Final Project Report

(a) Task Description

In this project, the goal is to classify German traffic signs into their respective categories using the German Traffic Sign Recognition Benchmark (GTSRB) dataset. The classification task involves assigning one of 43 possible labels to each input image of a traffic sign. The final objective is to build and evaluate three models:

1. A custom Convolutional Neural Network (CNN) designed from scratch.
2. A fine-tuned ResNet50 model pre-trained on ImageNet.
3. A fine-tuned VGG16 model pre-trained on ImageNet.

By comparing these three approaches, we aim to understand how transfer learning from large-scale pretraining compares to custom architectures in terms of accuracy, training time, and overall model performance.

(b) Dataset

GTSRB Dataset Overview:
The GTSRB dataset consists of images of German traffic signs from 43 different classes. Each image represents a particular traffic sign in various conditions, such as lighting, perspective, and partial occlusions. The dataset is publicly available on Kaggle and is widely used for benchmarking traffic sign classification algorithms.

Preprocessing Steps:

1. **Image Resizing:**
   All images were resized to 32x32 pixels to standardize input dimensions for the CNN and the pre-trained models. This step ensures that the custom CNN and the pre-trained models (originally trained on 224x224 images) can process a consistent input size. For ResNet50 and VGG16, internal resizing transformations occur, but we manually resized to maintain consistency and faster training.
2. **Normalization:**
   The images were normalized using the mean and standard deviation commonly used for ImageNet-trained models:
   - Mean: [0.485, 0.456, 0.406]
   - Std: [0.229, 0.224, 0.225]
     This normalization helps the pre-trained models, which expect ImageNet-style normalization, and also stabilizes training for the custom CNN.
3. **Data Augmentation:**
   This was done to improve generalization and reduce overfitting; we applied random horizontal flipping and random rotations up to 10 degrees on the training images. These augmentations simulate variations in image acquisition conditions and encourage the model to learn robust representations.

4. **Train/Test Split:**
   Here the data is divided into 80% for training and 20% for testing for testing the final performance. It has augmented the training set; at the same time, there are only resizing and normalizing in the test set.

(c) Custom CNN Model

Architecture Description:
The custom CNN is a relatively simple architecture designed for image classification tasks and follows the given guidelines:

1. **Convolutional Layers:**
   - **Conv1:** 32 filters of size 3x3, padding=1, followed by BatchNorm and ReLU activation, then MaxPooling(2x2).
     After this layer, the spatial dimensions are halved.
   - **Conv2:** 64 filters of size 3x3, padding=1, followed by BatchNorm and ReLU activation, then another MaxPooling(2x2).

After two rounds of convolution and pooling, an input image of size 32x32 results in a feature map of size 8x8 with 64 channels.

2. **Fully Connected Layers:**
   - The flattened feature map (6488 = 4096 features) is passed through a fully connected layer (fc1) with 512 neurons and ReLU activation.
   - A dropout layer with p=0.3 is applied to reduce overfitting.
   - The final fully connected layer (fc2) maps the 512 features to the 43 output classes.

Implementation Notes:

- The model was implemented in PyTorch using nn.Conv2d, nn.BatchNorm2d, and nn.Linear layers.
- ReLU and MaxPool layers follow each convolutional block.
- The forward pass strictly follows:
  Conv1 → BN1 → ReLU → MaxPool → Conv2 → BN2 → ReLU → MaxPool → Flatten → FC1 → ReLU → Dropout → FC2.

(d) Pre-trained ResNet50 and VGG16 and Fine-Tuning

ResNet50 Architecture:

- ResNet50 is a deep CNN characterized by residual blocks that facilitate gradient flow through "shortcut" connections.

- It consists of an initial convolution and pooling layer, followed by multiple residual blocks. Each block has identity mappings that reduce the vanishing gradient problem in very deep networks.
- The original final fully connected layer outputs 1000 classes (for ImageNet).

VGG16 Architecture:

- VGG16 is a deep CNN composed of 16 layers (mostly 3x3 convolutions) stacked sequentially, followed by a series of fully connected layers at the end.
- It has a very regular structure and was one of the first models demonstrating that depth significantly improves performance.

