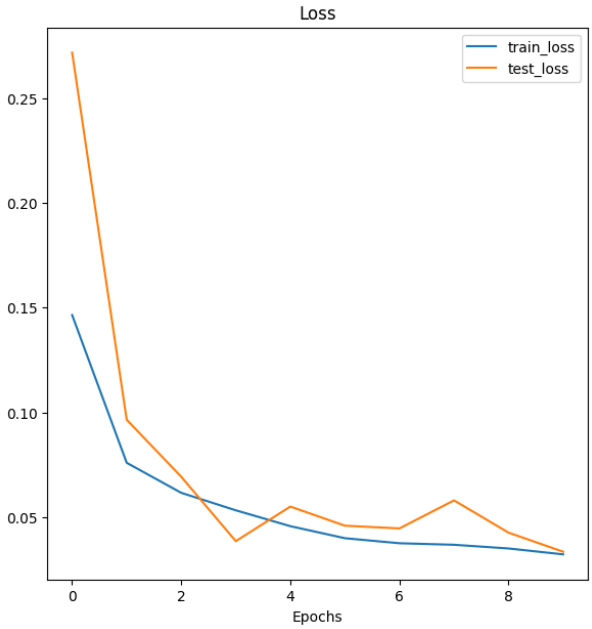


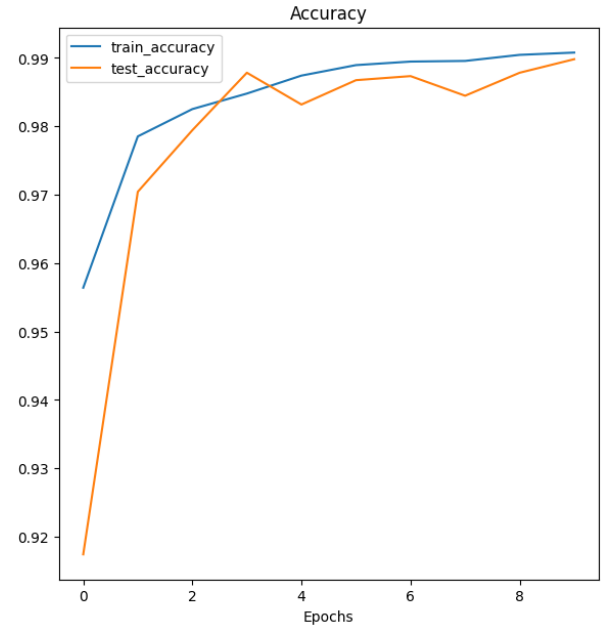
Training Curves for ResNet-18 Architecture:

current exp / total: 1 / 8

Training with: lr: 0.001, betas: (0.8, 0.888), eps: 1e-08, weight_decay: 0.001

LOSS & Accuracy Curves

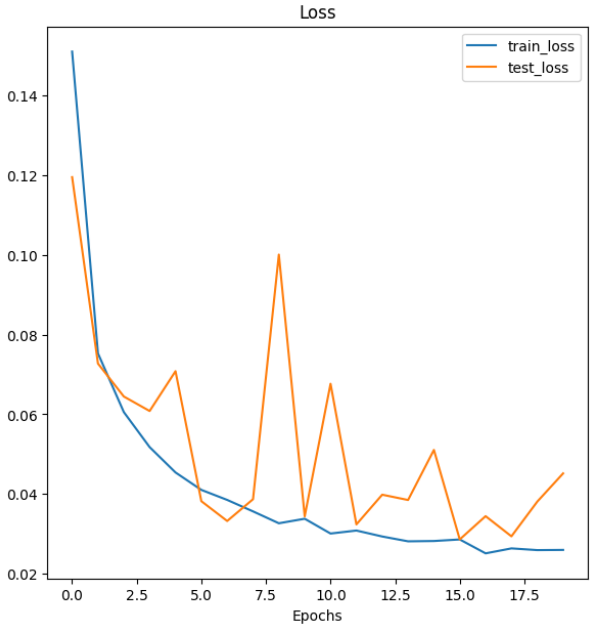


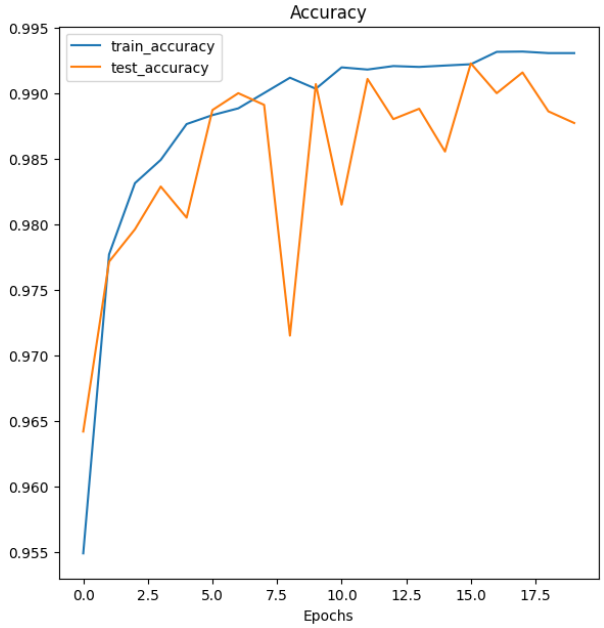


current exp / total: 2 / 8

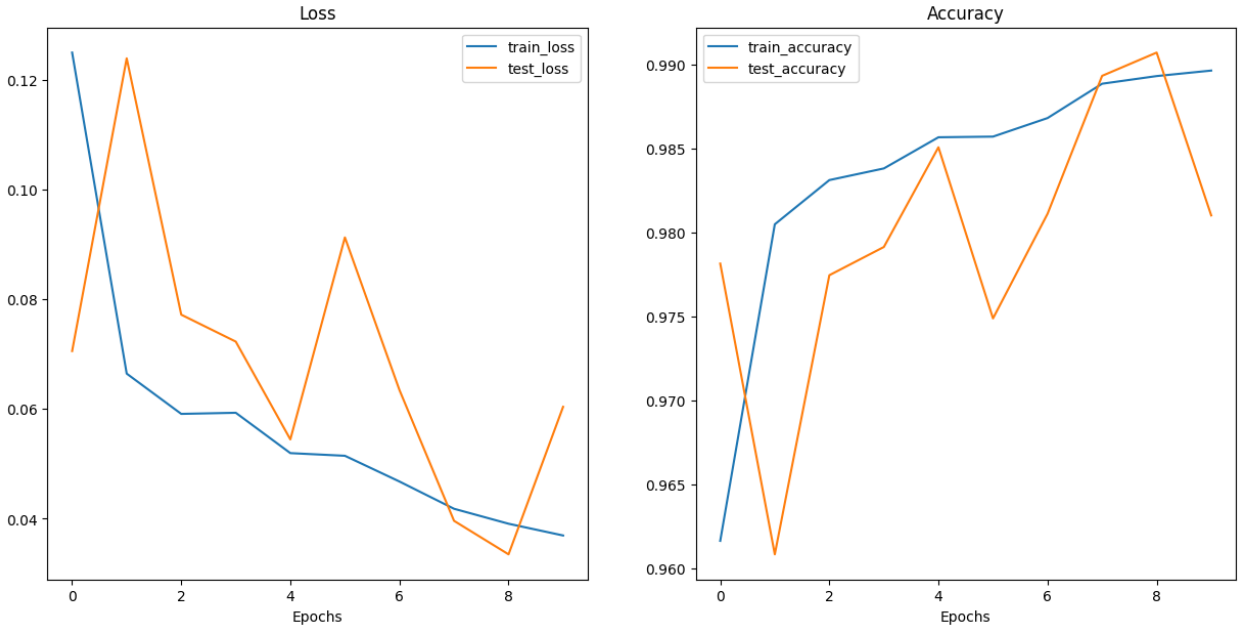
Training with: lr: 0.001, betas: (0.8, 0.888), eps: 1e-08, weight_decay: 0.001

LOSS & Accuracy Curves

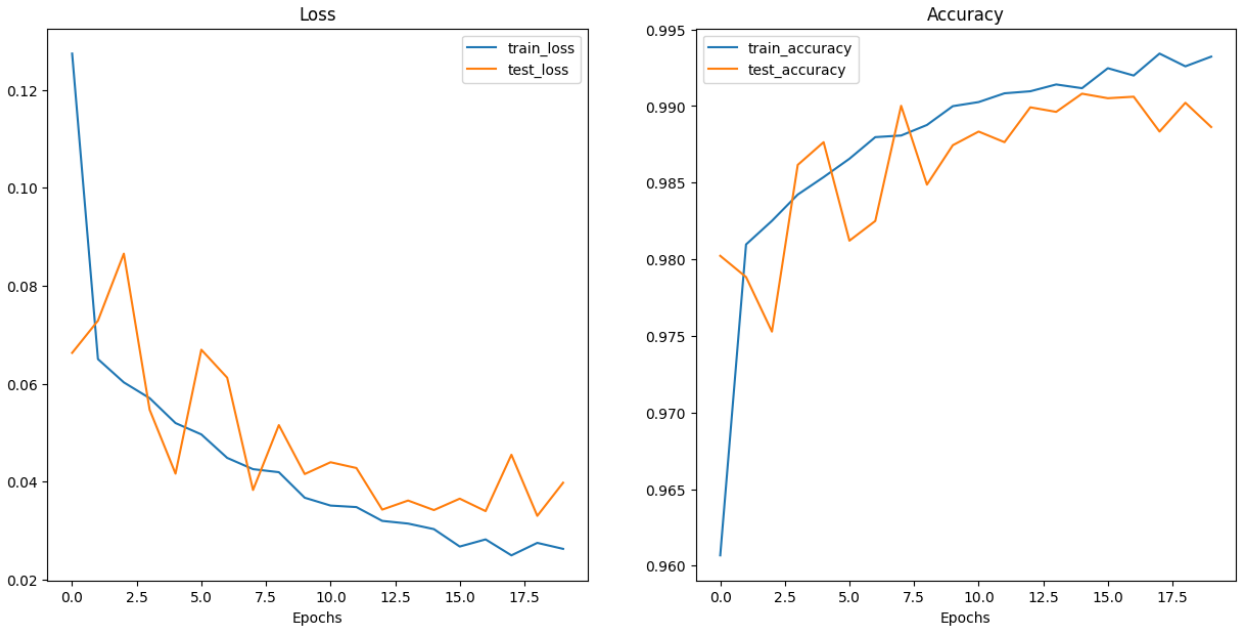




current exp / total: 3 / 8
Training with: lr: 0.001, betas: (0.9, 0.999), eps: 1e-08, weight_decay: 0.001
LOSS & Accuracy Curves

