AIN422 ASSIGNMENT 2

Hüseyin Yiğit Ülker

21993092

Department of AI Engineering

Hacettepe University

Ankara , Turkey

b21993092@hacettepe.edu.tr

1. Introduction

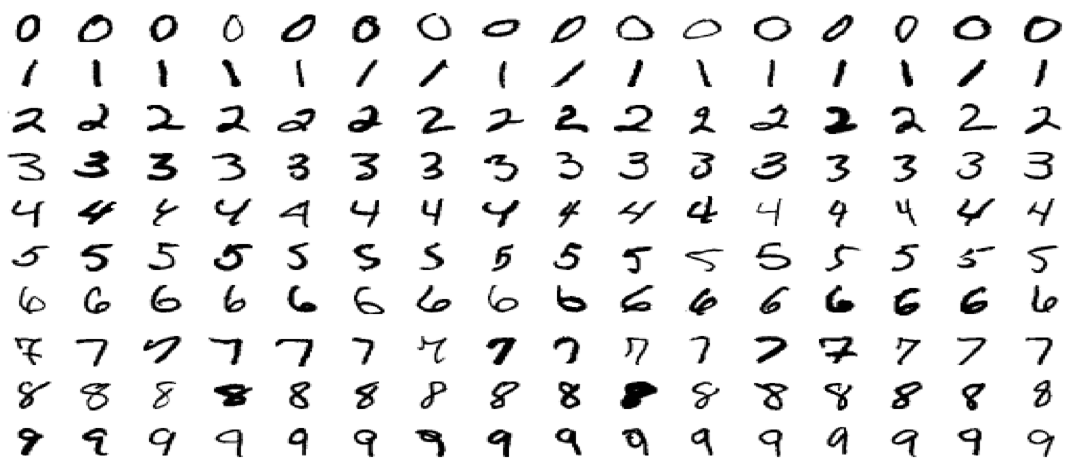
Image classification is a process of assigning a label or class to an image based on its visual content. It is a fundamental task in the field of computer vision and has numerous applications such as object recognition, facial recognition, and scene understanding. In this assignment, we will perform handwritten digit classification by developing CNN models on the MNIST data set. At the same time, we will create CNN models with different hyperparameters and examine the effect of the change of these parameters on the result.

2. Information about the MNIST Dataset

The MNIST dataset is a popular benchmark dataset in the field of machine learning and computer vision. It is a collection of 70,000 handwritten digits that are labeled with their corresponding numeric values, and is often used for training and testing image classification algorithms.

The dataset is divided into two parts: a training set of 60,000 images and a test set of 10,000 images. The images are grayscale and are normalized to fit in a 28x28 pixel bounding box, with each pixel represented as an integer value between 0 and 255. The labels range from 0 to 9, indicating the numeric value of the handwritten digit in each image.

The MNIST dataset has been used extensively in research and is often used as a benchmark to compare the performance of different machine learning algorithms, such as neural networks and support vector machines. It is widely regarded as a standard dataset for benchmarking image classification algorithms due to its simplicity, popularity, and well-defined task.



Sample images from MNIST test dataset (https://upload.wikimedia.org/wikipedia/commons/thumb/2/27/MnistExamples.png/320px-MnistExamples.png)

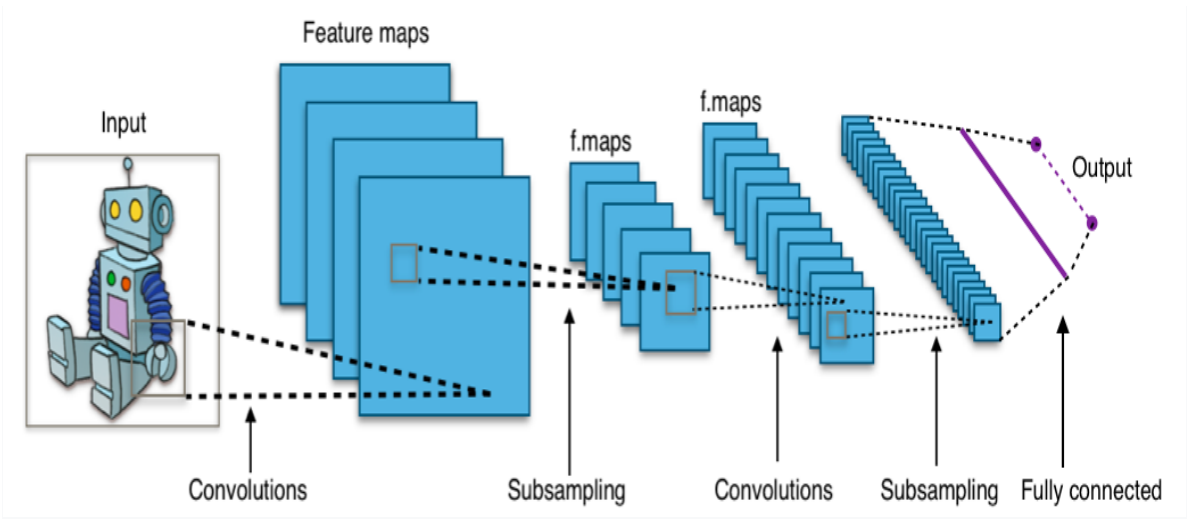
3. Convolutional Neural Networks (CNN) Approach

Convolutional Neural Networks (CNNs) are a type of deep neural network that are commonly used in image classification tasks. They are designed to automatically learn and extract meaningful features from input images by applying a series of convolutional and pooling layers.

The convolutional layers in a CNN use a set of filters to scan the input image and detect local features such as edges, corners, and other patterns. The filters are typically small in size and are slid across the image, producing a feature map that highlights the locations of the detected features.

