Developing a custom Convolutional Neural Network (CNN) for Image Classification using the Fashion MNIST Dataset

a1772276

Abstract

The study focuses on building a custom convolutional neural network (CNN) from scratch to explore and analyse its effectiveness in image classification. A major part of this study is to optimize the hyper-parameters and produce a CNN that compares to the performance of LeNet-5, AlexNet and ResNet on the Fashion-MNIST dataset. The custom CNN, when added with additional convolutional and dropout layers and used with a batch-size of 32, outperformed all the pre-trained models.

1. Introduction

The context of this report is to explore the use of Convolutional Neural Networks (CNN) for image classification.

Convolutional Neural Networks (CNNs) are a specialized class of neural networks designed for processing grid-like data, particularly images. They consist of multiple layers, including convolutional layers that extract features, pooling layers that reduce dimensionality, and fully connected layers that perform classification tasks.

Convolutional layers use small filters to slide over the input image, performing mathematical operations called convolutions, on the pixel-data, to capture important patterns such as edges and textures. This process creates feature maps that highlight these characteristics, allowing the network to learn hierarchical representations from simple features in earlier layers to complex objects in deeper layers[1][2][3].

Pooling layers, on the other hand, simplify these feature maps by reducing their size through operations like max pooling or average pooling, which help decrease computation and make the model more robust to variations in input [1][2][3].

The fully connected layer (FC) connects every neuron from the previous layer to its neurons, enabling the model to make predictions based on the features extracted by earlier layers. The final output layer typically uses an activation function like softmax or sigmoid to convert the output into probabilities for each class in classification tasks. The softmax function is particularly useful for multi-class classification, as it ensures that the sum of the output probabilities equals one, allowing for a clear interpretation of which class the model predicts as most likely based on the learned features. This combination of convolutional, pooling, and fully connected layers makes CNNs highly effective for tasks such as image recognition and classification, enabling advancements in fields like computer vision and artificial intelligence [3][4].

This study involves building a custom CNN from scratch, comparing it to existing CNN architectures such as AlexNet, LeNet-5 and ResNet and optimizing it by changing selected parameters based on the nature of the baseline architecture's training and validation – loss & accuracy curves.

The core of aim of this study is to perform various experiments on the baseline CNN by changing the different parameters in it such as the regularization values, the learning rate, optimizers, kernel or filters sizes, batch-sizes and, but not limited to using techniques like gradient-clipping and early-stopping. We will view the effect of changing these variables on the CNN's training and validation, loss & accuracy curves, and therefore, it's overall performance on the test-set and new unseen data.

1.1. Dataset

The dataset used for this experiment on understanding the use of CNNs for image classification is the widely known – Fashion MNIST dataset.

The Fashion MNIST dataset, originally developed in 2017, was published by researchers of an e-commerce company, Zalando Research [5]. The dataset contains 70,000, grayscale images of fashion products, with dimensions, 28x28; which are spread across 10 classification categories.

The originally published version of the dataset has no validation set. However, as the majority of this study will involve experimenting with various hyper-parameters and versions of a CNN – we decided to split the original training set into a train and validation (20% of training) set to use for experimentation. Overall – the dataset used in this study included a training set – 48,000 images; validation set – 12,000 images, and a test set – 10,000 images. The dataset included 10 classes – *[T-shirt, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle-boot]*. The dataset has an equal number of samples for each class and their visual distinction makes it an effective dataset for evaluation how CNNs learn hierarchical features.

The Fashion-MNIST dataset serves as a benchmark for evaluating and analysing various image classification machine learning (ML) algorithms. It serves as a direct alternative to the original MNIST dataset including images of hand-written digits as models were reaching high-level performance accuracies on it. Therefore, the Fashion-MNIST serves as a more challenging and diverse dataset, making it a benchmark for numerous ML algorithms [6].

1.2. Background and Competitors

Since Fashion-MNIST serves as the industry benchmark, it has already been used on several pre-trained models to evaluate their individual performance and understand their strengths and weaknesses.

For the purpose of this study, AlexNet, LeNet-5 and ResNet were chosen as the competitors to compare the CNN built for this study to. Since Fashion-MNIST includes all grayscale images, any part of the above 3-competing architectures that involved processed 3 RGB channels was modified to 1-channel.

