Image Classification of German Traffic Signs Using Capsule Neural Network and Convolutional Neural Network

Siddharth Das, Kartik Jain, Pavani Samala

1. Introduction

In this project, we explored Capsnet Neural Networks (CapsNet) and compared their performance with a Convolutional Neural Network (CNN). Both models were fine-tuned to perform image classification on the German Traffic Sign Image Dataset and were evaluated by plotting the accuracy and loss and calculating the f1-score. Lastly, we implemented Local Interpretable Model-agnostic Explanations (LIME) to understand the model predictions.

As a general overview of this paper, we will begin by discussing the results of our Exploratory Data Analysis (EDA). Next, we will provide background information on CapsNet and describe the models we used in our experiment. Then, we will discuss our results and explainability. Finally, we will conclude with our findings and conclusion thoughts.

2. Exploratory Data Analysis and Description of the data set.

The dataset used in this project was a collection of more than 50,000 German traffic sign images and 43 different classes, varying in lighting and quality. We split the images into training and testing with approximately 80-20 split for the experiment.

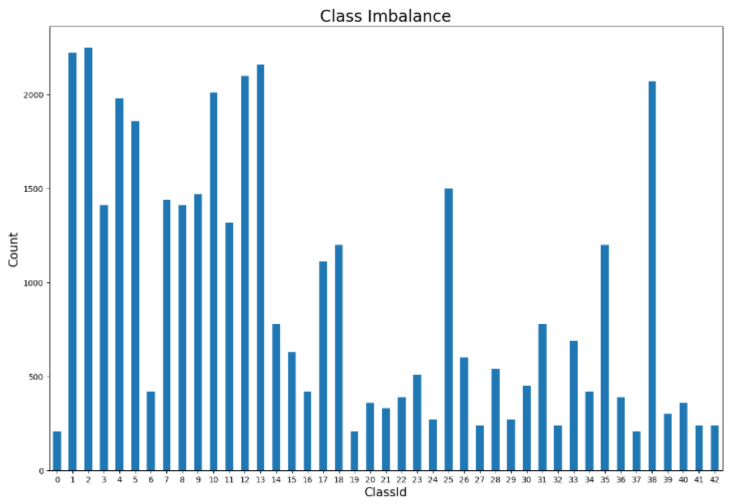


Figure 1. Count of Images per Class

The figure above shows the number of images per class. It seems that there is a slight class imbalance since a fraction of the classes have more images than the others. However, this is expected since some traffic signs are more widely used than others.

3. CapsNets

History and Background on CapsNet

Geoffrey Hinton, Sara Sabour and Nicholas Frosst developed Capsnet to overcome the shortcomings of a CNN. Although CNNs have changed deep learning greatly, their use of pooling layers causes spatial information between low level features and high level features to be lost. Hinton created a capsule, which is a group of neurons that outputs a vector that encapsulates spatial information about a specific object or part of an object. The length of this vector represents the probability of the object's presence in a particular region. This vector is valuable because it changes as the object's orientation and position changes in the image, but the probability of being detected stays the same.

How a capsule works

A capsule performed four operations to output a vector. The order of operations in a capsule are as follows: matrix multiplication of input vectors, scaling of scalar weighting, summing, and squashing. While discussing these steps, the figure below will be referenced.

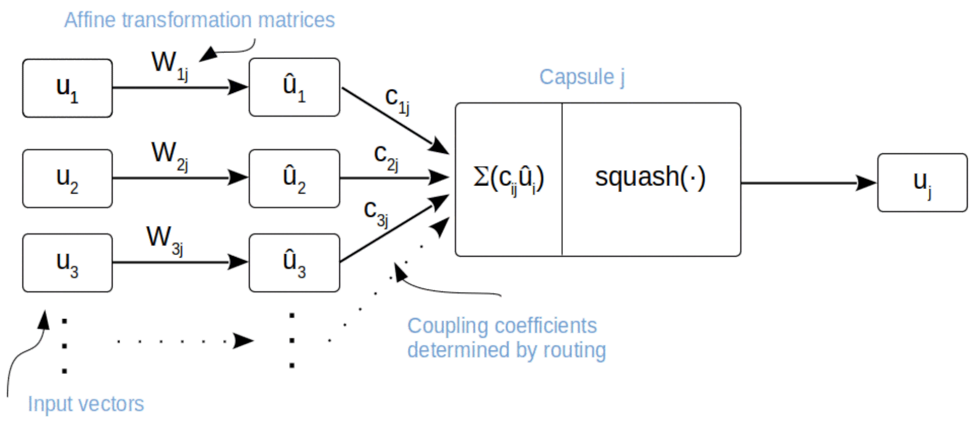


Figure 2: Capsule Configuration

Matrix Multiplication of Input Vectors:

The first step in a capsule is to multiply the inputs with weights matrices. As shown in figure 2, u_1, u_2, and u_3 are all vector inputs and they are either the initial inputs or are the outputs of previous capsules. The weight matrices are encoded with spatial information and are responsible for transforming u_1, u_2, and u_3 into the position of the predicted high level feature. So, after the matrix multiplication is completed, the resulting vectors, u_1, u_2, and u_3, tell us where the high level feature is located in respect to the low level feature's location. If they all point in the same orientation, then the high level feature will be predicted.

Scaling Scalar Weights with Dynamic Routing:

Just like a CNN, a capsule needs to update its scalar weights c_{ij}. This will help the low-level capsule decide which high level capsule to send its output to (note that the weights are scalar and are not the same as the weight matrices described in the previous step). However, rather than using backpropagation, capsules use an algorithm called Dynamic Routing and is illustrated in the figure below. First, the capsule needs to calculate the distance between its output and the other predictions from the other low level capsules, which are the clusters of dots. The minimum distance found, is the capsule that will receive the highest weight, and therefore the output from the capsule.The rest of the capsules will have weights assigned in a descending order.

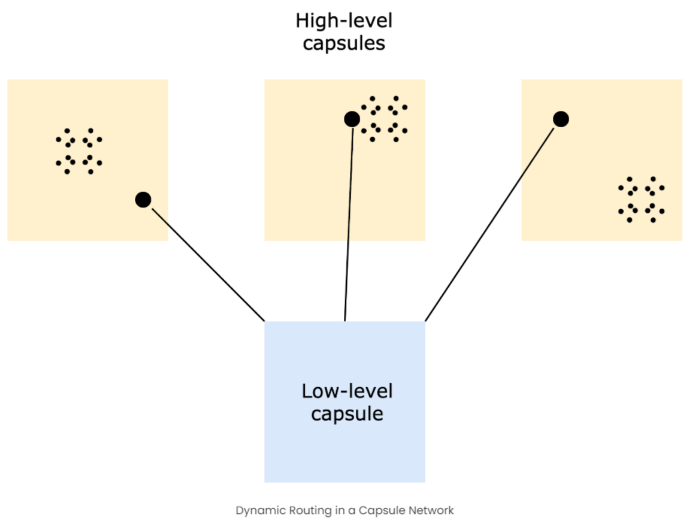


Figure 3: Dynamic Routing

Summing:

Referring to figure 2, after all the scalar weights are assigned, the multiplication of u_{ij} and c_{ij} are summed.

Squashing:

The last step a capsule needs to perform is converting the output vector to another vector using a non linear activation function known as the squashing function. The equation below describes squashing. s_i is the output from the summing step and the equation results in a vector with a length of between 0 and 1. It is important to note that while the length may change, the direction of the vector will not, preserving spatial information.

$$v_j = \frac{\|s_j\|}{1+\|s_j\|^2} \cdot \frac{s_j}{\|s_j\|}$$

Figure 4: Squashing

Capsnet Architecture

The CapsNet's architecture includes an encoder and decoder as shown in the figure below. The encoder has three layers: a convolution layer, PrimaryCaps layer, and DigitCaps layer. The decoder also has three layers, which are all fully connected layers. The decoder will take information from the DigitCaps and use it to reconstruct the image.

