ulm university universität
uulm

Ulm University | 89069 Ulm |Germany

Faculty of
Engineering, Computer Science and Psychology
Neural Information Processing

Iterative autoencoder networks for signal denoising

Bachelor thesis at Ulm University

Submitted by:
Manuel Lautaro Hickmann
lautaro.hickmann@uni-ulm.de
953591

Reviewer:
PD. Dr. Friedhelm Schwenker

2019

Version from December 16, 2019

Satz: PDF-LaTeX 2$_\varepsilon$

Abstract

English version

Inspired by the popularity of autoencoders and the properties of Denoising autoencoders [1], it was tried to corroborate that repeating the evaluation would cause the results to converge to a better solution. This was done by analysing the effects of iterative training and evaluating on the task of denoising a corrupt input, simulated by corrupted MNIST examples. Multiple architectures and noises were tested. It was discovered that, without any special constraints, the reconstruction of autoencoders tend to diverge. Nevertheless a special kind of dataset corruption used for training, represent all images of a class of numbers by one selected example of the same class, denominated 10_Targets, showed promising outcomes. To better evaluate the quality of the reconstructions, the encoded data, output of the encoder section of the autoencoder, was used as basis to generate a manifold and as input for a classifier. The proposed training dataset shows encouraging natural clustering. It cannot be clearly shown whether iterations offer a better solution, since the reconstructions behave very differently for different examples and do not follow any general norm. To further investigate this, additional experiments with another type of neural network (CNN) are suggested that could achieve improvements.

Deutsche Version

Inspiriert durch die Popularität von Autoencoder und den Eigenschaften von Denoising-Autoencoder [1] wird versucht zu bestätigen, dass eine Wiederholung der Evaluierung zu besseren konvergierenden Lösungen führen kann. Dazu werden die Auswirkungen von iterativem Training und Evaluierung auf die Aufgabe eine fehlerhafte Eingabe zu entrauschen analysiert. Diese werden durch korrupte MNIST-Beispiele simuliert. Es werden mehrere Architekturen und verschiedene Typen von Rauschen getestet. Dabei wird festgestellt, dass die Rekonstruktionen durch Autoencoder ohne spezielle Einschränkungen bei iterative Evaluierung tendenziell divergieren. Trotzdem zeigt eine besondere durch Verrauschung erzeugte Trainingsmenge, bei der alle Elemente einer Zahlenklasse durch ein ausgewähltes Beispiel derselben Klasse dargestellt werden, denominiert 10_Targets, vielversprechende Ergebnisse. Um die Qualität der Rekonstruktionen besser beurteilen zu können, werden die codierten Daten (die Ausgabe vom Codiererabschnitt des Autoencoders) als Grundlage zur Erzeugung einer Mannigfaltigkeit (manifold) und als Eingabe für einen Klassifizierer verwendet. Der vorgeschlagene Trainingsdatensatz zeigt ein ermutigendes natürliches Clustering. Es kann nicht eindeutig gezeigt werden, ob durch die Iterationen eine Optimierung erreicht wird, da sich die Rekonstruktionen bei verschiedenen Beispielen sehr unterschiedlich verhalten und keiner allgemeinen Norm folgen. Um dieses weiter zu untersuchen werden zusätzliche Experimente mit einer anderen Art von neuronale Netzen(CNN) vorgeschlagen, die dies beheben könnten.

Acknowledgement

I wish to express my sincere thanks to PD Dr. Friedhelm Schwenker, for sharing expertise, guidance and encouragement which helped me accomplish this thesis. I also thank my parents for the unceasing encouragement, support and attention.

Contents

1 Introduction

Artificial neural network is a mathematical and computational model composed of a large number of neurons that can simulate the structural and functional characteristics of biological neural network. They are a self-adaptive system, which changes the internal structure according to the external input, and is commonly used to model complex relationships between input and output.

