Advanced Machine and Deep Learning - Assignment-3

Heiko Röthl[1]

[1]*heiko.roethl@stud.unileoben.ac.at*, *m08635034*, *Montanuniversität Leoben*

Leoben, Austria, November 26, 2024

Dr. Ozan Özdenizci

Chair of Cyber-Physical-Systems
Montanuniversität Leoben, Austria

Contents

1 Outlook

This assignment involves developing a classification algorithm for the CIFAR-100 dataset, with the specific requirement of classifying images into the 20 superclasses only. The original 32x32 RGB images have been converted to grayscale as demanded in the assignment guidelines.

2 Training, Validation and Test Set Creation

The first step was a visual inspection of the provided data to get an initial sense of its characteristics. Here, two specific examples are highlighted: one where the image is clearly recognizable to the human eye , and another where it requires a great deal of imagination and interpretation to categorize the label correctly. Simply browsing through several images reveals the inherent challenge in achieving a high prediction accuracy with this dataset.

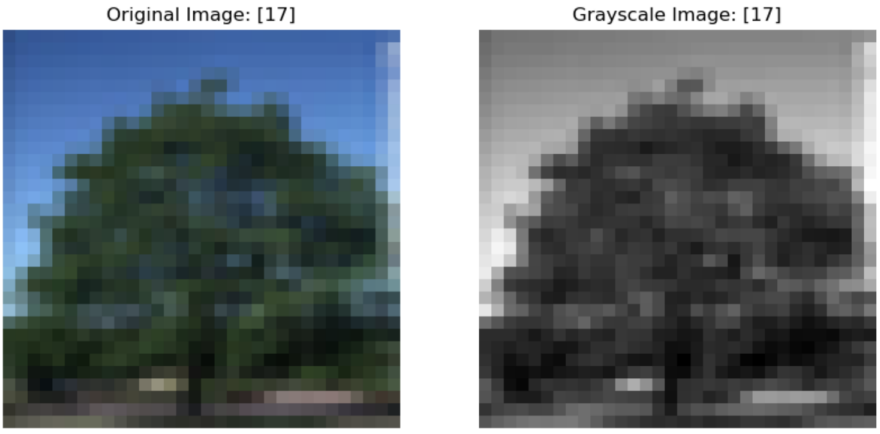

Figure 1: *Trees (Maple, Oak, Palm, Pine, Willow)*

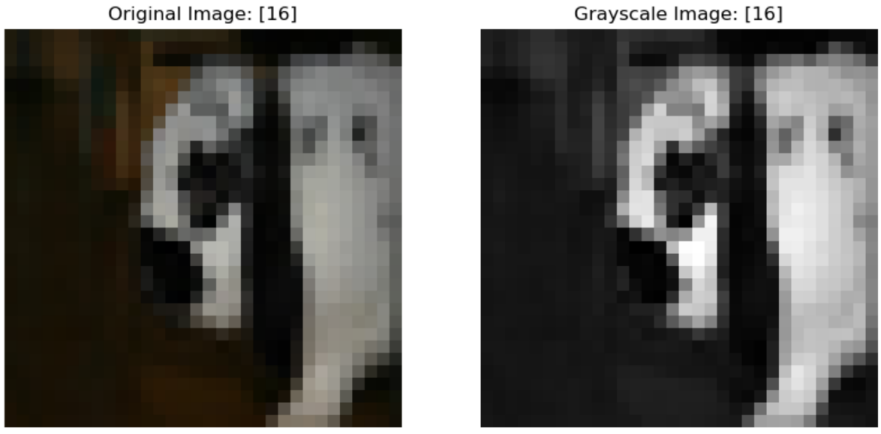

Figure 2: *Small Mammals (Hamster, Mouse, Rabbit, Shrew, Squirrel)*

A validation set was created with the aim of preserving the same label distribution as in the overall training set. This approach, also known as stratified sampling, is important to ensure that the validation process provides an unbiased assessment of model performance. When the validation set reflects the distribution of labels in the training data, it prevents

over- or under-representation of certain classes, which could otherwise lead to misleading performance metrics. In this case that would however not been necessary, as the distribution of the superclass is a even distribution.

3 Basic Model Design

Output Layer: As an output layer, a dense layer with 20 neurons was selected to match the 20 labels being predicted. The softmax activation function is used to provide a probability distribution across these classes, enabling the model to interpret the highest-probability label as its prediction.

Error Function: The categorical crossentropy loss function is used, as it is well-suited for multi-class classification tasks. It quantifies the difference between the predicted probabilities and the true class labels. The accuracy metric is chosen as the error function for evaluation, as it directly reflects the percentage of correct predictions, providing a clear indication of model performance.

Training was done using mini batches, with a size of 32. Mini batches with higher sizes were also tested, without detecting a notable difference by increasing the batch size.