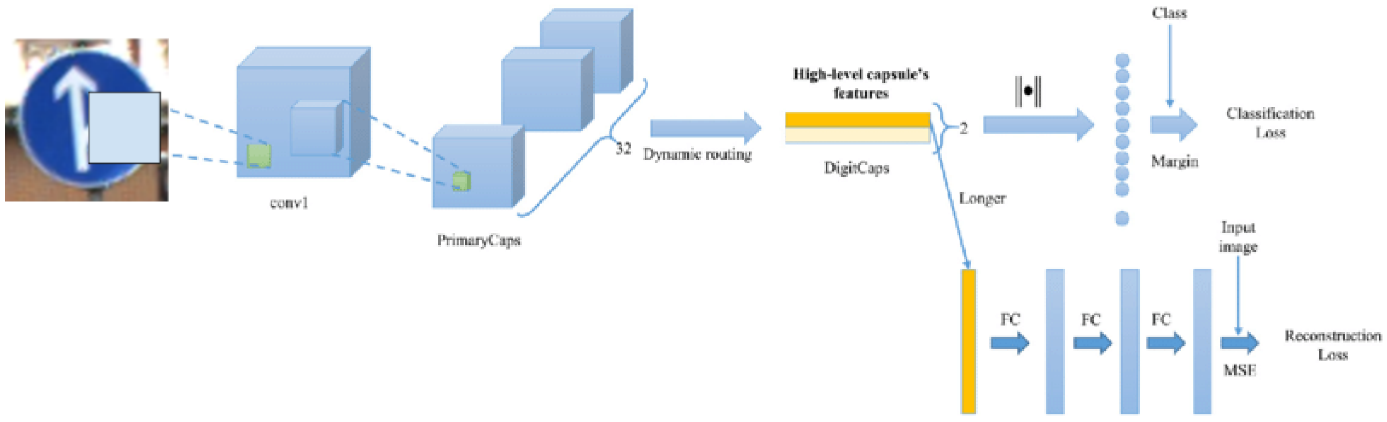


Figure 5: Capsule Neural Network Architecture

The loss is calculated in two parts. The encoder calculates a margin loss, meaning if a particular object is present in the image, that the square of the length of the vector cannot be less than 0.9. Conversely, if the object is not present, then the square of the length of the vector cannot be more than 0.1. The loss from the decoder is calculated by finding the euclidean distance between the reconstructed image and the input image using the following formula:

$$L_k = T_k \, max\left(0,\ m^+ - \left\|v_k\right\|\right)^2 + \lambda\left(1 - T_k\right) max\left(0,\ \left\|v_k\right\| - m^-\right)^2$$

In the end, the total loss is a sum of the margin loss and scaled down value of the reconstruction loss.

4. Methodology

Experimental Setup

The objective of our project is to run the CapsNet on the German Traffic Sign Benchmark dataset. We want to verify that capsule networks helped solve the issues with Convolutional Neural Networks. For this purpose, we do hyperparameter tuning to get an optimized model. We build a baseline architecture of a CNN network to serve as a comparison against the obtained metric scores of the capsule network on the dataset. Finally, we want to explain the results obtained from the two models to explain the inner workings of the algorithms using their results on sample data.

Fine-tuning

In the fine-tuning phase, we experimented with different number epochs, learning rate, batch size, and optimizer momentum. Some combinations that we tried are as follows:

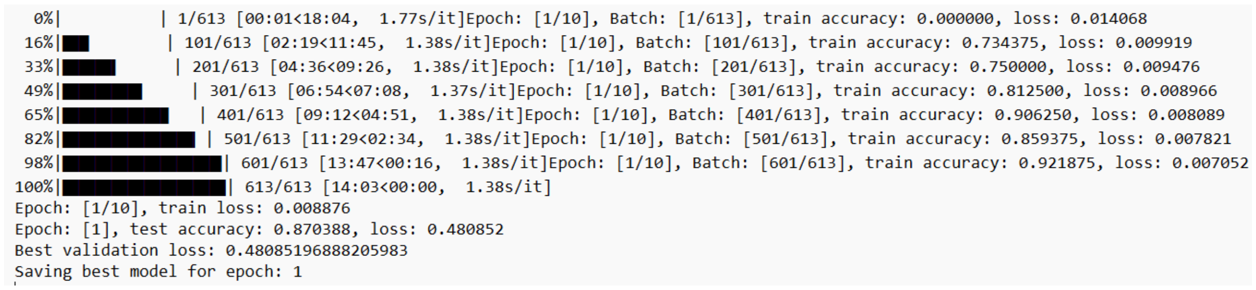
- Batch Size: We experimented with batch sizes 16, 32, and 64 and found similar results between the different batch sizes. However, we decided to use the largest batch size because the training was marginally faster because of the numerous transformations done within each batch.
- Number of epochs: We ran the model for 5 epochs, 10 epochs, and 25 epochs. We found that ten was the ideal number of epochs as the validation accuracy leveled off at this point.
- Learning rate: We experimented with different learning rates of 1e-3, 1e-4, and 1e-5 and chose the largest one because it helped us get to a good score faster without blowing up the gradient.

