Graz University of Technology

Institute of Theoretical Computer Science

Deep Learning

Correcting Images with Autoencoders

Title: Correcting Images with Autoencoders

Supervisor: Dr. Ozan Özdenizci

Name: Patrick Lovric, David Kerschbaumer, Linda Schweighofer, Felix Tischer

Student Number: 11707313, 11708776, 01530500, 01436798

Date: 25.01.2022 **SS:** 21

1 Introduction

1.1 Introduction to Autoencoders

Autoencoders are an unsupervised learning technique which uses neural networks. The autoencoder learns to encode a set of data by training the network to recognize structures and ignore "noise" for a given dataset. Autoencoders can also generate random new data that is similar to the input data. Autoencoders are used in many areas like image processing, anomaly detection, information retrieval, image compression and many more.

1.2 Task Description

Our task was to train an autoencoder which is able of correcting and re-generating clean images from their distorted versions. The dataset was to be chosen from chinese MNIST, Kuzushiji-MNIST and Kannada-MNIST. Also, it was our decision to adress a black and white or RGB colored dataset. We chose the greyscale Kannada-MNIST dataset. It contains images of 28 by 28 pixels, where each image represents a number.

1.3 Dataset

The dataset was downloaded from [1].

As this project is structured as a kaggle challenge it contains four csv-files, one training dataset, a testing set, a validation set (Dig-MNIST) and a sample submission file for the kaggle competition. In the training and validation sets, there is an additional column which indicates the target/label.

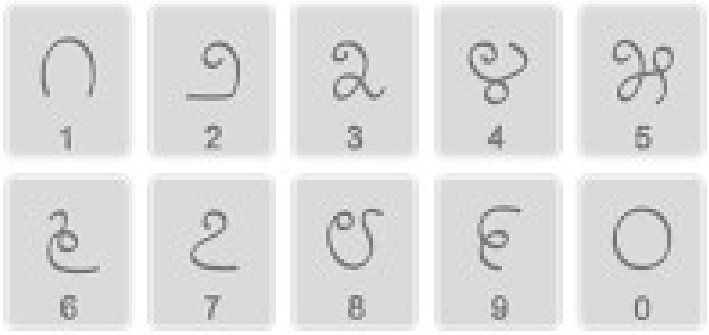

Figure 1: Kannada numbers [2]

The training dataset in the file train.csv has 785 columns. The first column indicates the label which is the digit that was drawn by the user. The rest of the columns are the pixel values of the associated image.

The test data set in the file test.csv is the same as the training set, but without the label.

All images are 28 pixel in height and 28 pixels in width, which leads to 784 pixels in total. Each pixel represents the lightness or darkness. The range of the pixel is from 0 to 255, where the 0 is the brightest 255 the darkest value.

1.4 Distortion

For the distortion of the input images, we added multiple methods which are implemented in the distortion_generator.py-file. Each clean image of the training set was distorted with one,

two, or three of these methods, chosen randomly.

The image could be flipped vertically or horizontally, or rotated by 90, 180, or 270 degrees. One of the three operations (up-down, left-right, rotation) was chosen with equal probability of 1 out of 3. The angle of rotation was chosen equivalently.

One of two noise generation methods was applied as well: either a gaussian distribution with $\mu = 0, \sigma = 30$, or a brightness gradient field across the image, where the position and magnitude of the maximum pixel are chosen randomly (where its brightness is distributed between 0 and 30). The brightness decreases then with distance r of the center by the factor $(1 + 0.1r)^{-1}$.

At last, a rectangular black patch can be put on the input image as well with a probabiltiy of 50%. The bottom left corner of the patch lies in the bottom left quadrant of the image and its side lengths lie between a quarter and half of the image side lengths. All these parameters are chosen randomly as well.

In summary, we applied one of three geometrical and one of two noise distortions to each image. To half of them, a partial occlusion has been applied too.