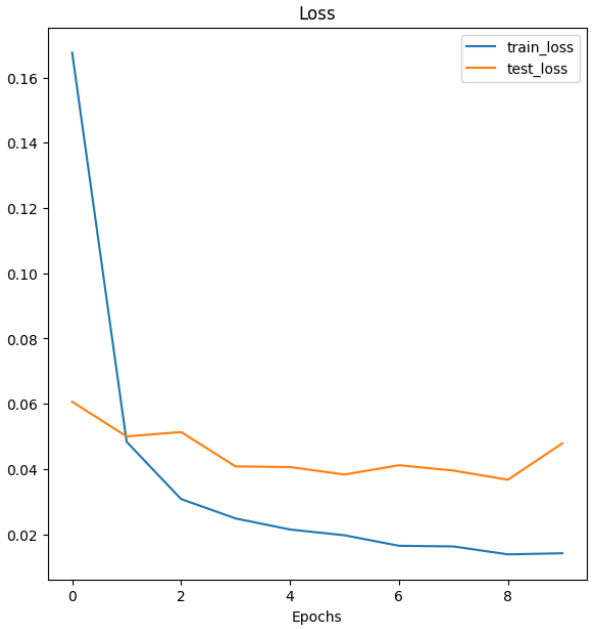


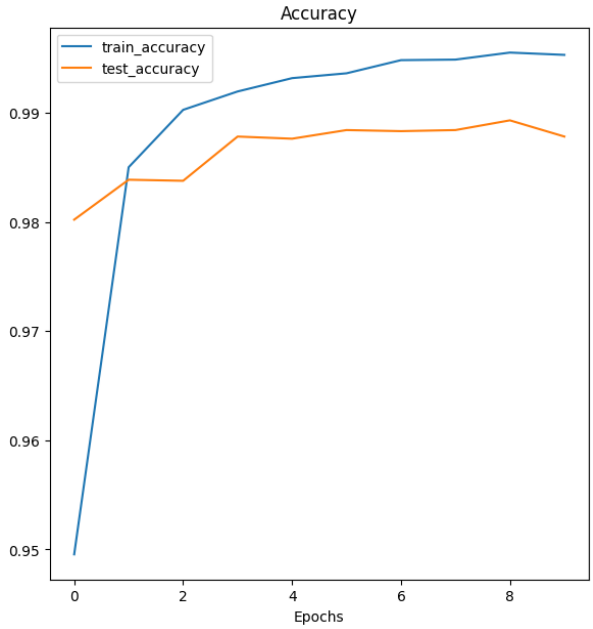
current exp / total: 4 / 8
Training with: lr: 0.001, betas: (0.9, 0.999), eps: 1e-08, weight_decay: 0.001
LOSS & Accuracy Curves



current exp / total: 5 / 8
Training with: lr: 0.0001, betas: (0.8, 0.888), eps: 1e-08, weight_decay: 0.001

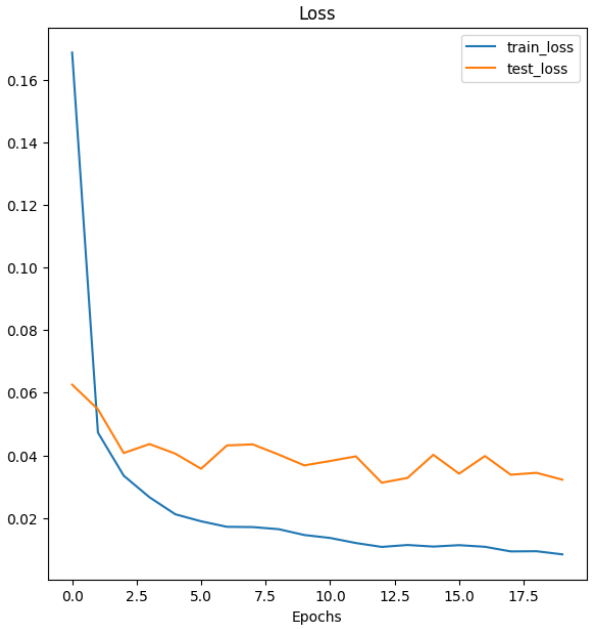
LOSS & Accuracy Curves

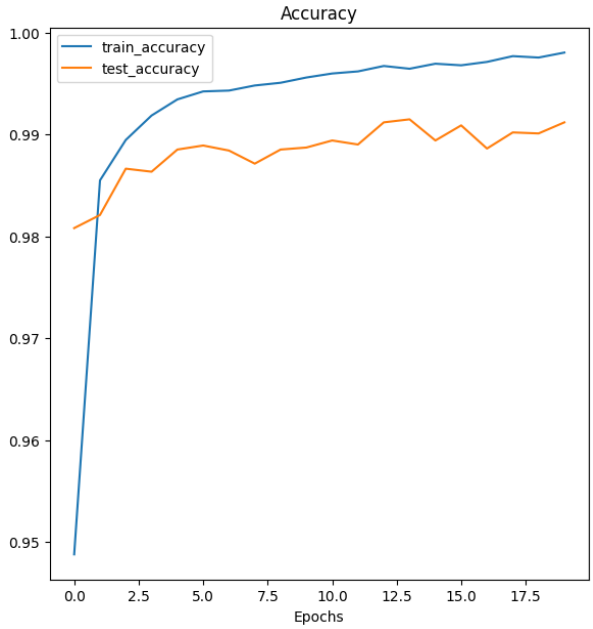




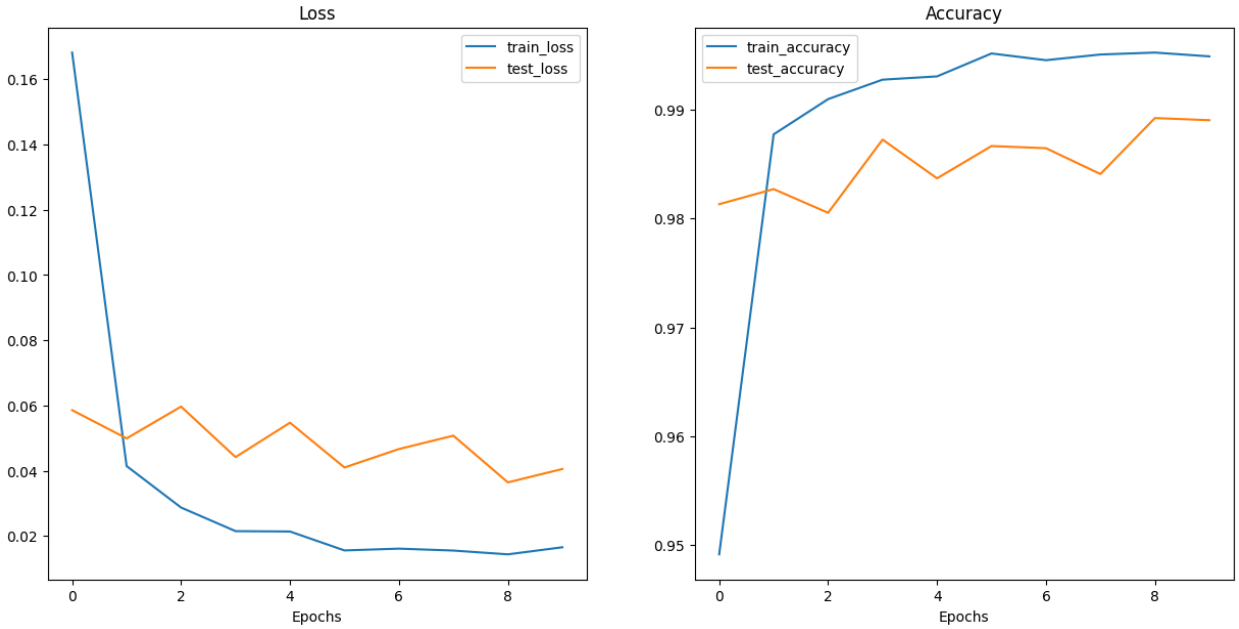
current exp / total: 6 / 8
Training with: lr: 0.0001, betas: (0.8, 0.888), eps: 1e-08, weight_decay: 0.001

LOSS & Accuracy Curves

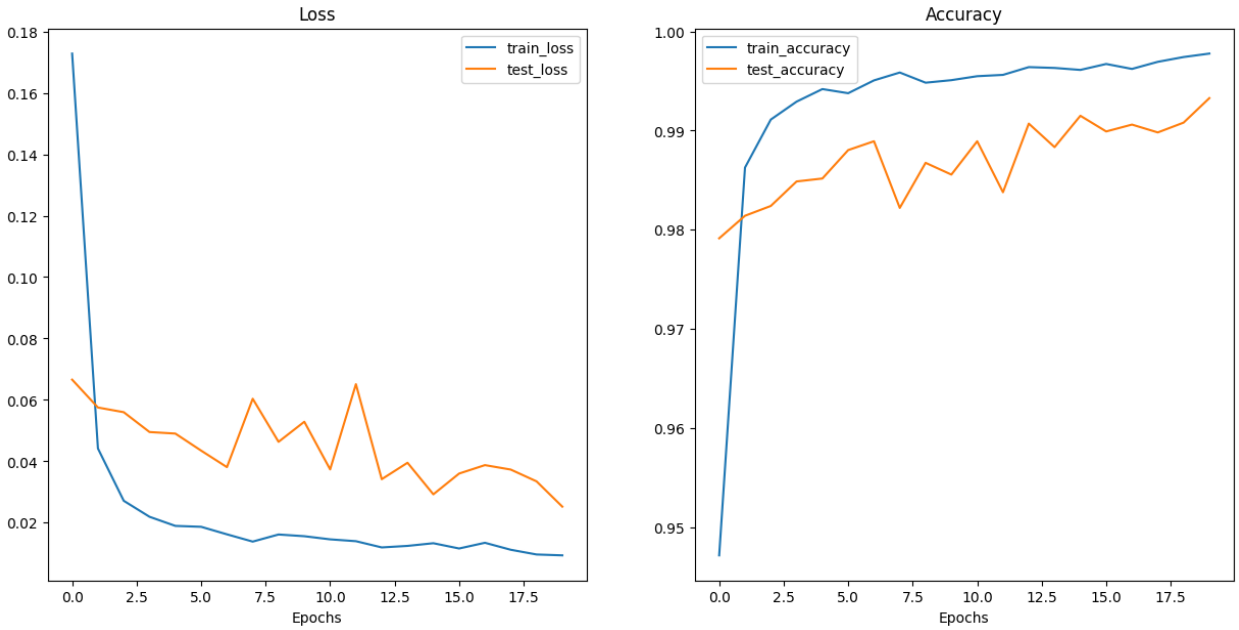




current exp / total: 7 / 8
Training with: lr: 0.0001, betas: (0.9, 0.999), eps: 1e-08, weight_decay: 0.001
LOSS & Accuracy Curves



current exp / total: 8 / 8
Training with: lr: 0.0001, betas: (0.9, 0.999), eps: 1e-08, weight_decay: 0.001
LOSS & Accuracy Curves



