Comparison of different types of auto-encoders for data cleaning

Kevin Alberts
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
k.j.alberts@student.utwente.nl

ABSTRACT

Using machine learning techniques for data cleaning has a lot of potential, for example in repairing corrupted data or restoring missing information. Previous research has given rise to a lot of different ways of using machine learning in this way, one of which being the auto-encoder. A lot of different types of auto-encoders have since emerged, which are usually tested on one dataset or compared to one other type. This begs the question which type is best and if auto-encoders can be used in a more general sense. In this research, we propose to experimentally compare five different auto-encoders (basic, sparse, contractive, denoising and variational) for cleaning and to see which types of auto-encoders are the most suited and most accurate for data cleaning for three different datasets, namely CIFAR-10, MNIST (images) and US Weather Data (tabular).
We implement a testing framework that allows easy implementation of different auto-encoders and datasets, and use this framework to test five different types of auto-encoders on two different image datasets. We find that for some types of auto-encoders there is no big difference in the type of dataset, but other types of auto-encoders work a lot better on certain types of data.

1 Introduction

1.1 Background

Data cleaning, is the process of detecting and removing errors and inconsistencies from data to improve the quality of the data [19]. In the past this has always been done by humans manually verifying the data and correcting mistakes, but humans can make mistakes or miss certain types of errors that are not as easily spotted, or datasets can become too large for humans to handle.
To avoid these mistakes, machine learning can be used to train artificial intelligence models to predict attributes of the data, based on the values of other attributes. If the amount of corruption in the original data is low, this can produce a good model even though there are errors in the data. After training, these models can then be used to correct suspicious values in the data. One type of model used is an auto-encoder. These models, when trained, generate a more general encoding of this data, which eliminates errors (see Section 3 for more details).
If we can create a method to correct corruption in data using machine learning, this has a lot of interesting use cases. One could for example imagine it being used to repair corrupted real-time sensor data that is sent over an unreliable connection. Or, one could even imagine using it to restore corrupted parts of photos, or even audio and video fragments. This intuitively can work because these types of data have a lot of patterns embedded in them, so a machine learning tool could be trained on the patterns from other, non-corrupted data to restore those patterns in the corrupted versions.

34th Twente Student Conference on IT Jan. 29st, 2021, Enschede, The Netherlands.

1.2 Related work

Automatic correction of data is a very interesting and useful topic. Previous research has been focused on developing new types of auto-encoders for de-noising images, video and audio data [16][17][23][20][13], or an existing type of auto-encoder is modified and then tested against one or more data sets for their accuracy in repairing the corrupted data [18][22][26][11]. Such research usually concludes with saying that the newly developed auto-encoder is indeed effective for data cleaning, but it is usually not clear if this auto-encoder is indeed better than other existing auto-encoders, or if it is maybe only better in certain cases or with certain types of data (i.e. structural, image, audio).

2 Research Question

Which types of auto-encoders are the best for repairing corruption in different types of datasets?

2.1 Sub-questions

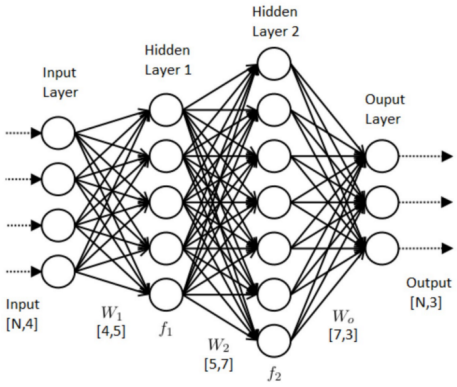
- What different types of auto-encoders exist that can repair corruption in a dataset?
- What different types of datasets can be used as inputs to auto-encoders?
- What parameters give the best results for each type of auto-encoder, based on previous research?
- How does each type of auto-encoder perform while repairing corruption in each dataset?
- How does the architecture of an auto-encoder impact the performance of it?

3 Background

3.1 Neural networks

A neural network is a machine learning structure consisting of layers of neurons. An example of a neural network with four input neurons, two hidden layers and three output neurons can be seen in Figure 1. It has an input layer, an output layer, and one or more hidden layers (f_1, f_2)

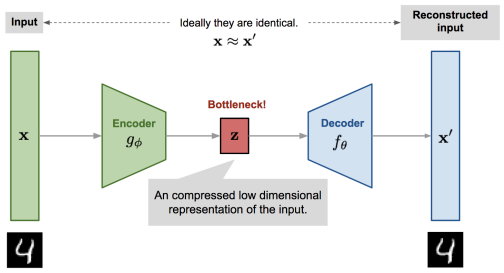
Figure 1: Example of a neural network [9]



in between. Each layer consists of neurons, which take a set of inputs, modifies them according to the weights of the connections (W_1, W_2, W_o), and then returns an output. Outputs of neurons can be modified by a propagation function before being used as inputs for other neurons, usually by adding a bias term [12]. The neurons in the output layer determine the result of the neural network. This can for example be the result of a classification (one output neuron per category), or an output image from an image generation neural network (one output neuron per output pixel). Weights are optimized by training the network on training data. Once the weights are all determined, a neural network is a really fast structure that is able to perform surprisingly complex tasks.

3.2 Auto-encoders

Figure 2: Structure of an auto-encoder [6]



An auto-encoder, as seen in Figure 2, is a type of neural network where the input and output layers have equal neurons. This allows it to produce output in the same format as the input. Furthermore, the hidden layers consist of less neurons than the input, which creates a *bottleneck*. This forces the network to figure out a more efficient encoding of the data [16]. The hidden layers consist of an encoder (g_ϕ), the bottleneck (z), and a decoder (f_θ).