- Momentum: We set the momentum value for the RMSProp optimizer to 0.9 instead of 0.5 because it gave the best scores. Ultimately, we decided to use the Adam optimizer because it has an in-built momentum.

Hyperparameter - tuning

We tested all our changes against a baseline model configuration run and chose the best parameters.

The base run results are:

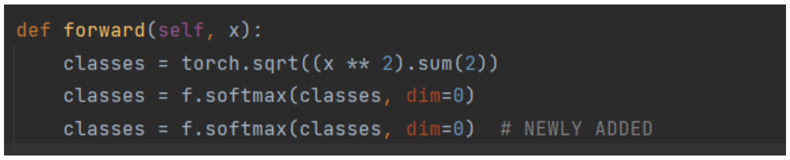


```
  0%|          | 1/613 [00:01<18:04,  1.77s/it]Epoch: [1/10], Batch: [1/613], train accuracy: 0.000000, loss: 0.014068
 16%|█         | 101/613 [02:19<11:45,  1.38s/it]Epoch: [1/10], Batch: [101/613], train accuracy: 0.734375, loss: 0.009919
 33%|███       | 201/613 [04:36<09:26,  1.38s/it]Epoch: [1/10], Batch: [201/613], train accuracy: 0.750000, loss: 0.009476
 49%|████▉     | 301/613 [06:54<07:08,  1.37s/it]Epoch: [1/10], Batch: [301/613], train accuracy: 0.812500, loss: 0.008966
 65%|██████▌   | 401/613 [09:12<04:51,  1.38s/it]Epoch: [1/10], Batch: [401/613], train accuracy: 0.906250, loss: 0.008089
 82%|████████▏ | 501/613 [11:29<02:34,  1.38s/it]Epoch: [1/10], Batch: [501/613], train accuracy: 0.859375, loss: 0.007821
 98%|█████████▊| 601/613 [13:47<00:16,  1.38s/it]Epoch: [1/10], Batch: [601/613], train accuracy: 0.921875, loss: 0.007052
100%|██████████| 613/613 [14:03<00:00,  1.38s/it]
Epoch: [1/10], train loss: 0.008876
Epoch: [1], test accuracy: 0.870388, loss: 0.480852
Best validation loss: 0.48085196888205983
Saving best model for epoch: 1
```

Figure 6: Baseline Case

We made the following changes to the Configuration class for our Capsule Net model:

- Extra softmax function on the output dimension: It has no change in the validation accuracy.

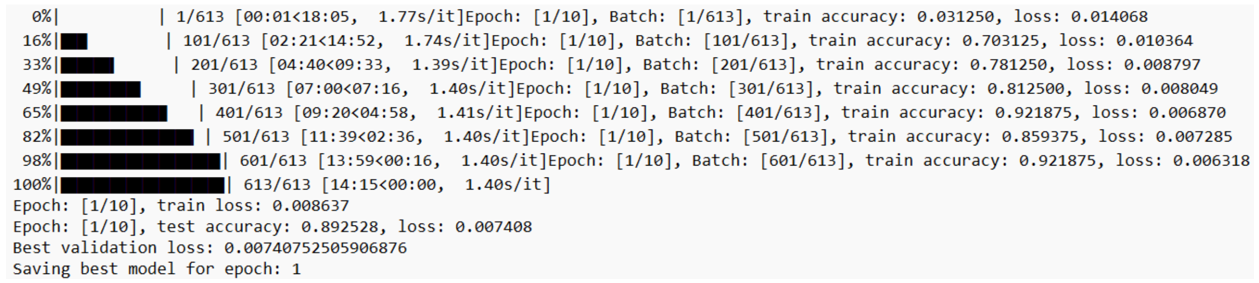


```
def forward(self, x):
    classes = torch.sqrt((x ** 2).sum(2))
    classes = f.softmax(classes, dim=0)
    classes = f.softmax(classes, dim=0)  # NEWLY ADDED
```