Confusion Matrix for Custom CNN Architecture:

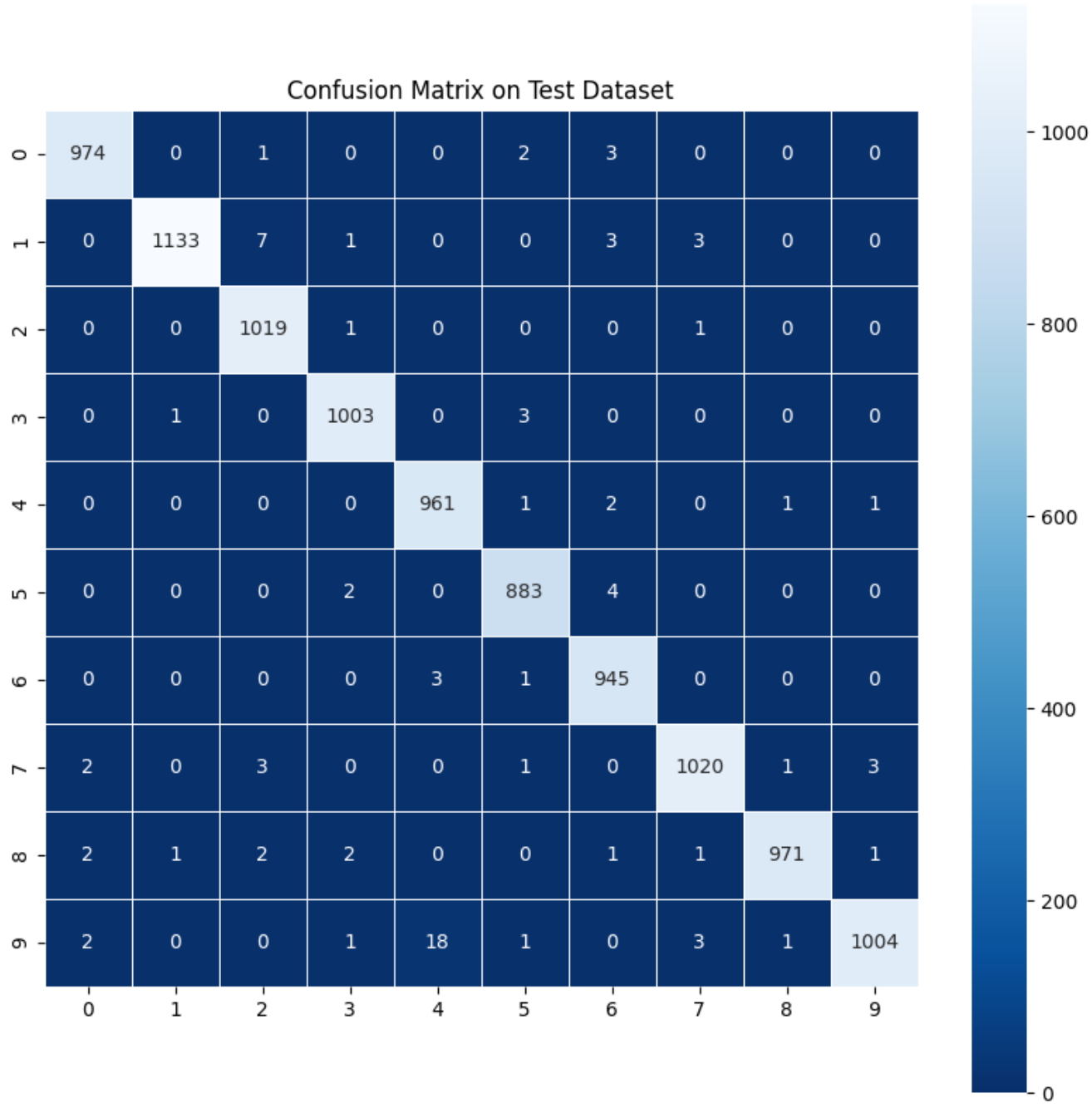
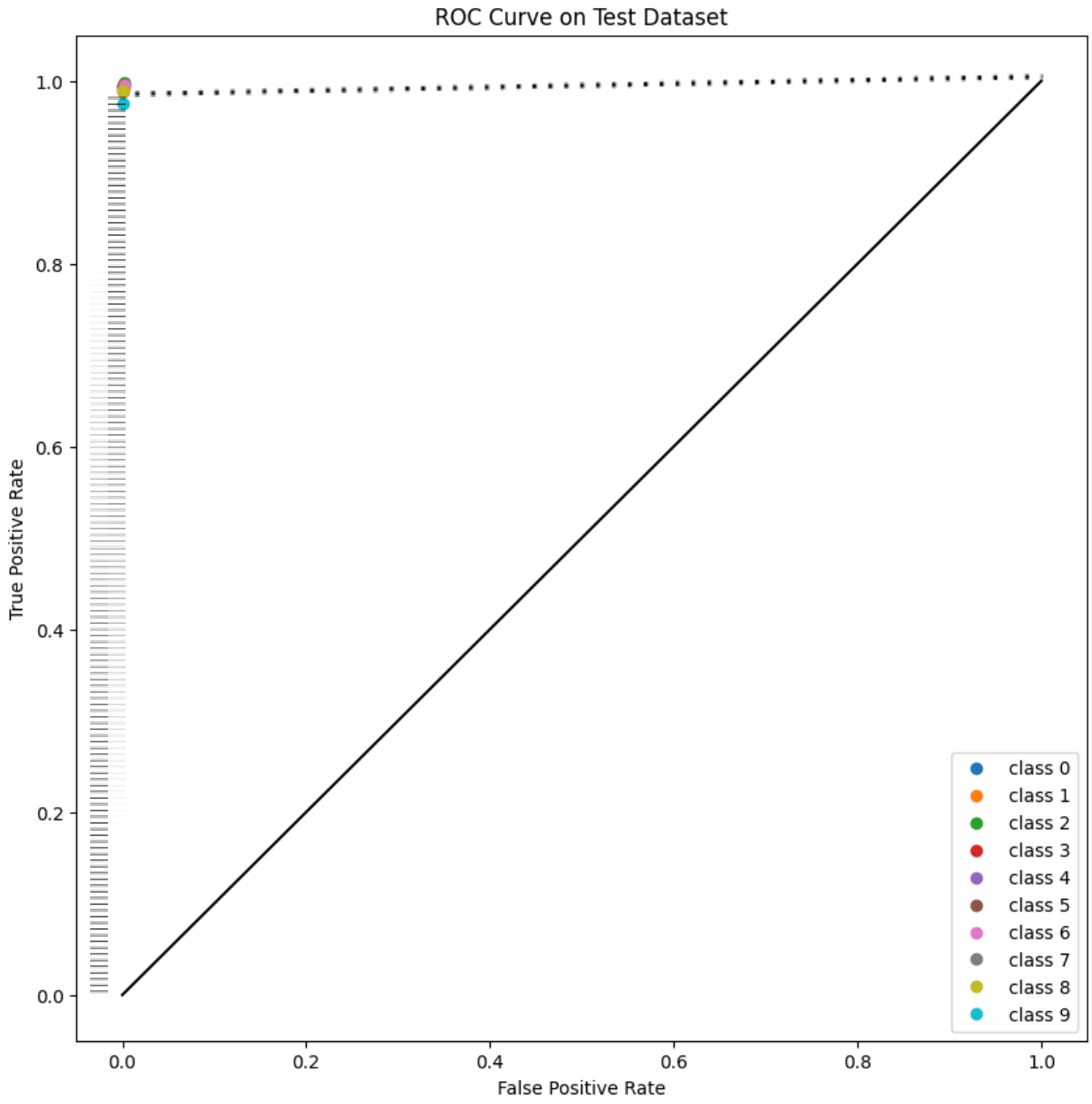


Fig: Confusion matrix for Custom CNN Architecture

ROC for Custom CNN Architecture:



Confusion Matrix for ResNet-18 Architecture:

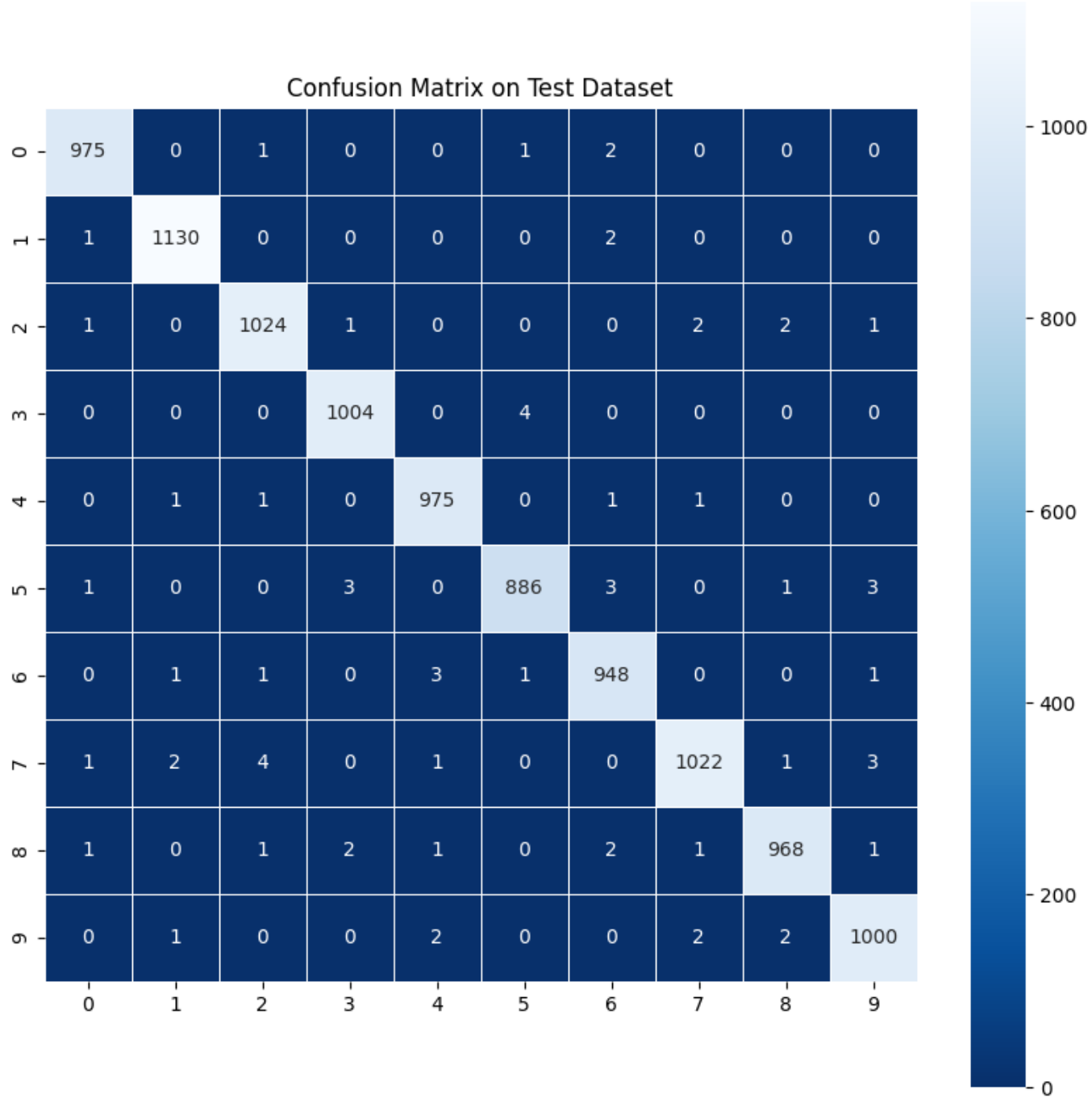
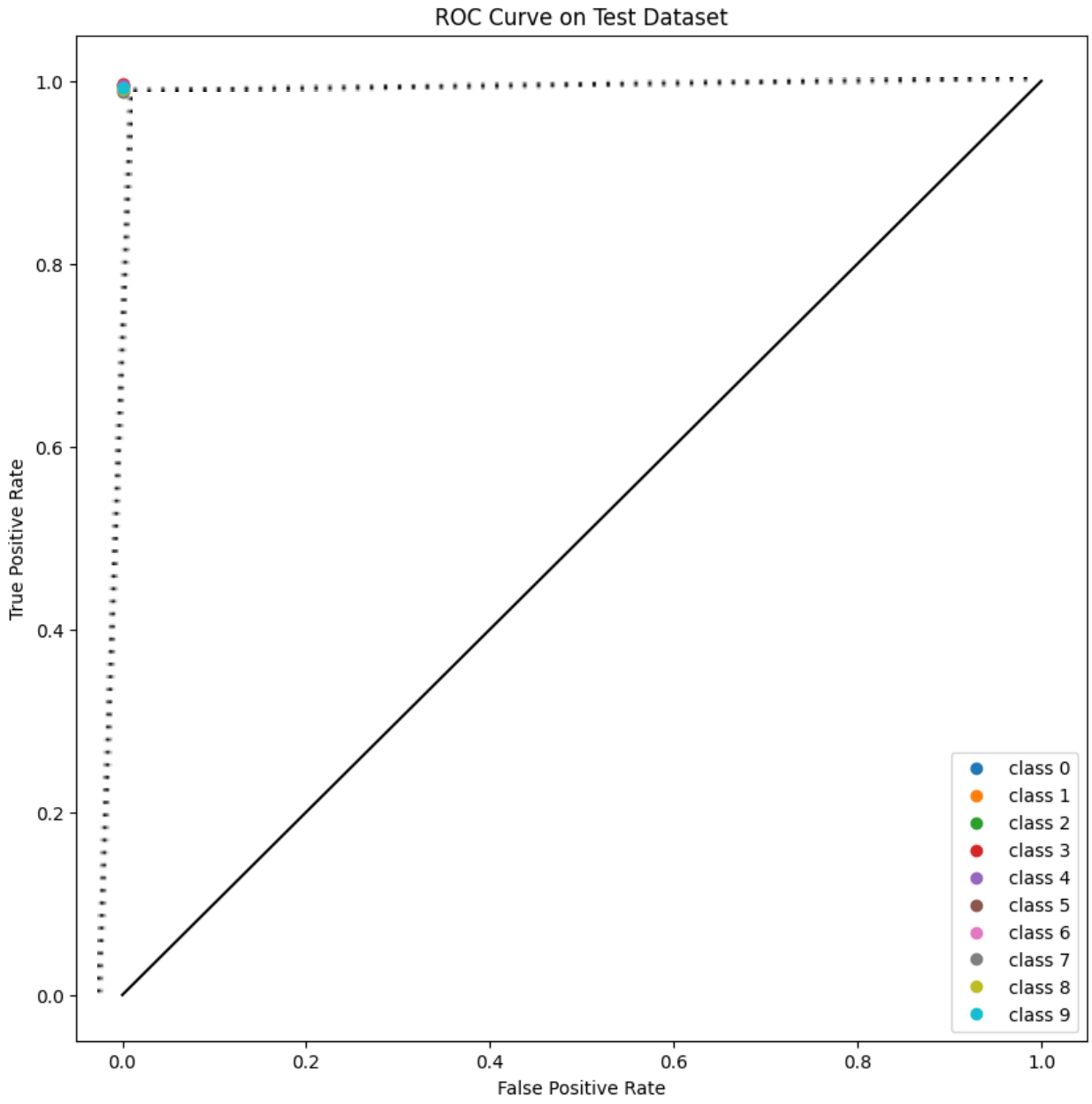


Fig: Confusion matrix for ResNet-18 Architecture

ROC for Custom ResNet-18 Architecture:



Performance of your CNN and resnet18:

- The performance of the custom CNN architecture and ResNet18 was evaluated on the MNIST dataset using the same hyperparameters for both models. The hyperparameters used were a learning rate of [0.001, 0.0001], betas of [(0.8, 0.888), (0.9, 0.999)], epsilon of 1e-08, weight decay of [0.001], and epochs of [10, 20].
- The results show that the custom CNN architecture achieved an overall accuracy of around 99.13% on the test dataset, while ResNet18 achieved an overall accuracy of around 99.32%. The confusion matrix for both models indicates that both models have high accuracy in most of the classes, with a few exceptions.
- From the training and validation loss and accuracy curves, it can be observed that both models achieved high accuracy and low loss on both the training and validation sets, indicating that there is no overfitting in either model.
- Overall, ResNet18 performed slightly better than the custom CNN architecture, but both models achieved high accuracy on the MNIST dataset. However, it is important to note that ResNet18 is a deeper and more complex model, which may not be necessary for simpler tasks such as MNIST classification. The choice of the model should depend on the specific task and the resources available.

Objectives:

Question 2.

- **Download the Flicker8k dataset [images, captions]. Implement an encoder-decoder architecture for Image Captioning. For the encoder and decoder, you can use resnet/densnet/VGG and LSTM/RNN/GRU respectivly. Perform the following tasks:**
  - **Split the dataset into train and test sets appropriately. You can further split the train set for validation. Train your model on the train set. Report loss curve during training.**
  - **Choose an existing evaluation metric or propose your metric to evaluate your model.**
- **Specify the reason behind your selection/proposal of the metric. Report the final results on the test set.**

Procedure:

- The first block of code downloads and unzips the Flicker8k_Dataset.zip file to the data directory if it does not already exist.
- The second block of code downloads and unzips the Flickr8k_text.zip file to the data/Flickr8k_text directory if it does not already exist.
- The code then creates a CSV file train_data.csv containing image paths, image names, and captions.
- It reads the captions from the file data/Flickr8k_text/Flickr8k.token.txt and retrieves the image names and captions for each image. If the image cannot be opened, it is skipped.
- Finally, it creates a Pandas DataFrame with the image paths, names, and captions and saves it as a CSV file data/Flickr8k_text/train_data.csv. If the file already exists, it reads the CSV file into a DataFrame.
- Moving on import necessary packages including PyTorch and its associated packages, spacy, pandas, and PIL.
- Defines a Vocabulary class for converting text to numerical values. It takes in a frequency threshold as an argument, which is used to build the vocabulary based on the words in the input text. It also includes methods for tokenizing the text and numericalizing the text.
- Defines a FlickrDataset class that inherits from PyTorch's Dataset class. It takes in the root directory, captions file, and transform as arguments. It loads the captions file using pandas, initializes a Vocabulary object, and builds the vocabulary. It also defines methods for getting the length of the dataset, getting an item from the dataset, and numericalizing the captions.
- Defines a MyCollate class for use in the DataLoader. It takes in a pad index as an argument and is used to pad the captions so that they are of the same length.
- Defines a get_loader function that takes in the root folder, annotation file, transform, batch size, number of workers, shuffle, and pin memory as arguments. It creates a FlickrDataset object, gets the pad index from the Vocabulary object, and creates a DataLoader object using the dataset, batch size, number of workers, shuffle, pin memory, and MyCollate object as arguments. It returns the DataLoader and dataset objects.