Figure 2 shows the 8 first images from the original clean (top) and distorted dataset (bottom). The most noticeable distortions are the occluding patches and the gaussian noise, while the others can be more subtle, like the horizontal flip of the 0 or the vertical flip and the brightness gradient of the 1 digit.

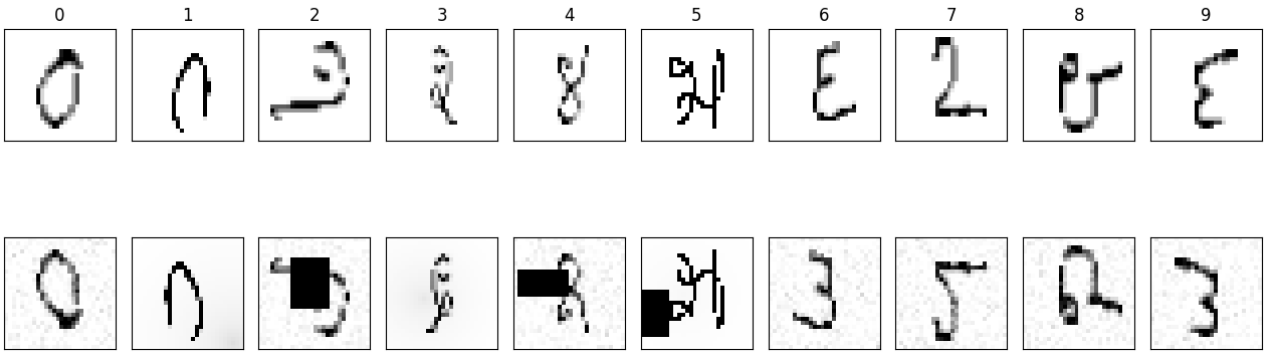

Figure 2: 8 examples of distorted images and their clean originals.

2 Methods

2.1 Mathematical Definiton

To calculate the loss, we chose the binary crossentropy.

$Loss = -\frac{1}{n}\sum_{i=1}^{n} y_i * \log(\hat{y}_i) * (1 - y_i) * \log(1 - \hat{y}_i)$

The loss is calculated by computing the average, from i=1 to n where n is the output size. $\hat{y}_i$ is the i-th scalar value in the output model and y_i is the corresponding target value.

2.2 Evaluation methods

We evaluate the performance of our autoencoder using 2 different methods. The first method is a visual evaluation, where we compare some example images as original version, as distorted version and then as the reconstructed version from our autoencoder. The results for this evaluation method are depicted in section 3.4.

As a second evaluation method, we reused the CNN from Assignment 3 from one of our team member (David) to evaluate how much improvement in classifying an image our autoencoder can provide. Therefore, we trained this evaluation-CNN with the original training set, downloaded from kaggle, and then evaluate the accuracy of 3 different sets:

1. the original training set
2. the distorted training set
3. the reconstructed training set

In most cases, one would not use the same dataset for training and evaluation, because the model is biased by the training set and of course the training set will perform better than a test set. But in this case, we do not want to show how well our model can predict the data. We only want to compare the accuracies of 3 different variations of the training set. Therefore, we think it is OK to use the training set to evaluate the improvement our autoencoder can generate. The results are depicted in section 3.5.

3 Results

3.1 Hyperparameter search

The model selection process was carried out via grid search over the following parameters:

1. Model type: We compared two different types of network: a dense layer NN and a CNN
2. Number of layers: 2, 3, or 4 hidden layers each on the encoder and decoder side
3. Number of neurons: 32, 64, 128 neurons in the bottleneck
4. Loss function: Two different loss functions for evaluation, binary crossentropy and mean squared error

The parameters were evaluated by accuracy, with the results in the following table 1. We can see that the convolutional models achieved a rather high accuracy here and there, but their loss was high and also the reconstructed images via the convolutional model didn't look nearly as good as the images from the dense model.

