# Assignment 2

April 18, 2018

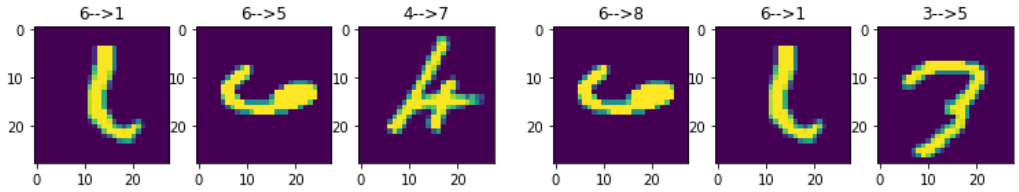
## 1 Task 1

### 1.1 Miss classified images

In order to detect the most misclassified images we explored the network's output which includes all probabilities for each image and each class. Softmax function produces the probability output for the classification task. We search only the instances(images) which were misclassified and in order to define the most misclassified images we calculate the absolute subtraction of the probability for the true class and the output (predicted) class. In this way, we can measure the probability error between the correct class and the one that is predicted from the network. Intuitively, we conclude that the three instances with the larger error-values are the most misclassified. We ran two different experiments on two different networks, MLP and CNN, using cross-entropy and mean-squared-error as the loss functions. Each experiment ran 50 times on the GPU servers. The results below visualize the three digits with the highest probabilistic-error and the highest misclassified frequency in 50 times for each experiment.

(a) MLP using Cross-entropy

(b) MLP using MSE




(c) CNN using Cross-entropy

(d) CNN using MSE

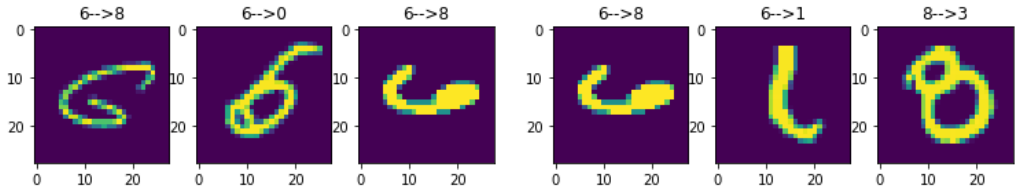



Figure 1: Top three misclassified images from MLP and CNN networks using "Cross-entropy" and "MSE" respectively. On the top of each images there is the actual label and the one that network predicted.

As it can be observed, there is one instance that is common in all the experiments and one instance which are common in three of the four experiments. It is important to mention that the aforementioned digits are also difficult to distinguish even for the human eye. We cannot tell if they represent six or one for example, so it totally makes sense that they are included in the most misclassified cases.

### 1.2 Randomness

Randomness and deterministic results are always an issue for prediction and classification tasks, especially when the experiments are running on GPU servers instead of CPU. There is an even bigger controversy around this issue when the network is trained and builded using keras. Multiple issues with random seed and reproducibility of neural networks' results may occur when they are running on GPU. One of the reasons this occurs, is that neural network algorithms are stochastic and they use randomness to initialize weights so it is quite unlikely that they will give the same

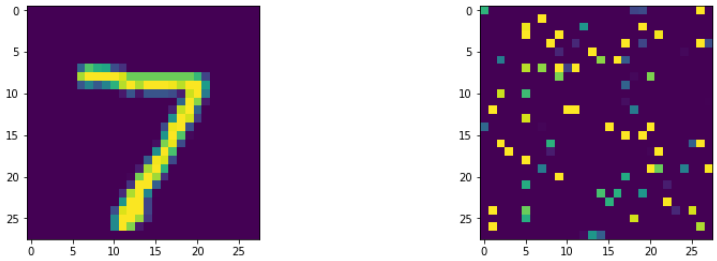
outcomes in a repeat of the experiment.
Definitely there are some tricks in order to stabilize or approximate the same results:

- setting a seed
- run multiple times the experiment

However, in our case it is not possible to set a random seed due to GPU's peculiarities. Probably, many algorithms build in CUDA behind Tensor-flow and Keras are non-deterministic so it is almost impossible to reproduce the same experiments and the same results. In particular, GPU servers use a library called cDNN which is developed for neural network training, deep learning and more. The problem is, that this library is non-deterministic, which means that it can not produce the same results even after the seed setting given that we are using Keras. Therefore, we chose a more traditional way to control the reproducibility of our results to run the network algorithm many times and using summary statistics, such as mean value, to gain an idea of the performance of our model.