- Defining an image transformation pipeline using transforms.Compose(), which includes resizing the image to (224, 224) and converting the image to a PyTorch tensor.
- The code loads the dataset using the get_loader() function from an external module, passing in the path to the image directory and the path to the CSV file containing the captions.
- Randomly spliting the dataset into training and testing sets using torch.utils.data.random_split(), with a split ratio of 0.8 and a random seed of 42.
- Creating data loaders for the training and testing sets using torch.utils.data.DataLoader(), setting the batch size to 32 and shuffling the training set.
- Printing out the sizes of the training and testing sets and their corresponding data loaders.
- Using Matplotlib to visualize the first 9 images and their captions in the training set.
- The code defines the neural network models for image captioning: EncoderCNN, DecoderRNN, and CNNtoRNN.
- EncoderCNN is a class that inherits from nn.Module and defines the convolutional neural network for extracting image features. It uses a pre-trained ResNet-18 model and replaces its fully connected layer with a linear layer with embed_size output features. It applies the ReLU activation function and a dropout layer with a dropout probability of 0.5 to the output.
- DecoderRNN is a class that inherits from nn.Module and defines the recurrent neural network for generating captions. It uses an embedding layer to embed the word indices in the caption, a multi-layer LSTM to generate hidden states based on the embeddings, and a linear layer to predict the word indices of the next word in the caption. It applies a dropout layer with a dropout probability of 0.5 to the embeddings.
- CNNtoRNN is a class that inherits from nn.Module and combines EncoderCNN and DecoderRNN to create the full image captioning model. It takes an image and a caption as input and generates a sequence of word indices as output.
- The forward() method of CNNtoRNN passes the image through the EncoderCNN to obtain image features, and then passes the image features and the caption through the DecoderRNN to generate the sequence of word indices.
- The caption_image() method of CNNtoRNN generates a caption for a given image using beam search. It takes an image, a vocabulary, and a maximum caption length as input, and returns a list of words representing the generated caption.
- EPOCHS = 100: The number of times the model will loop through the entire dataset during training.
- model = CNNtoRNN(embed_size=512, hidden_size=512, vocab_size=len(dataset.vocab), num_layers=1).to(device): This line initializes a CNNtoRNN model, which takes in an image and produces a caption for the image. The model has an embedding size of 512, hidden size of 512, and one layer. The to(device) part specifies that the model should be moved to the GPU if available, or else run on the CPU.
- loss_fn = torch.nn.CrossEntropyLoss(): This line initializes the loss function that will be used during training. The Cross-Entropy loss is commonly used for classification

problems, and it measures the difference between the predicted probabilities and the actual class labels.

- optimizer = torch.optim.Adam(params=model.parameters(), lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0): This line initializes the optimizer that will be used during training. The Adam optimizer is used here, with a learning rate of 0.001, betas of 0.9 and 0.999, and epsilon of 1e-08. The params=model.parameters() specifies which parameters in the model should be updated during training.
- The following code block contains a loop that trains the model for the specified number of epochs:
    - a. for epoch in tqdm(range(EPOCHS)):: This line starts a loop that will run for the specified number of epochs.
    - b. for idx, (imgs, captions) in tqdm(enumerate(loader), total=len(loader), leave=True):: This line starts a loop that will iterate over the dataset. loader here is the PyTorch DataLoader object that loads the data in batches. imgs and captions are the images and corresponding captions for each batch.
    - c. imgs = imgs.to(device): This line moves the image tensors to the GPU if available, or else keeps them on the CPU.
    - d. captions = captions.to(device): This line moves the caption tensors to the GPU if available, or else keeps them on the CPU.
    - e. outputs = model(imgs, captions[:-1]): This line passes the image and caption tensors through the model to generate the predicted captions. captions[:-1] is the input to the RNN, which excludes the last token of the caption, since that will be predicted by the model.
    - f. loss = loss_fn(outputs.reshape(-1, outputs.shape[2]), captions.reshape(-1)): This line calculates the loss between the predicted captions and the actual captions. The reshape function is used to convert the output and caption tensors to two-dimensional tensors that can be compared by the loss function.
    - g. loss_ += loss.item(): This line accumulates the loss for each batch.
    - h. optimizer.zero_grad(): This line resets the gradients of the model parameters, so that they can be updated correctly during backpropagation.
    - i. loss.backward(loss): This line calculates the gradients of the loss with respect to the model parameters using backpropagation.
    - j. optimizer.step(): This line updates the model parameters based on the calculated gradients and the specified optimization algorithm.
    - k. print(f"Epoch: {epoch}, Loss: {loss_/len(loader)}"): This line prints the average loss for the epoch.
- The for loop iterates over the range of epochs specified by the EPOCHS variable using the tqdm function to display a progress bar. Within each epoch, the loop iterates over each batch of data in the loader, where each batch consists of a set of images and their corresponding captions.
- The images and captions are moved to the device (CPU or GPU) using the to method.
- The model is then used to generate predictions for the given images and captions. The captions are shifted by one position and passed as input to the model, which generates

a sequence of output vectors of shape (batch_size, seq_len, vocab_size), where seq_len is the length of the sequence.

- The loss_fn is then used to compute the loss between the predicted output and the actual captions. The reshape method is used to flatten the predicted output and actual captions into 2D matrices of shape (batch_size*seq_len, vocab_size) and (batch_size*seq_len,) respectively.
- The loss is accumulated over all the batches in the epoch, and the optimizer is used to perform backpropagation and update the model parameters.
- At the end of each epoch, the average loss for the epoch is printed, and the loss is appended to the loss_list.
- Finally, the model is saved to a file called "image_captioning_model.pth" using the torch.save function.
- The loss_list is plotted using the plt.plot function to visualize the training loss over the epochs.
- Loading and inference
- model = CNNtoRNN(embed_size=512, hidden_size=512, vocab_size=len(dataset.vocab), num_layers=1).to(device): Initializes the model with embed_size of 512, hidden_size of 512, vocab_size equal to the length of the dataset vocabulary, and 1 layer. The model is then moved to the specified device.
- model.load_state_dict(torch.load("image_captioning_model.pth")): Loads the saved state of the trained model from the file image_captioning_model.pth.
- model.eval(): Sets the model to evaluation mode. This is necessary because there are some layers that behave differently during training and testing, such as dropout and batch normalization.
- evaluate(model, loader, dataset.vocab, device): Calls the evaluate() function to calculate the loss and perplexity of the model on the test dataset.
- from nltk.translate.bleu_score import sentence_bleu: Imports the sentence_bleu() function from the NLTK library, which is used to calculate the BLEU score of the model's predictions.
- for _ in range(5):: Loops over 5 random examples from the test dataset.
    - img, caption = test_data[idx]: Gets the image and caption corresponding to the current index.
    - img = img.to(device): Moves the image tensor to the specified device.
    - plt.imshow(img.cpu().permute(1, 2, 0)): Displays the image using matplotlib.
    - print("Actual:", " ".join([dataset.vocab.itos[idx] for idx in caption.tolist()[1:-1]])): Prints the actual caption for the image.
    - print("Prediction:", " ".join(model.caption_image(img, dataset.vocab))): Generates the predicted caption using the caption_image() method of the CNNtoRNN model and prints it. The caption_image() method takes the image tensor and the dataset vocabulary as inputs and returns the predicted caption as a string.
- Results:

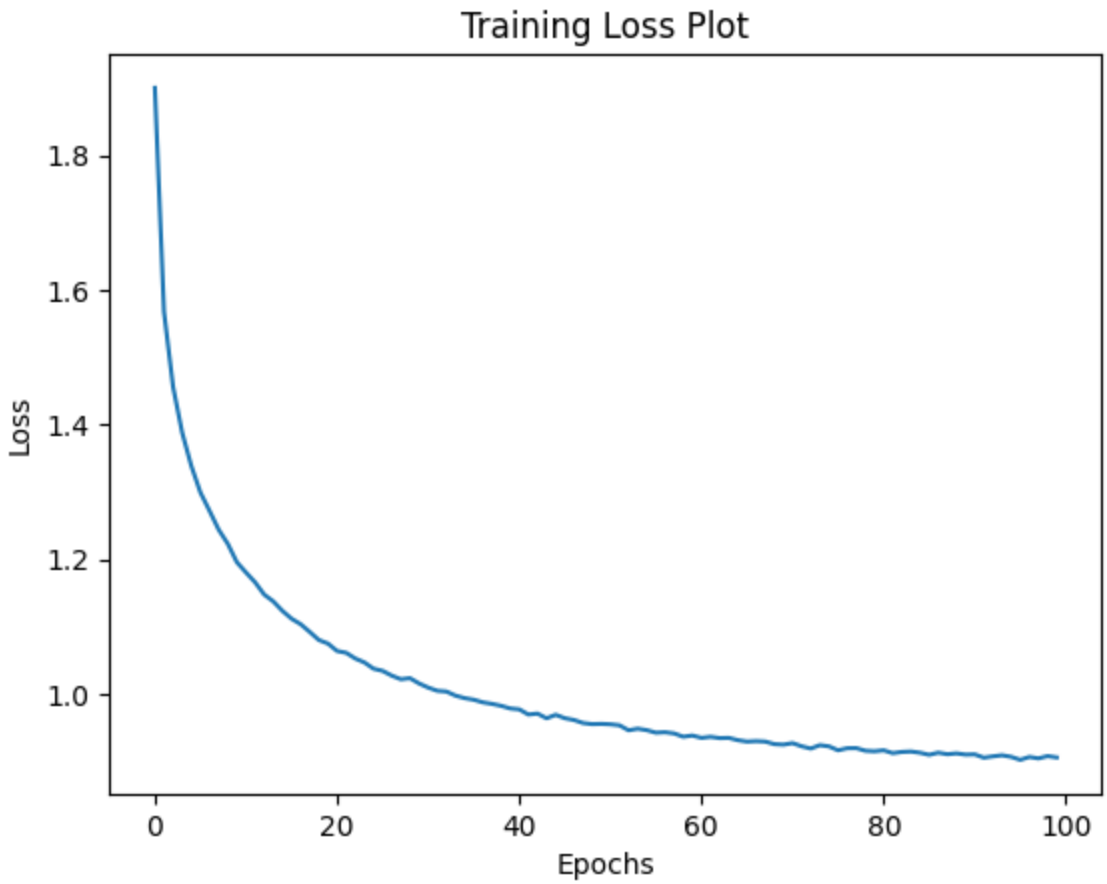


Fig: Loss Curve for training the CNNtoRNN Model

Loss: 0.699483488506008, Perplexity: 2.0195523009469856

Perplexity:

- Perplexity is a measure of how well a language model can predict a sequence of words. In image captioning, a language model generates a caption given an image. The perplexity metric indicates how well the model predicts the words in the caption.
- A lower perplexity score indicates that the model can more accurately predict the words in the caption. Perplexity can be calculated using the cross-entropy loss, which measures the difference between the predicted and actual distributions of words.
- In image captioning, perplexity can be used as an evaluation metric to assess the quality of generated captions. Lower perplexity scores indicate that the generated captions are more accurate and have a higher likelihood of being correct.
- However, it's important to note that perplexity is not a perfect metric for evaluating the quality of generated captions. It doesn't take into account the semantic coherence or overall quality of the generated caption, and it can sometimes favor shorter, simpler captions over longer, more descriptive ones.