The training of multi-layered networks was in general unsuccessful, with the exception of convolutional neural networks (CNN), until 2006, when breakthroughs were made by [2], [3] and [4] among others breaking a decade of almost no advance. A greedy layer-wise pre-training was proposed to initialize the weights of an entire network in an unsupervised manner, followed by a supervised back-propagation step. From then on, deep learning sweep across the industry and the academia like a wave. Recent research results have also demonstrated that deep learning indeed achieves state-of-the-art performances among various areas ([1], [5], [6], [7],[8], [9], [10], [11], [12], [13],[14]). Particularly, deep learning has already been successfully applied to the industry in speech area. Also achieving very favourable results in the field of image analysis and emotion recognition.

One big advantage of Neural Networks is their capability to analyse complex signals and images compared to conventional algorithms. But this presents also one of their biggest downsides then adversarial examples can very easily convert a very good classifier or estimator, among others, to a useless system. As explained in [15] several machine learning models, including state-of-the-art neural networks misclassify examples that are only slightly different from correctly classified examples drawn from the data distribution. This suggests that adversarial examples expose fundamental blind spots in the training algorithms. Normally adversarial examples are built with the objective of "breaking" such models but they can also happen because of missing or corrupted parts in the input signals, i.e. the kind of corruption is not present in the training phase.

Multiple constructs are proposed to solve this problem utilizing autoencoders as [16] and Denoising Autoencoders [9]. In this thesis an iterative approach to signal denoising is proposed, under the idea that repeating the evaluation of a model would cause the results to converge to a better solution, i.e. more similar to the uncorrupted signal.

1.1 Problem statement

Multiple system rely on correct input signal to accomplish their task, as for example self driving cars. Therefore clean reliable input data, signals, are necessary. As a solution to this problem an iterative approach is proposed. Unfortunately it was discovered that this tends to produce mostly divergent results. Therefore multiple architectures and corruptions are tested to determine which present better results.

1.2 Contributions and results

It was concluded that without special constraints, the iterative reconstruction of autoencoders tend to diverge. It could not be clearly shown whether iterations offer an improved solution, since the reconstructions behave very

differently on examples, not only on distinct examples but also on the same example when applied iteratively, and do not follow any general criterion. Nevertheless under specific constraints encouraging behaviour can be observed. To further investigate this, additional experiments with other types of neural networks, for example convolutional neural networks(CNN), and more complex constraints are suggested that could achieve improvements.

1.3 Structure of the thesis

In chapter 2 the theory fundamentals used in this thesis are explained and in section 2.2 the autoencoders and its properties presented. Then in chapter 3 the structure and idea behind the experiments are given and the choices of hyperparameters, in section 3.3 and frameworks , in section 3.9 justified. Continuing in chapter 4 the results of said experiments are presented in section 4.1 and in section 4.4 the conclusions are drawn and I reflect over the challenges and learnings of making this thesis. Finally in Chapter 5 possibilities how this architectures could be improved are presented.

2 Theory Fundamentals

Neural networks training is divided in two big groups on one hand unsupervised and on the other supervised learning algorithms, with a third in between option of semi-supervised algorithms. As described in [17] the unsupervised ones experience a dataset containing many features, then learn useful properties of the structure of this dataset. In the context of deep learning, the goal is to try and learn the entire probability distribution that generated the dataset, whether explicitly, as in density estimation, or implicitly, for tasks like synthesis or denoising. Supervised learning algorithms experience a dataset containing features, but each example is also associated with a label or target. For example the MNIST dataset is annotated with the class of each number, i.e. what number the image represents. A supervised learning algorithm can study the MNIST dataset and learn to classify the handwritten digits into ten different classes, $0, 1, \cdots, 9$. On the other hand if the labels are ignored a unsupervised algorithm, as are autoencoders, can be trained to reconstruct the images.

2.1 Deep Learning

A major source of difficulty in many real-world AI (artificial intelligence) applications, as denoted in [17], is that many of the variable factors influence every single piece of observable data. For example the colour of objects change depending on lighting or dust on a camera produces a non standard image. Therefore most applications require the separation and filtering of variation factors, remaining with useful ones to achieve a given task. Often discriminating these high-level, abstract features from raw data is very complicated, and a nearly human-level understanding of the data is required. In this case, when obtaining a representation is as difficult as solving the original problem, learning a representation of the data does not appear helpful. However recent theoretical studies indicate that deep architectures may be needed to efficiently model complex distributions and achieve better generalization performance on challenging recognition tasks, as described in [1].