The bottleneck makes sure that the input data needs to be compressed down into a lower number of neurons, which in theory means that any non-general data (errors, noise, etc) will be cut out. This is where the 'cleaning' comes from. The compressed representation is then decoded back to the original number of inputs, which is then the result. This result should then be a generalized form of the input, which in theory means that any noise, irregularities and corruption that does not line up with the rest of the data set is removed. Auto-encoders can be trained unsupervised, which is a major benefit for which saves time for researchers and makes it easier to implement them in practice.

3.3 Types of auto-encoders

Over the years, as more research in this field has been done, more and more different variations of auto-encoders have been devised. We give a short introduction of the most common types of auto-encoders.

3.3.1 *Basic auto-encoder*

This is the default version of the auto-encoder as described above, with no modifications.

3.3.2 *Sparse auto-encoder*

In a sparse auto-encoder, the bottleneck layer may actually have more neurons than the input data, but only a small number of them (usually the most active ones) are propagated to the decoding stage. This method allows the network to learn more complex patterns in the input data, because it has more neurons available to store the encoding in, while still preventing it from reconstructing the input using too many neurons [17]. It is shown that a sparsity factor using L1 regularization gives better results than the traditional KL divergence [25].

3.3.3 *Denoising auto-encoder*

In a denoising auto-encoder, a good representation is achieved by changing the reconstruction criterion. This means that instead of feeding the input directly into the auto-encoder, it is first corrupted using some form of noise. This corrupted version is then used as the input to train the auto-encoder. To evaluate the performance, the output of the network is compared to the original, uncorrupted version. By training the network in this way, it learns how to denoise the input, which leads to a better higher level representation of the input [23].

3.3.4 *Contractive auto-encoder*

A contractive auto-encoder adds an explicit regularizer to the inputs of the neurons that forces the model to be less sensitive to small variations of the input values. It essentially makes the neurons ignore small fluctuations in the data, and only react to larger, more important variations in the data. This 'penalty' is only applied during the training of the model, so when actually using the network it will not have any influence [20].

3.3.5 *Variational auto-encoder*

Variable auto-encoders produce a different form of encoding in their bottleneck layer. Instead of producing the encoding values directly, they produce a tensor of means, and a tensor of standard deviations. These two intermediate values are then used as a distribution, from which a sample is taken. This sample is then used as the input to the decoder network. This kind of auto-encoder is usually used for generative purposes (i.e. generating new images), but it can also be used for cleaning [13].

3.4 Types of noise

To test auto-encoders, usually a clean dataset is corrupted with some kind of noise. There are different noise types for each different data type. For this research, since we already have quite a few independent variables to test for (type of auto-encoder, type of data set), we constrain ourselves to one type of noise per data type. For image data we use *Gaussian noise* [14], which modifies each pixel to the value plus some random Gaussian distributed noise value. For structured data we will use *random errors* (where completely random fields are corrupted). The influence of other noise types can be part of further research.

4 The testing framework

We construct a testing framework, which is able to test an arbitrary implementation of an auto-encoder for an arbitrary dataset. Then, we can add our own implementation for each auto-encoder, add the datasets and automatically run the same tests on all of the different implementations. An added benefit of this is that we can collect exactly the metrics that we want from the frameworks, and it will be easily extendable with new types of encoders or datasets. The framework was built as a wrapper around the existing PyTorch classes, to make implementations as easy as possible. A class diagram of the framework can be found in Appendix A. The source code of the testing framework is available on GitHub [10].

The main structure is the `TestRun` class. This defines a combination of a dataset, encoder implementation and corruption type to use. Different test run instances can be configured in the configuration file of the framework. The dotted paths to the dataset, encoder and corruption classes that should be used are given there, as well as the parameter values for them (like the input shape). This allows a multitude of test runs for any combination of dataset, encoder or corruption.

The encoders are implemented as sub-classes of the `BaseEncoder` class, which is a subclass of the PyTorch `Module` class, which means they can be used directly in PyTorch. The class holds the network specification, optimizer and loss function, as well as methods to train and test the model, and hook functions that can be overridden to hook into specific points of the training or testing process. These hooks make it possible to implement all types of auto-encoders into the same framework. The default network consists of 4 layers, with the first two layers halving the input shape each time, and the last two layers doubling the neurons each time, thus creating the structure of an auto-encoder. The default optimizer is the Adam optimizer, and the default loss function is the Mean Squared Error loss. These are used by a lot of tutorials as a starting place[7][8][24], but of course these can be changed.

The datasets are implemented as sub-classes of the `BaseDataset` class, which itself is a subclass of the PyTorch `Dataset` class. The class offers two methods that return a training dataset or testing dataset, which are used during the model training and test runs. The rest of the implementation is mostly left to the individual dataset implementations, as the way datasets are structured can vary a lot (csv, images, etc). Three implementations for datasets are included in the framework. The `Cifar10Dataset` loads data from the CIFAR-10 dataset [1]. This dataset holds images and labels, but only the image data is used for this research. The `MnistDataset` loads its data from the MNIST dataset [4]. The `USWeatherEventsDataset` is built on the LTSW dataset [3]. The current implementation only uses the weather event type, severity, timezone and state. These columns are encoded using One-Hot encoding, so they can be used in a neural network [5]. This dataset does not use the default Mean Squared Error loss when training, but the Cross Entropy Loss, which is meant for data encoded in One-Hot encoding [21].