- Nonetheless, perplexity can be a useful tool for comparing the performance of different image captioning models and tracking their progress over time. By monitoring the perplexity score over the course of training, developers can identify when the model is overfitting or underfitting the data, and make adjustments to improve its performance.

Predictions for Random Samples:

Actual: these people are in uniform at a lacrosse type game .
Prediction: <SOS> a man in a white shirt kicked a goal shot . <EOS>

Actual: a small boy in blue crawls along a window <UNK> .
Prediction: <SOS> a little boy is jumping off a pier into a swimming pool . <EOS>

Actual: a dog running through snow .
Prediction: <SOS> a dog running through snow . <EOS>

Actual: a child seen from the waist down in a bathing suit standing on a slip and slide with <UNK> of water shooting up
Prediction: <SOS> a little girl in a bathing suit is holding a paddle in her hands . <EOS>

Actual: a yellow motorcycle drives down the road .
Prediction: <SOS> a man in a yellow helmet riding a motorcycle . <EOS>

Objectives:

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.**

Procedure:

- Dataset Prepration:
  - Upload your dataset images to the project.
  - Select the 'Bounding Box' annotation tool from the toolbar.
  - Draw a bounding box around the object of interest in the image.
  - Add a label to the bounding box that corresponds to the object class (e.g. 'person', 'car', 'tree', etc.) **in our case deers.**
  - Repeat steps 4 and 5 for all objects of interest in the image.
  - Save the annotations for the image and move on to the next one.
  - After annotating all images, export the annotations in the YOLO format (.txt files) using the 'Export' function.
  - The exported annotations will include the bounding box coordinates and object class label for each object in each image.
  - These annotations can then be used to train a YOLO object detection model on your dataset.
  - Images:

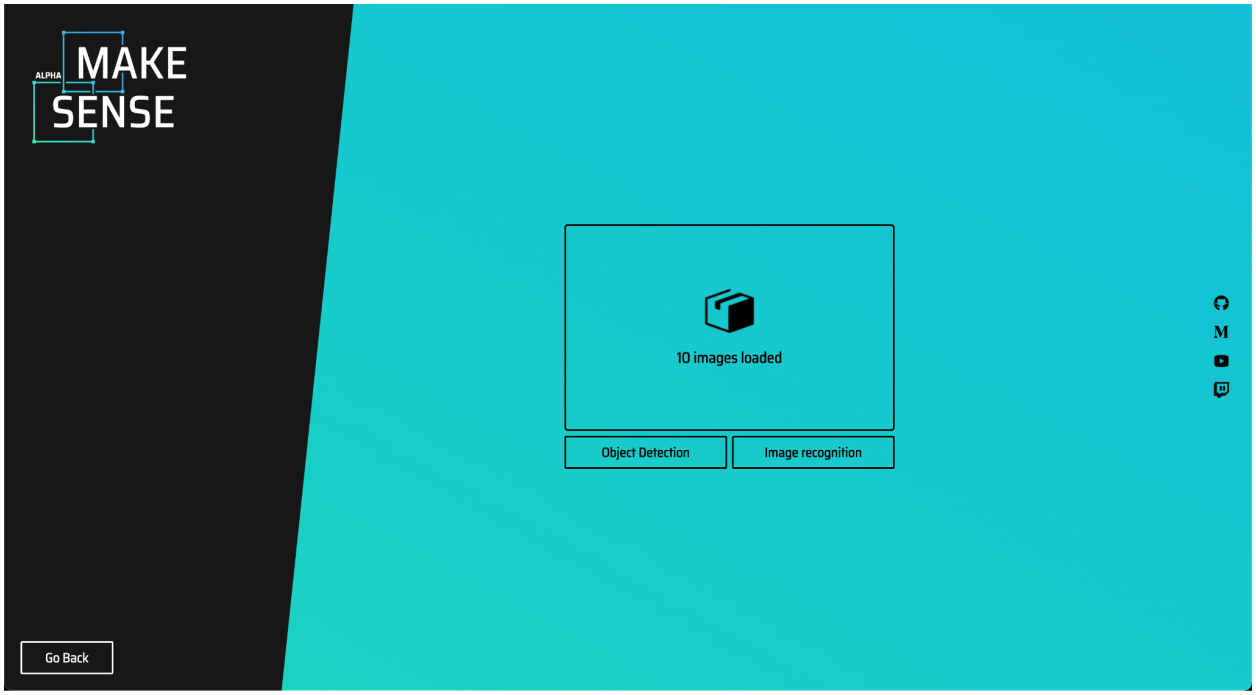


Fig: Upload the folder containing images and select object detection.