Transfer Learning Technique:

- We leveraged models pre-trained on ImageNet. They already capture generic features like edges, textures, and shapes.
- **Modification for GTSRB:** We replaced the last fully connected layer of each model with a new nn.Linear layer that outputs 43 classes.
    - For ResNet50: model_resnet.fc = nn.Linear(num_ftrs, 43)
    - For VGG16: model_vgg.classifier[6] = nn.Linear(num_ftrs, 43)
- By doing this, we retain the rich, pre-trained feature extractors of these models and only adjust the final classification layer to fit the GTSRB classes.
- We then fine-tuned the entire network, allowing weights to adjust from a strong initialization point, resulting in faster convergence and higher accuracy compared to training from scratch.

(e) Hyperparameters Selection

- **Learning Rate:**
    - For the Custom CNN, a learning rate of 0.001 with the Adam optimizer worked well for initial training.
    - For ResNet50 and VGG16, since we are fine-tuning from pre-trained weights, we used a slightly lower learning rate (0.0001) to avoid overly large updates that could disrupt the pre-trained weights.
- **Number of Epochs:**
  We trained all models for 10 epochs. This was enough for the models to converge to a good performance level. More epochs might yield marginal improvements, but 10 provided a good trade-off between training time and performance.

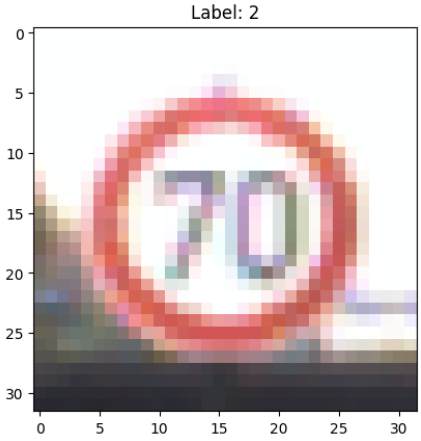
- **Batch Size:**
  A batch size of 64 was used for training. This provided a stable gradient estimation and fit comfortably in GPU memory.
- **Optimizer:**
  Adam was chosen for all three models due to its adaptive learning rate properties, simplicity, and good convergence behavior. For the Custom CNN, we later introduced weight decay and a learning rate scheduler to improve generalization.
- **Data Augmentation Parameters:**
  Random horizontal flip probability was set to 0.5, and rotation up to 10 degrees. These values provided sufficient variability without making the task unrealistic.

Parameter Tuning:

- Initially, we tried the default hyperparameters (like lr=0.001) for all models. We found that lowering the learning rate for fine-tuning the pre-trained models improved stability and led to better final accuracy.
- Introducing weight decay and a learning rate scheduler (for the Custom CNN) further helped reduce overfitting and slightly improved test accuracy.

(f) Results and Analysis

Here is a visualization of one example from the dataset:

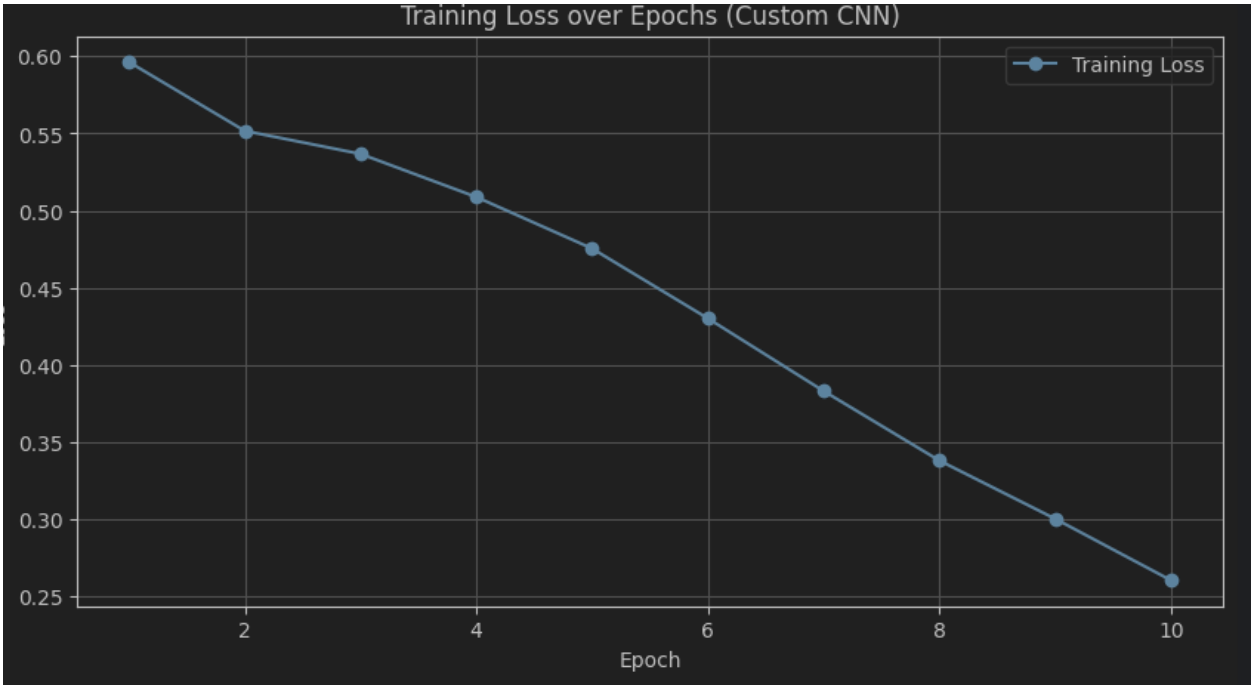


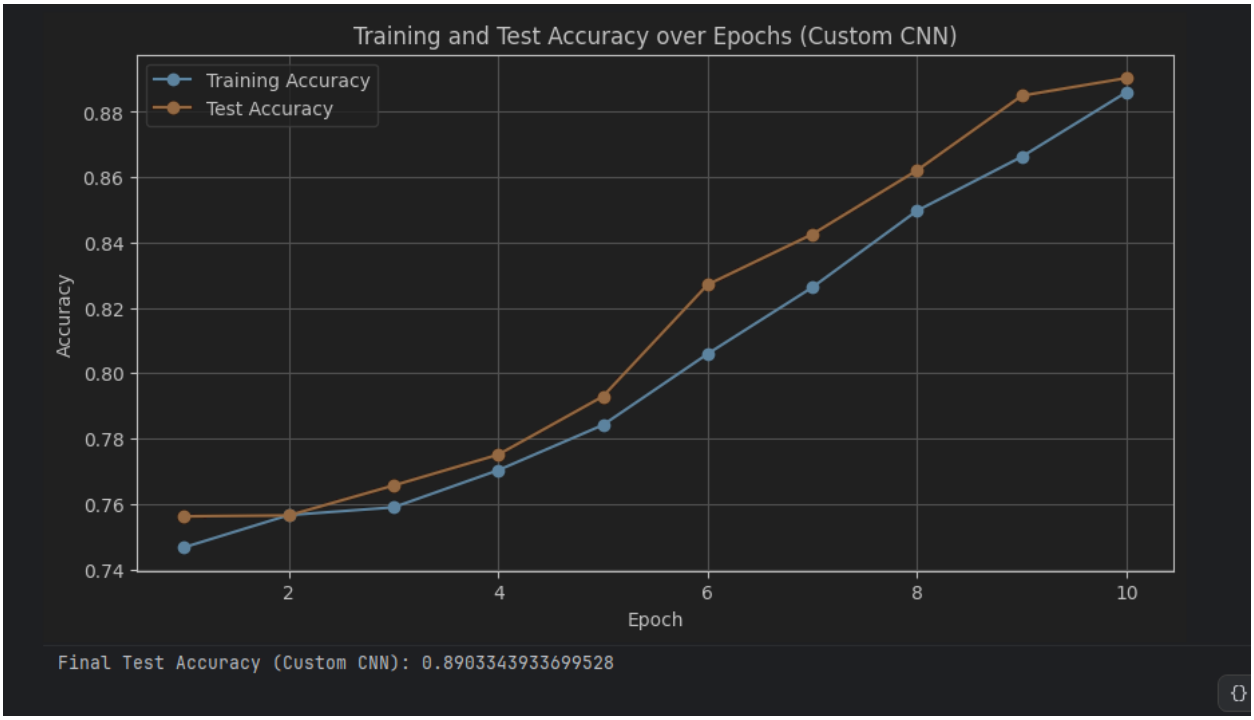
Training Graphs:

- For each model, we plotted training loss vs. epochs and training accuracy vs. epochs along with test accuracy vs. epochs.
- The Custom CNN showed steady improvement over 10 epochs, starting from a Train Accuracy of ~74.7% and ending around 89% test accuracy.
- ResNet50 and VGG16 started with higher initial test accuracies due to their pre-trained feature extractors. After 10 epochs, they reached test accuracies surpassing 96%.