## 1.3 Permuted Version of Images

The aim of this task was to rerun the two networks, MLP and CNN, but this time with randomly permuted pixels of the images. More precisely, each image is represented by a 28x28 matrix and the aim is to "shuffle" the order of its pixels. In this way we will define if the order of the pixels of each image has an impact on the network result. We permuted the pixels of each image on train and test set with the same random permutation. The following figure (2) visualizes an example of the original and the permuted image.



(a) Original Image (b) Randomly permuted image

Figure 2: Pixel Random Permutation

After we reran the two networks this time with randomly permuted pixels we define that the CNN produced lower test-accuracy, while the MLP produced exactly the same. This is reasonable because of the CNN's architecture which is based on smaller matrices(filters) that are used as masks on the initial images. These convolution-layers detect and learn better the structure of each image, so if the order and the geography of the image-pixels changes, then the network loses the aforementioned ability and produces worse results.

Table 1: Accuracy of the experiments

| | MLP | CNN |
|---|---|---|
| Permuted | 0.983 | 0.973 |
| Non-Permuted | 0.983 | 0.987 |

# 2 Task 2 : Recurrent Neural Networks

## 2.1 Intro

Recurrent Neural Networks, have several uses and are designed to predict data sequences, given an input data sequence, let's say x. They are quite effective in terms of prediction in a sense that they are able to keep the entire history of inputs we fed them and draw their information. More specifically, RNN train the network in such a way that the data dependencies are utilized in order to efficiently predict the output y via a probabilistic density produced by softmax function.

## 2.2 Experimenting with datasets

The implementation and training of a recurrent network can be easily understood with some examples given from three different scripts.

- Script 1 : Language Modeling
- Script 2 : Language Translation
- Script 3 : Adding two short integers

The idea in these three models is to train such a network which has the features of a RNN and thus it will be able to predict a word sequence, a translation and the result from adding two numbers.

1. **Language Modeling**

   For the first script, `lstm_text_generation.py` ,the aim is to predict the correct word sequence given a specific input. The process continues using the previous outcomes as inputs and deciding which information to keep and which not. First, the script was implemented on the data unchanged, that is with all the default set parameters. It was observed that in the early epochs the training process was in a premature state where the outcomes are mostly unordered sequences of words and 60 iterations after the outcomes were able to make some sense.

   It is known that tuning the parameters of an iterating procedure may give more accurate results. Moreover, depending on the data, the right tuning in parameters such like number of epochs batch size or in our case diversity may be proven crucial. So, we changed the number of epochs to 64, the batch size to 131 from 128 and tuned the word diversity a little bit experimenting with a new dataset that has also the form of sequences of words. Splitting the training process into batches requires less memory, thus the training process focuses on smaller data parts. Moreover, increasing the batch size may lead to more accurate estimates of the gradient descent improving out final predictions. Tuning diversity is all about the variety of words that will be predicted in each epoch. Focusing on the results of this training procedure we observed that the loss calculated with categorical cross entropy has reached 93% in the middle epochs and the predicted results can be seen in Console 1.1 and Console 1.2 :

   Console 1.1 : Early Iterations

   ```
    state as such has no raison d'etre and there is anarchists, social
   such and desirevably, which anarchists, and on justifipt by the
   first state, as anarchism without more race, there are region at same
   yes, ria nuele anarchists" a surree; who have how mutully best.  im
   considersure, or the contruly were all believed anarchist.  there are
   also subjectral anarchist is what working activity or feasind they
   anarcho-silltals with first many
   ```

   Clearly, most of the output does not make much sense in terms of syntax and language usage but, given the fact that we are at epoch 32 we can say that in terms of prediction we have an accurate RNN, thus there are parts of it which seem reasonable. Finally after 64 epochs the final output is this:

Console 1.2 : Final Iterations

```
there are still people who imagine that the diffecent citile
controliae of musscle protect concerns by banted be dovide is the
drained as.  ubver of hist, as resist to thhom freedoments of human
being in enfosion in the prisons, so onces.  must begurs with action
of his own more whe peopines the opportunity, and services of name
a "mall solibably kind becasia in man, which they ougrarce can gin
cloted some, this must contererty has no ruin year
```

It should be mentioned that we have a really large sample of words in this data so we expect our model to underestimate the output and we may need to experiment a little more with the parameters and the training samples. There are a lot of long-term dependencies between the words because we have to deal with a dataset taken from a book which means that most of the words and phrases are connected to each other. As a result, the maximum sequence length was increased form 40 to 50 so that gradient signal will backpropagate more than 40 time steps, and enable the model to find dependencies longer than the default length.