The pooling layers in a CNN are used to reduce the size of the feature maps and help the network become more robust to small changes in the input image. The most common type of pooling used in CNNs is max pooling, where the maximum value within a small region of the feature map is selected and retained while the other values are discarded.

After several convolutional and pooling layers, the output is flattened into a vector and passed through one or more fully connected layers to produce a final classification output.



Typical CNN architecture
(https://upload.wikimedia.org/wikipedia/commons/6/63/Typical_cnn.png)

4. Experiment

In this assignment, I trained a total of 8 models to classify 10 numbers in the MNIST dataset. Although these models are generally similar to each other, some hyperparameters have different features such as the quantity of the data set used in training and the use of pre-trained models.

4.1. Model_1 (Baseline Model)

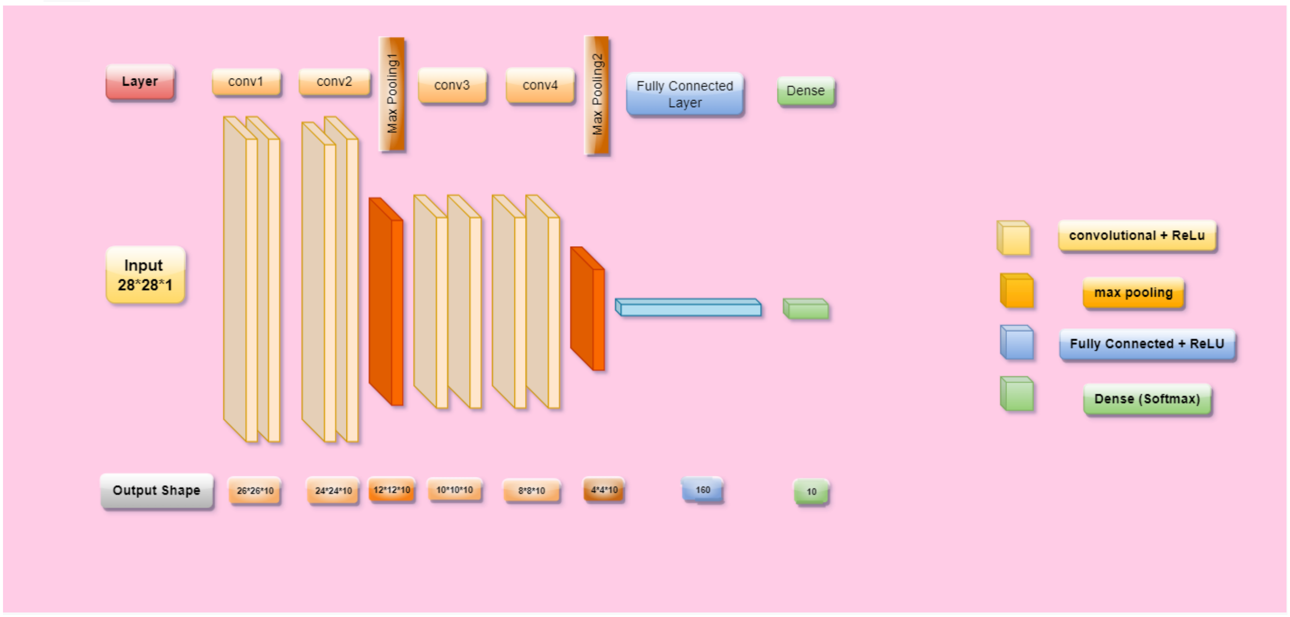
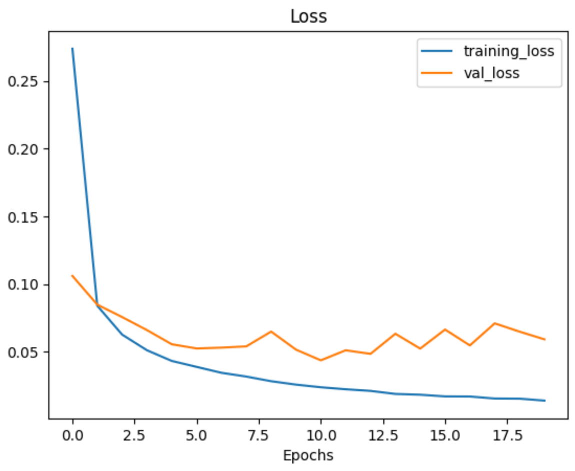


Image by Author

I determined the CNN model, which you see as the diagram above, as the baseline model, and I created the next models based on this model with some minor changes. If we go into the details of this model, the convolutional layers consist of 10 filters of 3 * 3 size. I used ReLu as the activation function. I used Adam as the optimizer with the learning rate value default (0.001).

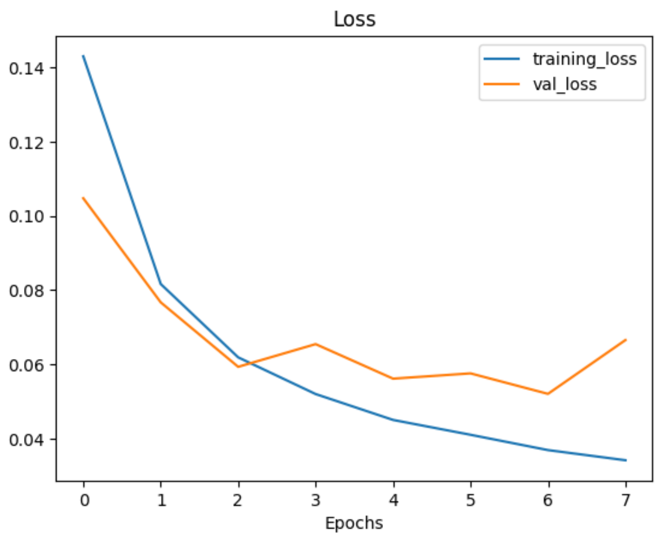


When we look at the loss epoch chart on the side, we see that our model is overfit after about the 8th epoch. In other words, we can say that while the model learns in the first 8 epochs, it memorizes the train dataset in the remaining 12 epochs. To solve this, we can make the epoch number 8, we can enlarge the data set with data augmentation, we can use a simpler model.