Deep learning solves this central problem in representation learning, following [17], by introducing representations that are expressed in terms of other, simpler representations. Deep learning enables the systems to build complex concepts out of simpler concepts. It can be regarded as the study of models that involve a greater amount of composition of either learned functions or learned concepts than traditional machine learning does. Nevertheless deep learning is kind of like a “black-box” and there is no strict theoretical system to support it. So a problem as mentioned by [5] is: "we have impressive performance using deep learning but we do not know why theoretically". In deep learning, a number of researchers tend to make progress by employing increasingly deep models and complex unsupervised learning algorithms.

Further as connoted in [8] deep learning consist of multiple hidden layers of representation, it greatest advantage is that the features of each hidden layer are not designed manually, rather they are learned from the input data automatically.

Deep learning algorithms achieve state-of-the-art results and received much attention in the machine learning and applied statistics literature in recent years. They are used for research in dimensionality reduction, as [5], [6], and

[7], analyse and predict machinery faults [8], multiples are concentrated in extracting features and representations, as [1], [9], [10], [11], and [12], or clustering data [13], as a recommender system for Netflix [14], or for multimodal affection recognition and annotation as [18], and [19].

Training of multi-layered networks was in general unsuccessful, with the exception of convolutional neural networks (CNN), until 2006, when breakthroughs were made by [2], [3] and [4] among others, by proposing a greedy layer-wise pre-training to initialize the weights of an entire network in an unsupervised manner, followed by a supervised back-propagation step. As denoted in [20] the inclusion of the unsupervised pre-training step appeared to be the missing ingredient which then lead to significant improvements over the conventional training schemes.

2.1.1 Feature representation

A "good" or useful feature representation is one that will eventually be helpful for resolving the tasks of interest, by enabling the system to quickly achieve a higher performance on those tasks, compared to if it hadn't first learned to form the representation. Specifically as described in [9] based on the objective measure typically used to assess algorithm performance, this might be phrased as "A good representation is one that will yield a better performing classifier". Final classification is often used to objectively compare algorithms, it should be denoted that final classification doesn't mean classification of the last output layer, in autoencoders for example it can also be the classification of the encoded layer. However [9] remarks further the error signal from a single narrowly defined classification task should not be the only nor primary criterion used to guide the learning of representations. First because it has been shown experimentally that beginning by optimizing an unsupervised criterion, oblivious of the specific classification problem, can actually greatly help in eventually achieving superior performance for that classification problem. Second it can be argued that the capacity of humans to quickly become proficient in new tasks builds on much of what they have learnt prior to being faced with that task.

Deep learning methods are therefore as specified in [10] formed by the composition of multiple non-linear transformations, with the goal of yielding more abstract – and ultimately more useful – representations.

Manifold

A manifold is a connected region. Mathematically, it is a set of points associated with a neighbourhood around each point, as defined in [10]: a probability mass concentrated near regions that have a much smaller dimensionality than the original data. From any given point, the manifold locally appears to be a Euclidean space. The concept of a neighbourhood surrounding each point implies the existence of transformations that can be applied to move on the manifold from one position to a neighbouring one. Although there is a formal mathematical meaning to the term "manifold", in machine learning, as described in [17], it tends to be used more loosely to designate a connected set of points that can be approximated well by considering only a small number of degrees of freedom, or dimensions, embedded in a higher-dimensional space. Each dimension corresponds to a local direction of variation.

A useful property of manifolds is natural clustering as explained in [10], different values of categorical variables, classes, are associated with separate geometric structures in the manifold. Mathematically, the local variations on the manifold tend to preserve the value of a category, and a linear interpolation between examples of different classes in general involves going through a low density region, i.e., $P(X|Y = i)$ for different i tend to be well separated and not overlap much. In [10] following hypothesis is proposed: humans have named categories and classes because of such statistical structure (discovered by their brain and propagated by their culture), and machine learning tasks often involves predicting such categorical variables.