2. **Language Translation**

Continuing the implementation of RNN in different kinds of data, we introduce the second script, `lstm_seq2seq.py`, that has to do with translation. The idea here is to feed our network a sequence of words in English and after the requiring iterative procedure inside the network, to get the translated version of this sentence. We implemented the script unchanged in the default data set `eng-french.txt` and examined how accurate the network translated the English sentences to the relative French. The results were pretty accurate in 100 epochs and a batch size of 70, thus most of the sentences were translated correctly. The training and validation loss, were fluctuating among 0.09 and 0.7 respectively until the middle epochs,and reached 0.04 and 0.8 at the final epoch. In cases like this, when the loss of training data is much lower than the validation loss the network is overfitting the data.

Afterwards, the network was fed with a different set of sentences, obtained from `eng-ell.txt` dataset, and this time we are expecting the Greek translation of the given input. Despite the change of the dataset we also experimented with the parameters of the network in a similar way as the previous language modeling dataset. In particular, we changed the number of epochs to 126 and the batch size to 75 for the reasons analyzed above. Taking also into account the overfitting problem from the previous experiment we made an attempt to change the amount of validation data to 40% of the training data to examine weather the results will change in terms of loss equilibrium. To gain some idea of the network performance, some of the predicted output can be seen in Console 2.

Console 2 : Predicted Outcome

```
Input sentence:  Find Tom.
✔Decoded sentence:  Βρες τον Τομ.
Input sentence:  Grab him.
✔Decoded sentence:  ΄Αρπαξέ τον.
Input sentence:  Have fun.
✖ Decoded sentence:  ΄Εχετε αράστιις.
Input sentence:  He tries.
✔Decoded sentence:  Προσπαθεί.
Input sentence:  Help Tom.
✔Decoded sentence:  Βοηθήστε τον Τομ.
```

This output is indicative of the accuracy of the network and shows that it was able to predict most of the sentences correct. Implementing the aforementioned parameter tuning, the training loss and validation loss stood at 1.1 and 1.4 respectively in the first epoch, and kept fluctuating ever since. Validation loss took higher values relating to the previous unchanged script while training loss fluctuated around the same values as before. They did not distance as much as in the implementation of the unchanged script, but still there is overfitting. The last attempt to solve this problem was to increase the dropout from 0 to 0.5. Dropout is a regularization method where some of the network nodes are excluded

randomly during the weight estimating procedure with a given probability(in our case 50%). We observed that both training loss and validation loss kept dropping steadily and had smaller distance than the previous experimentations indicating a more accurate predictive power on the test set with training loss at 0.25 and validation loss at 0.72.

3. **Adding two short integers**

The last recurrent network is aiming to predict the outcome of an addition of two numbers. That is, given an input of the form "28 + 10" the network should be able to learn how to add those numbers and therefore, produce "38". First, we put in practice the unchanged script with all the default set parameters, `addition_rnn.py`. The number of iterations is equal to 199 and during those iterations we kept track of the accuracy, training and validation loss along with the predicted output of the network. While the first epochs kept running, it was observed that the network did not manage to produce the desired output and until the 13th iteration it kept returning wrong results, shown in Console 3.1:

Console 3.1 : Early Iterations

```
Q 268+62 T 330 ✔ 330
Q 349+45 T 394 ✖ 494
Q 48+12 T 60 ✔ 60
Q 278+80 T 358 ✖ 368
Q 534+51 T 585 ✔ 585
Q 89+692 T 781 ✖ 771
Q 71+417 T 488 ✔ 488
Q 645+4 T 649 ✖ 659
Q 584+125 T 709 ✔ 709
Q 556+6 T 562 ✔ 562
```