I used 8 epochs when training other models as 20 epochs cause overfits. (to save time and energy)

4.2. Model_2 (epoch = 8)

The second model shares identical hyperparameter values with the baseline model, with the exception of the training duration. In contrast to the baseline model which utilizes a 20-epoch training regimen, the second model is only trained over a period of 8 epochs.

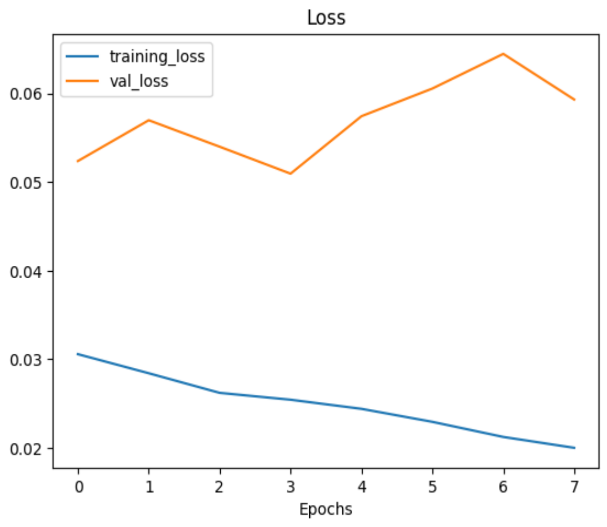


By training the model for fewer epochs, we have achieved a satisfactory level of accuracy and approximate loss value, while also ensuring that the model has a high degree of generalization.

Note: I set the epoch value to 8 for the remaining models since there wasn't a significant difference between epoch 20 and 8. The baseline model can also be considered as model_2.

4.3. Model_3 (batch = 64)

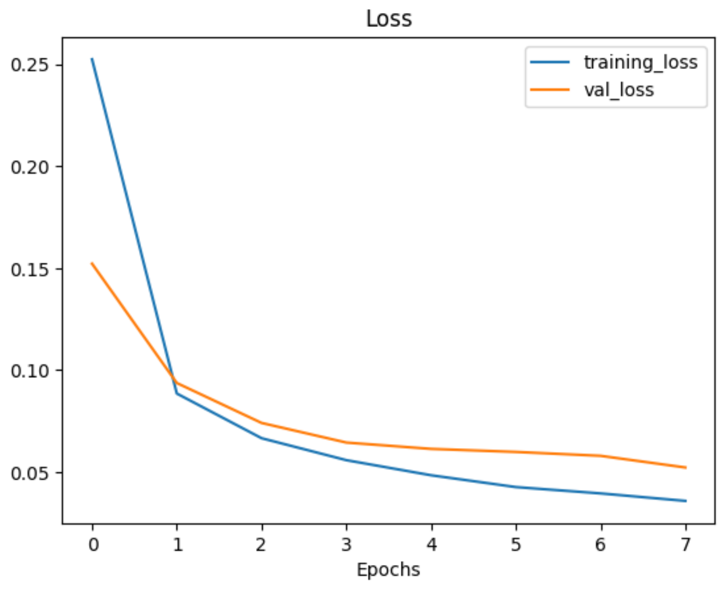
Model_3 differs from the baseline model solely in its batch size value, which is set to 64 rather than 32**.**Increasing batch size from 32 to 64 can have a significant impact on the training process of deep neural networks. Using a larger batch size can result in faster convergence during training, as larger batches allow for more efficient computation and better utilization of hardware resources such as GPUs. This can reduce the time taken to train models, making it a useful optimization for larger datasets and more complex architectures.



After analyzing the graph, it was observed that using a batch size of 64 resulted in a faster decrease in training loss values and validation loss values.

4.4. Model_4 (lr = 0.01)

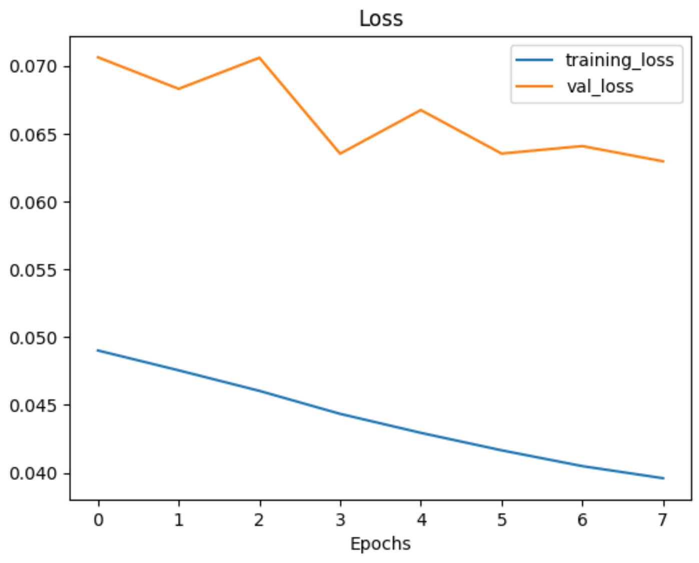
In Model_4, I adjusted the learning rate hyperparameter by setting it to lr = 0.01 for the Adam optimizer, instead of the default value of lr = 0.001.



After adjusting the learning rate from 0.001 to 0.01, I observed that the model converged more efficiently. Specifically, the loss value decreased from 0.0471 to 0.0347, while the accuracy value increased from 0.9855 to 0.9894. Overall, these results indicate that using a learning rate of 0.01 resulted in a more successful model.