layers	neurons	model type	loss function	loss	val-loss	acc	val-acc
2	32	dense	binary crossentropy	0.10859	0.12267	0.88312	0.87995
2	64	dense	binary crossentropy	0.09859	0.11276	0.88623	0.88286
2	128	dense	binary crossentropy	0.09282	0.10792	0.88795	0.88419
3	32	dense	binary crossentropy	0.10815	0.10223	0.88314	0.88987
3	64	dense	binary crossentropy	0.09970	0.11433	0.88577	0.88216
3	128	dense	binary crossentropy	0.09882	0.11183	0.88608	0.88281
4	32	dense	binary crossentropy	0.10947	0.12388	0.88258	0.87936
4	64	dense	binary crossentropy	0.10186	0.11688	0.88497	0.88142
4	128	dense	binary crossentropy	0.10114	0.11475	0.88520	0.88171
2	32	dense	mse	0.01688	0.02154	0.88506	0.88056
2	64	dense	mse	0.01450	0.01854	0.88685	0.88292
2	128	dense	mse	0.01212	0.01661	0.88875	0.88475
3	32	dense	mse	0.01749	0.02195	0.88449	0.88020
3	64	dense	mse	0.01399	0.01853	0.88725	0.88311
3	128	dense	mse	0.01271	0.01753	0.88826	0.88391
4	32	dense	mse	0.01640	0.02139	0.88521	0.88028
4	64	dense	mse	0.01480	0.01932	0.88647	0.88209
4	128	dense	mse	0.01403	0.01913	0.88710	0.88229
2	32	conv	binary crossentropy	0.09484	1.29154	0.88702	0.64586
2	64	conv	binary crossentropy	0.08526	0.53478	0.88955	0.80479
2	128	conv	binary crossentropy	0.08517	0.58997	0.88958	0.78767
3	32	conv	binary crossentropy	0.09976	0.19213	0.88563	0.86751
3	64	conv	binary crossentropy	0.09747	0.23336	0.88631	0.87717
3	128	conv	binary crossentropy	0.08355	0.11610	0.88985	0.88411
4	32	conv	binary crossentropy	0.09822	4.62407	0.88584	0.86451
4	64	conv	binary crossentropy	0.09268	0.15781	0.88742	0.87693
4	128	conv	binary crossentropy	0.08913	0.51533	0.88837	0.87953
2	32	conv	mse	0.01032	0.05677	0.88966	0.83360
2	64	conv	mse	0.01173	0.02810	0.88856	0.88032
2	128	conv	mse	0.00893	0.01988	0.89089	0.88348
3	32	conv	mse	0.01212	0.55866	0.88818	0.35231
3	64	conv	mse	0.01281	0.06574	0.88764	0.86445
3	128	conv	mse	0.00898	0.06066	0.89068	0.86454
4	32	conv	mse	0.01212	0.55866	0.88818	0.35231
4	64	conv	mse	0.01281	0.06574	0.88764	0.86445
4	128	conv	mse	0.00898	0.06066	0.89068	0.86454

Table 1: Results of our hyperparameter search

3.2 Final autoencoder

The best model architecture is a dense layer network with 3 layers, 32 neurons in the bottleneck, and binary crossentropy as the loss function.

We used these best hyperparameters to create our final autoencoder model and trained it with the full training set. The accuracy and the loss evolution over epochs are depicted in

figure 3.