The initial phase in investigating the effect of model architecture follows the approach to first see the effect of making the model deeper. The starting point was a simple model containing 3 *convolution* layers with kernel size 3x3, accompanied by a *batch normalization* and *Max-Pooling* layer followed by two *dense* layers, and the final output layer using *softmax* activation. Filter and Pooling size stayed the same throughout the sequence, the number of filers is increasing.

```
def m1(input_shape=(32, 32, 1), num_classes=20):
    model = models.Sequential([
        layers.Input(shape=input_shape),
        layers.Conv2D(32, (3, 3), activation='relu', ),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(128, (3, 3), activation='relu'),
        layers.BatchNormalization(),
        layers.Flatten(),
        layers.Dense(128, activation='relu'),
        layers.Dense(64, activation='relu'),
        layers.Dense(num_classes, activation='softmax')
    ])
    return model
```

The subsequent models followed the same architecture, only that there was an additional block of convolution layer, batch normalization and pooling layer added, with the numbers of neurons in the dense layers adjusted accordingly.

The results of these three models are compared in Figure 3. It is evident that these model suffer from over-fitting, as the validation losses increase after only a few epochs. There is no significant difference between the 4 and 5 convolution layer models, however the computation time in the larger 5 layer model is significantly larger. For further investigations therefore the 4 layer model was used, which is a trade-off between performance and training time requirement.

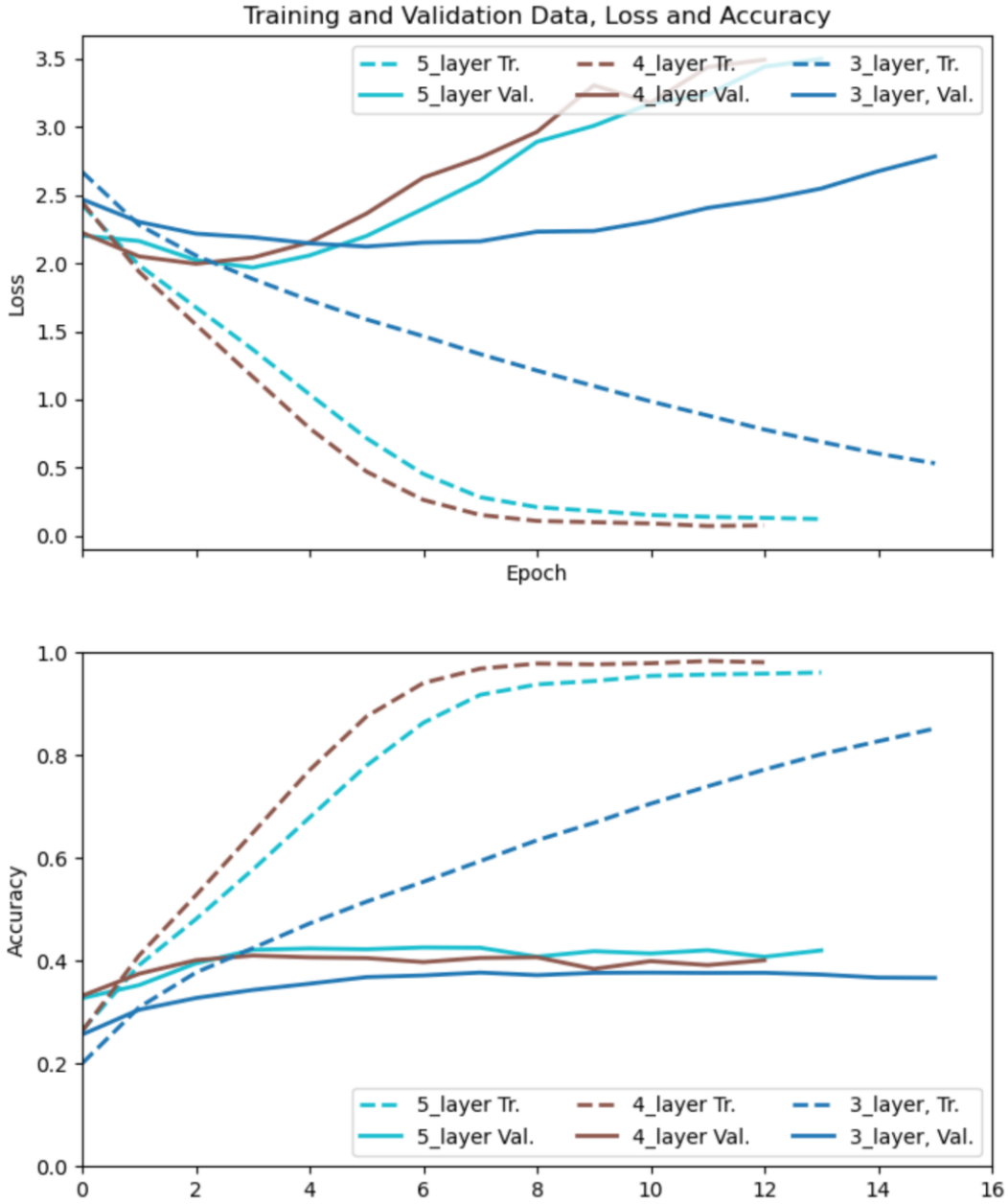

Figure 3: *3, 4 and 4 Convolution Layer Models*

Further to testing the effect of Number of convolution layers, it was also tested, how the size of the convolution kernel influences the performance. Two additional 4 layer model were created, one with increasing kernel sizes, from 2x2 to 5x5, and a model with decreasing kernel sizes from 5x5 to 2x2. The results are shown in Figure 4. Having an architecture with the kernel size increasing gives a clear advantage in validation accuracy and was kept for further models.