AlexNet has demonstrated 92% accuracy on Fashion-MNIST, with an even higher accuracy when transfer learning was used[7][8]. Though numbers were not mentioned, a comparative study by Ganesh et al. (2023), revealed that ResNet had outperformed both LeNet-5 and AlexNet [9]. Another study revealed that particularly, ResNet-20 had achieved an accuracy of 93% [10]. Finally, LeNet-5 had the lowest accuracy of 88% as it is a less modern architecture and was originally built for digit recognition [10]. The training and validation loss & accuracy graphs, including the classification reports for the above architectures are presented in Appendix 3 and 4, respectively. These will be discussed further in the Experiment and Analysis section.

This study will aim to build a custom CNN architecture that will match the performance of the above pre-trained models, if not outperform them through experimentation and hyper-parameter optimization.

2. Method Description

This section will describe the steps I followed to build a CNN from scratch, evaluate its performance, compare it to the performance of pre-trained models and how I hyper-parameter optimized it.

1. I started by installing and importing all the necessary libraries – mainly `Tensorflow, Keras, Numpy,`

`Scikit Learn` and any other that I came across during the the development.

2. I loaded the original Fashion-MNIST dataset into train and test sets using the line of code:

```
(x_train,   y_train),   (x_test,
y_test)                        =
fashion_mnist.load_data()
```

3. The `x_train` and `y_train` variables were split by 20% in order to create an `x_val` and `y_val`, validation sets.
4. The data was then normalized using mean and standard deviation values. I used mean and std values, instead of `255`, to normalize the data as these would centre the values around zero which will be better for the convergence of the gradient descent and help the model learn better.

```
x_train  =  (x_train  -  0.2860)  /
0.3530
x_val = (x_val - 0.2860) / 0.3530
x_test  =  (x_test  -  0.2860)  /
0.3530
```

5. The data was then reshaped to include its colour channel dimension. The x-values were reshaped from `(28,28)` → `(28,28,1)` to make it appropriate for the keras model input.
6. Once the data was prepared, a basic CNN-architecture was built , it is demonstrated in in Appendix 1. The custom CNN architecture included 2 convolutional layers, 2 max-pooling layers, 1 fully-connected layer and 1 output layer. It was compiled using the `'adam'` optimizer' with a `'sparse_categorical_loss'` loss function and `accuracy` as the key-performance metric.
7. The baseline model was then fit to the training and validation sets and produced the accuracy and loss curves shown in Appendix 2.
8. This process was then repeated with AlexNet, LeNet-5 and ResNet. The training and validation curves for the pre-trained models are shown in Appendix 3. Although the architectures of these pre-trained models were retained – some elements such as channel dimensions of had to be changed. For example, since AlexNet was originally made for RGB Images of size `224x224` and 3 channels, the input had to be resized to `(28, 28, 1)`.

9. After looking at the training and validation curves, I made a list of all the hyper-parameters (shown in the Experiment and Analysis section) that I wanted to experiment one. The key feature of the plots that led to me choosing the relevant hyper-parameters, was that the validation curve was quite erratic. The baseline plots, shown in Appendix 2 produced similar results for training and validation curves. However, despite being close together, their fluctuating nature suggested that the learning-rate is too high and indicated potential overfitting.

10. For every hyper-parameter I chose, I modified the CNN using various values for the parameter, based on the training and validation loss & accuracy curves. The procedure for each experiment will be outlined in the Experiment and Analysis section.

    I then picked the version of the CNN with the highest validation accuracy but also performed this judgement in relation to which graph had curves that were least erratic. I then used this CNN on the test-dataset to evaluate the model's performance on unseen data.

11. The architecture of the best version of the optimised CNN will be described in the experiment and analysis section.

3. Method Implementation

The Fashion-MNIST.ipynb file outlines loading the dataset and all the baseline models. All other files are the individual experiments for the various hyper-parameters. The title of each file and the sub-headings in the notebooks describes its experiment. The experiments were separated into different notebooks although this meant loading the dataset repeatedly, for the ease of running multiple experiments at the same time. Every experiment included plotting training and validation curves to understand the model's performance.

All experiments were performed on JupyterHub. Therefore, there is not commit history on GitHub, however if required, revision history can be retrieved from JupyterHub.

- `Code Repository:` [GitHub Repository]

4. Experiments and Analysis

All experiments were run on the train and validation datasets using 50 epochs, 'adam' optimizer and the 'sparse_categorical_crossentropy' as the loss function.

The key feature of the model I wanted to address through hyper-parameter optimization is overfitting. The nature of the training and validation curves of the baseline CNN were quite erratic, indicating potential overfitting of the model to the training data.

Hence, all the focus of hyper-parameter optimization was to find a version of the CNN model which will produce stable curves. The curves for each experiment are provided in the relevant Jupyter Notebook.