2.1.2 Denoising

As described in [17] Denoising is the task of predicting the original clean signal $x \in \mathcal{R}^n$ starting from an, through an unknown process corrupted, input $\tilde{x} \in \mathcal{R}^n$, i.e. learn to predict the conditional probability $p(x|\tilde{x})$.

Types of Corruption Considered

- Additive Gaussian noise (GS): $\tilde{x}|x \sim \mathcal{N}\left(\mu, \sigma^2\right)$, with different considerations for the μ and σ^2.
- Masking noise (MN): a fraction ν of the elements of x (chosen at random for each example) is forced to 0, i.e. is "turned off".
- Salt-and-pepper noise (SP): a fraction ν of the elements of x (chosen at random for each example) is set to their minimum or maximum possible value (in the dataset used for this thesis ± -1) according to fair coin flip.

Gaussian noise is the most common noise type, it is the natural choice for real valued inputs, as denoted in [9]. The salt-and-pepper noise is a natural choice for datasets which can be considered as binary or near binary such as black and white images.

Masking noise can be viewed as turning off components considered missing or replacing their value by a default value, it is a common technique for simulating missing values. Consequently removing all information about these masked components in a particular input pattern. A Denoising autoencoder, as described in Section 2.2.3, can be considered as trained to fill-in these artificially introduced “blanks”.

On the other hand both salt-and-pepper and masking noise only corrupt a fraction of the elements while leaving the others unchanged. Reconstructing the original x is only possible leveraging the dependencies between dimensions in high dimensional distributions. As detailed in [9] the idea of Denoising training, as described in Section 2.2.3, is to capture these dependencies.

2.1.3 Activation functions

The central challenge in machine learning is that an algorithm must perform well on new, previously unseen inputs, not just those on the ones the model was trained on. The ability to perform well on previously unobserved inputs is called generalization.

Typically, when training a machine learning model there is a training set of n ,(input, target) pairs $D_n = \{(x^{(1)}, t^{(1)}), \cdots, (x^{(n)}, t^{(n)})\}$, from this set a error measurement can be computed, the so called training error. This error is then minimized so far being only a optimization task. As described in [17] what separates machine learning from optimization is the objective of a generalization error, also called the test error, to be low as well. The generalization error is defined as the expected value of the error on a new input. Here the expectation is taken across different possible inputs, drawn from the distribution of the expect inputs the system could encounter in practice.

The two central challenges in machine learning as denoted in [17], are underfitting and overfitting.

- Underfitting occurs when the model is not able to obtain a sufficiently low error value on the training set.
- Overfitting occurs when the gap between the training error and test error is too large.

To control whether a model is more likely to overfit or underfit by, as described in [17], altering its ability to fit a wide variety of functions. Models with low capacity may struggle to fit the training set. Models with high capacity

can overfit by memorizing properties of the training set that do not serve them well on the test set. The activation function and loss function influence this capacity.

Specifically the activation functions introduce non-linear properties to the Network. Their main purpose is to convert a input signal to an output signal. That output signal is then used as a input in the next layer in the stack. For a detailed explanation the reader is advised to Chapter 6 of [17].

Following is a brief explanation of the activations used in this thesis.

Rectified linear unit (ReLU)

The "rectified linear unit" (ReLU) activation function is the identity for positive arguments and zero otherwise. Besides producing sparse codes, the main advantage of ReLUs is that they alleviate the vanishing gradient problem ([21]). However ReLUs are non-negative and, therefore, have a mean activation larger than zero. And as described in [21] units that have a non-zero mean activation act as bias for the next layer. If such units do not cancel each other out, learning causes a bias shift for units in next layer. The more the units are correlated, the higher their bias shift.

The ReLu is defined as:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}, \quad f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{2.1}$$

Exponential Linear Units (ELUs)

The “exponential linear unit” (ELU) function proposed in [21] speeds up learning in deep neural networks and leads to higher classification accuracies. Like rectified linear units (ReLUs), leaky ReLUs (LReLUs) and parametrized ReLUs (PReLUs), ELUs alleviate the vanishing gradient problem via the identity for positive values. However ELUs have improved learning characteristics compared to the units with other activation functions. In contrast to ReLUs, ELUs have negative values which allows them to push mean unit activations closer to zero like batch normalization but with lower computational complexity. Mean shifts toward zero speed up learning by bringing the normal gradient closer to the unit natural gradient because of a reduced bias shift effect. ELUs saturate to a negative value with smaller inputs and thereby decrease the forward propagated variation and information. Saturation means as proposed by [21] a small derivative which decreases the variation and the information that is propagated to the next layer. Therefore the representation is both noise-robust and low-complex.