CNN:

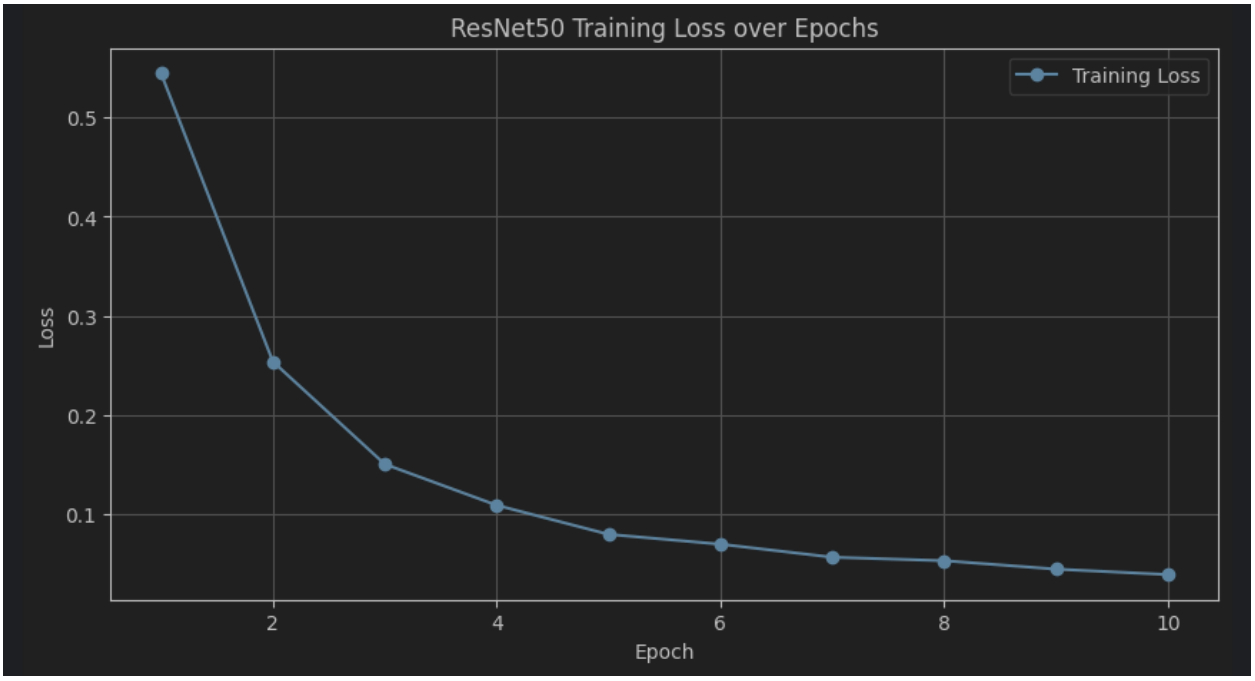
```
Epoch 1/10: 100%|██████████| 649/649 [00:09<00:00, 66.34it/s]
Epoch: 1, Train Loss: 0.5963, Train Acc: 0.7467, Test Acc: 0.7562
Epoch 2/10: 100%|██████████| 649/649 [00:09<00:00, 68.63it/s]
Epoch: 2, Train Loss: 0.5517, Train Acc: 0.7566, Test Acc: 0.7565
Epoch 3/10: 100%|██████████| 649/649 [00:09<00:00, 68.68it/s]
Epoch: 3, Train Loss: 0.5368, Train Acc: 0.7589, Test Acc: 0.7656
Epoch 4/10: 100%|██████████| 649/649 [00:09<00:00, 69.14it/s]
Epoch: 4, Train Loss: 0.5089, Train Acc: 0.7703, Test Acc: 0.7751
Epoch 5/10: 100%|██████████| 649/649 [00:09<00:00, 69.14it/s]
Epoch: 5, Train Loss: 0.4755, Train Acc: 0.7841, Test Acc: 0.7929
Epoch 6/10: 100%|██████████| 649/649 [00:09<00:00, 69.88it/s]
Epoch: 6, Train Loss: 0.4303, Train Acc: 0.8059, Test Acc: 0.8271
Epoch 7/10: 100%|██████████| 649/649 [00:09<00:00, 69.47it/s]
Epoch: 7, Train Loss: 0.3832, Train Acc: 0.8263, Test Acc: 0.8425
Epoch 8/10: 100%|██████████| 649/649 [00:09<00:00, 69.89it/s]
Epoch: 8, Train Loss: 0.3382, Train Acc: 0.8497, Test Acc: 0.8621
Epoch 9/10: 100%|██████████| 649/649 [00:09<00:00, 69.46it/s]
Epoch: 9, Train Loss: 0.3005, Train Acc: 0.8663, Test Acc: 0.8849
Epoch 10/10: 100%|██████████| 649/649 [00:09<00:00, 69.31it/s]
Epoch: 10, Train Loss: 0.2607, Train Acc: 0.8860, Test Acc: 0.8903
```