4.1. Regularization

- L2=0.01:

```
Final training accuracy: 0.9590
Final validation accuracy: 0.9072
Final training loss: 0.2225
Final validation loss: 0.3764
```

- L2=0.1:

```
Final training accuracy: 0.8888
Final validation accuracy: 0.8840
Final training loss: 0.4234
Final validation loss: 0.4489
```

Due to the erratic nature of the validation accuracy and loss curves from the baseline CNN model, I decided to add L2 regularisation to each convolutional layer in my architecture.

L2 adds a penalty to the loss function based on the square of the coefficients' magnitudes. This penalty discourages the model from assigning excessively large weights to any individual feature, effectively reducing model complexity and promoting smoother, more generalized predictions. By constraining the size of the coefficients, L2 regularization minimizes the risk of capturing noise in the training data, leading to improved performance on unseen data and more stable validation metrics.

I experimented with the values 0.01 and 0.1. L2 regularizer, when set to 0.1, produced smoother curves compared to 0.01. Although this stabilized the loss curves, the validation accuracy curve was still quite erratic as it can be seen in Figure 1. This is because the larger the co-efficient, the stronger the penalty. Hence, 0.1, causing stronger penalties for overfitting, resulted in smaller loss function and a smoother curve.

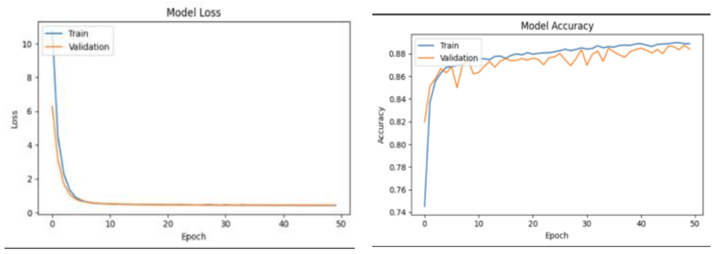


Figure 1: Validation and Training curves for L2 Regularizer=0.1

Although erratic, the proximity of the two curves suggests that the model is not drastically overfitting anymore, but the instability in accuracy is a sign that the model requires further adjustments or finetuning other hyperparameters to enhance generalization and stabilize validation performance.

4.2. Adding layers(dropout/convolutional)

- Batch-Size=16:
```
Final training accuracy: 0.9013
Final validation accuracy: 0.9040
Final training loss: 0.2887
Final validation loss: 0.2971
```
- Batch Size=32:
```
Final training accuracy: 0.9113
Final validation accuracy: 0.9115
Final training loss: 0.2512
Final validation loss: 0.2749
```

The second experiment I performed on my CNN model was making it more complex by adding 1 additional convolutional layer and 3 dropout layers in total. The architecture of this CNN is given in Appendix 5.

Dropout layers generally work by randomly setting a fraction of the input units to zero during training, which helps prevent overfitting by forcing the network to learn redundant representations and reducing reliance on specific neurons. This randomness encourages the model to generalize better to unseen data.

On the other hand, adding more convolutional layers gives the network more opportunities to extract patterns and features. It allows the network to learn increasingly complex features from the input data at different levels of abstraction. This would've been helpful for the model to capture patterns like buttons or zippers in the context of this dataset.
Each layer can capture different patterns, leading to improved feature extraction and representation. Together, these techniques were used to improve the model's robustness so that it can achieve higher accuracy and better generalization on validation datasets.

In addition, this version of the CNN was fitted with both 16 and 32 as batch sizes, separately. As this dataset has quite small images, a smaller batch size will be better suited as it typically leads to faster convergence due to more frequent
updates to the model's parameters, allowing for better exploration of the loss landscape and potentially escaping local minima. As predicted, the model with the batch-size 16 produced smoother curves than that with batch-size 32. However, the model with the larger batch size has a slightly better final validation accuracy and loss values. For this model, a learning rate of 0.006, found through an extensive grid-search was used.

4.3.Grid-Search for Kernels and Filters

- Best Validation Accuracy: 0.8619
- Best Hyperparameters: {'model__filters_1': 32, 'model__filters_2': 64, 'model__kernel_size': (5, 5), 'model__learning_rate': 0.001}

A grid-search was performed on the baseline-CNN with 2 convolutional, 2 max-pooling, 1 fully-connected layer and 1 output layer. The grid-search checked what filter, kernel or input sizes will be the most suitable for each layer of the CNN. After running through 50-epochs (while eliminating every parameter combination trial if it produced smaller accuracy than the best one found yet), it restored the weights of the epoch that produced the best results and used the version of the model to plot the curves.