The exponential linear unit (ELU) with $0 < \alpha$ is defined as

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}, \quad f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ \alpha \cdot \exp(x) & \text{if } x \leq 0 \end{cases} \tag{2.2}$$

The ELU hyperparameter α controls the value to which an ELU saturates for negative net inputs.

Experimental results in [21] show that ELUs significantly outperform other activation functions on different vision datasets. .

Softmax

The Softmax function calculates the probabilities of each target class over all possible target classes, in other word when a probability distribution over a discrete variable with n possible values should be represented. Mathematically is a form of multinomial logistic regression that normalizes an input value into a vector of values, in range $[0, 1]$, that represent a probability distribution, total sums up to 1.

Softmax functions are most often used as the output of a classifier, to represent the probability distribution over n different classes. Formally, the softmax function is given by [17]:

$$\operatorname{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \tag{2.3}$$

With $z = W^T \cdot h + b$ the output of the network, i.e. the unnormalized log probabilities predicted by a linear layer.

Hyperbolic Tangent ($\tanh$)

The $\tanh$ is a nonlinear activation function that squashes a real-valued number to the range $[-1, 1]$. It offers the advantage that negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero.

$$\tanh(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{\mathrm{e}^z - \mathrm{e}^{-z}}{\mathrm{e}^z + \mathrm{e}^{-z}} \tag{2.4}$$

It is often used as an output layer activation function, also known as output function.

2.1.4 Loss Functions

Most deep learning algorithms involve optimization of some sort, referring to the task of minimizing or maximizing a function $f(x)$, called the objective function, or criterion, by altering x. Usually $f(x)$ is minimized in this case the function may be referred to as the cost function,loss function, or error function.

This function is a form of discrepancy measurement between the output of the network and some target; this can be defined as labels for supervised learning, a condition or property for unsupervised ones, among other possibilities.

Mean squared error (MSE)

One way of measuring the performance of the model is to compute the mean squared error of the model on the test set.

The mean squared error is given by:

$$\mathrm{MSE} = \frac{1}{m} \sum_i (\hat{\boldsymbol{y}} - \boldsymbol{y})_i^2 = \frac{1}{m} \|\hat{y} - y\|_2^2 \tag{2.5}$$

with $\hat{y}$ the predictions and y the targets of the model.

MSE provides the advantage of removing the sign of the error by squaring it, so that the only magnitude of the errors influences the average error measure.

Categorical Crossentropy

Categorical crossentropy is used for single label categorization, i.e. when each data point corresponds only to one category. By comparing the distribution of the predictions (the activations in the output layer, one for each class) with the true distribution, 1 for correct class 0 for the rest. In other words, the true class is represented as a one-hot encoded vector, and the closer the model's outputs are to that vector, the lower the loss.

Mathematically as defined by [22] is:

$$R_{\mathcal{L}}(f) = \mathbb{E}_D \left[\mathcal{L}\left(f(\boldsymbol{x}), y_{\boldsymbol{x}}\right) \right] = -\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{c} \boldsymbol{y}_{ij} \log f_j\left(\boldsymbol{x}_i\right) \tag{2.6}$$

with loss function $\mathcal{L}$, usually $softmax$; classifier $f : \mathcal{X} \Rightarrow \mathbb{R}^c$ maps an input feature, in feature space $\mathcal{X} = \{1, \cdots, n\} \subset \mathbb{R}^d$, to a label, in label space $\mathcal{Y} = \{1, \cdots, c\}$. Further $\boldsymbol{y}_{ij}$ corresponds to the j'th element of one-hot encoded label of the sample x_i and f_j corresponds to the j'th element of f. Note that, $\sum_{j=1}^{n} f_j(\boldsymbol{x}_i) = 1$, and $f_j(\boldsymbol{x}_i) \geq 0, \forall j, i$, when using a softmax output layer, as will be done for the experimentation see 3.6.