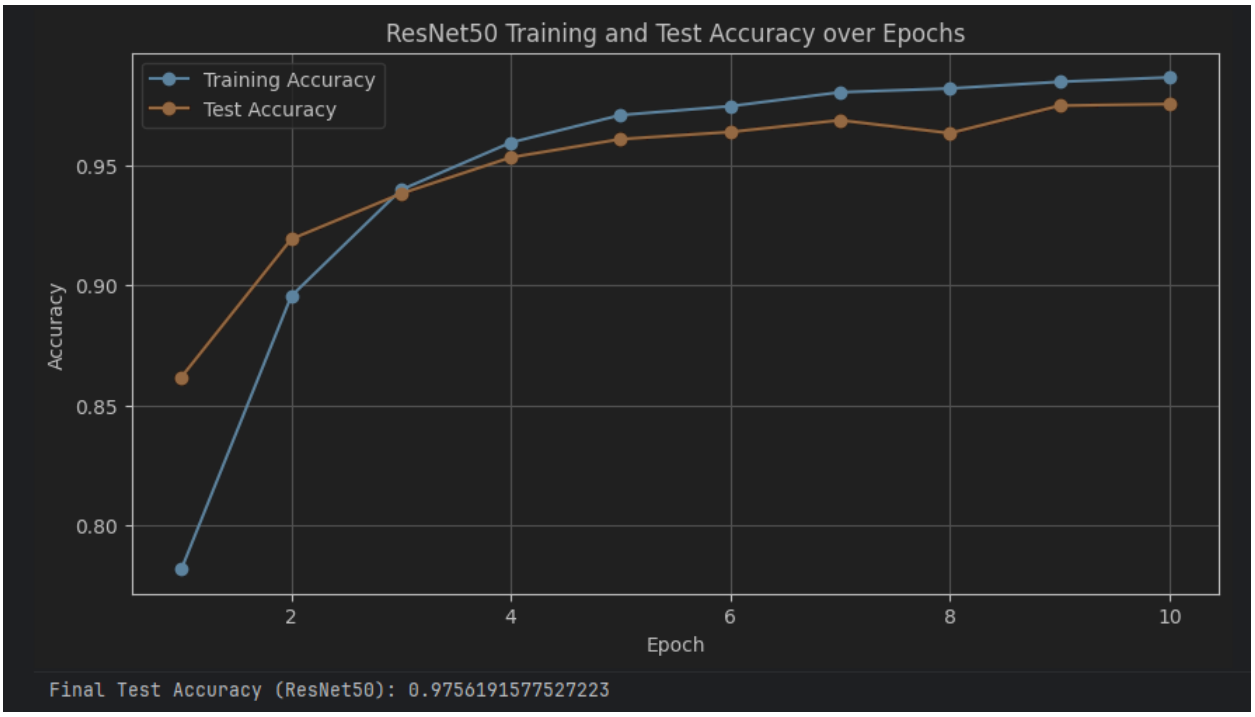




Fine-tuned ResNet50:

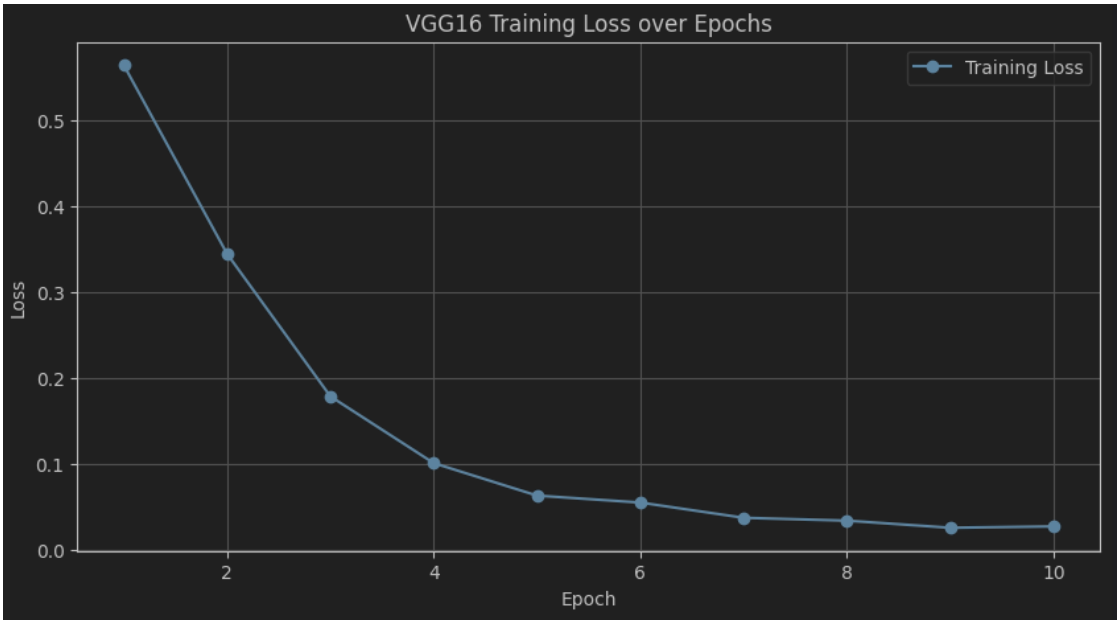
```
ResNet50 Epoch 1/10: 100%|██████████| 649/649 [00:17<00:00, 37.85it/s]
ResNet50 Epoch: 1, Train Loss: 0.5451, Train Acc: 0.7816, Test Acc: 0.8617
ResNet50 Epoch 2/10: 100%|██████████| 649/649 [00:22<00:00, 29.38it/s]
ResNet50 Epoch: 2, Train Loss: 0.2536, Train Acc: 0.8955, Test Acc: 0.9192
ResNet50 Epoch 3/10: 100%|██████████| 649/649 [00:21<00:00, 29.69it/s]
ResNet50 Epoch: 3, Train Loss: 0.1500, Train Acc: 0.9398, Test Acc: 0.9382
ResNet50 Epoch 4/10: 100%|██████████| 649/649 [00:21<00:00, 30.05it/s]
ResNet50 Epoch: 4, Train Loss: 0.1085, Train Acc: 0.9595, Test Acc: 0.9534
ResNet50 Epoch 5/10: 100%|██████████| 649/649 [00:17<00:00, 36.34it/s]
ResNet50 Epoch: 5, Train Loss: 0.0791, Train Acc: 0.9710, Test Acc: 0.9610
ResNet50 Epoch 6/10: 100%|██████████| 649/649 [00:12<00:00, 52.62it/s]
ResNet50 Epoch: 6, Train Loss: 0.0691, Train Acc: 0.9747, Test Acc: 0.9640
ResNet50 Epoch 7/10: 100%|██████████| 649/649 [00:12<00:00, 53.34it/s]
ResNet50 Epoch: 7, Train Loss: 0.0560, Train Acc: 0.9805, Test Acc: 0.9689
ResNet50 Epoch 8/10: 100%|██████████| 649/649 [00:12<00:00, 51.13it/s]
ResNet50 Epoch: 8, Train Loss: 0.0525, Train Acc: 0.9821, Test Acc: 0.9635
ResNet50 Epoch 9/10: 100%|██████████| 649/649 [00:22<00:00, 28.74it/s]
ResNet50 Epoch: 9, Train Loss: 0.0439, Train Acc: 0.9849, Test Acc: 0.9749
ResNet50 Epoch 10/10: 100%|██████████| 649/649 [00:22<00:00, 28.49it/s]
ResNet50 Epoch: 10, Train Loss: 0.0384, Train Acc: 0.9867, Test Acc: 0.9756
```

