RIS522 - Deep Learning and Classification Techniques
German Traffic Sign Recognition Benchmark
Term Project Report

1. Abstract

Image processing is one of the core areas of interest in Deep Learning. It has numerous different applications in a wide spectrum such as Image Correction, Sharpening, and Resolution Correction, Medical Image Processing and Computer Vision. [1] Computer vision includes tasks such as image classification, object localization and detection. Image classification, as can be understood from its name, is a way to classify images depending on what class they can possibly belong to. [2] For example, fed with a dataset of cat and dog images, an image classification model can be trained to discriminate these two and classify each. Various architectures of models based on Convolutional Neural Networks (CNNs) have been proposed by researchers, LeNet5, [3] AlexNet [4], ResNet [5] and VGG [6] are some examples of them. This project has been prepared as an instance of a LeNet5 application using the German traffic signs dataset [7]. I also would like to give the article on [8] credit since I got help with some of their implementational details during the preprocessing of the dataset.

2. Objective of the Project

My main goal to accomplish while implementing this project was to illustrate a multi-class classification task using one of the proposed architectures mentioned above. Furthermore, I wanted to see how changing or tuning the hyper parameters such as learning rate, type of optimizer etc. would affect the overall performance of prediction of the model. I did not add or remove layers since I wanted to measure the performance on a previously proposed model and changing the structure of the model was out of the scope of this project.

3. Methodology

After deciding to perform a classification task, I decided to seek a proper dataset for this goal. Although numerous different datasets are available on the internet, some of them were not sufficient, namely either they did not contain enough data, the data they contained was corrupted or they were not labeled properly. Finally I found the dataset of the German traffic signs and decided that it would be proper for my project. The main reasons of this decision were the followings: first of all, sizes of the images in the dataset were more or less around 30 x 30, which was beneficial since the resources I have were restricted and this made the process of reshaping all of the images to the same shape of 32 x 32 computationally efficient, as LeNet5 architecture requires. This was also a reason for the choice of LeNet5 as architecture. Moreover, the dataset contained 43 different classes, which in my opinion was a reasonable amount for the implementation of this project. Finally, the data was very well labeled, which is essential to perform such a task efficiently.

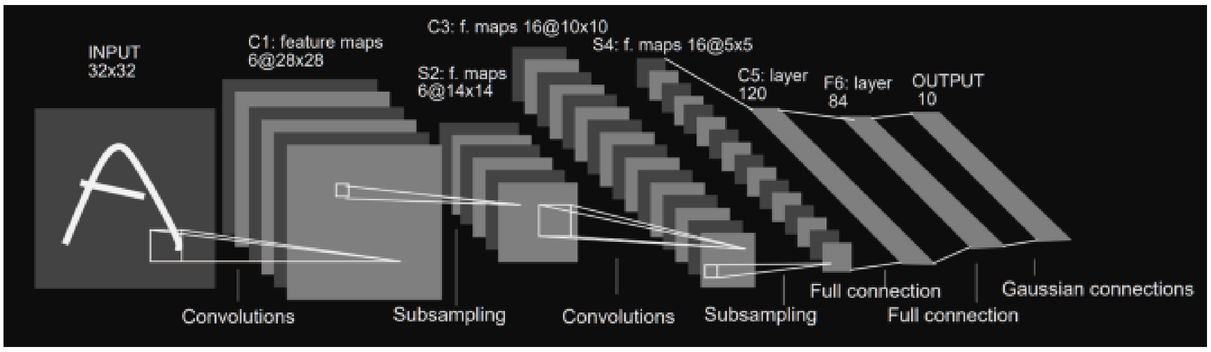


Fig. 1: Visual illustration of LeNet5 architecture. Different from this image, my inputs had a shape of 32x32x3 since I was working on RGB images and I used 43 neurons instead of 10 in the output layer, as there are 43 different possible classes in the dataset.

I also would like to perform the same task using AlexNet, ResNet and VGG models on the same dataset and compare the results, however these required larger sizes of images at the beginning (for example: 224 x 224 for AlexNet [4]), which made the process of reshaping the images quite heavy in terms of computational complexity and time. (Reshaping all the images in the dataset to the size of 224 x 224 required more than 30 gigabytes of memory). Applying these models to possibly another dataset could be the subject of another project.
After the decision on the dataset, it was time to preprocess the data, divide the data to training, validation and test sets, shuffle the data, build the code of the LeNet5 model, train the model with our data and obtain the results. Implementational details and acquired results are touched upon in the below sections.