The corruption methods are implemented as sub-classes of the `BaseCorruption` class. This is a class which has two methods to corrupt one instance from a dataset, and to corrupt an entire dataset. The exact method of corruption is left up to the implementations. Two implementations are included in the framework; `NoCorruption`, which does not corrupt the data in any way, and `GaussianCorruption`, which adds noise taken from a Gaussian distribution [14].

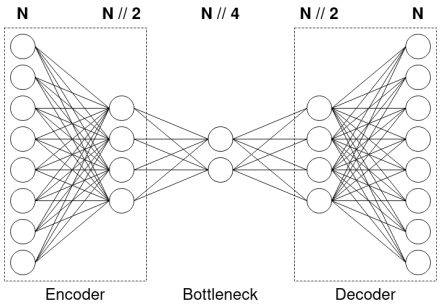
5 Methodology and approach

5.1 Implementations of auto-encoders

For this experiment, we want to implement one of each type of auto-encoder explained in Subsection 3.3, using the parameters from previous research.

5.1.1 *Basic auto-encoder*

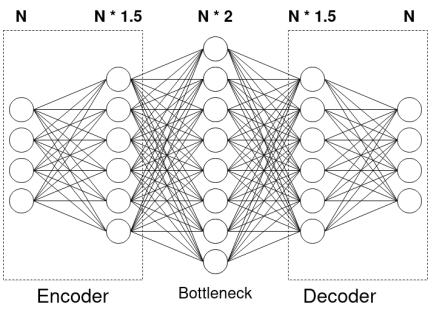
Figure 3: Basic auto-encoder network



This implementation is based on the original auto-encoder layout as proposed by Kramer[16]. The layout of the network is illustrated in Figure 3, where N is the number of inputs (for a 10 by 10 pixel image with 3 colour channels, $N = 300$, so the second layer would have $N//2 = 150$ neurons, and the bottleneck layer would have $N//4 = 75$ neurons). This is the same auto-encoder as implemented in the the base framework, so no modifications are necessary. It has 4 layers, connected with a ReLU (Rectified Linear Unit) activation function. This activation function will return 0 if the input is negative, or the input as-is if the input is positive. The two encoder layers halve the neurons each time, and the two decoder layer doubles the neurons each time. Neuron outputs are only influenced by their weights, nothing else.

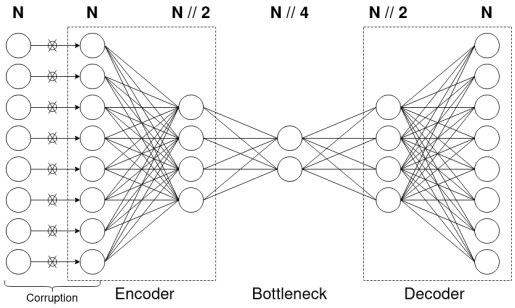
5.1.2 *Sparse L1 auto-encoder*

Figure 4: Sparse L1 auto-encoder network



This implementation is based on the sparse auto-encoder with an L1 regularization term as the sparsity factor [25]. The layout of the network is illustrated in Figure 4. As can be seen this encoder has more neurons in the bottleneck than the input and output. For a 10 by 10 pixel image with 3 colour channels ($N = 300$), the second layer has $N * 1.5 = 450$ neurons and the bottleneck layer has $N * 2 = 600$ neurons. To prevent input copying, these neurons are limited by the sparsity factor, which in this case is the L1 regularization. In practice this is achieved by adding the sparsity penalty to the loss when the network is trained. This is done by multiplying a regularization parameter (set to 0.001 by default) with the L1 loss, which is the sum of the means of the absolute values of each layer in the network. The rest of the auto-encoder behaves just like a regular auto-encoder network.

Figure 5: Denoising auto-encoder network



5.1.3 *Denoising auto-encoder*

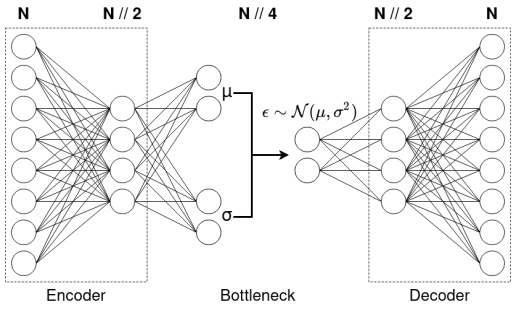
This implementation is based on the original definition of the stacked denoising auto-encoder [23]. The layout of the network is illustrated in Figure 5. As can be seen the input is manipulated before it enters the encoder (At the connections with the $\otimes$ symbol). The input data is corrupted before entering the encoder, but the end result is compared to the uncorrupted image. This makes the auto-encoder learn how to denoise the image. This corruption is only done during training. The rest of the network functions the same as the basic auto-encoder. Implementing this encoder only requires an extra parameter, namely the type of corruption to apply to the network. The only thing different from the basic auto-encoder in the implementation is an overridden `process_train_features` method, where the input to the network during training is corrupted.

5.1.4 *Contractive auto-encoder*

The contractive auto-encoder implementation is based on the original definition [20]. The network layout is exactly the same as the basic auto-encoder from Figure 3. The only implementation difference is in the loss function, where the regular MSE loss is summed with the weighted L2-norm of the Jacobian of the hidden units with respect to the inputs. The technical implementation of this loss function is inspired by an implementation by A. Kristiadi [2].