Console 3.2 :Final Iterations

```
Q 291+7 T 298 ✔ 298
Q 814+32 T 846 ✔ 846
Q 344+27 T 371 ✔ 371
Q 360+6 T 366 ✔ 366
Q 260+41 T 301 ✔ 301
Q 357+58 T 415 ✔ 415
Q 194+56 T 250 ✔ 250
Q 46+601 T 647 ✔ 647
Q 47+76 T 123 ✔ 123
Q 416+595 T 1011 ✔ 1011
```

Where T indicates the true value of the addition and the results can be seen in the last 'column' of the consoles. It is observed that the predicted values are quite close to the true ones with an accuracy of 89%. After some iterations the network made progress and learned to predict the correct outcome with at most one mistake, that is reaching an accuracy of 98-99% approximately. Finally from iteration 160 and so on it managed to compute all the desired outcomes, reaching 100% accuracy and producing outputs like the second table.

Consequently, we will train the network to learn other operations as well such like subtraction of two digits. Changing the parts of the script where the encoding was referring to the addition symbol, the outcome of the experimentation was more or less the same as the previous one. More precisely, as is also shown in the consoles below, the network did not have much difficulty to learn the subtraction operation thus, in the final iterations it achieved the correct prediction of the outcome. It was stabilized at accuracy 100% since iteration 114 while in the previous iterations it kept fluctuating at 98-99%.

Console 3.3 : Early Iterations

```
Q 823-48 T 775 ✔ 775
Q 27-724 T -697 ✔ -697
Q 36-47 T -11 ✔ -11
Q 78-889 T -811 ✔ -811
Q 89-979 T -890 ✔ -890
Q 786-4 T 782 ✔ 782
Q 573-7 T 566 ✔ 566
Q 114-694 T -580 ✔ -580
Q 46-575 T -529 ✔ -529
Q 402-181 T 221 ✖ 231
```

Console 3.4 : Final Iterations

```
Q 200-956 T -756 ✔ -756
Q 2-915 T -913 ✔ -913
Q 891-3 T 888 ✔ 888
Q 268-78 T 190 ✔ 190
Q 662-2 T 660 ✔ 660
Q 53-560 T -507 ✔ -507
Q 112-50 T 62 ✔ 62
Q 74-652 T -578 ✔ -578
Q 759-54 T 705 ✔ 705
Q 4-29 T -25 ✔ -25
```

Where, Q indicates the question referring to the integers of the operation and T stands for the true value of the outcome of the subtraction.

# 3 Task 3: Auto-Encoders

## 3.1 Intro

Auto-encoders are artificial neural networks used for unsupervised learning. The basic goal of auto-encoders is to learn to represent the same input. Of course the output will never be exactly the same as the input, due to the stochastic procedure of neural networks. More precisely, auto-encoders use the same architecture as classical neural networks (RNN, CNN, MLP) but the output layer has to be exactly the same (same number of nodes) as the input layer. In this way, the network compresses the input product in the hidden layer which has to be smaller compared to the input-layer, and finally it decodes the compressed material in order to approximately reconstruct it. So, auto-encoders aim in learning the initial structure of data, while they are trying to represent them and can be useful on pre-training weights and dimensionality reduction. They are also used on image-denoising and anomalies detection in datasets.

## 3.2 Experiments

We run three different experiments with various datasets and different auto-encoder architectures. The first experiment is made on bird-sounds dataset and all three datasets are listed below:

- bird sounds dataset
- cifar10 dataset
- noisy cifar10 dataset

1. **Simple autoencoder applied to bird-sounds dataset**

   Data consists of 8000 raw wav files each one 11 sec duration. After we imported the .wav files and transform them to a 21167 float-vector, we split them into 70% train and 30% test set respectively. The first auto-encoder we built on this dataset was a simple fully connected auto-encoder with three hidden layers. The encoder/decoder layer consists of 256 nodes while the other layers consist of 1000. All hidden layers use the sigmoid activation function except from the last one which uses the tanh. We used the ADAM optimizer and the mean absolute error for the loss function. After 200 epochs the network converges with train-loss: 0.7 and test-loss: 158 . Unfortunately this type of auto-encoder seems to fail in producing a representation of the input, probably because of the sound-series complexity but also because of the small number of the input sample. Moreover, a reasonable fact that impacts on network results is the big variance of the input vector which makes difficult the sound representation from the auto-encoder. The following images visualize the actual waveform in the first line and the predicted one in the second.