Figure 7: Extra Softmax

- Hardswish function in the convolution layer: This decreased model validation accuracy to 0.77 after the first epoch. We reverted this change.
- Number of input and output channels in the Conv and primary capsule layer: cnn_out_channels and pc_in_channels were changed to 384 from 256 and we saw an increase in model performance.
- Pc_kernel_size was changed to 8 and pc_num_routes and dc_num_routes were changed to 32 * 7 * 7 to fit the output dimension size. The model score was slightly lower than the baseline so this change was reverted as well.

- Pc_kernel_size was changed to 10 and this resulted in better accuracy. This was because the matrix boundary was not getting dropped anymore because of the stride of 2. This resulted in the following scores over the first epoch:



```
  0%|          | 1/613 [00:01<18:05,  1.77s/it]Epoch: [1/10], Batch: [1/613], train accuracy: 0.031250, loss: 0.014068
 16%|██        | 101/613 [02:21<14:52,  1.74s/it]Epoch: [1/10], Batch: [101/613], train accuracy: 0.703125, loss: 0.010364
 33%|████      | 201/613 [04:40<09:33,  1.39s/it]Epoch: [1/10], Batch: [201/613], train accuracy: 0.781250, loss: 0.008797
 49%|██████    | 301/613 [07:00<07:16,  1.40s/it]Epoch: [1/10], Batch: [301/613], train accuracy: 0.812500, loss: 0.008049
 65%|████████  | 401/613 [09:20<04:58,  1.41s/it]Epoch: [1/10], Batch: [401/613], train accuracy: 0.921875, loss: 0.006870
 82%|██████████ | 501/613 [11:39<02:36,  1.40s/it]Epoch: [1/10], Batch: [501/613], train accuracy: 0.859375, loss: 0.007285
 98%|████████████| 601/613 [13:59<00:16,  1.40s/it]Epoch: [1/10], Batch: [601/613], train accuracy: 0.921875, loss: 0.006318
100%|████████████| 613/613 [14:15<00:00,  1.40s/it]
Epoch: [1/10], train loss: 0.008637
Epoch: [1/10], test accuracy: 0.892528, loss: 0.007408
Best validation loss: 0.00740752505906876
Saving best model for epoch: 1
```

Figure 8: Results after changing the Kernel Size

Interpretation with LIME

Evaluating models based on accuracy is not always sufficient; hence, we sought ways to explain and interpret our model's behaviors to understand the reasons behind why certain classes were predicted. LIME, as mentioned before, stands for Local Interpretable Model-agnostic Explanations. This tool explains a model by approximating the local linear behavior. It generates new data points around a given instance and uses the model to classify them. Then, it weighs the data points based on their distance to the original instance and fits them using a linear classifier. This can tell us whether increasing or decreasing a value can change the prediction. After running its algorithm, LIME masks a portion of the image in green if it has a strong positive correlation and contributes greatly to the prediction and masks the image in red if it has a negative correlation and contributes trivially to the prediction.

To avoid any bias, we built code that randomly chooses an image and utilized the lime package to generate masked images. Below are the results from using LIME on four random images that were predicted by the CapsNet and CNN model. Figures, a and b, are organized as follows: top left is the original image, top right is the segmented image that is the output of the segmentation algorithm used by lime, bottom left is the masked image using the CapsNet model, and bottom right is the masked image from CNN model.

Looking at our result, we can see that the CapsNet and CNN model correctly predicted both images. However, when we take a closer look at the highlighted regions, some of the results are unexpected. In figure 7, both models use the image outside the traffic sign to classify the image. Intuitively, this does not make sense. In figure 8, the CapsNet model again uses the image outside the traffic sign; the CNN on the other hand correctly uses the inside of the traffic sign to make the correct prediction.