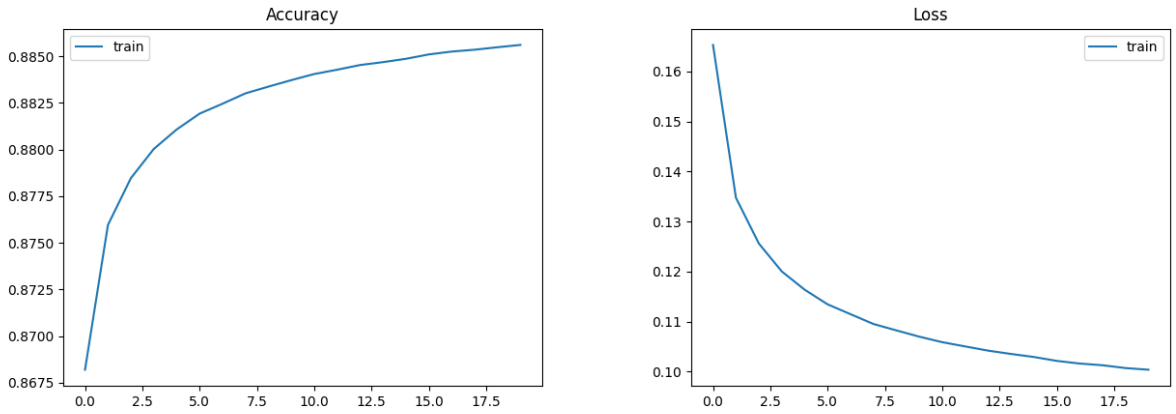

Figure 3: Accuracy and loss over epochs of the final autoencoder model.

```
Model: "autoencoder"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 Image Import (InputLayer)   [(None, 28, 28, 1)]       0

 flatten (Flatten)           (None, 784)               0

 dense (Dense)               (None, 512)               401920

 dense_1 (Dense)             (None, 512)               262656

 dense_2 (Dense)             (None, 256)               131328

 dense_3 (Dense)             (None, 64)                16448

 dense_4 (Dense)             (None, 784)               50960

 reshape (Reshape)           (None, 28, 28, 1)         0

=================================================================
Total params: 863,312
Trainable params: 863,312
Non-trainable params: 0
_________________________________________________________________
```

Above we can see the final architecture. We have 5 layers with trainable parameters, where our encoder consists of the first three dense layers. The bottleneck is the dense_3 and the decoder is the dense_4 layer.

3.3 Regularization

After determining the final model architecture, we tried to add regularization routines to the model as well, to prevent overfitting. This was done in two different ways: At first, we applied L1 and L2 (one of them as well as both at the same time) regularization on weights, biases, and the activation, and at the second attempt only on weights. A parameter search quickly revealed that the best results came from not using any regularization, as is obvious in the following figures 4 and 5.

Due to regularization, the network tried to minimize the loss on average, regardless of the input. In the first attempt, this led to completely meaningless output. The second at least produced output images that resemble a kind of average over the input images. Compare these with the results from 6 and 7, which used the same architecture and no regularization.

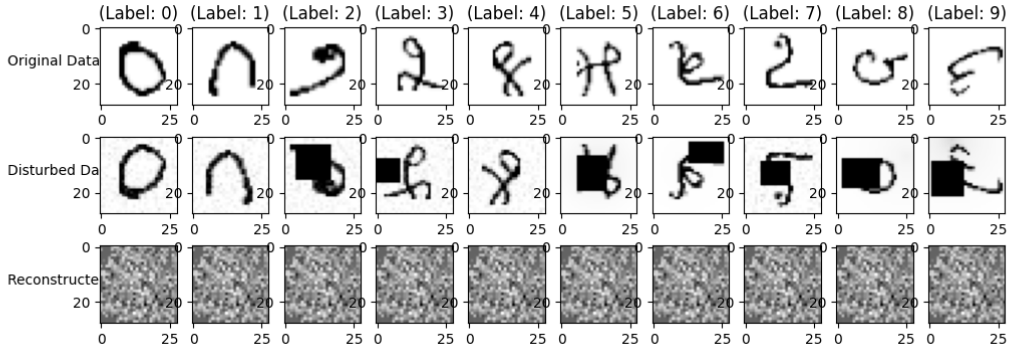

Figure 4: Best results from the regularization parameter search, first attempt. Regularization was applied to weights, biases, and activation.