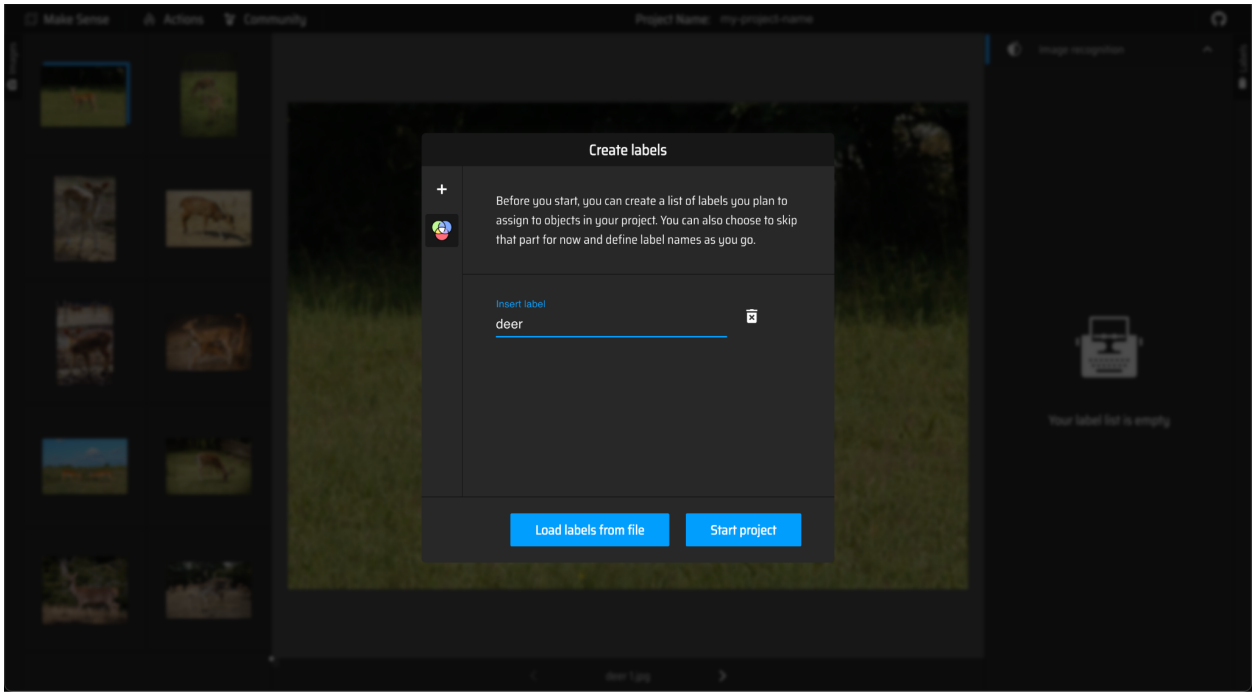


Fig: Creating Labels for Objects

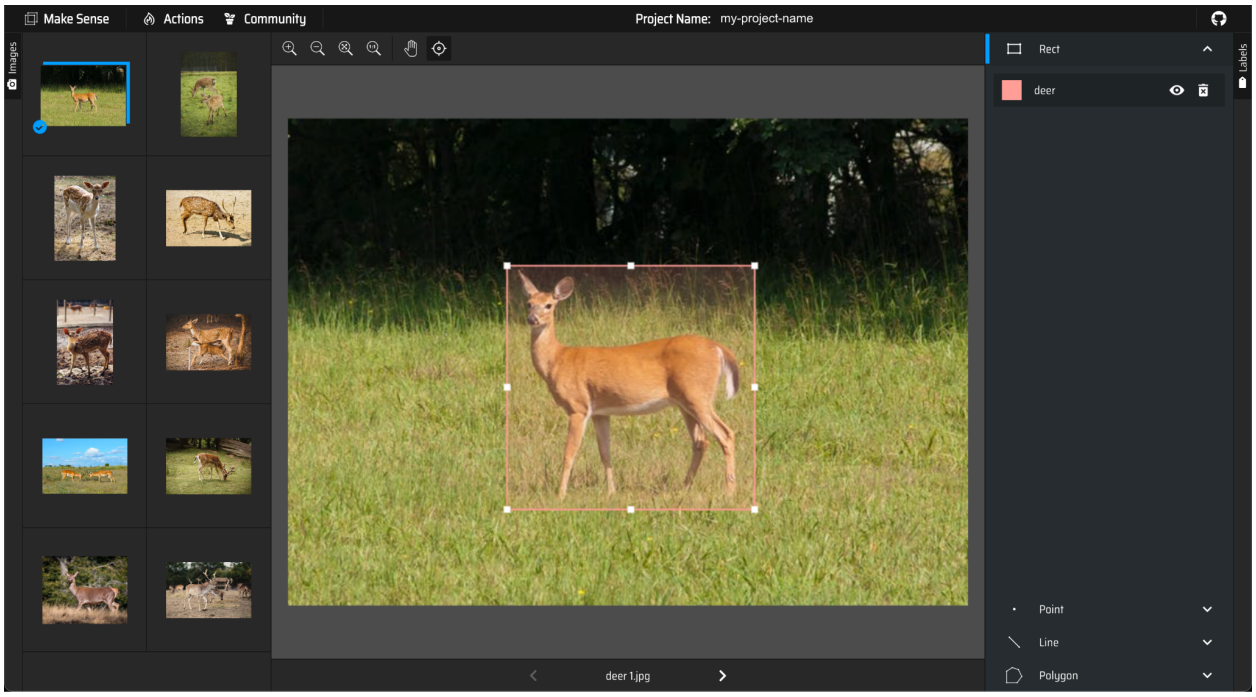


Fig: Annotating by drawing a bounding rectangle

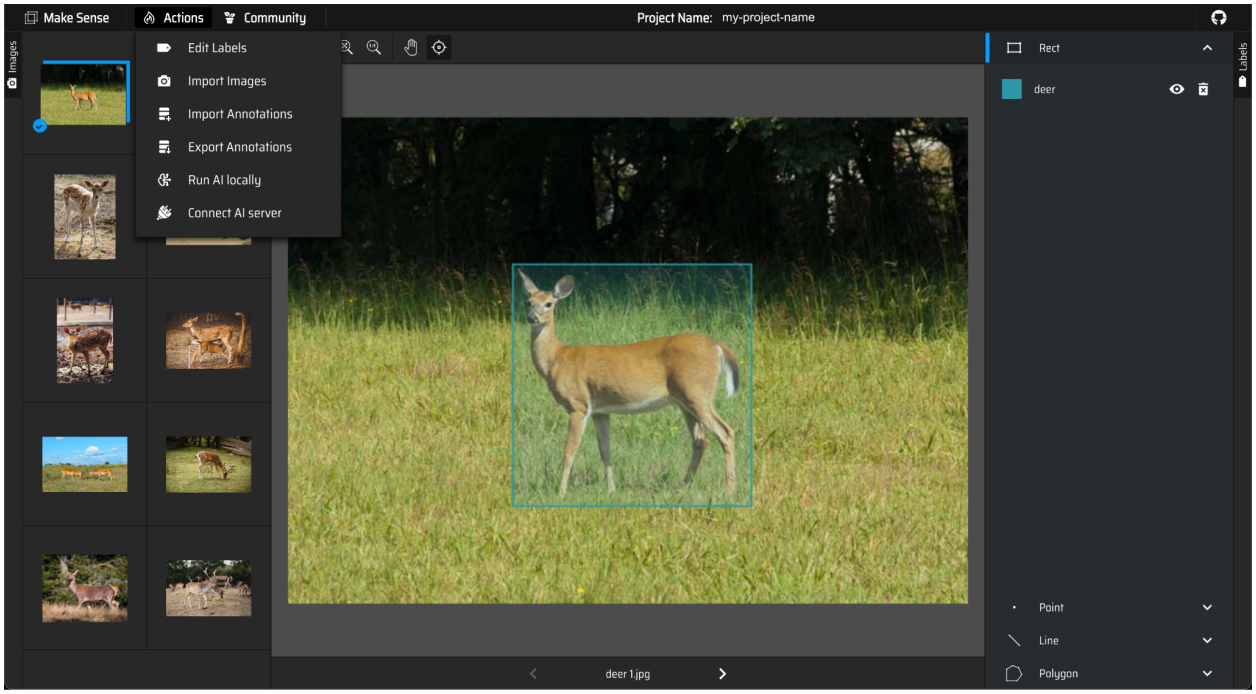


Fig: Selecting Actions to Export Annotations

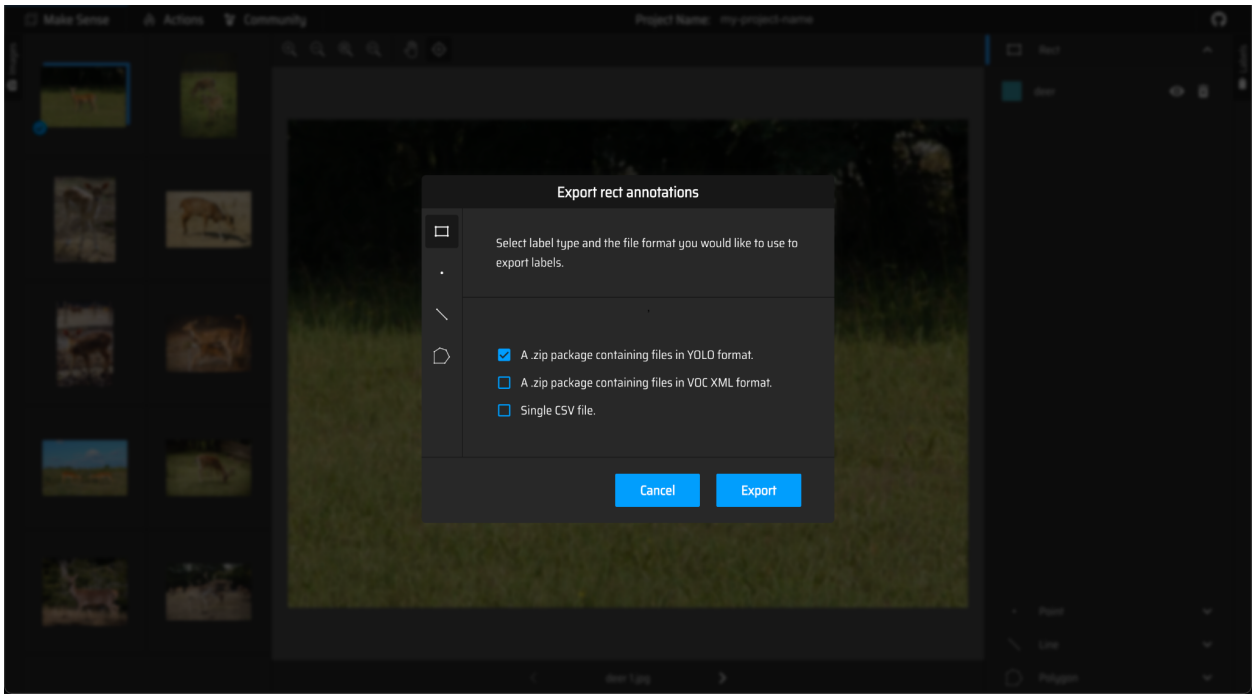


Fig: Selecting the YOLO format and export, this will download a zipped file

Training Model:

- !pip -q install -U ultralytics: This installs the ultralytics package, which provides an implementation of the YOLOv8 object detection model and related utilities.
- ultralytics.checks(): This performs some checks to ensure that the system has the necessary requirements to run YOLOv8.
- import os: This imports the os module, which provides a way to interact with the operating system.
- os.chdir(os.getcwd()): This changes the current working directory to the current directory.
- !yolo train model=yolov8n.pt data="./datasets/deer_dataset.yml" epochs=100 imgsz=640 batch=10: This trains a YOLOv8 model on the deer dataset with the following options:
- model=yolov8n.pt: This specifies the model architecture to use (yolov8n.pt is a custom model).
- data="./datasets/deer_dataset.yml": This specifies the location of the dataset configuration file.
- epochs=100: This specifies the number of training epochs.
- imgsz=640: This specifies the size of the input images (in pixels).
- batch=10: This specifies the batch size to use during training.

Results:

Fig: Ground Truth

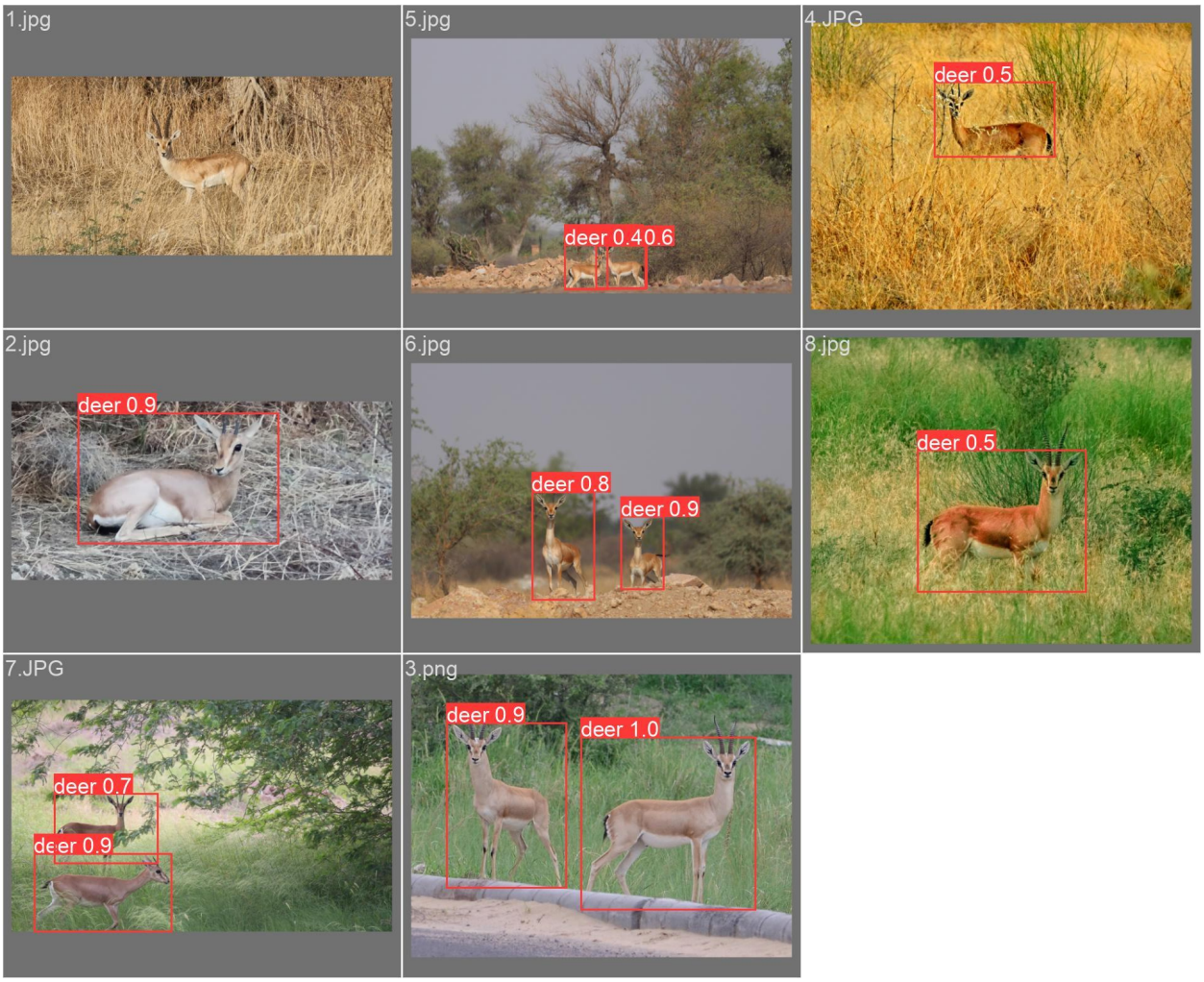


Fig: Predictions by the model with confidence

Note: For the 1st image model is unable to detect the deer.