4. Implementation

The project has been implemented using the Python programming language. At first, in order to preprocess the data, namely to create the arrays of each class instance, to connect the labels with those instances, to sketch graphs and visualize the data, NumPy, Pandas and Matplotlib libraries have been used. Distribution of the number of instances per class can be examined in the figure below.

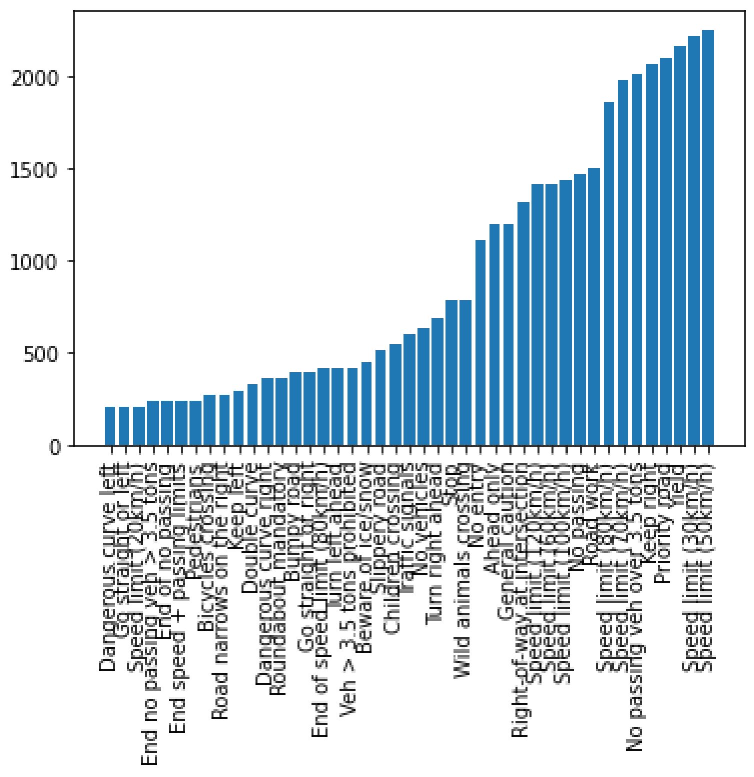
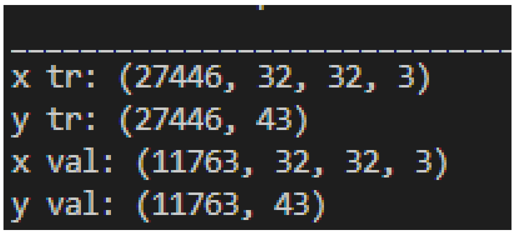