4.5. Model_5 (activation function = sigmoid)

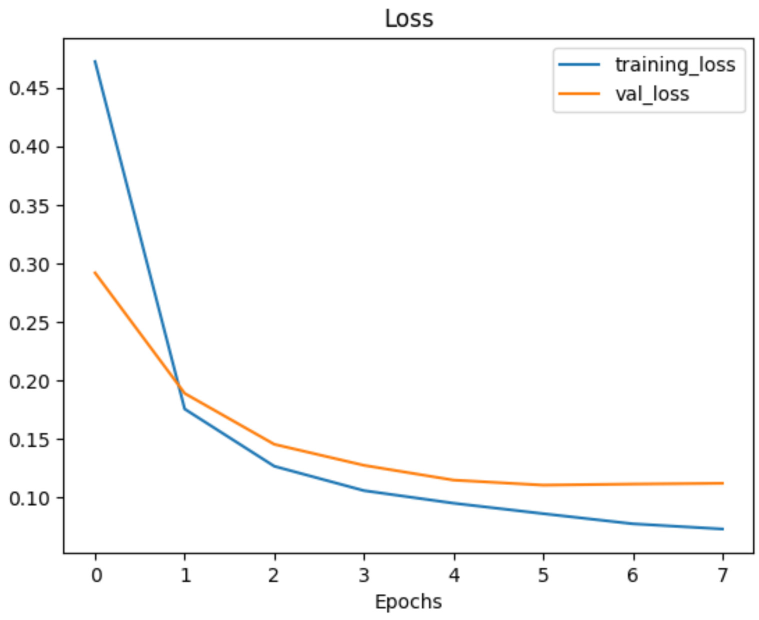
In this assignment section, I compared activation functions for model_5 against the baseline model. Instead of using the ReLU activation function as in the baseline model, I used sigmoid. Although ReLU is known to give better results in theory, I wanted to test this assumption by trying sigmoid as an alternative activation function.



Upon analyzing the graph, it can be observed that while the loss value is similar to the baseline model. I expected it to be worse

4.6. Model_6 (stride = 2)

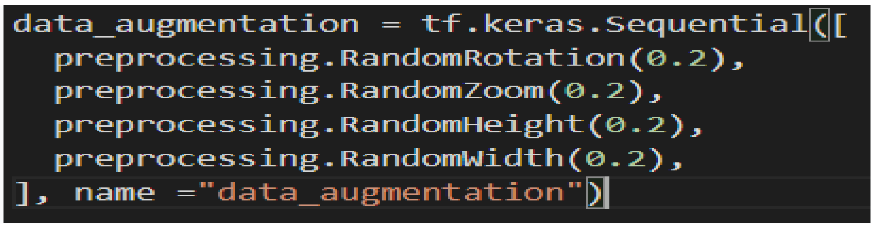
In this assignment section, I modified the stride value from the default (1,1) to (2,2) and evaluated the performance of the model.



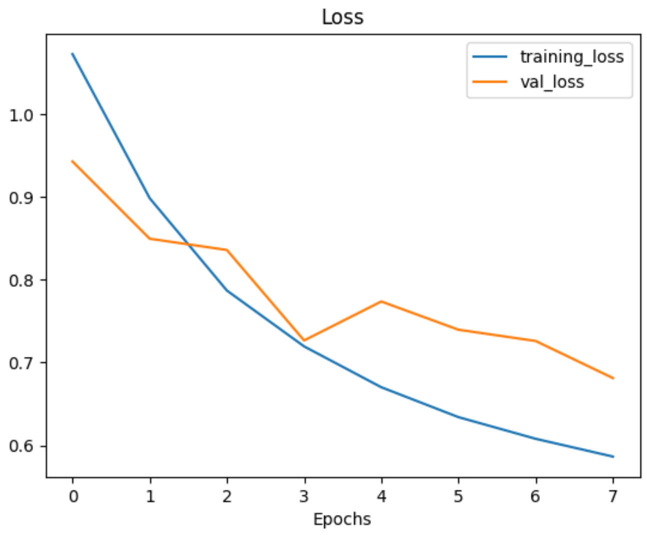
Modifying the stride value resulted in an increase in the loss and a decrease in the overall performance of the model.

4.7. Model_7 (trained augmented data)

In this section, rather than adjusting the model's hyperparameters, I performed data augmentation operations to manipulate the dataset.



```
data_augmentation = tf.keras.Sequential([
  preprocessing.RandomRotation(0.2),
  preprocessing.RandomZoom(0.2),
  preprocessing.RandomHeight(0.2),
  preprocessing.RandomWidth(0.2),
], name ="data_augmentation")
```



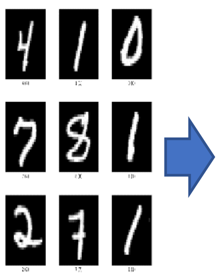
When we examined the success of the model, I saw that the loss value was quite high (compared to other models).

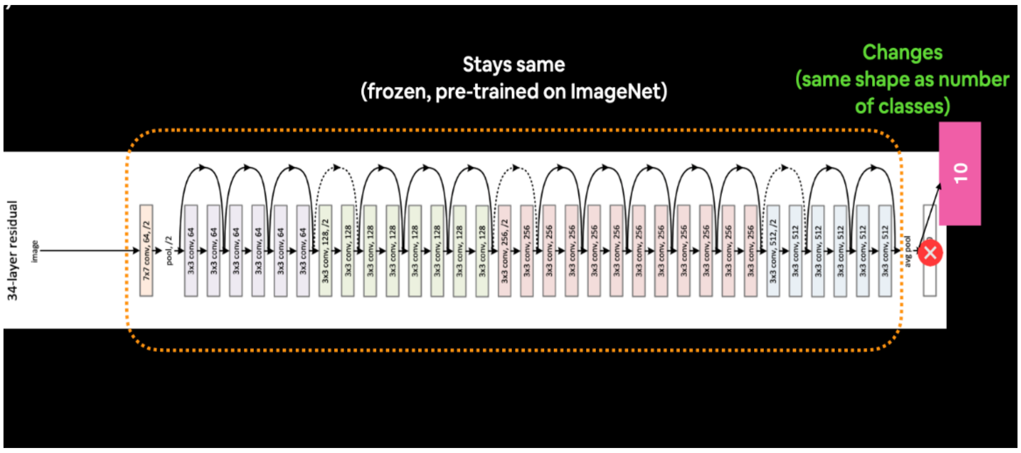
4.8. Model_8 (using transfer learning)