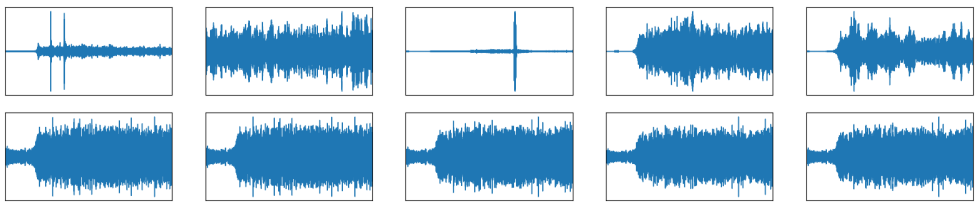


Figure 3: Simple auto-encoder output on bird-sounds

We can easily observe that the network can not represent the initial input data. Moreover, in occasions where the input signal is weak(e.g first and third), the network produces a noise-distortion, probably because it can not understand the structure of the raw signal. In some other cases, such as the second and the fifth signal, it seems that the network is able to detect some pitches and some signal-peaks but again it is not able to reproduce the same sound as the input. Clearly, there is no need to mention that raw sound data need lot of experimentation and detailed preprocessing in order to become manageable. So, after various tries of different feature extraction methods, such as, mel-frequency coefficients and fast-Fourier-transformations, we finally decided to feed the network with the original frequency-series of each wav file. The previous features are better for sound-classification and detection not for auto-encoders. This decision was made due to encoders/decoders peculiarities. The following figure visualizes the error function through the epochs.

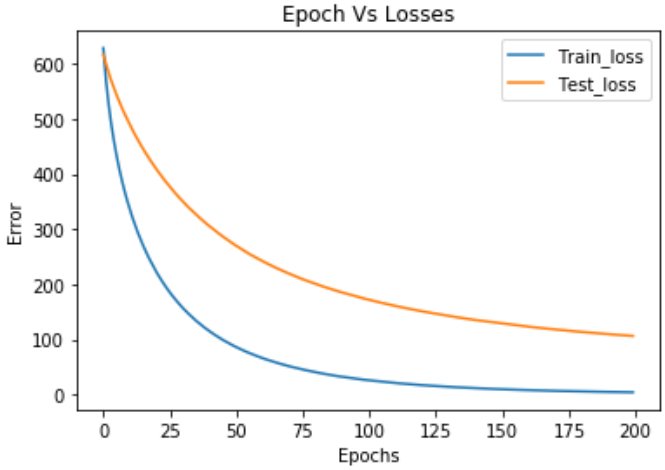



Figure 4: Simple auto-encoder loss_function

2. **Convolutional autoencoder applied to cifar10 dataset**

There are several famous image datasets which can be used in deep learning, classification and autoencoding. The one we chose for the implementations of this task is cifar10 dataset(also included in keras). The dataset consists of 50000 images of various things such as cars objects or animals, each image is 32×32 pixels, given in RGB representation.
The idea is to build an encoder-decoder network and train it to approximate the exact input image, as precise as possible. One difference here, comparing to other datasets like mnist, is that we have RGB images, thus the code should be adapted accordingly. At first, we tried to pass the data through the simple autoencoder used in the first experiment with the sound data. It is clear from figure 4 that this kind of autoencoder does not apply here because the structure of the data is more complex. Mainly it produces blur versions of the colors of the images. So, we will continue with convolutional autoencoding.

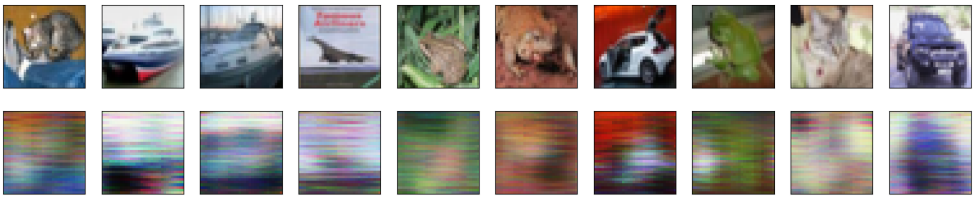


Figure 5: Simple auto-encoder output on cifar10 data

After normalizing the data, in order to take values between 0 and 1, we have two convolutional layers with 32 and 16 nodes which will 'scan' the input image with 3×3 matrices and then produce its encoded version. It should be pointed out, that as the dimensions of the convolution-kernel decrease, the prediction will be more detailed but also more ad-hoc. Proceeding to the decoding part, we chose again two convolutional layers with the same number of nodes and therefore, the training process should begin. The model was trained at 50 epochs using adam optimizer thus these choices give the optimal result and using this optimizer the algorithm converges faster. Proceeding in the deeper parts of the network, we chose relu activation function for the inner layers and MSE as the loss function. Of course, we experimented with other loss and activation functions for the decoded layer and the results in Table 2 led us to choose the ones which give the smaller error value in the test set.