5.1.5 *Variational auto-encoder*

Figure 6: Variational auto-encoder network



This implementation is based on the standard variational auto-encoder proposed by Doersch [13]. The network, shown in Figure 6, is split up at the bottleneck. The mean (μ) and variance (σ^2) are calculated from the outputs of the encoder. These variables are used as the mean and variance of a normal distribution. Then, instead of passing the outputs of the encoder to the decoder, a sample is taken from that normal distribution and passed to the decoder. The rest of the network functions as a normal auto-encoder. Details for the implementation of the reparameterization and the KL-divergence were taken from [13] and [15].

5.2 Chosen datasets

For this experiment we want varied datasets so we can make conclusions about the accuracy on different data types. The datasets should be usable with all or the majority of the auto-encoders we want to test. Three datasets were chosen and implemented for this research. Two with image data (one general and one specialized), and one dataset with tabular data.

The CIFAR-10 dataset contains small 32x32 pixel colour images of everyday things like cars, cats and ships. This dataset was chosen because it is very broad. This makes it harder for the neural network to converge, and it will be interesting to see if any type of auto-encoder will get good results with such a general dataset. This dataset contains 50000 images for training, and 10000 images for testing.

The MNIST dataset has been used frequently when testing auto-encoders. It consists of handwritten numbers from 0 to 9. This dataset was chosen because it is commonly used to test the accuracy of auto-encoders, and this will give a good baseline value of the performance of the encoder. This dataset contains 60000 images for training, and 10000 images for testing.

The US weather dataset is based on tabular data of weather events in the United States. It contains data about the type of event (e.g. rain, snow, fog), the severity (e.g. light, moderate, severe), the timezone (e.g. eastern, central, pacific), and the state (e.g. Arizona, California, Texas). The original data contains more columns but these were left out due to implementation difficulties and time constraints. The original dataset contains more than 5 million entries, but for this research 250000 random entries were chosen for both the training and testing dataset.

5.3 Metrics

For image data the accuracy is calculated using the SSIM (Structural Similarity Index). This score considers the luminance, contrast and structure of two images and combines these [27]. It is meant to be used to compare the visual similarity of two images for human eyes, given as a number between -1 and 1, where -1 means the image is completely different from the other, and 1 means they are practically the same image.

For tabular data the accuracy is calculated by comparing each cell to the original value, and taking the percentage of correct values, so the amount of correct cells divided by the amount of total cells.

6 Results

To get these results, each auto-encoder is trained for 50 epochs using the training data from the dataset. Then, the testing data from the dataset is corrupted, and run through the trained network. The resulting images are compared to the original images and the average accuracy score is calculated. The training loss for each auto-encoder over the 50 epochs have been graphed and they can be found in Appendix B. For the image datasets, a sample of the original, corrupted, and reconstructed images from all implemented auto-encoders can be found in Appendix C. The average SSIM (for image datasets) and accuracy (for tabular dataset) scores that were achieved on the testing datasets can be seen in Figure 7. The numerical data that makes up this chart can be found in Figure 8. As can be seen, the scores for the MNIST dataset are consistently higher than the scores for the CIFAR-10 dataset. Performance on those datasets seems to be similar for the basic, sparse and contractive auto-encoders, but the denoising auto-encoder has a lot better score on the MNIST dataset,

Figure 7: Accuracy scores for all implemented auto-encoders and datasets.

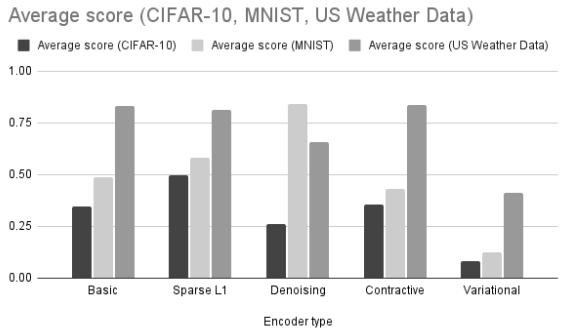


Figure 8: Accuracy scores for all implemented auto-encoders and datasets. (rounded to 4 decimals)

Encoder type	CIFAR-10	MNIST	US Weather
Basic	0.3441	0.4898	0.8320
Sparse L1	**0.4962**	0.5817	0.8145
Denoising	0.2631	**0.8431**	0.6582
Contractive	0.3533	0.4327	**0.8379**
Variational	0.0804	0.1225	0.4141

while having the second lowest score on CIFAR-10. Also, note that the score of the variational auto-encoder is very low in all three datasets compared to the other encoders. Lastly, it is interesting to note is that the scores for the US weather dataset seem to be pretty good and quite high across the board when compared to the image datasets, though this is misleading as will be discussed in the next section.

7 Analysis

Overall, the simplicity of the implemented networks lead to quite moderate scores on all encoders. To get higher scores more hidden layers are required, and one needs to do hyper-parameter optimization for each specific dataset. This all was not done for this experiment, as it is not necessary for a global comparison of the types. Also, in all cases except the US weather data the default Mean Squared Error loss was used during training to keep implementations as simple as possible, while there may be other types of loss functions that are better suited for images, like SSIM, which was used for the score calculation in the end.

Most types of encoders perform similarly for all datasets, and similar to each other. The basic auto-encoder, sparse L1 auto-encoder and the contractive auto-encoder all have roughly similar performances on the all datasets. It is quite interesting to see that even the most basic auto-encoder still performs pretty well.

The most interesting thing is the large difference in the performance of the denoising auto-encoder. This type of auto-encoder seems to be very good at cleaning when used with the MNIST dataset, but not very good at all with the CIFAR-10 dataset. This is probably due to the simplicity of the images in the MNIST dataset, where clean images only consist of a black background and white lines, so Gaussian noise can be easily recognised and removed. This does not work well with the CIFAR-10 dataset because those are pretty varied pictures with a lot of colours and shapes, so the denoising auto-encoder cannot find large enough differences between a noisy image and some other image that just looks different.