Fig. 2: Distribution of data per class

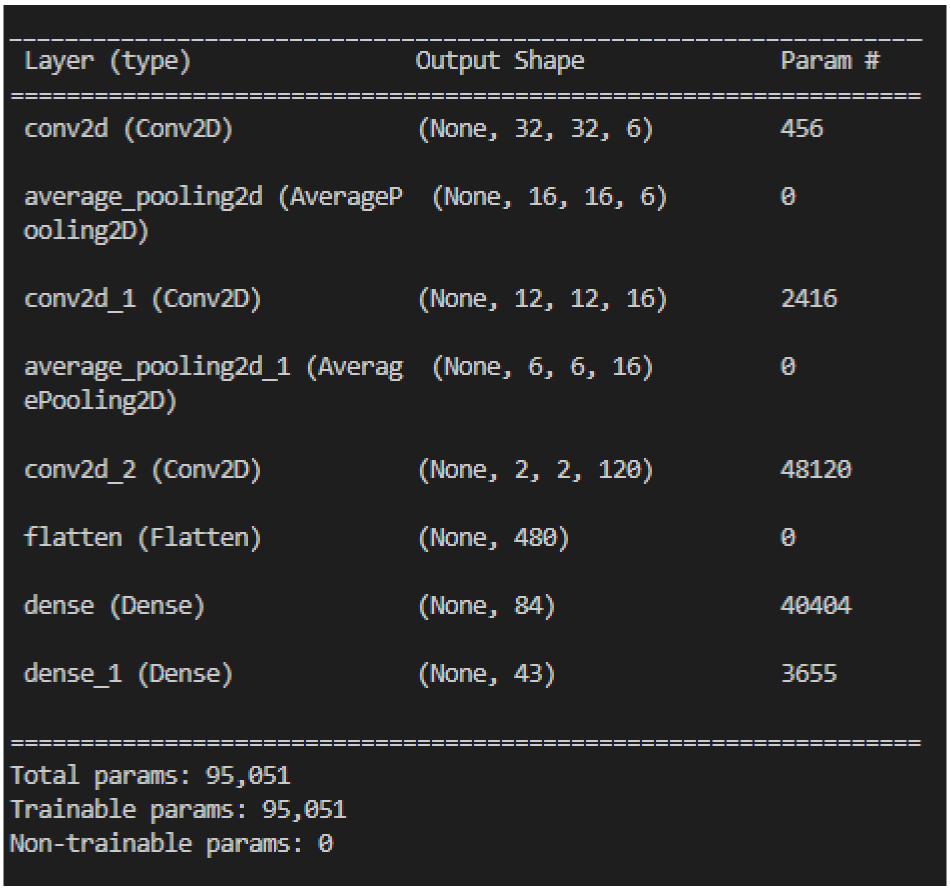
Afterwards, two NumPy arrays have been created, one for images and another for labels of those images. Each image in the array has been reshaped to the size of 32 x 32, which was required by the LeNet5 architecture. Since I was working on RGB images, each image after preprocessing had a shape of 32 x 32 x 3.
The sets created have been then shuffled in order to prevent the model from being trained by learning the order of the data, hence leading to bias. After shuffling, I split the data to training and validation sets of input and outputs in a way that 30% of the data to be used as validation data, and the rest as training data. This operation has been performed using the `train_test_split` method of the sklearn library. Then, I divided the sets into 255, in order to normalize the intensity of pixels to an interval between 0-1 [9] so that the model could converge faster and provide more accurate results. Finally, labels of each class number have been converted to one-hot vectors, and by loading and preprocessing the test data, the preprocessing phase of the project has been finished. The preprocessing of the test data has been achieved the same way as the training data, two NumPy arrays have been created for labels and images, the images have been reshaped and the set of images have been divided by 255. Below can be seen the shapes of the training and the validation sets.



```
------------------------------
x tr: (27446, 32, 32, 3)
y tr: (27446, 43)
x val: (11763, 32, 32, 3)
y val: (11763, 43)
```

Fig. 3: Shapes of training and validation sets.

After the preprocessing, it was time to create and compile the model and train the model with data. The model has been coded, compiled and fit using keras, using different optimizers and number of epochs. Different results obtained from these will be discussed in the results section. The summary of the model has been shown in the figure below.



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 6)         456       
                                                                 
 average_pooling2d (AverageP  (None, 16, 16, 6)        0         
 ooling2D)                                                       
                                                                 
 conv2d_1 (Conv2D)           (None, 12, 12, 16)        2416      
                                                                 
 average_pooling2d_1 (Averag  (None, 6, 6, 16)         0         
 ePooling2D)                                                     
                                                                 