| | sigmoid | softmax |
|---|---|---|
| MSE | 0.003 | 0.007 |
| Bin.Cross Entropy | 0.60 | 0.724 |

Table 2: Error Results with different activation functions on the decoded layer

Eventually, within 50 epochs the encoder-decoder network managed to produce approximately the same input images. The network has the parameters mentioned before and

consist of four layers in total. Taking a look at the below figure it is shown that there is a really small difference between input and output images.

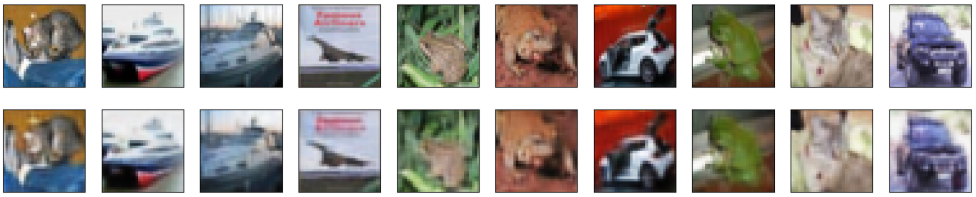


Figure 6: Convolutional auto-encoder output on cifar10 data

Finally the iterating procedure in terms of error functions can be visualized in the below plot. We have plotted the loss functions for training and test data versus the number of epochs for the chosen network structure.

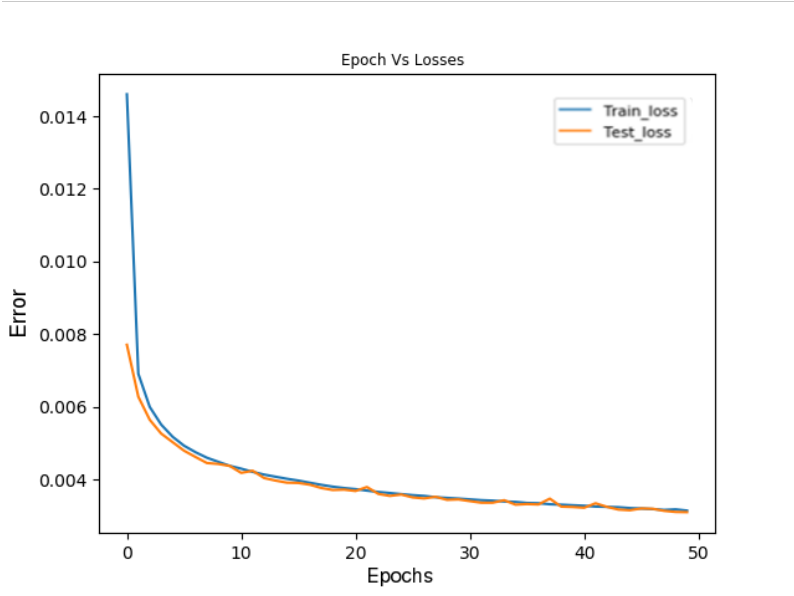



Figure 7: Convolution auto-encoder on cifar10 data

Both error functions begin with small values, 0.014 for train set and 0.008 approximately for test set. As the epochs increase the loss functions decrease sharply at first and steadily from iteration 10 until the final one. It can be seen that the algorithm converges at iteration 50 with loss function values standing at 0.003 for both sets.