Another interesting thing is that the scores for the US weather dataset seem to be a lot higher than the image datasets. This however does not mean that the encoders perform a lot better on this type of data. The dataset is corrupted using a random corruption, which is implemented so that it corrupts zero columns in 10% of the rows, one column in 90% of the rows, and it has a 10% chance of corrupting a second column if one was already corrupted. So, if we just compare a regular row to a corrupted row, we would expect a score of around 0.9, without running it through any kind of auto-encoder. However, all scores we measured are lower than 0.9, which means that the auto-encoders actually are corrupting cells that were not even wrong. This could probably be avoided in any number of ways, but it is a side-effect of auto-encoders operating on the entire row, and reconstructing the entire row of data in one go.

It can also be seen that the variational auto-encoder was not able to converge on a well-performing model in all tests. As can be seen in the sample images from Appendix C, every reconstructed image of this auto-encoder is very similar to each other and not similar to the original input at all. This is especially clearly visible in the MNIST reconstruction. This is probably due to the random sampling from the distribution, maybe requiring more training epochs or some parameter optimization, but obviously programming errors are always a possible cause as well.

8 Conclusion

This research has shown that it depends greatly on the type of dataset which forms of auto-encoder works the best. A denoising auto-encoder works a lot better on a dataset with similar-looking images, but this type of auto-encoder does not work well at all for datasets with a lot of variation. Conversely, other types of encoders like the basic auto-encoder or the sparse L1 auto-encoder work similarly well on datasets with both varied and similar images, but give overall less accuracy than if an auto-encoder that is better at handling a specific type of dataset is used. We also show that auto-encoders can effectively be used on tabular data, but extra measures have to be taken to ensure that the encoders do not accidentally corrupt data that was valid in the first place.

Furthermore, this research resulted in a testing framework where different types of auto-encoders, datasets and corruption methods can easily be implemented, trained and tested. This framework can allow future research to quickly train, test and compare new types of auto-encoders or even more general neural networks, or to test existing implementations against new datasets or with new types of corruption.

8.1 Future work

Researchers that are interested in furthering this research could look into implementing more types of auto-encoders into the framework, or do hyper-parameter optimization on the existing auto-encoders to find out if the score differences become more apparent.

Furthermore, researchers can implement new datasets of different types. This research focused on image and tabular data, but it would be very interesting to see if auto-encoders are also effective at removing noise or correcting errors in for example audio data. Also, the tabular data used was only of a categorical type, it is very interesting to see if this also works for other types of columns like timestamps, or actual measurement values.

And lastly, this research did not compare these implementations of the different auto-encoders to other implementations of the same type of encoder. It might be interesting to see if there are implementation differences that impact the performance of the encoders.

9 References

[1] CIFAR-10 and CIFAR-100 datasets. Accessed: 23-01-2021. URL: https://www.cs.toronto.edu/~kriz/cifar.html.

[2] Deriving Contractive Autoencoder and Implementing it in Keras - Agustinus Kristiadi's Blog. Accessed: 23-01-2021. URL: http://wiseodd.github.io/techblog/2016/12/05/contractive-autoencoder/.

[3] LSTW: Large-Scale Traffic and Weather Events Dataset - Sobhan Moosavi. Accessed: 23-01-2021. URL: https://smoosavi.org/datasets/lstw.

[4] MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges. Accessed: 23-01-2021. URL: http://yann.lecun.com/exdb/mnist/.

[5] What is One Hot Encoding? Why and When Do You Have to Use it? | Hacker Noon. Accessed: 23-01-2021. URL: https://hackernoon.com/what-is-one-hot-encoding-why-and-when-do-you-have-to-use-it-e3c6186d008f.

[6] From Autoencoder to Beta-VAE, August 2018. Accessed: 23-01-2021. URL: https://lilianweng.github.io/2018/08/12/from-autoencoder-to-beta-vae.html.

[7] Implementing Deep Autoencoder in PyTorch -Deep Learning Autoencoders, December 2019. Accessed: 23-01-2021. URL: https://debuggercafe.com/implementing-deep-autoencoder-in-pytorch/.

[8] Abien Fred Agarap. Implementing an Autoencoder in PyTorch, October 2020. Accessed: 23-01-2021. URL: https://medium.com/pytorch/implementing-an-autoencoder-in-pytorch-19baa22647d1.

[9] Jayesh Bapu Ahire. The Artificial Neural Networks handbook: Part 1, September 2020. Accessed: 23-01-2021. URL: https://medium.com/coinmonks/the-artificial-neural-networks-handbook-part-1-f9ceb0e376b4.

[10] Kevin Alberts. Kurocon/AutoEncoderComparison, January 2021. Accessed: 23-01-2021. URL: https://github.com/Kurocon/AutoEncoderComparison.

[11] J. Dai, H. Song, G. Sheng, and X. Jiang. Cleaning Method for Status Monitoring Data of Power Equipment Based on Stacked Denoising Autoencoders. *IEEE Access*, 5:22863–22870, 2017. Conference Name: IEEE Access. doi:10.1109/ACCESS.2017.2740968.

[12] CHRISTIAN W. DAWSON and ROBERT WILBY. An artificial neural network approach to rainfall-runoff modelling. *Hydrological Sciences Journal*, 43(1):47–66, February 1998. Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/02626669809492102. doi:10.1080/02626669809492102.