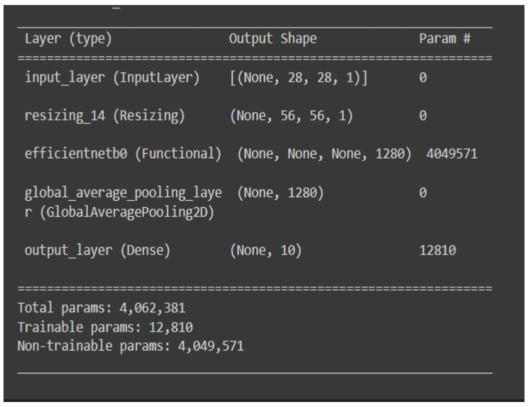
In this section, I utilized the EfficientNetB0 model, pre-trained with the Imagenet dataset, for classification purposes. Rather than fine-tuning the model, I conducted feature extraction by freezing the model's weights. I then added a dense layer to the end of the pre-trained model, updating the weights for 10 classes instead of the original 1000 classes.

Model

Input

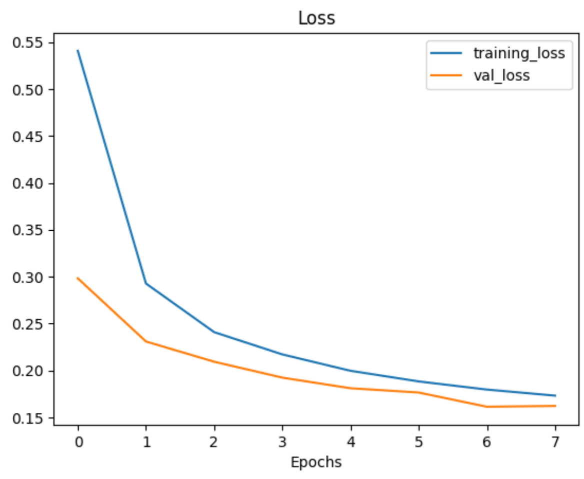



Stays same
(frozen, pre-trained on ImageNet)
Changes
(same shape as number of classes)
10
34-layer residual
image
avg pool



```
 Layer (type)                Output Shape              Param #
=================================================================
 input_layer (InputLayer)    [(None, 28, 28, 1)]       0

 resizing_14 (Resizing)      (None, 56, 56, 1)         0

 efficientnetb0 (Functional)  (None, None, None, 1280)  4049571

 global_average_pooling_laye  (None, 1280)             0
 r (GlobalAveragePooling2D)

 output_layer (Dense)        (None, 10)                12810

=================================================================
Total params: 4,062,381
Trainable params: 12,810
Non-trainable params: 4,049,571
_________________________________________________________________
```

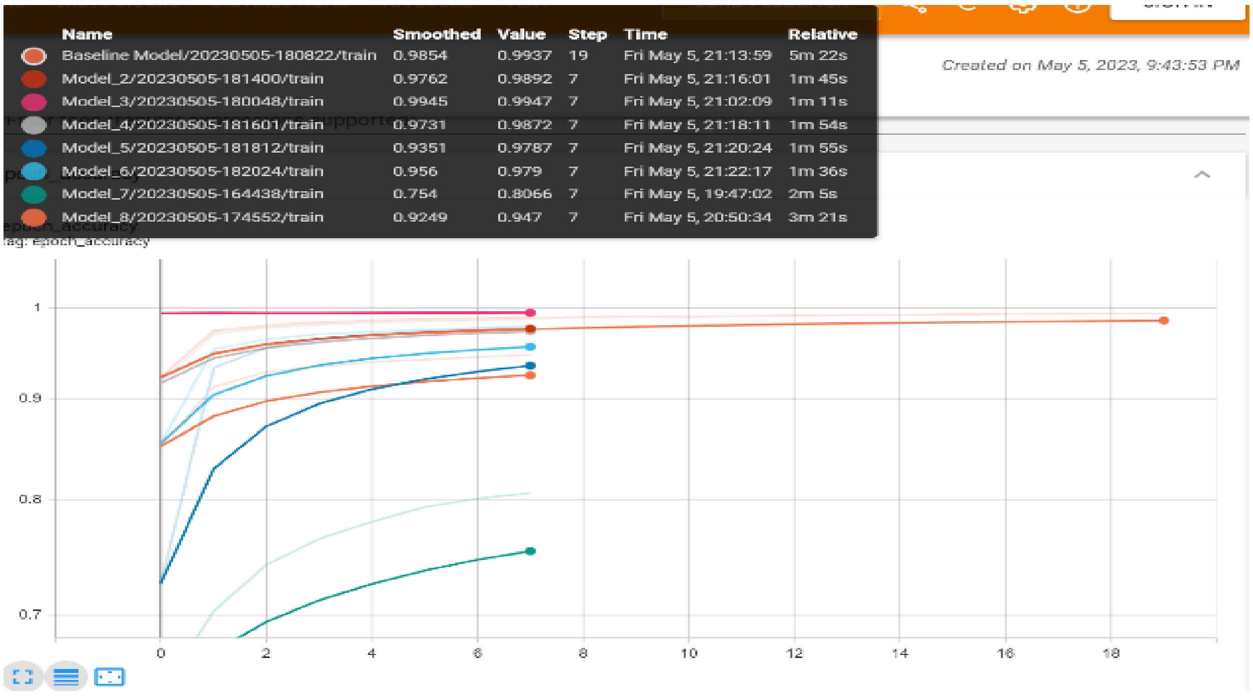
EfficientNetB0 is a convolutional neural network architecture that was developed using a neural architecture search method. It is designed to be efficient in terms of both computation and parameter usage while still achieving state-of-the-art performance on various computer vision tasks. It was trained on the ImageNet dataset, which is a large-scale image classification dataset that contains over one million labeled images spanning 1000 different classes. The pre-trained EfficientNetB0 model can be fine-tuned on other datasets for various computer vision tasks such as object detection, semantic segmentation, and image classification. The transfer learning capabilities of EfficientNetB0 make it a popular choice for tasks where training a deep neural network from scratch is not practical due to limited computational resources.