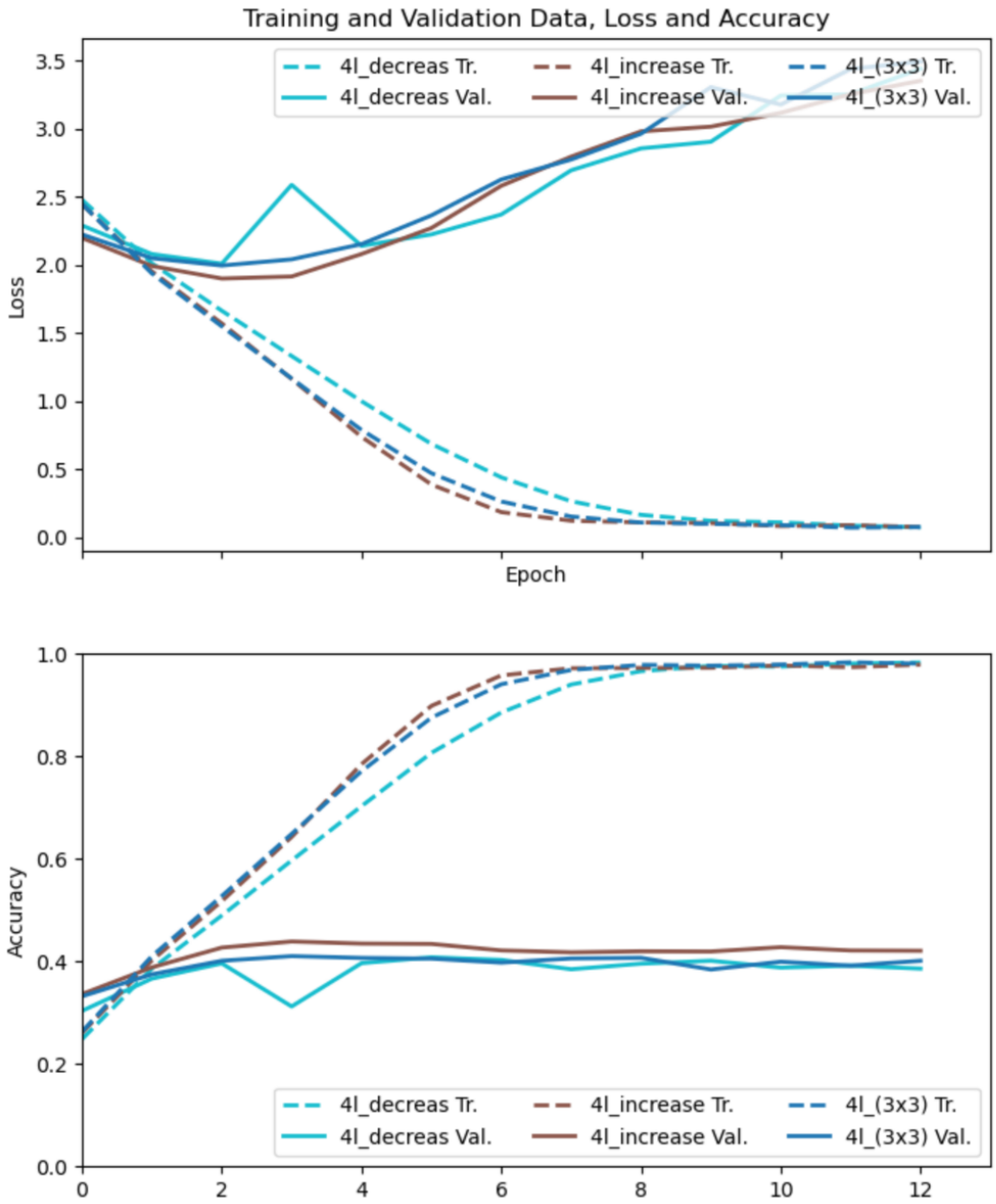

Figure 4: *4 Convolution Layer Models with varying Kernel Size*

4 Testing Weight Regularizers and Layer Dropout

For regularization two methods were tested, namely using dropout layers and L2 regularization. Both methods were individually tested to fully evaluate their effect. As a first step the drop out method was used, starting with adding drop out layers only after the final dense layers. For comparison a second model was added, where the dropout after each convolution layer has been added as shown in the code below.

```
def m7(input_shape=(32, 32, 1), num_classes=20):
    model = models.Sequential([
        layers.Input(shape=input_shape),
```

```
        layers.Conv2D(32, (2, 2), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.2),
        layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.3),
        layers.Conv2D(128, (4, 4), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.4),
        layers.Conv2D(256, (5, 5), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.Flatten(),
        layers.Dense(512, activation='relu'),
        layers.Dropout(0.5),
        layers.Dense(128, activation='relu'),
        layers.Dropout(0.3),
        layers.Dense(num_classes, activation='softmax')
    ])
    return model
```

The comparison of the result for the two models is shown in Figure 5. A regularization using dropout layers in the section of the final dense layers is not as effective as putting dropout layers also after the convolution layers. For the first case the model is still massively over-fitting with the validation loss increasing again.