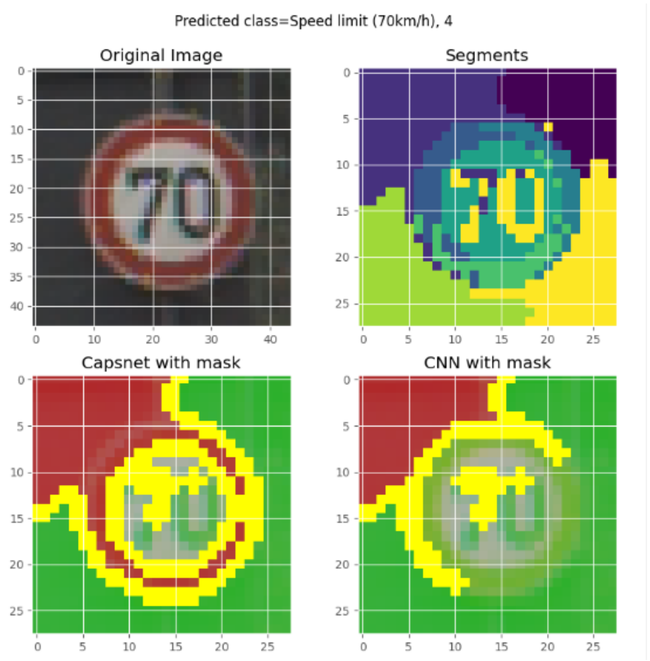


Figure 9: Random LIME Result #1

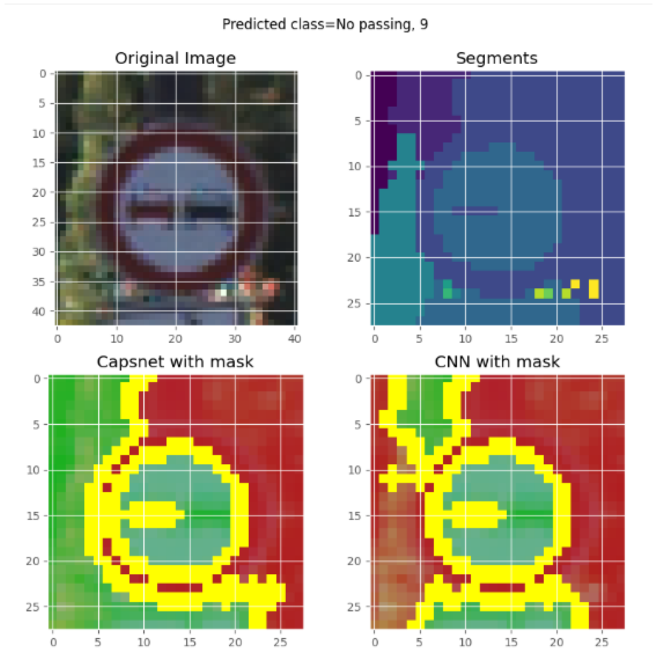


Figure 10: Random LIME Result #1

5. Results.

We fixed the optimized hyperparameter values like learning rate, the number of epochs, etc., and ran both the baseline and Capsnet models and compared their performance. We can see that the loss is very different for Capsule networks as opposed to CNNs and expected. It is because Capsnet uses a custom loss function that combines the margin loss (loss between encoder output and target) and reconstruction loss (reconstruction and the input data). The margin loss gets precedence and the reconstruction loss with weights 0.005.

The validation accuracy and F1 scores show that the two models are performing similarly for the most part, and CNN performs better for a higher number of epochs. Hence, we conclude that one model is not inherently better than the other, and more work is needed to tune the parameters and test on new datasets.