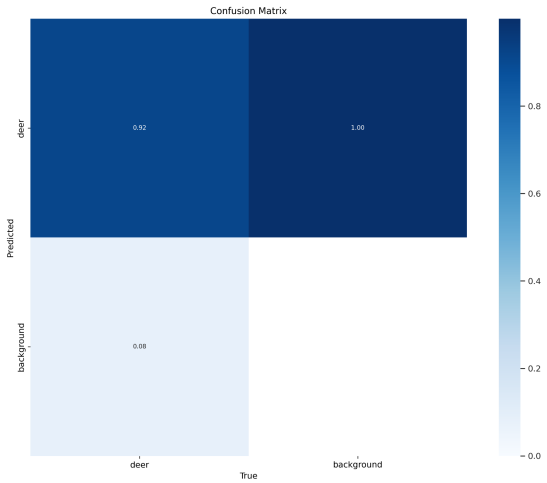


Fig: Confusion Matrix

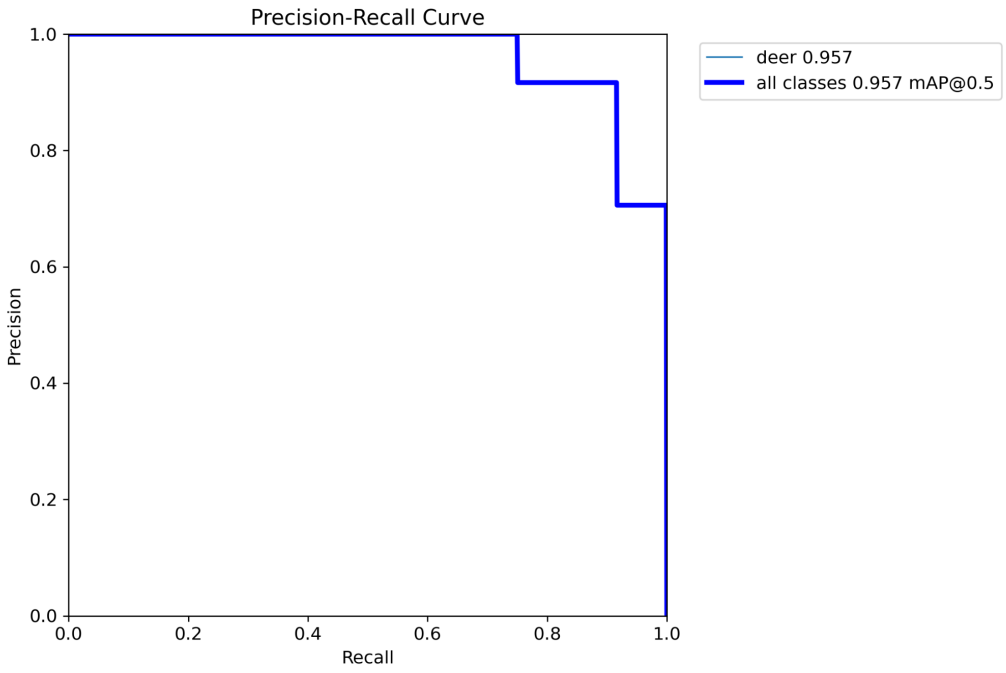


Fig: P-R Curve

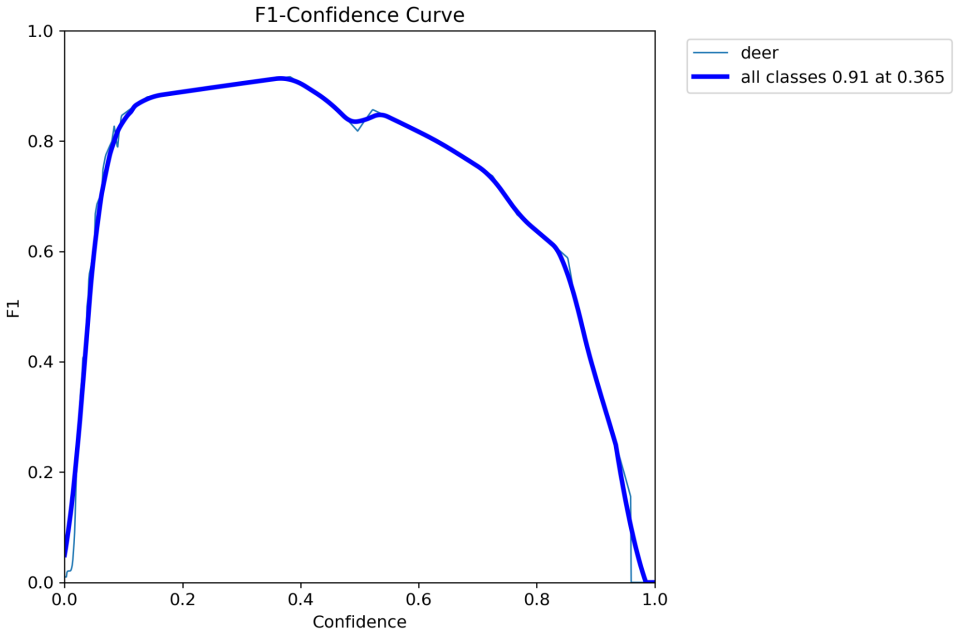


Fig: F1 Curve

HOG Detector with SVM Classification:

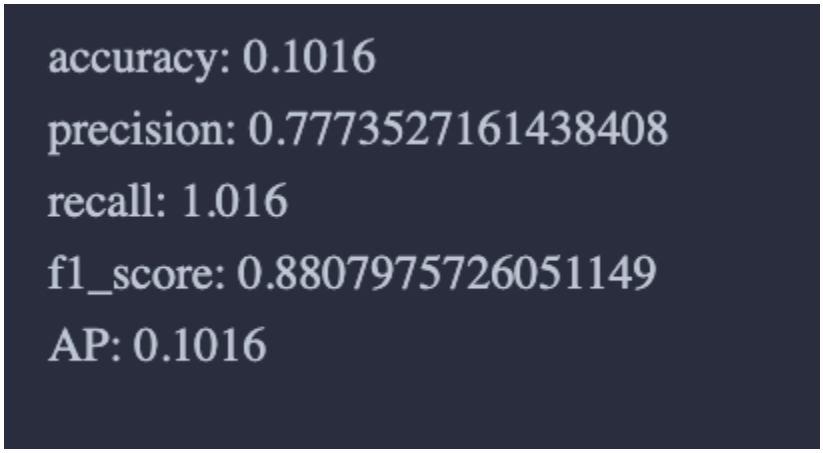


Fig: Linear SVM permormance

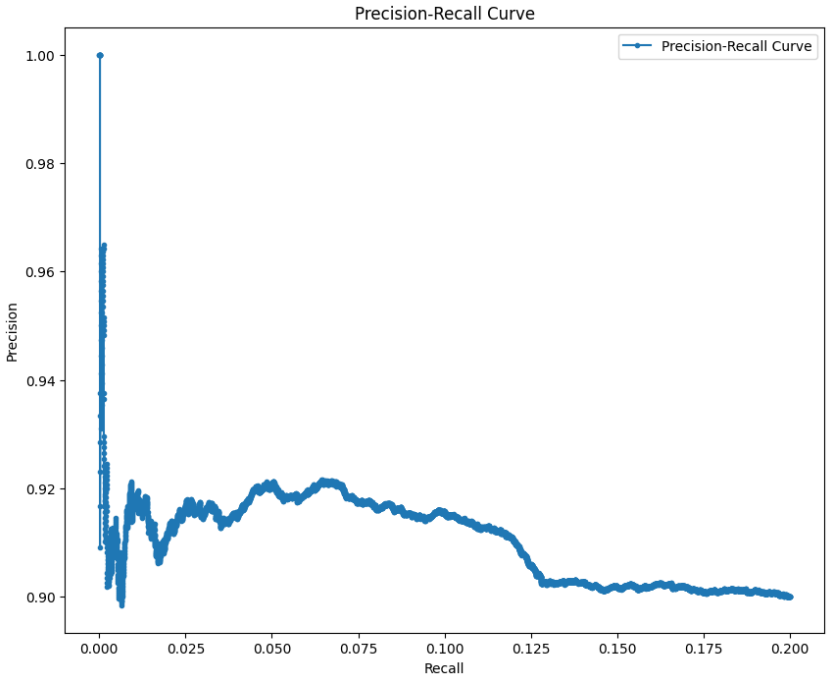


Fig: P-R Curve for Hog Detector

YOLO Detector:

- YOLO stands for "You Only Look Once".
- It is a deep learning-based object detection algorithm that is very fast and can detect objects in real-time.
- YOLO works by dividing the image into a grid and then making predictions for each cell of the grid.
- It uses a single neural network to predict the bounding boxes and class probabilities for the objects in the image.

- YOLO is very accurate and can detect multiple objects in a single image.

HOG Detector:

- HOG stands for "Histogram of Oriented Gradients".
- It is a traditional computer vision-based object detection algorithm.
- HOG works by calculating the gradient orientation and magnitude for every pixel in the image.
- It then creates a histogram of these gradients for different regions of the image.
- This histogram is then used to train a classifier that can detect objects based on the gradient information.
- HOG is slower than YOLO but can be very accurate in certain cases, especially when combined with other computer vision techniques.

NOTE: YOLO is a deep learning-based algorithm that is fast and accurate in object detection, while HOG is a traditional computer vision-based algorithm that is slower but can be very accurate in some instances.

References:

- https://pytorch.org/docs/stable/index.html
- https://pytorch.org/vision/stable/models.html
- https://pytorch.org/vision/stable/models/resnet.html
- https://pytorch.org/vision/stable/datasets.html
- https://pytorch.org/tutorials/
- https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html
- https://www.guru99.com/seq2seq-model.html
- https://www.youtube.com/watch?v=y2BaTt1fxJU
- https://arxiv.org/abs/1502.03044
- https://github.com/aladdinpersson/Machine-Learning-Collection/tree/master/ML/Pytorch/more_advanced/image_captioning
- https://github.com/ultralytics/ultralytics
- https://colab.research.google.com/github/ultralytics/ultralytics/blob/main/examples/tutorial.ipynb
- https://github.com/ultralytics/ultralytics/blob/main/examples/tutorial.ipynb
- https://www.makesense.ai/
- https://www.youtube.com/watch?v=GRtgLlwxpc4

Links:
ColabNotebooks + Data:- CV_A4

Note: Use institute id only for accessing the above link.