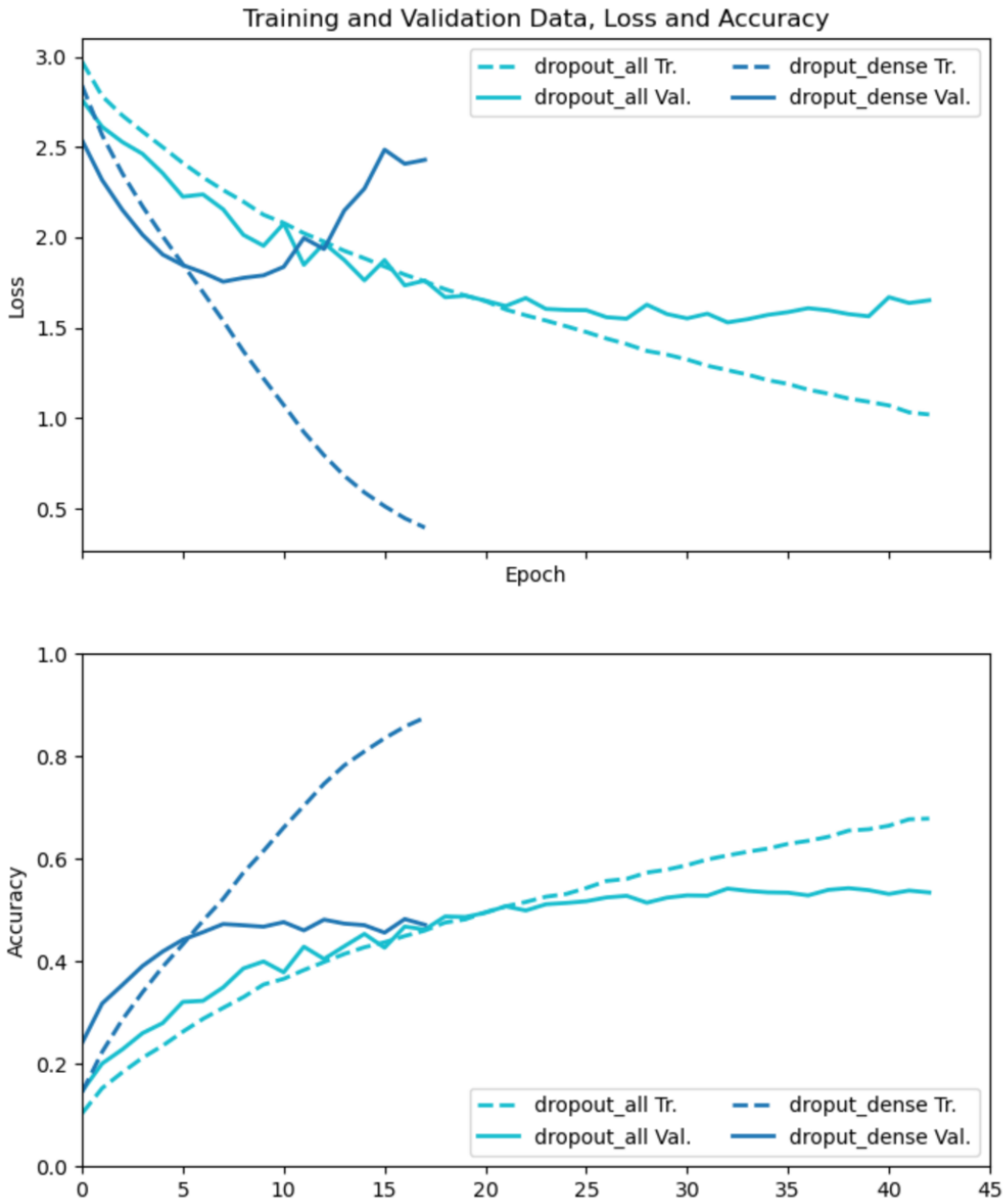

Figure 5: *4 Convolution with Drop Out Layers*

The same procedure as for the drop out layers was used to evaluate the L2 kernel regularization effect, with first using Kernel regularization only in the final dense layers, compared to the kernel regularization in all activated layers. The result is shown in Figure 6 and is somehow different from what has been observed for the drop out layers. The effect of L2 regularization when done only on the final dense layers is almost as high as when done in al activated layers.