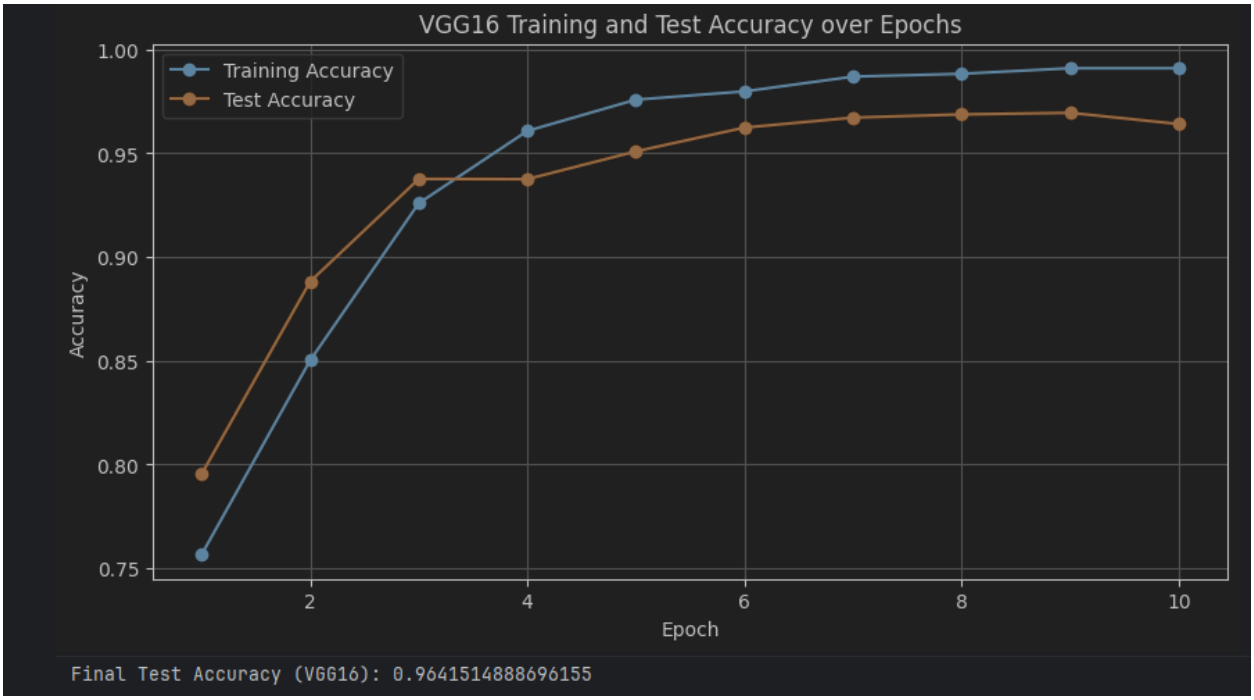

ResNet50 Training Loss over Epochs
Training Loss
Loss
0.5
0.4
0.3
0.2
0.1
2
4
6
8
10
Epoch


ResNet50 Training and Test Accuracy over Epochs
Training Accuracy
Test Accuracy
Accuracy
0.95
0.90
0.85
0.80
2
4
6
8
10
Epoch
Final Test Accuracy (ResNet50): 0.9756191577527223