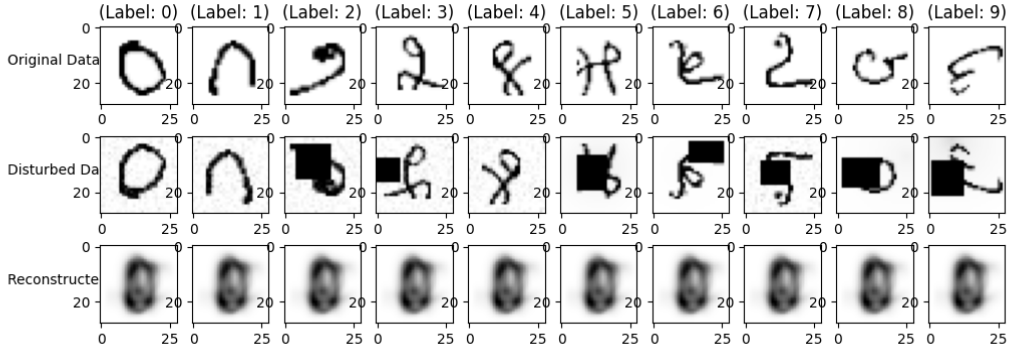

Figure 5: Best results from the regularization parameter search, second attempt. Regularization was applied only to weights.

3.4 Visual evaluation

The first evaluation method we used is visual evaluation. Therefore we compare the same example images in 3 different phases. Once as original and unmodified image from kaggle (in row original data), once as distorted version and once as reconstructed version.

Moreover, we had 2 different approaches in distorting the data. At single distortions at figure

6, we only used one of our distortion approaches (e.g. only flipping) and reconstructed it, while at multiple distortions at figure 7 we applied 2 or 3 distortions on the same image (e.g. flip and occlusion) and tried to reconstruct it.

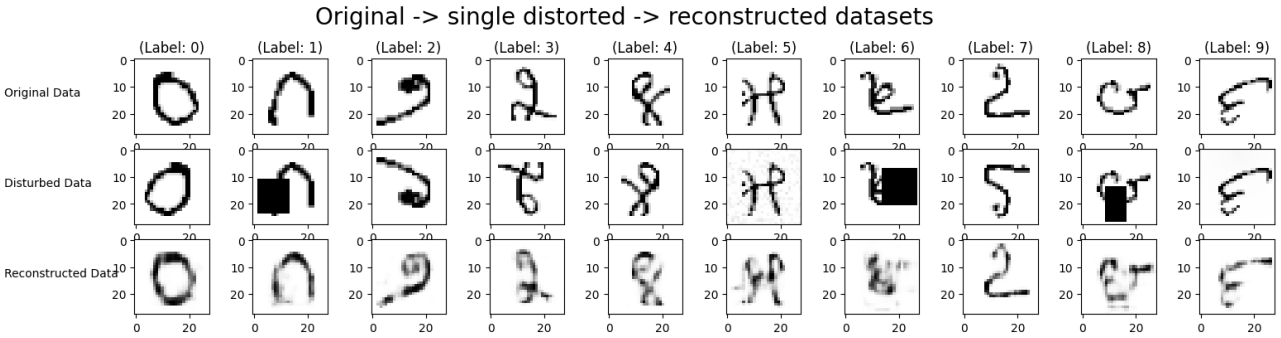

Figure 6: Comparison of original, distorted and reconstructed data using single distortions.