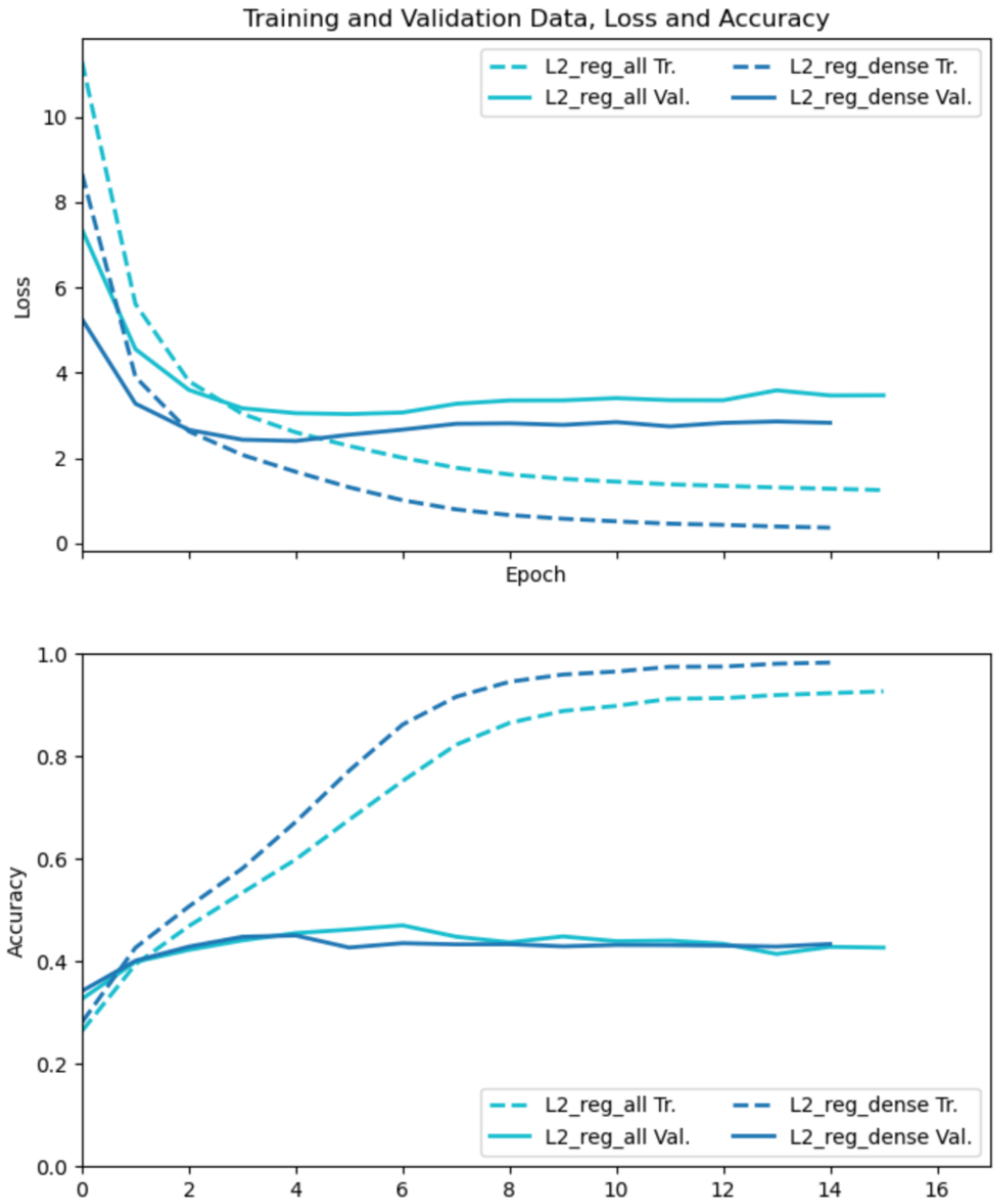

Figure 6: *4 Convolution Layers with L2 regularization*

The comparison of the regularized model results using exclusively either drop-out layers or L2 regularization is shown in Figure 7. Two combination of regularizers were tested, one model being full regularized by drop-out and L2 regularizers in every possible position, and the other with a somehow limited regularization, using drop-out for the convolution layers and L2 regularization in the dense layers. The latter option yields the best results followed by regularization with drop-out layers only. The metod using drop-out and L2 regularizers in a limited fashion als provides a good coherence with the training and the validation metrics. The table that summarizes this behavior is shown in Table 1.