 conv2d_2 (Conv2D)           (None, 2, 2, 120)         48120     
                                                                 
 flatten (Flatten)           (None, 480)               0         
                                                                 
 dense (Dense)               (None, 84)                40404     
                                                                 
 dense_1 (Dense)             (None, 43)                3655      
                                                                 
=================================================================
Total params: 95,051
Trainable params: 95,051
Non-trainable params: 0
```

Fig. 4: Summary of the model

An issue here that I faced was that the first layer should have output feature maps shaped (28 x 28), as shown in Fig. 1 and also according to the calculation using the given formula in Fig. 5, however the model summary does not meet this result although the model implementation has been done correctly.

$$n_{out} = \left\lfloor \frac{n_{in} + 2p - k}{s} \right\rfloor + 1$$

n_{in}:	number of input features
n_{out}:	number of output features
k:	convolution kernel size
p:	convolution padding size
s:	convolution stride size

Fig. 5: Formula for output size calculation. [10]

5. Results

As mentioned earlier, the model has been compiled and trained using the same data with different hyper parameters to illustrate some possible effects of hyperparameter tuning on training and test accuracy.

- At the beginning, the model was compiled using Adam Optimizer and trained through 10 epochs. The default value of learning rate is 0.001 for Adam Optimizer and it was not changed.
  The graph of training and validation losses and accuracies are given in Fig. 6 below. It was a good sign that both training and validation loss were decreasing while both training and validation accuracy were increasing. The model was trained several times under these conditions and the highest result obtained was 92.01%, while the result in general was changing between 91-92%.

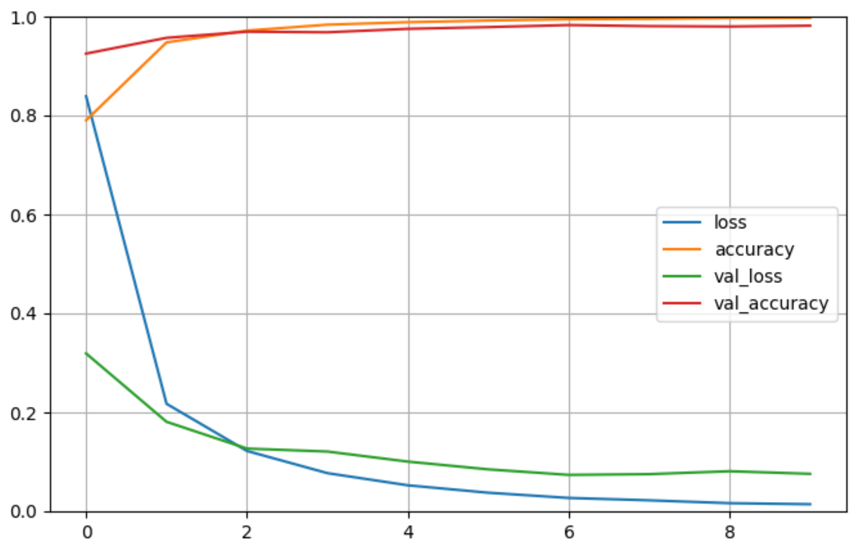
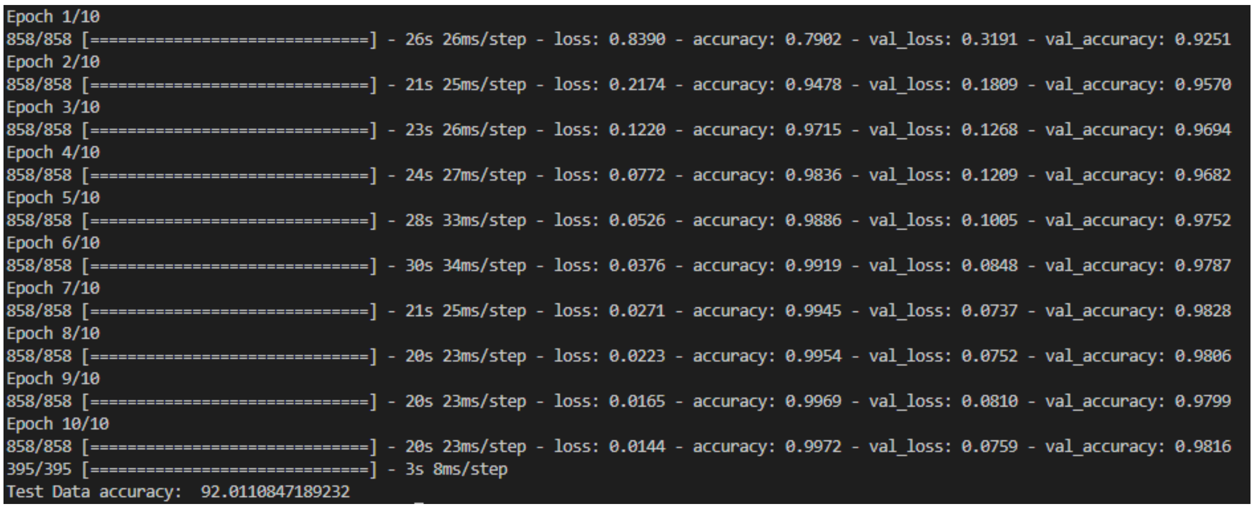


Fig. 6: Change in training and validation losses and accuracies through 10 epochs using Adam Optimizer.