The validation and training curves were very close in proximity. The validation accuracy curve although fluctuating initially, stabilized itself towards the end. Conversely, the validation loss curve was still quite unstable towards the end of the 50 epochs.

Next-time this grid search will be performed on the version of the CNN model derived in 4.2 as it had more opportunities to retrieve features and for pattern recognition.

4.4. Early Stopping

- Validation Loss:

```
Final training accuracy: 0.8535
Final validation accuracy: 0.8593
Final training loss: 0.4039
Final validation loss: 0.4217
```

- Validation Accuracy:

```
Final training accuracy: 0.8697
Final validation accuracy: 0.8591
Final training loss: 0.3641
Final validation loss: 0.4805
```

I used early-stopping techniques with both validation loss and validation accuracy, in separate models, as the criteria. Early stopping was implemented so that if there was no improvement in loss or accuracy after 5 epochs – the model will restore the weights from the epoch that produced the best accuracy or lowest loss value (separate models). On average, both models, stopped around 15-18 epochs and produced similar curves. They were less unstable with both training and validation curves being close to each other.

However, instead of early-stopping, I believe I should have let the CNN model run for all 50 epochs and restore the best weights instead as I feel that this experiment meant that the model stopped too early, preventing it from learning the data properly. Alternatively, next time, I can increase the tolerance to 10 epochs before it stops early.

4.5. Analysis

Other parameters that could have been explore include learning-rates and different optimizer values. However, during an extensive grid search, it could be seen that 'adam' was the best suited optimizer and learning rate was most efficient around 0.0001-0.0006.

Overall – the model found in 4.2, with additional dropout and convolutional layers, fit with 32 batch size had the highest validation accuracy. The classification report of this model, when evaluated in the test set is given in Appendix 4, along with the classification reports of the pre-trained models.

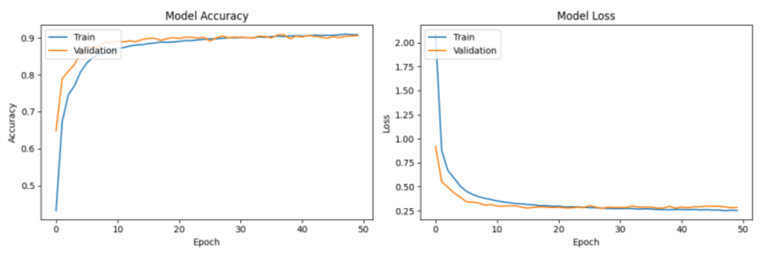


Figure 2: Training and Validation curves of the model with the highest Validation Accuracy

The validation and train curves for ResNet and AlexNet were the most erratic, while LeNet-5 produced smoother curves in comparison. All the pre-trained models produced training and validation curves that were not the closest in proximity. These can be found in Appendix 3.

Based on the classification-reported provided in the appendix, it can be seen that the custom CNN architecture built from scratch – after hyper-parameter optimization has matched, if not outperformed the individual class prediction accuracy of all the competing pre-trained models.

5. Reflection

My key-learning from this study is that general model hyper-parameter optimization includes checking for and experimenting with things like learning rate, activation functions, optimizers etc. However, from my experiments, I have learnt that for CNNs, additional parameters hold more importance. These may include dropout layers, choosing the right filter, kernel or batch sizes for your convolutional layers.

Regularization techniques and ensure that your dataset is well normalized and shaped is also quite important. If the pixel data is not normalized around 0-1, it can skew the results by over ~45%.

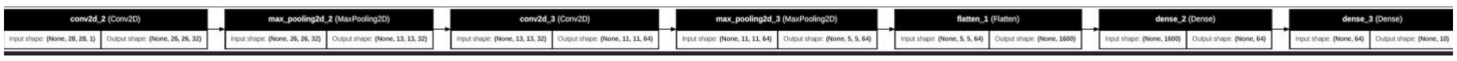
From viewing and comparing the classification report of my custom CNN to the ones of the pre-trained models. I have also learnt that pre-trained models will not always suit your use-case the best and sometimes building your own models or networks from scratch and customizing them to your data-requirements will produce better results. For example, AlexNet was primarily made for RGB images, while LeNet-5 is an older model, better suited for classifying digits. Similarly, ResNet has an architecture such that, it may be too complex for a simple dataset like Fashion-MNIST, making it difficult to capture local and simpler patterns in the data.