[13] Carl Doersch. Tutorial on Variational Autoencoders. *arXiv:1606.05908 [cs, stat]*, August 2016. arXiv: 1606.05908. URL: http://arxiv.org/abs/1606.05908.

[14] Sukhjinder Kaur. Noise types and various removal techniques. *International Journal of Advanced Research in Electronics and Communication Engineering (IJARECE)*, 4(2):226–230, 2015. Publisher: Citeseer.

[15] Diederik P. Kingma and Max Welling. Auto-Encoding Variational Bayes. *arXiv:1312.6114 [cs, stat]*, May 2014. arXiv: 1312.6114. URL: http://arxiv.org/abs/1312.6114.

[16] Mark A. Kramer. Nonlinear principal component analysis using autoassociative neural networks. *AIChE journal*, 37(2):233–243, 1991. Publisher: Wiley Online Library.

[17] Alireza Makhzani and Brendan Frey. k-Sparse Autoencoders. *arXiv e-prints*, 1312:arXiv:1312.5663, December 2013. URL: http://adsabs.harvard.edu/abs/2013arXiv1312.5663M.

[18] Ben Poole, Jascha Sohl-Dickstein, and Surya Ganguli. Analyzing noise in autoencoders and deep networks. *arXiv:1406.1831 [cs]*, June 2014. arXiv: 1406.1831 version: 1. URL: http://arxiv.org/abs/1406.1831.

[19] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[20] Salah Rifai, Xavier Muller, Xavier Glorot, Gregoire Mesnil, Yoshua Bengio, and Pascal Vincent. Learning invariant features through local space contraction. *arXiv:1104.4153 [cs]*, April 2011. arXiv: 1104.4153. URL: http://arxiv.org/abs/1104.4153.

[21] Stacey Ronaghan. Deep Learning: Which Loss and Activation Functions should I use?, August 2019. Accessed: 23-01-2021. URL: https://towardsdatascience.com/deep-learning-which-loss-and-activation-functions-should-i-use-ac02f1c56aa8.

[22] Dan Stowell and Richard E. Turner. Denoising without access to clean data using a partitioned autoencoder. *arXiv:1509.05982 [cs]*, September 2015. arXiv: 1509.05982. URL: http://arxiv.org/abs/1509.05982.

[23] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, Pierre-Antoine Manzagol, and Léon Bottou. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of machine learning research*, 11(12), 2010.

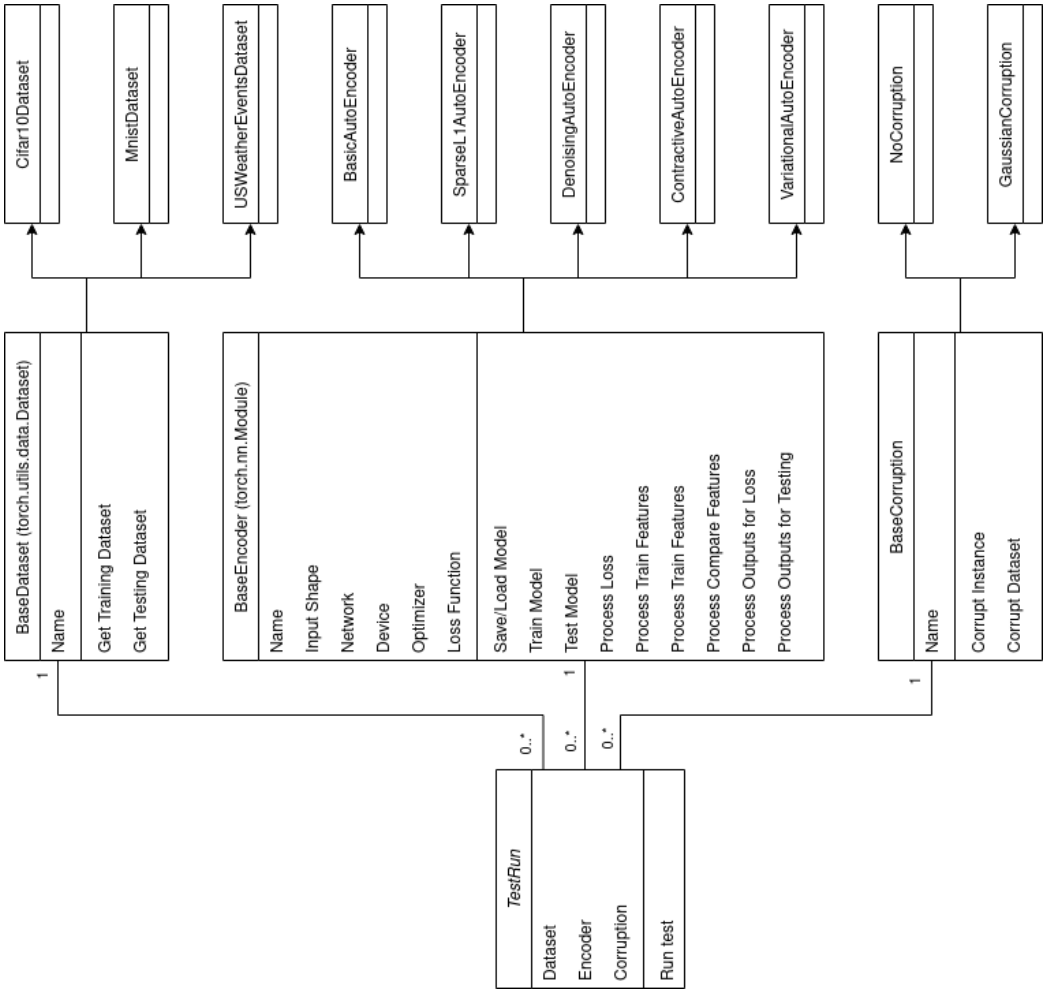
[24] Orhan G. Yalçın. Image Noise Reduction in 10 Minutes with Convolutional Autoencoders, November 2020. Accessed: 23-01-2021. URL: https://towardsdatascience.com/image-noise-reduction-in-10-minutes-with-convolutional-autoencoders-d16219d2956a.