2.1.5 Optimizer Adam

The Adam optimizer proposed in [23], is a method for efficient stochastic optimization that only requires first-order gradients with little memory requirement. The method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients; the name Adam is derived from adaptive moment estimation. It combines the advantages of AdaGrad ([24]), which works well with sparse gradients, and RMSProp ([25]), which works well in on-line and non-stationary settings.

For a detailed algorithm and further explanation the reader is advised to [23].

2.1.6 Regularization

Regularization is as defined in [17] any modification made to a learning algorithm with the objective of educing the generalization error but not its training error, i.e. to avoid overfitting.. Regularization is one of the central concerns of the field of machine learning, rivalled in its importance only by optimization.

A continuation two types of regularization used in this work are describe, but there are many more as $L1$ and $L2$ norms or special weights initializations.

Dataset Augmentation (Noise)

As described in [17] adding noise in the input to a neural network can be seen as a form of data augmentation. For many classification and even some regression tasks, the task should still be possible to solve even if small random noise is added to the input. Neural networks prove not to be very robust to noise. One way to improve the robustness of neural networks is simply to train them with random noise applied to their inputs some unsupervised learning algorithms, such as the Denoising autoencoder proposed in [1], see Section 2.2.3.

Early Stopping as a Form of Regularization

When training large models with sufficient representational capacity to overfitt the task, it often happens and it can be observed that the training error decreases steadily over time, but validation error begins to rise again. This means a model with better validation error can be obtained by reverting to the parameter setting at the point in time with the lowest validation error. The algorithm terminates when no parameters have improved over the best recorded validation error for some pre-specified number of iterations. This strategy is known as early stopping . It is probably the most commonly used form of regularization in deep learning. Its popularity is due to its effectiveness, simplicity and by stopping before all epochs are done it can produce time savings.

As described in [17] and [26] early stopping acts as a regularizer by utilizing that the effect of restricting the optimization procedure to a relatively small volume of parameter space in the neighbourhood of the initial parameter value θ. More specifically, taking θ optimization steps (corresponding to τ training iterations) and with learning rate ϵ . The product $\epsilon \cdot \tau$ can be defined as a measure of effective capacity. Assuming the gradient is bounded, restricting both the number of iterations and the learning rate limits the volume of parameter space reachable from θ. In this sense, $\epsilon \cdot \tau$ behaves as if it were the reciprocal of the coefficient used for weight decay. [17] it is shown that a trajectory of length θ ends at a point that corresponds to a minimum of the L^2 -regularized objective. It is further argument that early stopping typically involves monitoring the validation set error in order to stop the trajectory at a particularly good point in space. Early stopping therefore has the advantage over weight decay in that it automatically determines the correct amount of regularization while weight decay requires many training experiments with different values of its hyperparameters

2.1.7 Batch and Mini-Batch

Batch : where the batch size is equal to the total dataset thus making the iteration and epoch values equivalent.

mini-batch mode: where the batch size is greater than one but less than the total dataset size. The mini-batch size B is typically chosen between 1 and a few hundreds, e.g. $B = 32$ is a good default value. The impact of B is mostly computational, i.e., larger B yield faster computation (with appropriate implementations) but requires visiting more examples in order to reach the same error, since there are less updates per epoch. According to [27] in theory, this hyper-parameter should impact training time and not so much test performance[1], so it can be optimized separately of the other hyperparameters, by comparing training curves (training and validation error vs amount of training time), after the other hyper-parameters (except learning rate) have been selected.

2.1.8 Metrics

To evaluate how well the machine learning algorithm performs on data metric functions are used. Keeping in mind that it is important to know how it performs on data that has not been seen before, since this determines how well it will work when deployed in the real world. Therefore these performance measures are evaluated using a test set of data that is separate from the data used for training the machine learning system.

[1] In my experiments batch size did influence the results, it could be because of the Keras implementation of the program.