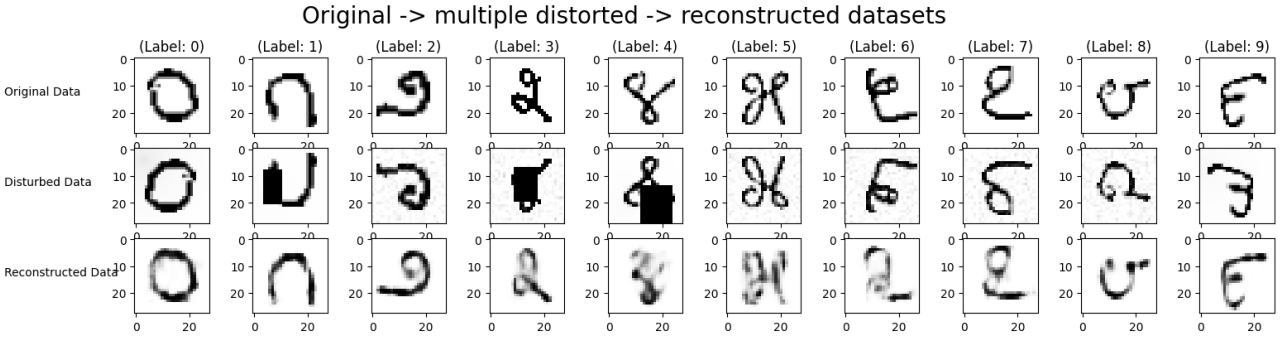

Figure 7: Comparison of original, distorted and reconstructed data using multiple distortions.

One can see that with single and multiple distortions, our autoencoder can reconstruct the original image pretty well and the original is most of the time recognizable for the human eye.

3.5 CNN evaluation

We also wanted to evaluate our autoencoder on a neuronal network and reused one CNN from assignment 3.

As depicted in table 2 we reach an accuracy of 99% on the original training set (ID = 1). This seems very high, but we must not forget that it is the same dataset we used for training the model.

Then we evaluated on the single and multiple distorted training sets (ID = 2 and ID = 3). Where at the single distorted training set we only included one distortion (e.g. occlusion or rotation), at the multiple distorted dataset we combined 2-3 distortions for each image (e.g. occlusion and rotation). In both cases our CNN can't even begin to recognize as many images as in the original data. We reach an evaluation accuracy of 58% at single distortions and a very bad accuracy of 26% for the multiple distorted dataset. This means that our generated distortions (using multiple distortions) are "good" enough to reduce the accuracy of an CNN by 73%.

To show how much improvement our autoencoder can deliver, we evaluated the reconstructed dataset and reach an accuracy of 97%, which means we have increased the accuracy by 71% and are only 3% accuracy away from the original training set.

ID	dataset	loss	accuracy
1	original training set	0.0152	0.9952
2	single distorted training set	3.4219	0.5853
3	multiple distorted training set	4.9394	0.2681
4	reconstructed training set	0.1047	0.9682

Table 2: Evaluation results of different datasets on an evaluation-CNN.

Looking at the confusion matrix 9 we see that images with label 3 and 5 are the hardest to predict from the reconstructed dataset. When checking this in figure 6, these are the images which are the most squiggly ones and are therefore harder to reconstruct for the autoencoder than others.