```
Epoch 1/10
858/858 [==============================] - 26s 26ms/step - loss: 0.8390 - accuracy: 0.7902 - val_loss: 0.3191 - val_accuracy: 0.9251
Epoch 2/10
858/858 [==============================] - 21s 25ms/step - loss: 0.2174 - accuracy: 0.9478 - val_loss: 0.1809 - val_accuracy: 0.9570
Epoch 3/10
858/858 [==============================] - 23s 26ms/step - loss: 0.1220 - accuracy: 0.9715 - val_loss: 0.1268 - val_accuracy: 0.9694
Epoch 4/10
858/858 [==============================] - 24s 27ms/step - loss: 0.0772 - accuracy: 0.9836 - val_loss: 0.1209 - val_accuracy: 0.9682
Epoch 5/10
858/858 [==============================] - 28s 33ms/step - loss: 0.0526 - accuracy: 0.9886 - val_loss: 0.1005 - val_accuracy: 0.9752
Epoch 6/10
858/858 [==============================] - 30s 34ms/step - loss: 0.0376 - accuracy: 0.9919 - val_loss: 0.0848 - val_accuracy: 0.9787
Epoch 7/10
858/858 [==============================] - 21s 25ms/step - loss: 0.0271 - accuracy: 0.9945 - val_loss: 0.0737 - val_accuracy: 0.9828
Epoch 8/10
858/858 [==============================] - 20s 23ms/step - loss: 0.0223 - accuracy: 0.9954 - val_loss: 0.0752 - val_accuracy: 0.9806
Epoch 9/10
858/858 [==============================] - 20s 23ms/step - loss: 0.0165 - accuracy: 0.9969 - val_loss: 0.0810 - val_accuracy: 0.9799
Epoch 10/10
858/858 [==============================] - 20s 23ms/step - loss: 0.0144 - accuracy: 0.9972 - val_loss: 0.0759 - val_accuracy: 0.9816
395/395 [==============================] - 3s 8ms/step
Test Data accuracy:  92.0110847189232
```

Fig. 7: Loss and Accuracy values in each epoch and overall result of test data accuracy.

- Secondly, different optimizers have been tried on the model. One of them was Nadam, and the other one was RMSprop. Although they did not cause a dramatic change in the results, Adam optimizer was more successful. The test data accuracy was around 90-91%.