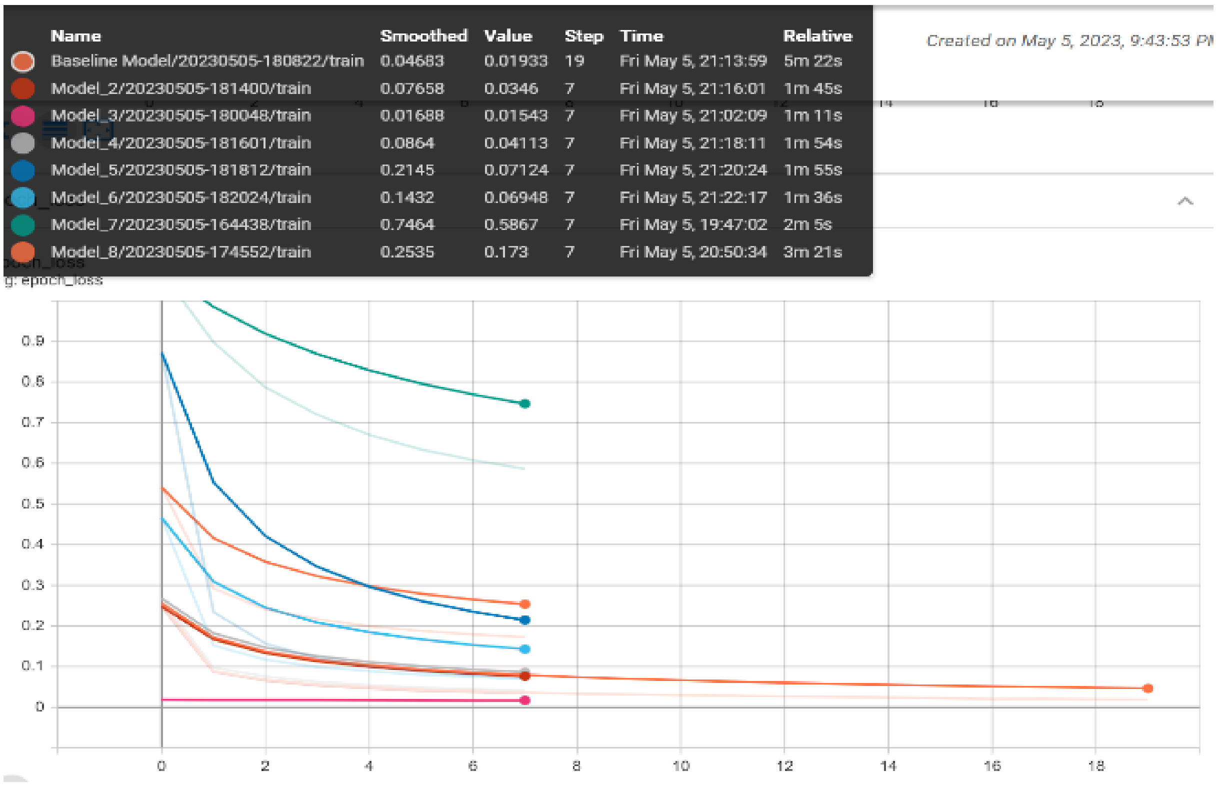
Upon examining the loss chart, it is observed that the model's loss value is comparatively higher than other models. This could be attributed to the fact that we only performed feature extraction, without fine-tuning the model. It is possible that the loss value could have been reduced if we had fine-tuned the model.