References

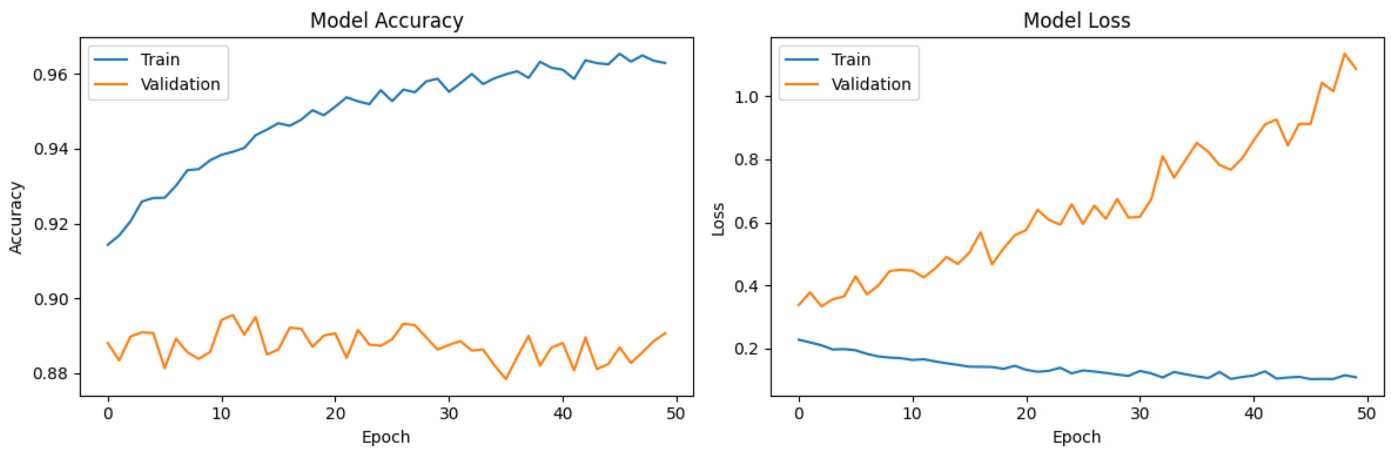
1. Towards Data Science, 2021. Convolutional Neural Networks Explained. Available at: https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939?gi=da15e48830ff [Accessed 2 Nov. 2024].
2. PyImageSearch, 2021. Convolutional Neural Networks (CNNs) and Layer Types. Available at: https://pyimagesearch.com/2021/05/14/convolutional-neural-networks-cnns-and-layer-types/ [Accessed 30 Oct. 2024].
3. Towards AI, 2021. Introduction to Pooling Layers in CNN. Available at: https://towardsai.net/p/l/introduction-to-pooling-layers-in-cnn [Accessed 2 Nov. 2024].
4. UpGrad, 2021. Basic CNN Architecture. Available at: https://www.upgrad.com/blog/basic-cnn-architecture/ [Accessed 10 Oct. 2024].
5. Zalando Research, n.d. Fashion-MNIST. Available at: https://github.com/zalandoresearch/fashion-mnist [Accessed 1 Nov. 2024].
6. Papers with Code, n.d. Fashion-MNIST Dataset. Available at: https://paperswithcode.com/dataset/fashion-mnist [Accessed 2 Nov. 2024].
7. Yoss, A. and Harrison, C., 2021. Transfer Learning using Pre-Trained AlexNet Model and Fashion-MNIST. Available at: https://towardsdatascience.com/transfer-learning-using-pre-trained-alexnet-model-and-fashion-mnist-43898c2966fb [Accessed 1 Nov. 2024].
8. Tiiktak, 2021. Fashion-MNIST with AlexNet in PyTorch: 92% Accuracy. Available at: https://www.kaggle.com/code/tiiktak/fashion-mnist-with-alexnet-in-pytorch-92-accuracy [Accessed 4 Nov. 2024].
9. K. Ganesh, K. Haripriya, K. Indrasena Reddy, T. Pavan Raj and Prof. V. Sravanthi, 2023. Fashion-MNIST Classification Using CNN. International Research Journal of Modernization in Engineering Technology and Science, 5(7). Available at: https://www.irjmets.com/uploadedfiles/paper/issue_7_july_2023/42990/final/fin_irjmets1692082330.pdf [Accessed 1 Nov. 2024].
10. Iowa State University, n.d. Title of the Document. Available at: https://dr.lib.iastate.edu/server/api/core/bitstreams/9f1b5107-0ff4-41bb-a4e5-e6b1bc555d74/content [Accessed 2 Nov. 2024].