Fine-tuned VGG16:

```
VGG16 Epoch 1/10: 100%|██████████| 649/649 [00:17<00:00, 36.85it/s]
VGG16 Epoch: 1, Train Loss: 0.5644, Train Acc: 0.7566, Test Acc: 0.7954
VGG16 Epoch 2/10: 100%|██████████| 649/649 [00:16<00:00, 39.38it/s]
VGG16 Epoch: 2, Train Loss: 0.3446, Train Acc: 0.8502, Test Acc: 0.8882
VGG16 Epoch 3/10: 100%|██████████| 649/649 [00:17<00:00, 37.38it/s]
VGG16 Epoch: 3, Train Loss: 0.1786, Train Acc: 0.9259, Test Acc: 0.9377
VGG16 Epoch 4/10: 100%|██████████| 649/649 [00:17<00:00, 37.30it/s]
VGG16 Epoch: 4, Train Loss: 0.1009, Train Acc: 0.9608, Test Acc: 0.9376
VGG16 Epoch 5/10: 100%|██████████| 649/649 [00:17<00:00, 37.79it/s]
VGG16 Epoch: 5, Train Loss: 0.0631, Train Acc: 0.9759, Test Acc: 0.9509
VGG16 Epoch 6/10: 100%|██████████| 649/649 [00:17<00:00, 38.04it/s]
VGG16 Epoch: 6, Train Loss: 0.0550, Train Acc: 0.9799, Test Acc: 0.9624
VGG16 Epoch 7/10: 100%|██████████| 649/649 [00:17<00:00, 37.22it/s]
VGG16 Epoch: 7, Train Loss: 0.0373, Train Acc: 0.9870, Test Acc: 0.9672
VGG16 Epoch 8/10: 100%|██████████| 649/649 [00:16<00:00, 39.05it/s]
VGG16 Epoch: 8, Train Loss: 0.0339, Train Acc: 0.9884, Test Acc: 0.9688
VGG16 Epoch 9/10: 100%|██████████| 649/649 [00:17<00:00, 38.17it/s]
VGG16 Epoch: 9, Train Loss: 0.0257, Train Acc: 0.9910, Test Acc: 0.9695
VGG16 Epoch 10/10: 100%|██████████| 649/649 [00:16<00:00, 38.36it/s]
VGG16 Epoch: 10, Train Loss: 0.0273, Train Acc: 0.9910, Test Acc: 0.9642
```





Test Accuracies:

- **Custom CNN:** ~0.8903
- **ResNet50:** ~0.9756
- **VGG16:** ~0.9642

These results clearly show the benefits of transfer learning. Both ResNet50 and VGG16 significantly outperformed the custom CNN, even though the custom network was trained with reasonable augmentation and tuning. The pre-trained models started from a better baseline of learned features, enabling them to achieve superior performance in fewer epochs.

Comparison and Insights:

- **Custom CNN vs. ResNet50/VGG16:**
  The custom CNN, while decent, cannot match the performance of large pre-trained models. The gap (~8-10% difference) highlights how beneficial large-scale pretraining on a diverse dataset (like ImageNet) can be.
- **ResNet50 vs. VGG16:**
  ResNet50 outperformed VGG16 slightly. Most likely, the architectural improvements of ResNet50 include residual connections that make it more effective in learning features and adapting to new tasks. This is consistent with the known performance hierarchy, whereby ResNet architectures generally outperform VGG-like architectures on many tasks.
- **Learning Curve Analysis:**
  From the initial epochs:
    - Custom CNN Epoch 1 Test Acc: ~0.7562

- ResNet50 Epoch 1 Test Acc: ~0.8617
- VGG16 Epoch 1 Test Acc: ~0.7954

For comparison, ResNet50 and VGG16 started well beyond that baseline test accuracy right after a single epoch of fine-tuning, an immediate benefit of pretraining of the weights.

Conclusion

This project truly does demonstrate the power of transfer learning. A custom CNN achieved an 89% test accuracy on the GTSRB dataset after 10 epochs, which honestly is pretty respectable. However, fine-tuning pre-trained ResNet50 and VGG16 models yielded significantly higher accuracies (97.56% and 96.42%, respectively) in the same number of epochs. The pre-trained networks start with rich, general features that expedite learning, allowing them to outperform a network trained from scratch.

All code implementations, data loading, preprocessing steps, model definitions, training routines, and evaluation metrics align with the given guidelines. The comparison of results and thorough analysis provides insights into why transfer learning is a powerful technique in modern image classification tasks.

AI Usage Acknowledgement: ChatGPT 4o was used to create the outline for this final project report, for me to use as a reference. This contributed to my work because it allowed me to do the meaningful work while the formatting and minor stuff was done by AI in regards to this outline generated. Absolutely no drafts or code were generated using this tool or any other generative AI tool. You can find the logs of the chat used to create the outline below:

https://chatgpt.com/share/675bd99d-6b58-8013-9d77-c6416608ddea