5. Compare Results

Train Data

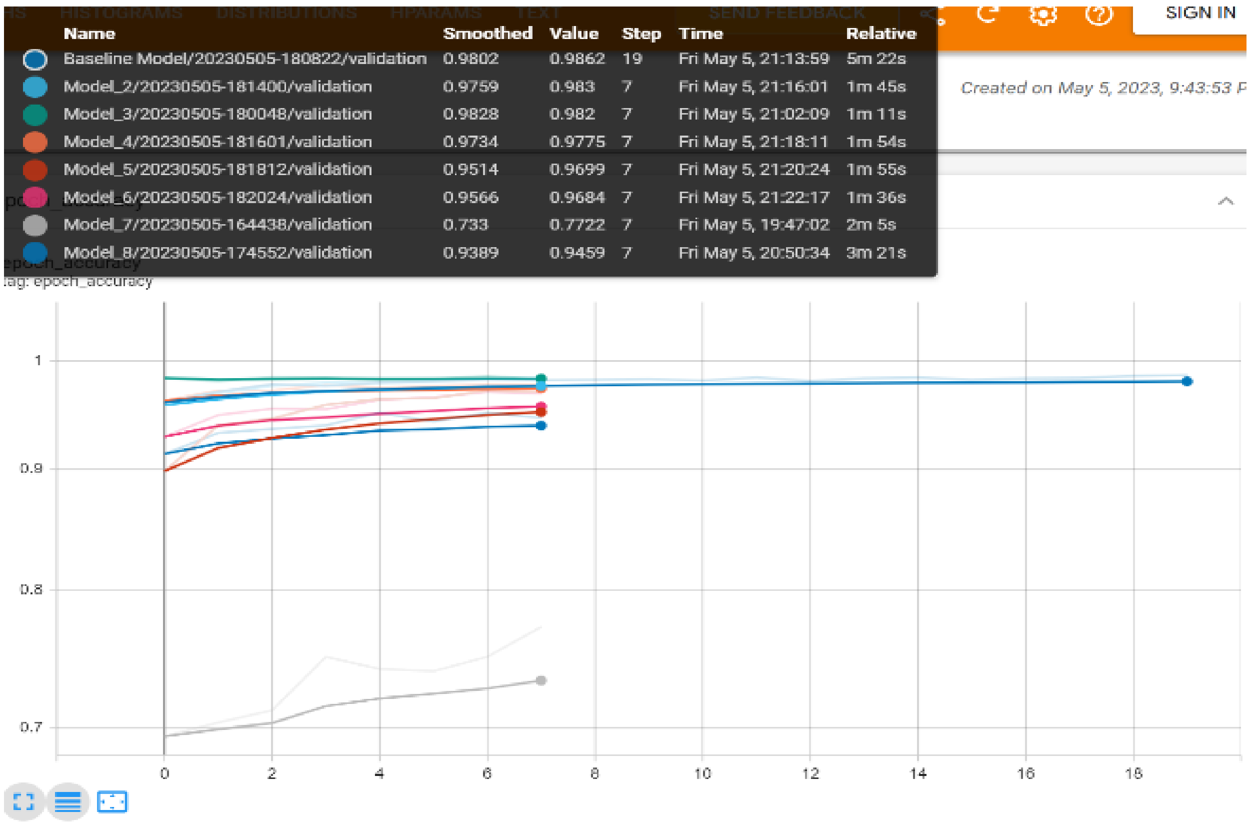


	Name	Smoothed	Value	Step	Time	Relative
●	Baseline Model/20230505-180822/train	0.9854	0.9937	19	Fri May 5, 21:13:59	5m 22s
●	Model_2/20230505-181400/train	0.9762	0.9892	7	Fri May 5, 21:16:01	1m 45s
●	Model_3/20230505-180048/train	0.9945	0.9947	7	Fri May 5, 21:02:09	1m 11s
●	Model_4/20230505-181601/train	0.9731	0.9872	7	Fri May 5, 21:18:11	1m 54s
●	Model_5/20230505-181812/train	0.9351	0.9787	7	Fri May 5, 21:20:24	1m 55s
●	Model_6/20230505-182024/train	0.956	0.979	7	Fri May 5, 21:22:17	1m 36s
●	Model_7/20230505-164438/train	0.754	0.8066	7	Fri May 5, 19:47:02	2m 5s
●	Model_8/20230505-174552/train	0.9249	0.947	7	Fri May 5, 20:50:34	3m 21s



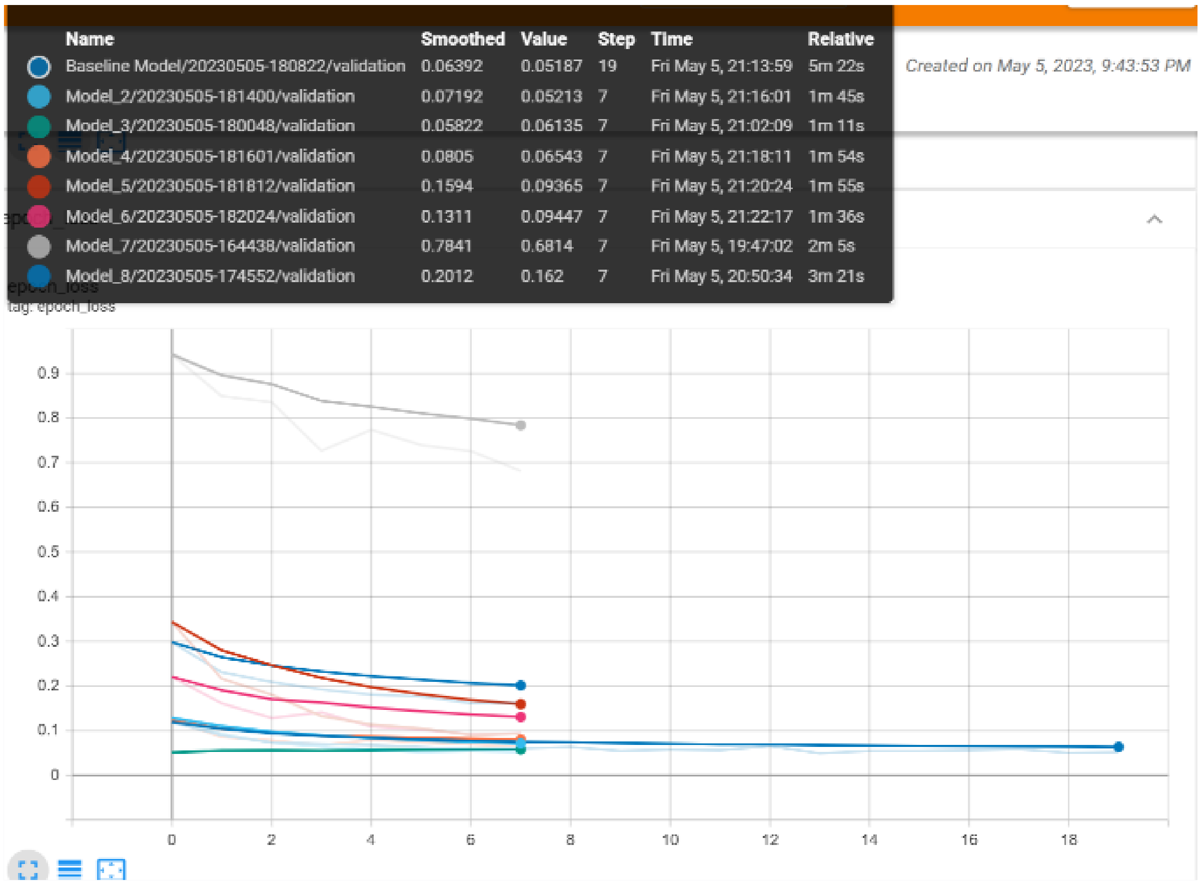
	Name	Smoothed	Value	Step	Time	Relative
●	Baseline Model/20230505-180822/train	0.04683	0.01933	19	Fri May 5, 21:13:59	5m 22s
●	Model_2/20230505-181400/train	0.07658	0.0346	7	Fri May 5, 21:16:01	1m 45s
●	Model_3/20230505-180048/train	0.01688	0.01543	7	Fri May 5, 21:02:09	1m 11s
●	Model_4/20230505-181601/train	0.0864	0.04113	7	Fri May 5, 21:18:11	1m 54s
●	Model_5/20230505-181812/train	0.2145	0.07124	7	Fri May 5, 21:20:24	1m 55s
●	Model_6/20230505-182024/train	0.1432	0.06948	7	Fri May 5, 21:22:17	1m 36s
●	Model_7/20230505-164438/train	0.7464	0.5867	7	Fri May 5, 19:47:02	2m 5s
●	Model_8/20230505-174552/train	0.2535	0.173	7	Fri May 5, 20:50:34	3m 21s