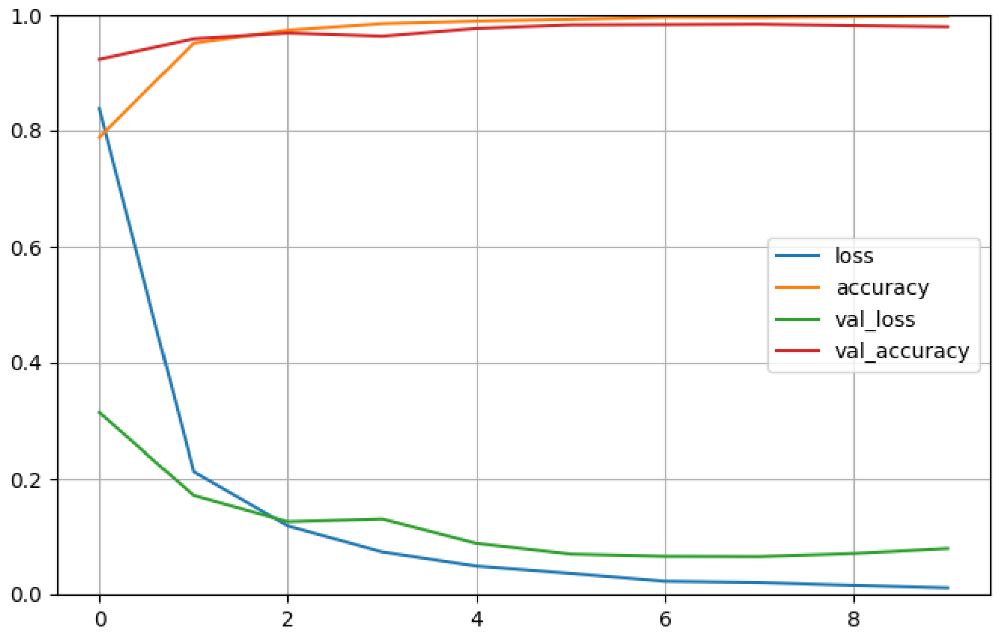


Fig. 8: Change in training and validation losses and accuracies through 10 epochs using Nadam Optimizer.

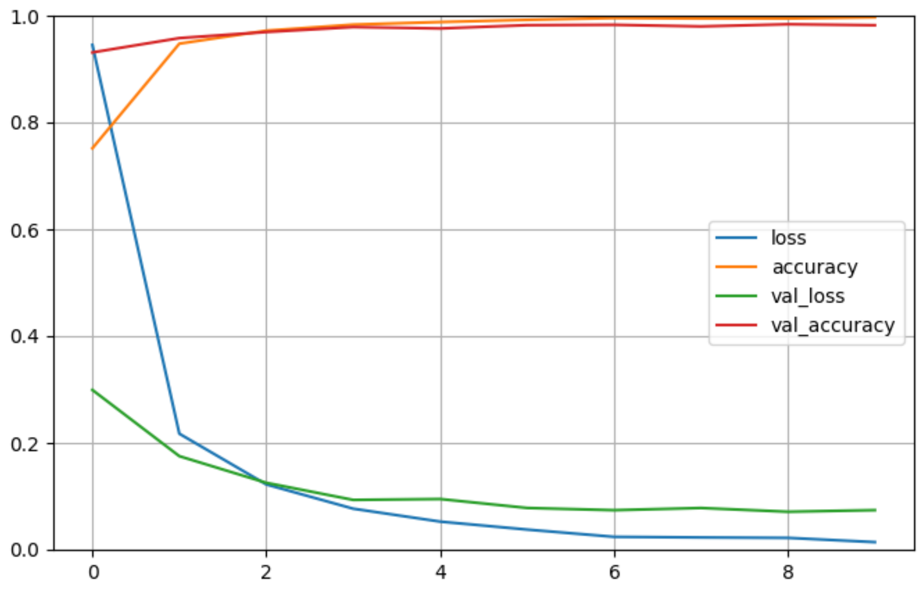


Fig. 9: Change in training and validation losses and accuracies through 10 epochs using RMSprop Optimizer.

- Afterwards, batch normalization has been added after each convolutional layer. The graph again did not change dramatically; however, although I was expecting an increase in the test data accuracy, it provided a result of 90.43%.
- It was tried to train the model without shuffling the data, and the test data accuracy dropped to a value around 89%, as expected.
- The learning rate has been tuned to the value of 0.01, which resulted in the model not being able to be trained. The accuracy values were close to zero in the graph while loss values were not even present. The test data accuracy was 5.70%.

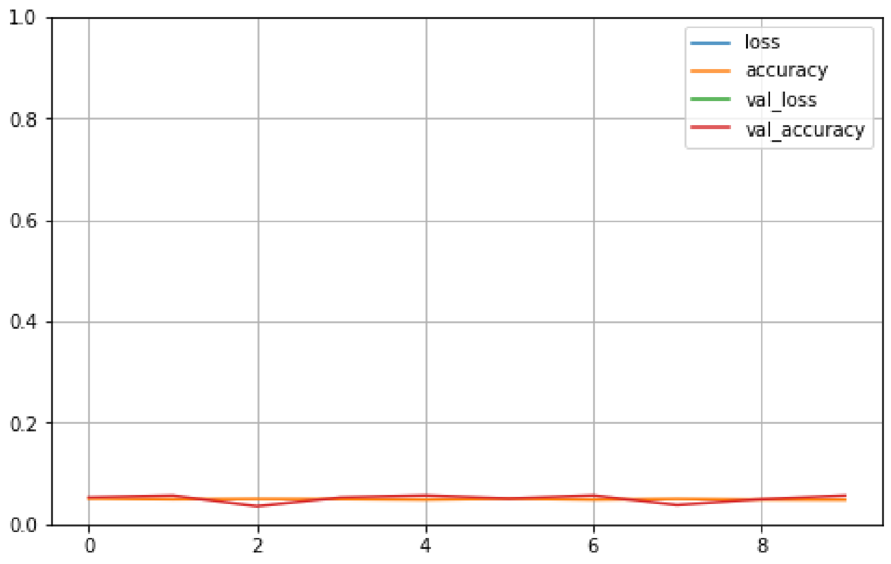


Fig. 10: Change in training and validation losses and accuracies through 10 epochs using Adam Optimizer, with learning rate = 0.01.

- Finally, the number of epochs were gradually increased:
- With 30 epochs, the model provided a more accurate result of 93.05%. The graph was as in the following figure, we can easily examine that the training accuracy reaches 100%, validation accuracy converges close to 100% and the loss reaches 0 approximately after 22-23 epochs.

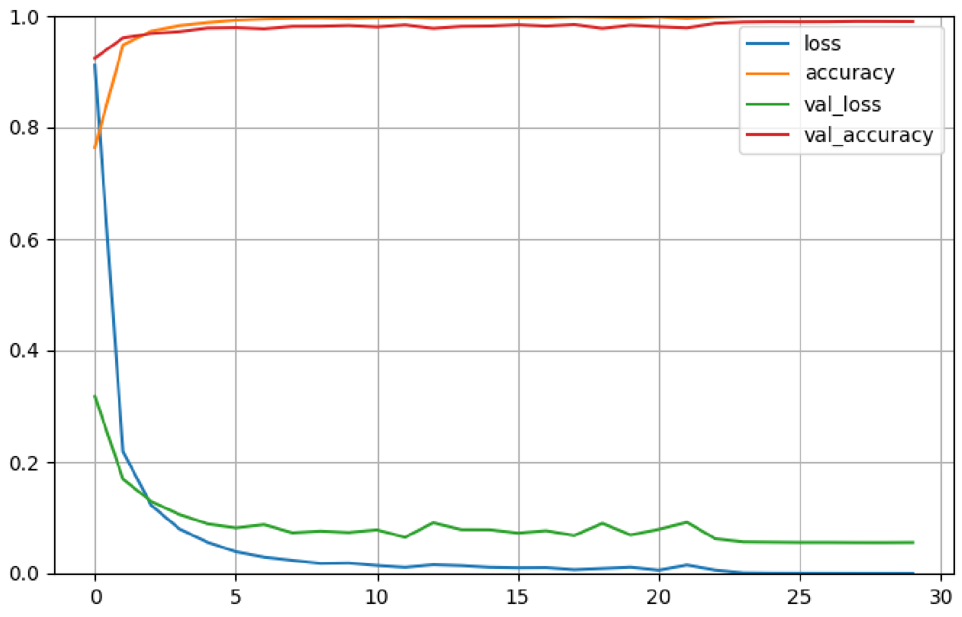


Fig. 11: Change in training and validation losses and accuracies through 30 epochs using Adam Optimizer.