[25] Li Zhang, Yaping Lu, Bangjun Wang, Fanzhang Li, and Zhao Zhang. Sparse Auto-encoder with Smoothed $$l_1$$l1Regularization. *Neural Processing Letters*, 47(3):829–839, June 2018. doi:10.1007/s11063-017-9668-5.

[26] Chong Zhou and Randy C. Paffenroth. Anomaly Detection with Robust Deep Autoencoders. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 665–674, New York, NY, USA, August 2017. Association for Computing Machinery. doi:10.1145/3097983.3098052.

[27] Zhou Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, April 2004. Conference Name: IEEE Transactions on Image Processing. doi:10.1109/TIP.2003.819861.

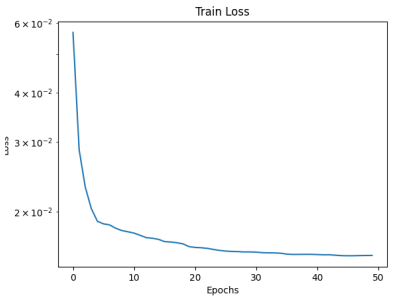
Appendix A: Testing framework class diagram



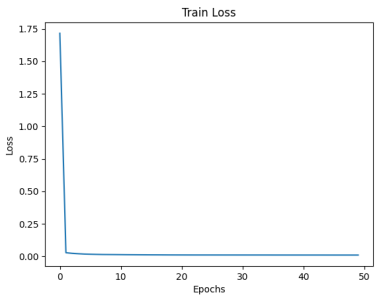
Appendix B: Loss graphs for the training of the auto-encoders.

Figure 9: Loss graphs - CIFAR-10

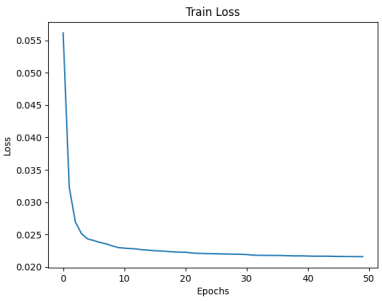
(a) Basic auto-encoder



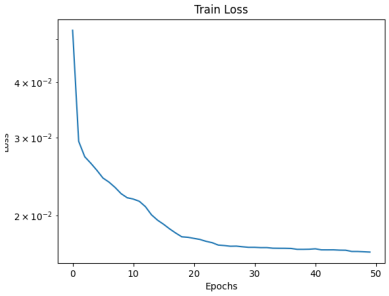
(b) Sparse L1 auto-encoder



(c) Denoising auto-encoder



(d) Contractive auto-encoder



(e) Variational auto-encoder

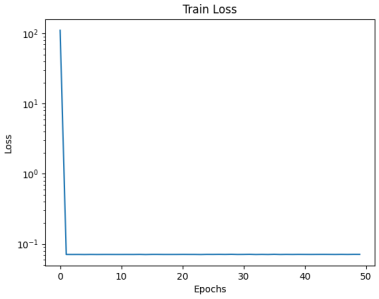
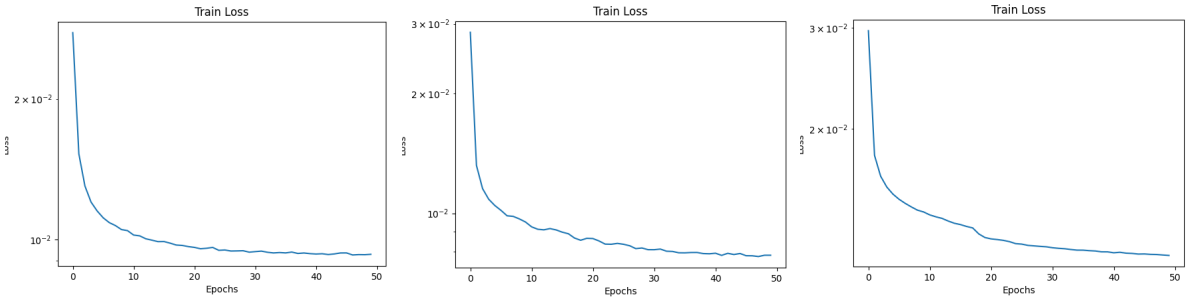