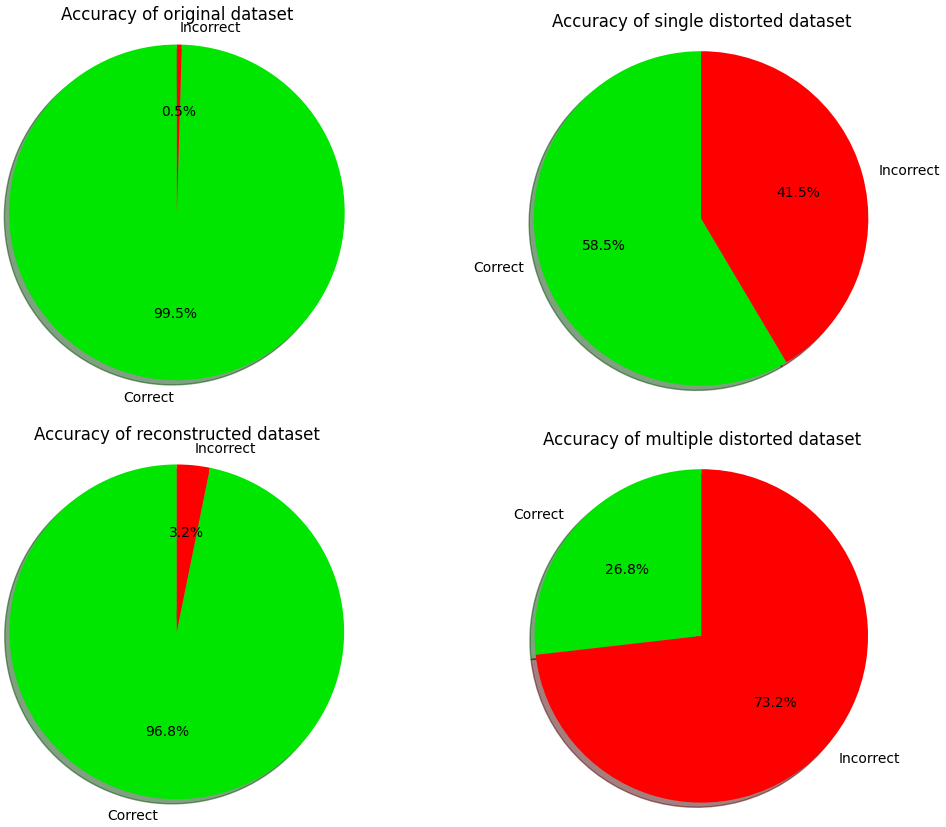

Figure 8: Pie charts showing the percentage of correctly predicted images of different datasets.

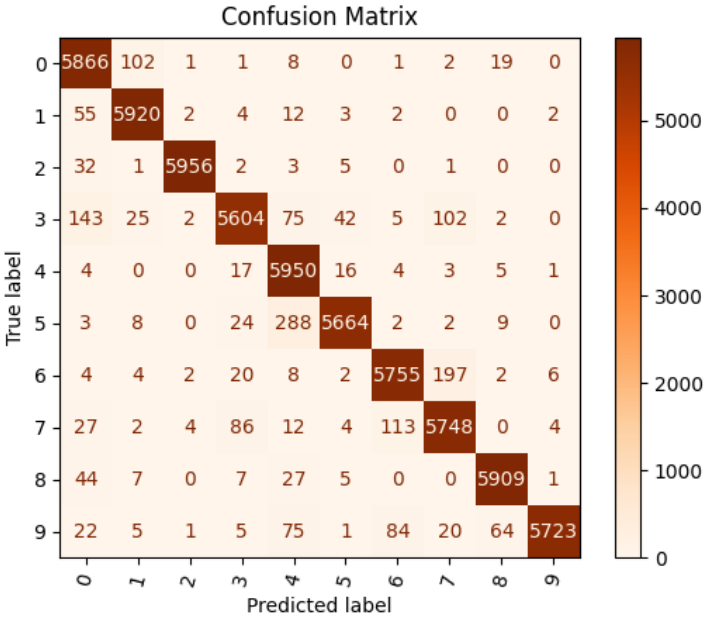

Figure 9: Confusion matrix of evaluation CNN predicting the reconstructed dataset

4 Discussion

4.1 Example Use-Case

We would like to show the major benefit of our autoencoder by an example use-case. Imagine, our task is to classify handwritten digits of the language Kannada (45 million native speakers in southwestern India) but our dataset is partially noisy. Some images have a high resolution, but some of them include noise, occlusions, changes in brightness, rotations, or flips. Our CNN would suffer predicting these distorted images and only predicts the high quality images well. This might lead to an average or low accuracy even though we have a strong model.

Using our autoencoder before predicting the images with the CNN will modify the distorted images and reconstruct them optimized for the model. These reconstructed images than can be predicted with the CNN and will have a much higher accuracy. This use-case often occurs in real life, because you very rarely have clean datasets without noise. As depicted in section 3.5 our autoencoder can increase the accuracy of those CNNs tremendously.

References

[1] KAGGLE. Kannada-minst dataset. https://www.kaggle.com/c/Kannada-MNIST/data, 2022.

[2] vectorstock. Kannada numbers. https://www.vectorstock.com/royalty-free-vector/set-of-monochrome-icons-with-kannada-numbers-vector-15469802, 2022.