Table 1: CapsNet Summary

Model	Epoch	Validation Loss	Validation Accuracy	Validation F1-Macro
Capsule Networks	8	4.6e-3	0.943	0.903
	9	4.5e-3	0.947	0.913
	10	4.4e-3	0.948	0.914

Table 2: CNN Summary

Model	Epoch	Validation Loss	Validation Accuracy	Validation F1-Macro
Baseline CNN	8	0.265	0.94	0.91
	9	0.236	0.943	0.91
	10	0.249	0.947	0.923

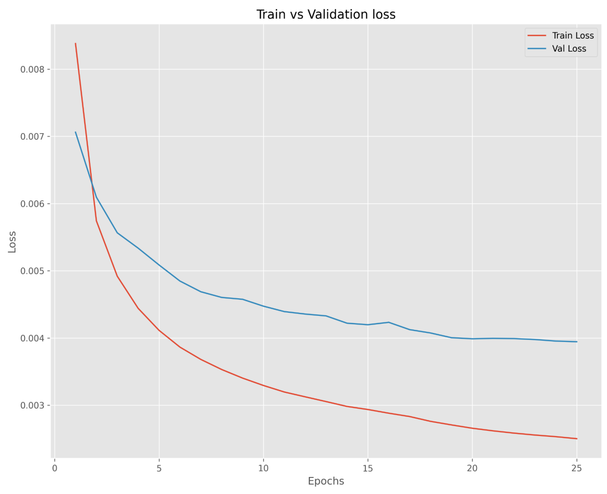


Figure 11: Train and Validation Loss

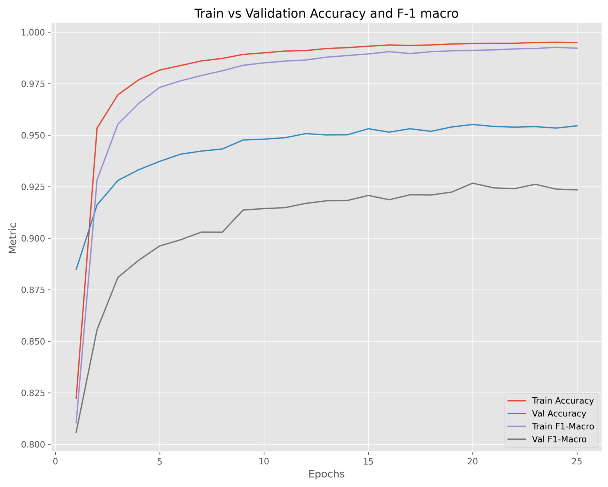


Figure 12: Train and Validation Accuracy and f1-score

We see that the train and validation losses decrease for all 25 epochs. It is a good sign because even though we have a very complex model with many parameters, there is no overfitting on the train data. We see that the train and validation losses decrease for all 25 epochs. It is a good sign because even though we have a very complex model with many parameters, there is no

overfitting on the train data. Additionally, we see that the validation and F1 scores level off after about ten epochs at 95% and 92.5%, respectively. It is the reason we set the number to ten for our comparison experiment.

6. Summary and Conclusion

This project focused on comparing the performance of a Capsule Neural Network and a Convolutional Neural Network in classifying the German Traffic Sign Data. CapsNet was fine tuned by changing batch size, epochs, learning rate, and momentum. It was also fine-tuned by changing activation functions, number of in channels, kernel size, and number of dynamic routing. The CNN was built using the same number of parameters and was fine tuned by changing the batch size, epoch, learning rate, and In the end, we saw that the CapsNet did not outperform the CNN, since they had similar accuracy and f1-scores.

After implementing LIME, we were able to see which regions were impacting the model prediction positively and negatively. The results were unexpected and can be further analyzed in future work.

Although we did not see improved results in the CapsNet, it is still important to keep them as a tool. It may be the case that CapsNets works more efficiently on certain types of dataset than others. For example, it has excelled in predicting healthcare images such as histological and retina images.

7. References

- https://pechyonkin.me/capsules-1/
- GitHub - jindongwang/Pytorch-CapsuleNet: An easy-to-follow Pytorch implementation of Hinton's Capsule Network
- https://christophm.github.io/interpretable-ml-book/lime.html
- https://proceedings.neurips.cc/paper/2017/file/2cad8fa47bbef282badbb8de5374b894-Paper.pdf
- https://eudl.eu/pdf/10.4108/eai.13-7-2018.158416
- https://arxiv.org/abs/1807.07559
- https://github.com/marcotcr/lime
- https://medium.com/dataman-in-ai/explain-your-model-with-lime-5a1a5867b423