3. **Application of autoencoder to image-denoising to cifar10 dataset**

For the final experiment, the cifar10 dataset is used again except that this time it will be used in image denoising. The idea is, to distort the initial input images by adding noise from the standard normal distribution. This way, we aim to 'confuse' the autoencoder in order to test whether it can perform as efficiently in distorted data as in the previous experiment. In this case we estimated that more epochs will be needed for the training process, because the input is distorted and it may take more time for the network to decode it. We trained the network in 100 epochs and by changing the same parameters as before we came up with the results in the test set are shown in the following table:

| | sigmoid | softmax |
|---|---|---|
| MSE | 0.011 | 0.079 |
| Bin.Cross Entropy | 0.572 | 0.723 |

Table 3: Error Results with different activation functions on the decoded layer

Given the above results, we ended up with the 4-layered encoder-decoder network, sigmoid activation function for the decoded layer and MSE loss function in combination with adam optimizer. The outcome can be seen in Figure 7 and we observe that the network performed quite well in the distorted images. Nevertheless, output images seem a little blurry but we can distinguish the general shape of the object in each image.

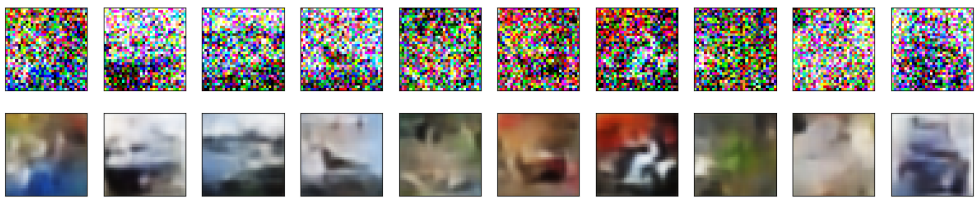


Figure 8: Simple auto-encoder for denoising on cifar10 data

Finally, the loss function of the chosen network structure in train and test data is visualized in the figure below.

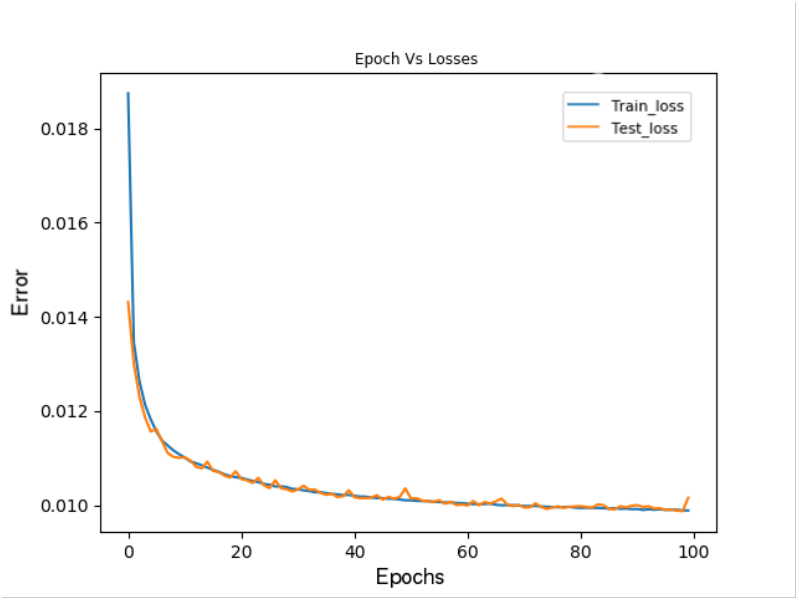



Figure 9: Convolutional auto-encoder on noisy cifar10 data

Both error functions seem to drop quite sharply in the first iterations and there are some fluctuations in the test set from iteration 20 and on. The loss functions have different starting values, 0.019 for train test and 0.0142 approximately for test set and in the final iteration the former converges at 0.009 while the latter flags a slight increase reaching 0.011.

# 4 Final Remarks

The aim of the analysis presented in this report, was to experiment with different datasets and different types of networks in order to obtain the desired results. We managed to examine which are the most misclassified digits in mnist data using MLP and CNN by combining some probabilistic approaches in Task 1. We also had the chance to make a little research for the problem of controlling the reproducibility of our results. Moreover, implementing recurrent networks in Task 2 drove us to train a network which was able to predict word sequences given two different kinds of input. In addition, we were able to predict the outcomes of arithmetic operations such as addition and subtraction. In the last Task 3, we experimented with autoencoders in three different cases and managed to train a network that is able to predict the same output as the input. To conclude, it should be mentioned that experimenting with the inner-network parameters such as, number of layers, epochs, different activation functions and so on, can be proved really crucial for the final results. In order to reach the desired output, we should take into account the peculiarities of the data, the target of each task and learn to distinguish between all network parameters.