- With 50 epochs, the result was even more accurate: 93.50%. However, this was not a significant increase in the accuracy; hence it can be considered as redundant to train the model for this long, since it is computationally much heavier and does not provide a significant difference. In the graph, that training loss again reaches 0 approximately after 22-23 epochs which shows us that training this model for approximately 25 epochs would be sufficient.

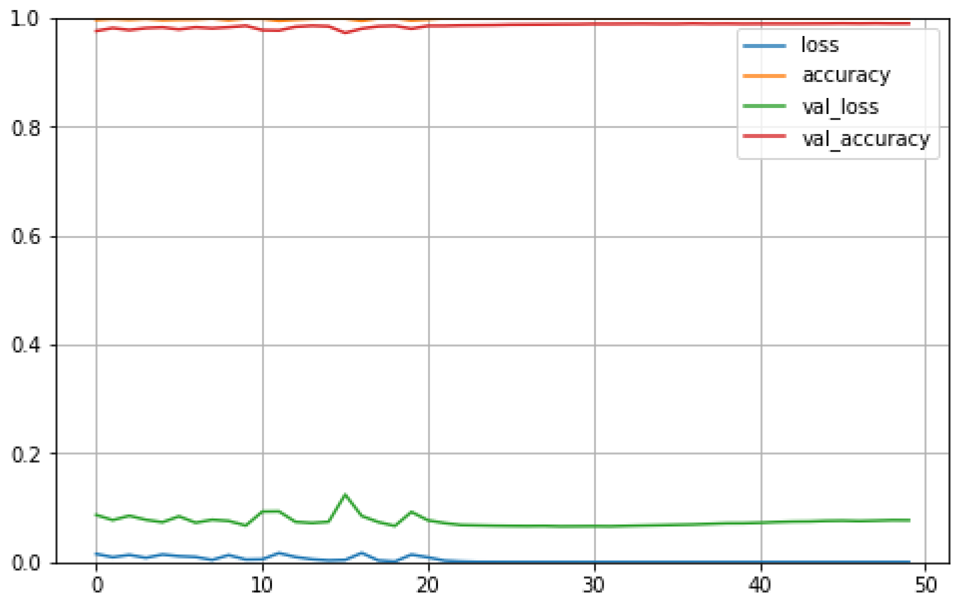


Fig. 12: Change in training and validation losses and accuracies through 50 epochs using Adam Optimizer.

6. Conclusion

Convolutional neural networks (CNNs) are widely used in image classification tasks and different architectures to obtain more accurate results have been developed by researchers, LeNet5 is one of them. The illustrations done in this project have shown that it is possible to increase the accuracy of predictions using different optimizers, shuffling the data or training for more epochs, mainly by tuning the hyper parameters. One other significant outcome is that using a more complex model or training it for a longer time does not necessarily provide significantly better results, what is important is that we should find or create the optimal solution which meets our requirements, also considering the computational efficiency.
Although the success of LeNet5 architecture has been scientifically proven, it is still possible to build different models with different layers and get better results, for example, the authors of [8] built their own model and they claim that they have a test accuracy of 98% in the same dataset we used, which is a significantly better outcome compared to what we obtained using LeNet5 architecture.
This project can be improved further and used as a base for other projects, for instance, an application that uses this model and detects the traffic signs instantly seen by a camera can be developed. This can be the subject of another project.