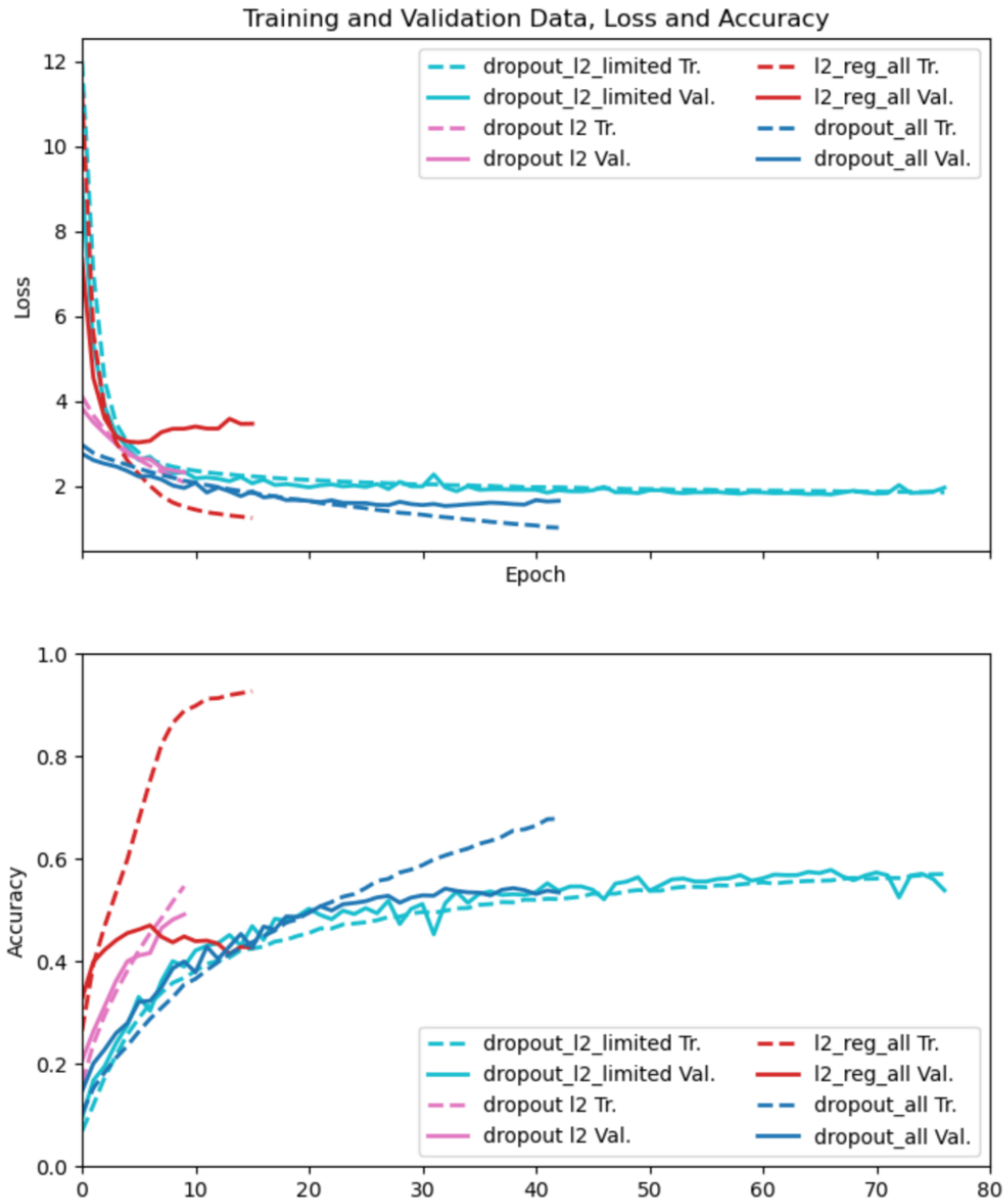

Figure 7: *Comparison of Regularization Methods*

Model	Name on Plots	Training		Validation			
		min Loss	max Acc.	min Loss	last Loss	max Acc.	last Acc.
m6	dropout dense	0.39	0.87	0.39	0.39	0.48	0.47
m7	dropout_all	1.02	0.68	1.02	1.02	0.54	0.53
m8	L2_reg_dense	0.37	0.98	0.37	0.37	0.45	0.43
m9	L2_reg_all	1.25	0.93	1.25	1.25	0.47	0.43
m10	L2 and Drop-out	2.09	0.55	2.09	2.09	0.49	0.49
m11	L2 Drop-out lim.	1.42	0.59	1.80	2.09	0.58	0.57

Table 1: *Training and Validation Results*

5 Final Model Architecture

As being demonstrated in Section 3, there is a slightly better performance of a deeper 5 convolution layer model compared to the 4 Layer model. Thus two deeper models with 5 convolution layers were created, one with a regularization using drop-out layers as only means of regularization and one making a combination of L2 and drop-out as described in the previous section. The results are depicted in Figure 8. The 5 convolutional layer model shows the best performance, although it also has the longest training time, with almost 120 epochs to go before early stopping is triggered.