Appendix

- Appendix 1: Architecture of Baseline CNN:

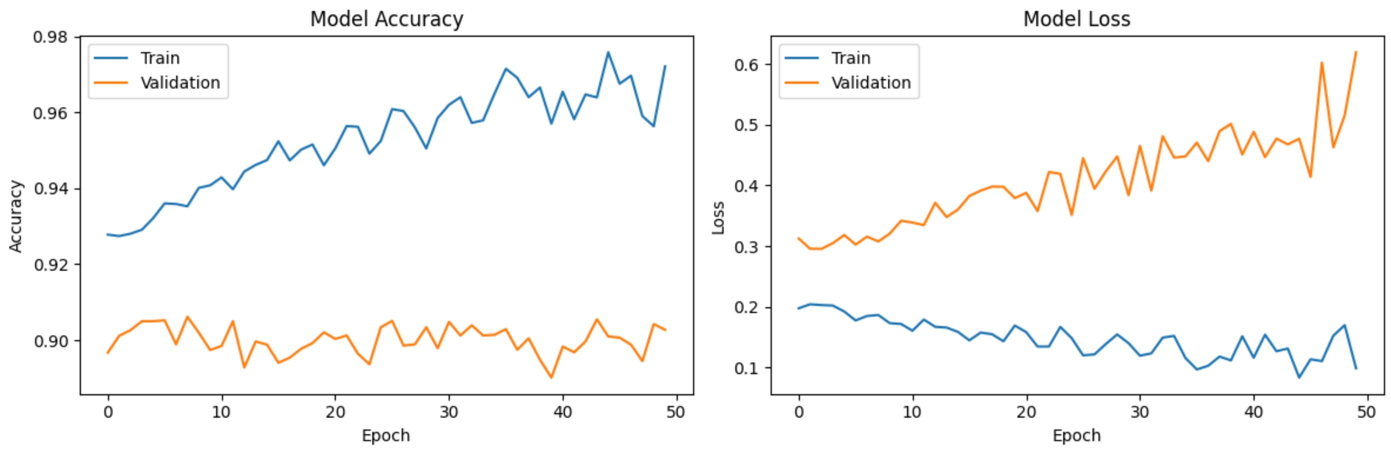

conv2d_2 (Conv2D)
Input shape: (None, 28, 28, 1)
Output shape: (None, 26, 26, 32)
max_pooling2d_2 (MaxPooling2D)
Input shape: (None, 26, 26, 32)
Output shape: (None, 13, 13, 32)
conv2d_3 (Conv2D)
Input shape: (None, 13, 13, 32)
Output shape: (None, 11, 11, 64)
max_pooling2d_3 (MaxPooling2D)
Input shape: (None, 11, 11, 64)
Output shape: (None, 5, 5, 64)
flatten_1 (Flatten)
Input shape: (None, 5, 5, 64)
Output shape: (None, 1600)
dense_2 (Dense)
Input shape: (None, 1600)
Output shape: (None, 64)
dense_3 (Dense)
Input shape: (None, 64)
Output shape: (None, 10)

- Appendix 2: Training and Validation Curves for Baseline CNN:

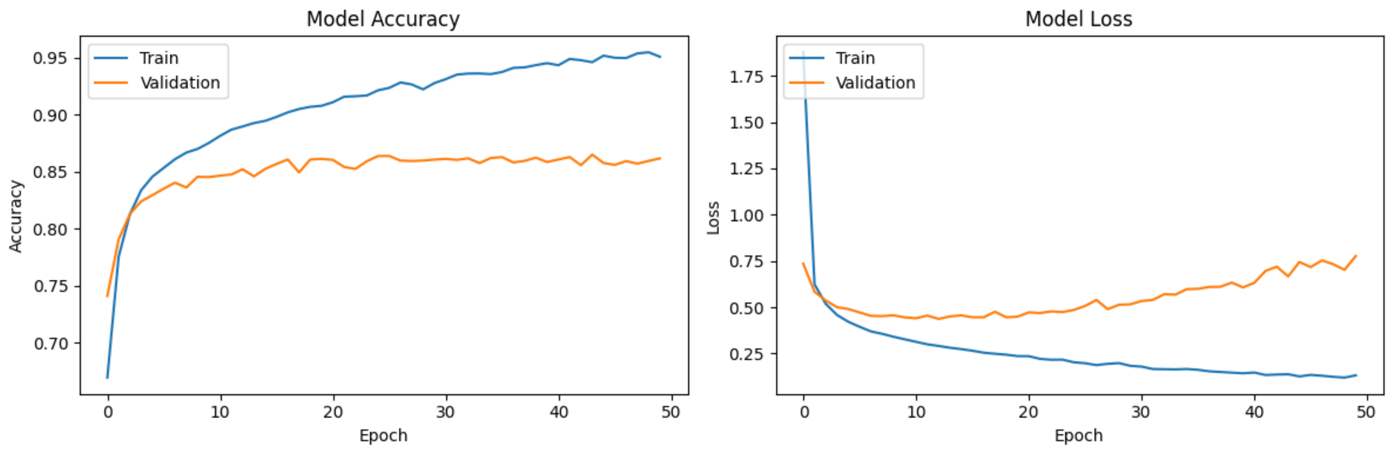


- Appendix 3: Training and Validation Graphs for pre-trained models:

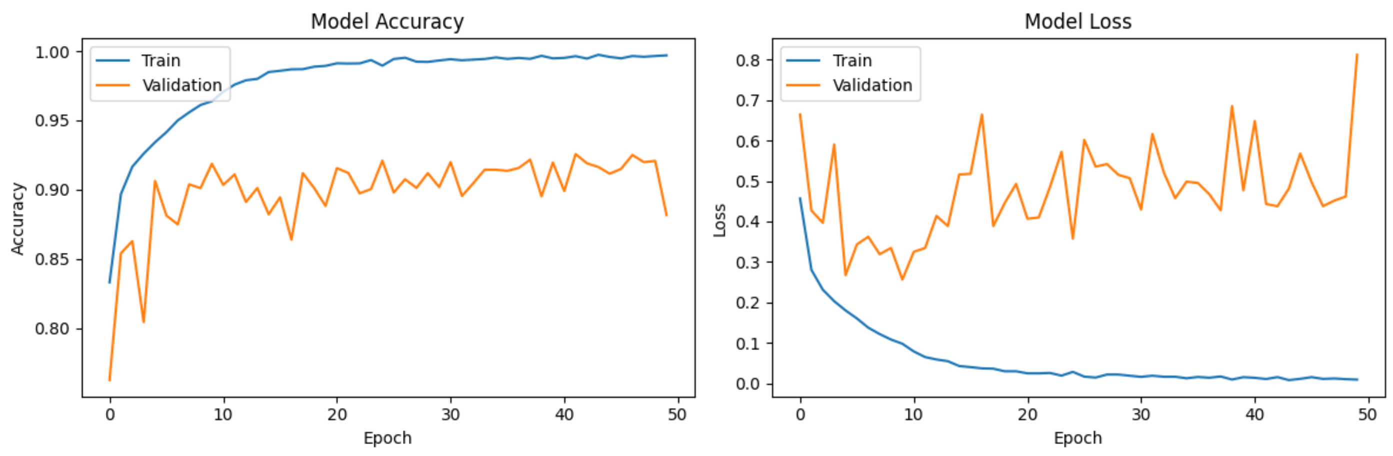
  - AlexNet:



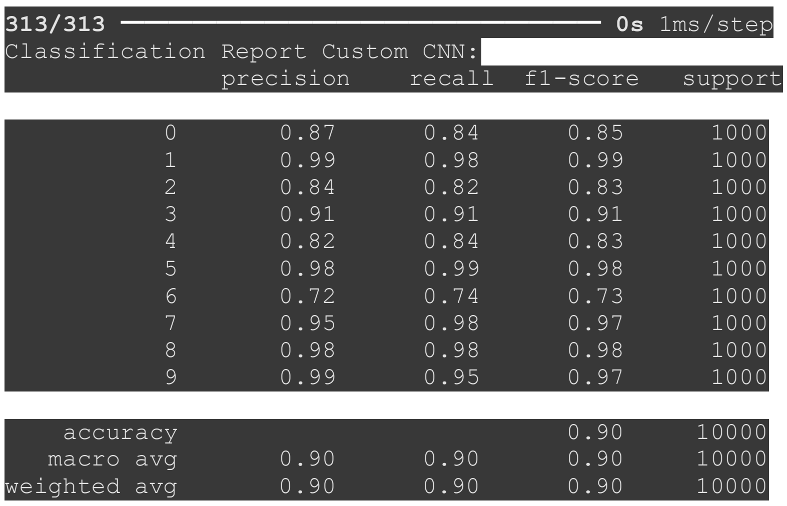
  - LeNet-5:



- ResNet:



- Appendix 4: Classification Report of Final CNN Model + Pretrained Models when used on the test-dataset.
  - Custom CNN:

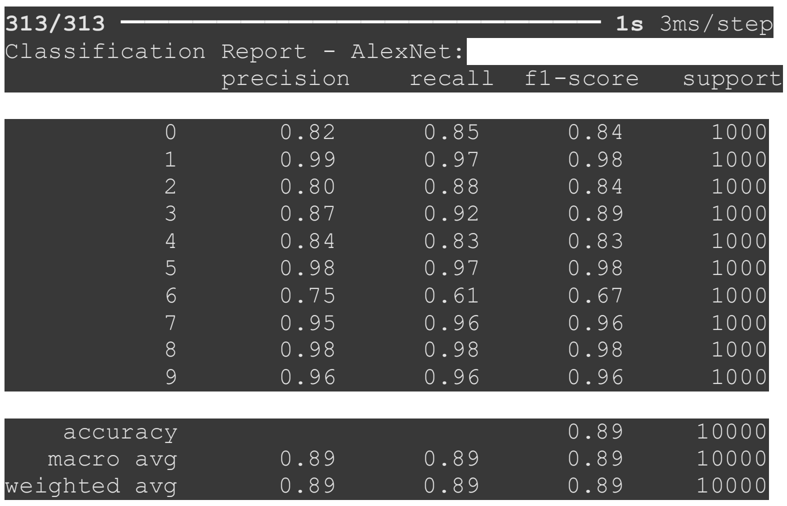


```
313/313 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
Classification Report Custom CNN:
              precision    recall  f1-score   support

           0       0.87      0.84      0.85      1000
           1       0.99      0.98      0.99      1000
           2       0.84      0.82      0.83      1000
           3       0.91      0.91      0.91      1000
           4       0.82      0.84      0.83      1000
           5       0.98      0.99      0.98      1000
           6       0.72      0.74      0.73      1000
           7       0.95      0.98      0.97      1000
           8       0.98      0.98      0.98      1000
           9       0.99      0.95      0.97      1000

    accuracy                           0.90     10000
   macro avg       0.90      0.90      0.90     10000
weighted avg       0.90      0.90      0.90     10000
```

  - AlexNet:



```
313/313 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step
Classification Report - AlexNet:
              precision    recall  f1-score   support

           0       0.82      0.85      0.84      1000
           1       0.99      0.97      0.98      1000
           2       0.80      0.88      0.84      1000
           3       0.87      0.92      0.89      1000
           4       0.84      0.83      0.83      1000
           5       0.98      0.97      0.98      1000
           6       0.75      0.61      0.67      1000
           7       0.95      0.96      0.96      1000
           8       0.98      0.98      0.98      1000
           9       0.96      0.96      0.96      1000

    accuracy                           0.89     10000
   macro avg       0.89      0.89      0.89     10000
weighted avg       0.89      0.89      0.89     10000
```

  - LeNet-5:

```
313/313 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
Classification Report - LeNet5:
              precision    recall  f1-score   support

           0       0.75      0.82      0.79      1000
           1       0.98      0.96      0.97      1000
           2       0.77      0.78      0.77      1000
           3       0.88      0.86      0.87      1000
           4       0.74      0.79      0.76      1000
           5       0.96      0.95      0.96      1000
           6       0.67      0.55      0.60      1000
           7       0.93      0.95      0.94      1000
           8       0.92      0.96      0.94      1000
           9       0.95      0.95      0.95      1000

    accuracy                           0.86     10000
   macro avg       0.85      0.86      0.85     10000
weighted avg       0.85      0.86      0.85     10000
```

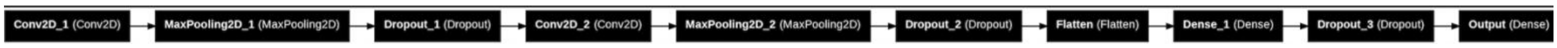
- ResNet:

```
313/313 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step
Classification Report - ResNet:
              precision    recall  f1-score   support

           0       0.77      0.93      0.84      1000
           1       1.00      0.94      0.97      1000
           2       0.71      0.94      0.81      1000
           3       0.89      0.89      0.89      1000
           4       0.99      0.47      0.64      1000
           5       0.99      0.95      0.97      1000
           6       0.69      0.75      0.72      1000
           7       0.91      0.99      0.95      1000
           8       0.99      0.96      0.98      1000
           9       0.99      0.94      0.96      1000

    accuracy                           0.88     10000
   macro avg       0.89      0.88      0.87     10000
weighted avg       0.89      0.88      0.87     10000
```

- <u>Appendix 5:</u> Final Optimized CNN Architecture


Conv2D_1 (Conv2D)
MaxPooling2D_1 (MaxPooling2D)
Dropout_1 (Dropout)
Conv2D_2 (Conv2D)
MaxPooling2D_2 (MaxPooling2D)
Dropout_2 (Dropout)
Flatten (Flatten)
Dense_1 (Dense)
Dropout_3 (Dropout)
Output (Dense)