The code written for this project can be found on: https://github.com/mehmetcesitli/GTSRB---German-Traffic-Sign-Recognition-Benchmark

7. References

[1] Majumder, Prateek. “Digital Image Processing Applications | Image Processing in Python.” *Analytics Vidhya*, 28 May 2021, https://www.analyticsvidhya.com/blog/2021/05/digital-image-processing-real-life-applications-and-getting-started-in-python/. Accessed 7 June 2022.

[2] “What is Computer Vision?” *IBM*, https://www.ibm.com/topics/computer-vision. Accessed 7 June 2022.

[3] LeCun, Yann, et al. "Gradient-based learning applied to document recognition." *Proceedings of the IEEE* 86.11 (1998): 2278-2324.

[4] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." *Advances in neural information processing systems* 25 (2012).

[5] He, Kaiming, et al. "Deep residual learning for image recognition." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016.

[6] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." *arXiv preprint arXiv:1409.1556* (2014).

[7] “GTSRB - German Traffic Sign Recognition Benchmark.” *Kaggle*, https://www.kaggle.com/datasets/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign. Accessed 7 June 2022.

[8] Sharma, Shivank. “GTSRB - CNN (98% Test Accuracy).” *Kaggle*, https://www.kaggle.com/code/shivank856/gtsrb-cnn-98-test-accuracy. Accessed 7 June 2022

[9] Alake, Richmond. “Understanding and Implementing LeNet-5 CNN Architecture (Deep Learning).” *Towards Data Science*, 25 June 2020, https://towardsdatascience.com/understanding-and-implementing-lenet-5-cnn-architecture-deep-learning-a2d531ebc342. Accessed 7 June 2022.

[10] Dang Ha The Hien. “A guide to receptive field arithmetic for Convolutional Neural Networks.” *ML Review*, 5 April 2017, https://blog.mlreview.com/a-guide-to-receptive-field-arithmetic-for-convolutional-neural-networks-e0f514068807. Accessed 7 June 2022.