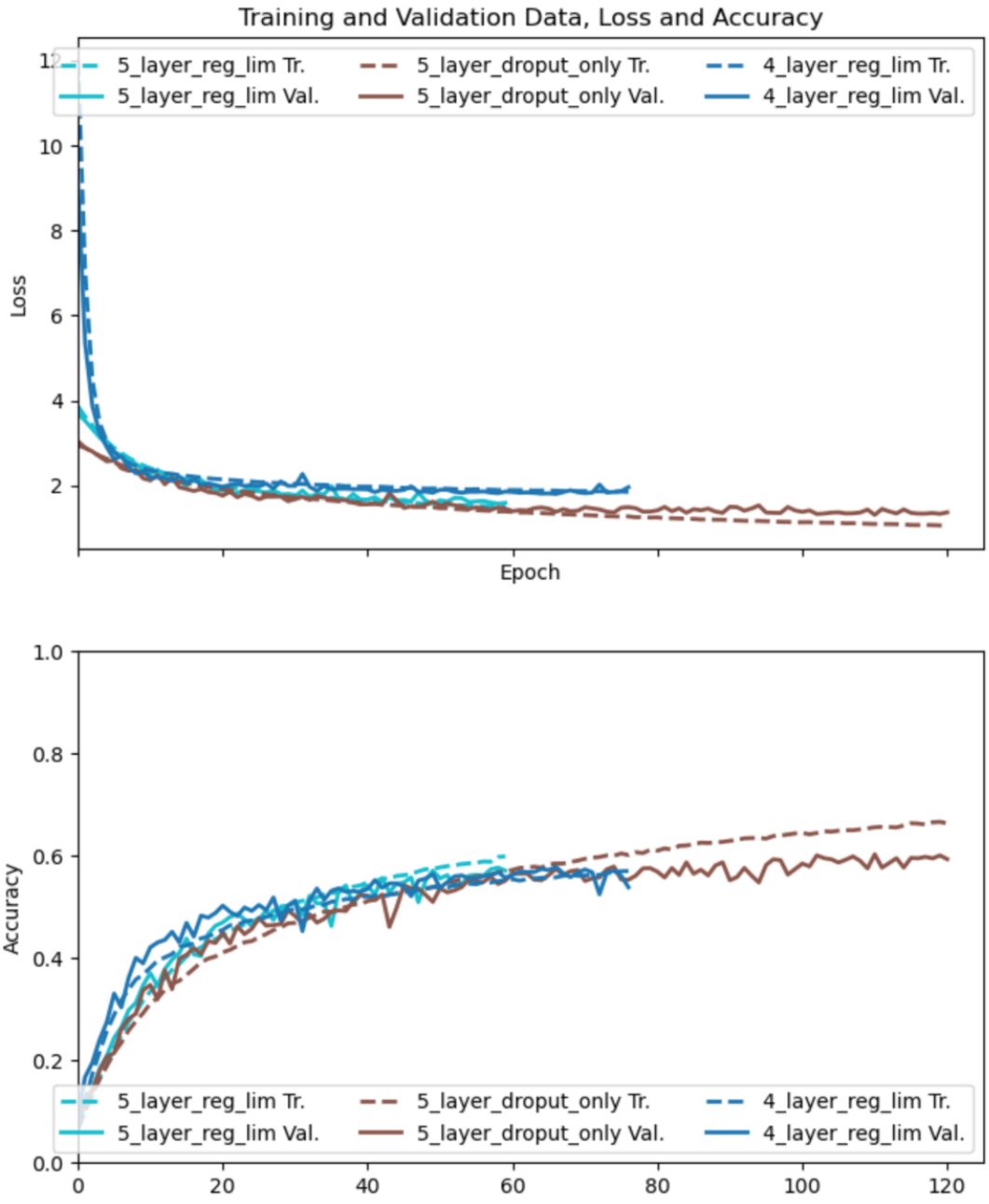

Figure 8: *Final Models - Best: 5 Layer Model with dropout only Regularization*

The code for the final model is shown below. The model has in total 6,587,732 trainable parameters. The test accuracy for this model is 59.48% on the test set and a loss of 1.3602.

```
def m12(input_shape=(32, 32, 1), num_classes=20):
    model = models.Sequential([
        layers.Input(shape=input_shape),
        layers.Conv2D(32, (3, 3), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.2),
        layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.3),
        layers.Conv2D(128, (3, 3), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.4),
        layers.Conv2D(256, (4, 4), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.4),
        layers.Conv2D(512, (5, 5), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(1024, activation='relu', ),
        layers.Dropout(0.4),
        layers.Dense(512, activation='relu', ),
        layers.Dropout(0.3),
        layers.Dense(128, activation='relu', ),
        layers.Dropout(0.2),
        layers.Dense(num_classes, activation='softmax')
    ])
    return model
```