Figure 10: Loss graphs - MNIST

(a) Basic auto-encoder (b) Sparse L1 auto-encoder (c) Denoising auto-encoder



(d) Contractive auto-encoder (e) Variational auto-encoder

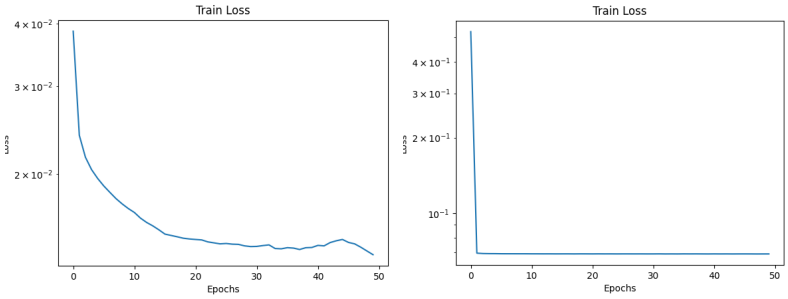
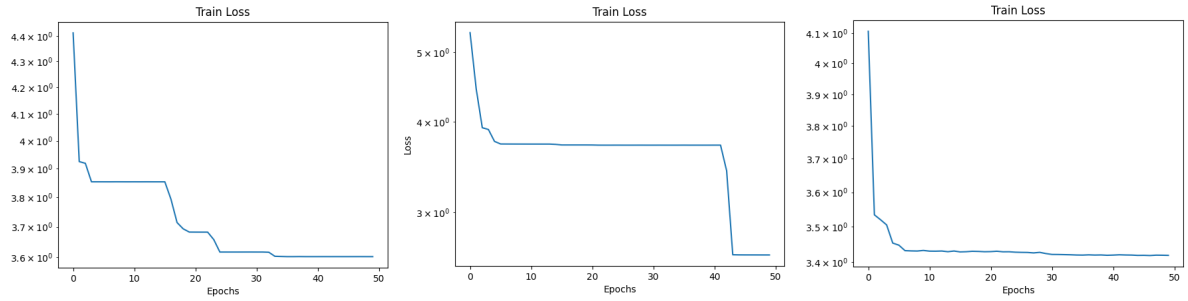
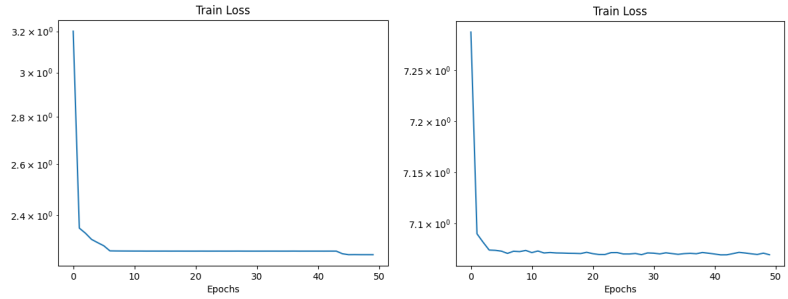


Figure 11: Loss graphs - US Weather Data

(a) Basic auto-encoder (b) Sparse L1 auto-encoder (c) Denoising auto-encoder



(d) Contractive auto-encoder (e) Variational auto-encoder



Appendix C: Reconstructed images from the implemented auto-encoders.

Figure 12: Original, corrupted and reconstructed test images - CIFAR-10

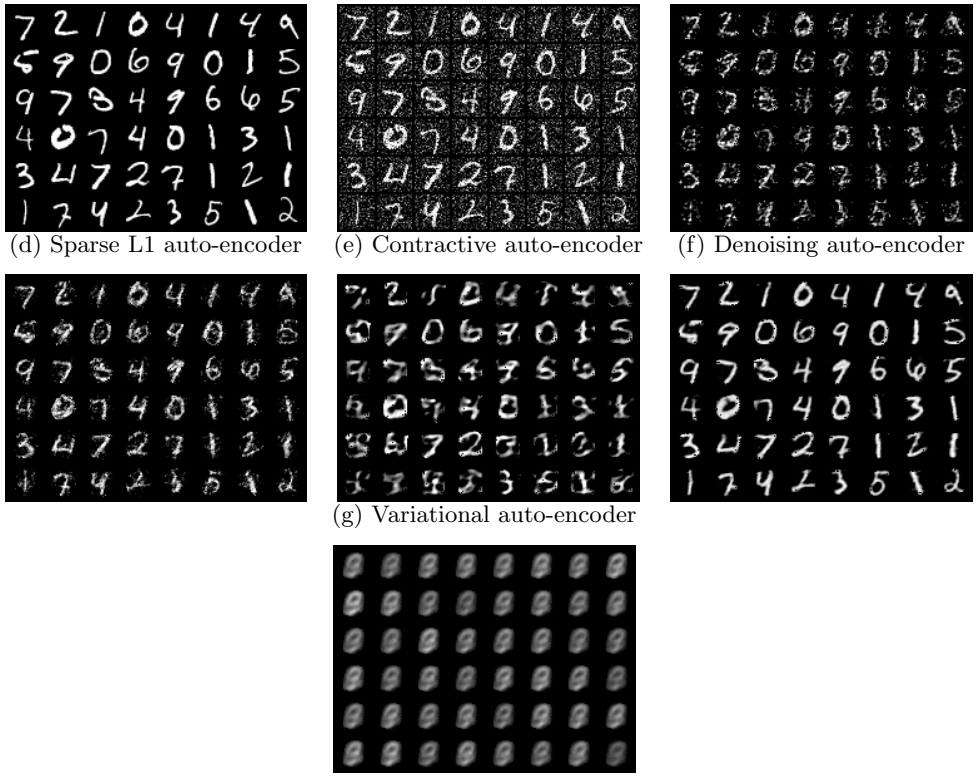
(a) Original (b) Corrupted (c) Basic auto-encoder

(d) Sparse L1 auto-encoder (e) Contractive auto-encoder (f) Denoising auto-encoder

(g) Variational auto-encoder

Figure 13: Original, corrupted and reconstructed test images - MNIST

(a) Original (b) Corrupted (c) Basic auto-encoder



(d) Sparse L1 auto-encoder (e) Contractive auto-encoder (f) Denoising auto-encoder

(g) Variational auto-encoder