-Test Data



Created on May 5, 2023, 9:43:53 P

	Name	Smoothed	Value	Step	Time	Relative
●	Baseline Model/20230505-180822/validation	0.9802	0.9862	19	Fri May 5, 21:13:59	5m 22s
●	Model_2/20230505-181400/validation	0.9759	0.983	7	Fri May 5, 21:16:01	1m 45s
●	Model_3/20230505-180048/validation	0.9828	0.982	7	Fri May 5, 21:02:09	1m 11s
●	Model_4/20230505-181601/validation	0.9734	0.9775	7	Fri May 5, 21:18:11	1m 54s
●	Model_5/20230505-181812/validation	0.9514	0.9699	7	Fri May 5, 21:20:24	1m 55s
●	Model_6/20230505-182024/validation	0.9566	0.9684	7	Fri May 5, 21:22:17	1m 36s
●	Model_7/20230505-164438/validation	0.733	0.7722	7	Fri May 5, 19:47:02	2m 5s
●	Model_8/20230505-174552/validation	0.9389	0.9459	7	Fri May 5, 20:50:34	3m 21s



Created on May 5, 2023, 9:43:53 PM

	Name	Smoothed	Value	Step	Time	Relative
●	Baseline Model/20230505-180822/validation	0.06392	0.05187	19	Fri May 5, 21:13:59	5m 22s
●	Model_2/20230505-181400/validation	0.07192	0.05213	7	Fri May 5, 21:16:01	1m 45s
●	Model_3/20230505-180048/validation	0.05822	0.06135	7	Fri May 5, 21:02:09	1m 11s
●	Model_4/20230505-181601/validation	0.0805	0.06543	7	Fri May 5, 21:18:11	1m 54s
●	Model_5/20230505-181812/validation	0.1594	0.09365	7	Fri May 5, 21:20:24	1m 55s
●	Model_6/20230505-182024/validation	0.1311	0.09447	7	Fri May 5, 21:22:17	1m 36s
●	Model_7/20230505-164438/validation	0.7841	0.6814	7	Fri May 5, 19:47:02	2m 5s
●	Model_8/20230505-174552/validation	0.2012	0.162	7	Fri May 5, 20:50:34	3m 21s

https://tensorboard.dev/experiment/P7gbikD3TwuLevCJrQUQAg/

Model Name	**Train Loss**	**Train Accuracy**	**Test Loss**	**Test Accuracy**
Baseline Model	0.0193	0.9937	0.0376	0.9894
Model_2	0.0346	0.9892	0.0471	0.9855
Model_3	0.0154	0.9947	0.0351	0.9900
Model_4	0.0411	0.9872	0.0346	0.9894
Model_5	0.0712	0.9787	0.0431	0.9868
Model_6	0.0695	0.9790	0.0755	0.9756
Model_7	0.5867	0.8066	0.5772	0.8159
Model_8	0.1730	0.9470	0.1620	0.9459

This table summarizes the performance of different models for image classification using the MNIST dataset. The Baseline Model achieved the best results with a train accuracy of 0.9937 and a test accuracy of 0.9894, followed closely by Model_3 with a train accuracy of 0.9947 and a test accuracy of 0.9900

Model_2, which used a smaller epoch size than the Baseline Model, had slightly lower accuracy values. Model_4, which used a higher learning rate for the optimizer, had similar accuracy values to the Baseline Model. Model_5, which used a different activation function, had lower accuracy values compared to the Baseline Model.

Model_6, which used a different stride value, had lower accuracy values for both train and test sets. Model_7, which used a pre-trained model with feature extraction, had much lower accuracy values compared to other models. Finally, Model_8, which used a pre-trained model with fine-tuning, had similar accuracy values to the Baseline Model.

Overall, the results suggest that some hyperparameters and architecture choices can have a significant impact on the performance of the model, while others may not have a significant effect. It also demonstrates the importance of experimentation and evaluation in the development of effective deep learning models.