The confusion matrix is shown in Figure 9

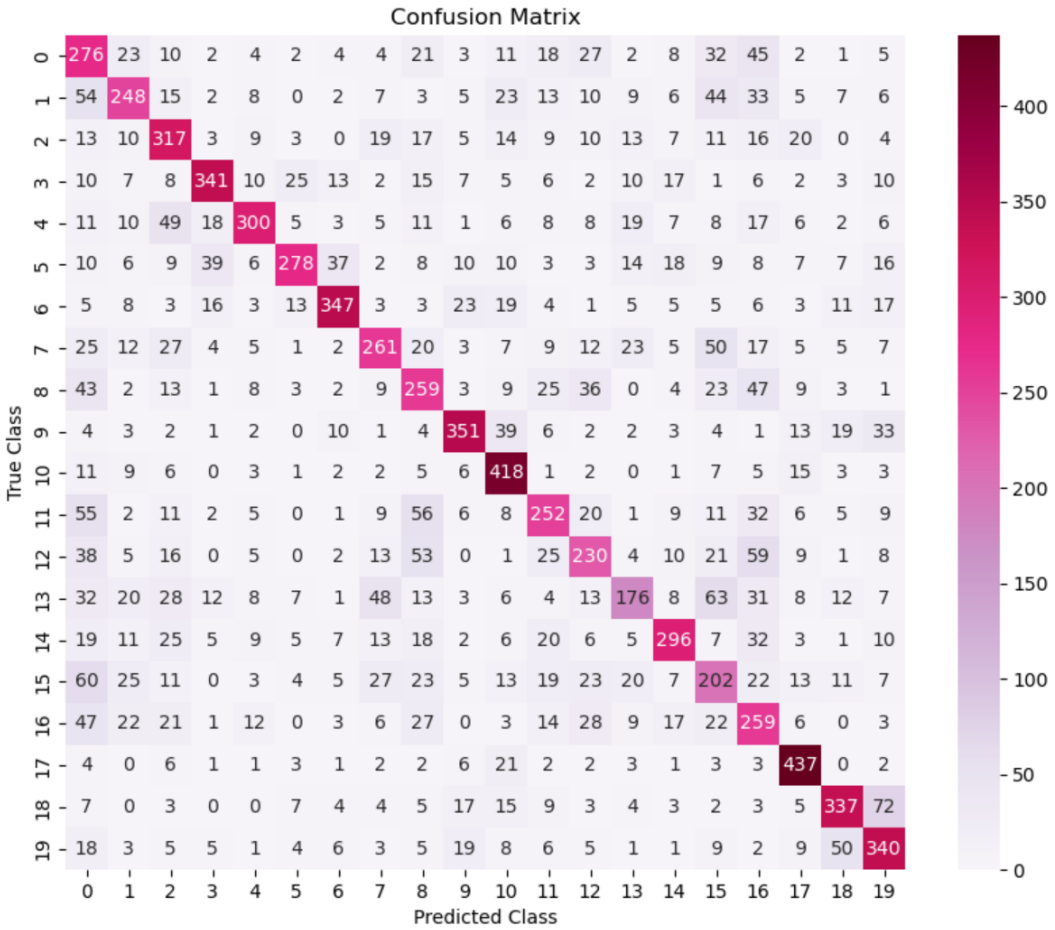

Figure 9: *Final Model - Confusion Matrix*

6 Using a Perturbed Test Set

Using the final Model with the test data of the perturbed set, the Test accuracy drops from 59.5% to 52.4%, which is quite a significant drop.

The first observation on the perturbed data is that there is some kind of Noise on the data. In a first attempt it was simply tried to mimic this noise. A visual inspection revealed that there are maybe around 3% of the pixels affected by noise. The training data was modified to reflect that behavior by adding random noise by following code:

```
def add_noise(x_data, noise_percentage=0.03):
    noisy_data = np.copy(x_data)
    total_pixels = x_data.shape[1] * x_data.shape[2]
    num_noise_pixels = int(total_pixels * noise_percentage)
    batch_size = x_data.shape[0]
    for i in range(batch_size):
        noise_indices = np.random.choice(total_pixels, num_noise_pixels,
            replace=False)
        x_coords, y_coords = np.unravel_index(noise_indices, (x_data.shape[1],
            x_data.shape[2]))
        noisy_data[i, x_coords, y_coords] = np.random.rand(num_noise_pixels)
    return noisy_data
```

The idea was to add the noise affected data to the training data set and retrain the model, using the weights from the previous training run. For practical reason this exercise was not done for the final model, as that was simply too big and would require a GPU. Instead the *m7* model, *(dropaout all)* as shown in Figure 7 was used. The re-training with the added data-set was done using the same amount of training data (i.e. 40000 samples). The result from this process was that, using this additional noisy data changing the accuracy for the original test data from 53.4% to 54.4%, the increase in accuracy for the perturbed data set was not as high but also changed from an accuracy of 44.3% to 47.9%. The process could be refined trying also a varying degree of noise across the samples, eg from 2% to 4%

A further method to attempt to make the model more robust against noise is to use image augmentation. Image augmentation was introduced using the keras *ImageDataGenerator* class. The image augmentation was done using rotation of 10 deg, vertical and horizontal shift by a factor of 0.1 and a flip of the image. The model was run, however the results could not match expectations, with the training stopped with a training accuracy stalled at 45% and the validation accuracy reaching only 42%, which is far away from the original model. The accuracy for the perturbed model was computed for completeness, giving a value of 35%, rendering this approach useless.
The effect of batch normalization could only evaluated backwards, as the model used was already implemented with batch normalization after each convolutional layer. Therefore a model was created without batch normalization *(m7_nbn)*. The results are as expected. On the original dataset the accuracy dropped from 53.4% to 50.1%, on the perturbed set the accuracy dropped from 44.3% to 42.2%. That shows that batch normalization has an effect on the generalization of the model.
Using a network with residual layers was envisaged, however due to the lack of access to a gpu and limited task, this task could not be implemented. Res nets are known to be less prone to over-fitting and being insensitive to noise.