Accuracy

For many tasks such as classification and classification with missing inputs, often the accuracy of the model is measured . Accuracy, is as defined in [17] the proportion of examples for which the model produces the correct output. Equivalent information can be obtained by measuring the error rate, the proportion of examples for which the model produces an incorrect output. The error rate is often referred to as the expected $0-1$ loss. The $0-1$ loss on a particular example is 0 if it is correctly classified and 1 if it is not.

For tasks such as density estimation, it does not make sense to measure accuracy, error rate, or any other kind of $0-1$ loss. Instead, we must use a different performance metric that gives the model a continuous-valued score for each example such as MAE.

Mean absolute error (MAE)

One way of measuring the performance of the model is to compute the mean mean absolute error (MAE).

MAE is given by:

$$\mathrm{MAE} = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n} \tag{2.7}$$

with $\hat{y}$ the predictions and y the targets of the model.

2.1.9 Mutual information

The Mutual information (I) of two random variables denotes how much dependence there is between the two variables. It quantifies how much information can be obtained about one random variable by observing the other one. It can be decomposed into an entropy and a conditional as $I(X;Y) = H(X) - H(X|Y)$. The observed input X comes from an unknown distribution $q(X)$, this makes $H(X)$ an unknown constant.

In [9] it is shown that maximizing the lower bound of the mutual information between input X and a learnt representation Y is possible using the infomax principle and considering a parametric distribution $p(X|Y;\theta')$, parametrized by θ' and a deterministic mapping from X to Y , that is, representation Y is to be computed by a parametrized function $Y = f_\theta(X)$ or equivalently $q(Y|X;\theta) = \delta\left(Y - f_\theta(X)\right)$ (where δ denotes Dirac-delta) it can be shown that since $q(X)$ is unknown, but there are samples from it, the empirical average over the training samples can be used instead as an unbiased estimate. Coming to following maximization equation:

$$\max_{\theta,\theta'} \mathbb{E}_{q^0(X)}\left[\log p\left(X|Y = f_\theta(X);\theta'\right)\right] \tag{2.8}$$

2.2 Autoencoders

Autoencoders are as denoted in [17] the prototypical example for a representation learning Neural Network. An autoencoder consists of two parts: an encoder, that encodes the input to a restricted feature space and a decoder that reconstructs this to the input space again. The neural network is then trained to reconstruct input data attempting to copy its input as output. Figure 2.1 shows a deep autoencoder.

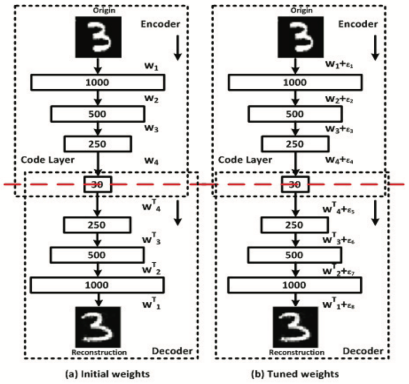

Figure 2.1: Deep autoencoder proposed in [6], with tied weights, before and after fine-tuning, i.e. trained with pre-training, see section 2.2.6.

The encoder, as defined in [9], generates a deterministic mapping $f_\theta : M^d \mapsto M^h$, note that M can be $\mathbb{R}^d$ or $[0,1]^d$, that transforms an input vector x into a hidden representation y, of dimension h, it allows the straightforward and efficient computation of a feature vector $y = f_\theta(x)$ from an input x. For each example $x^{(t)}$ from a data set $\{x^{(1)}, \cdots, x^{(T)}\}$, $y^{(t)} = f_\theta(x^{(t)})$ where $y^{(t)}$ is defined as the feature-vector or representation or code computed from $x(t)$. Its typical form is an affine mapping followed by a nonlinearity: $f_\theta(\mathbf{x}) = s(\mathbf{W}\mathbf{x} + \mathbf{b})$ Its parameter set is $\theta = \{W, b\}$, where W is a $d' \times d$ weight matrix and b is an offset vector of dimensionality d'.

The resulting hidden representation y is then mapped back to a reconstructed $d-$dimensional vector z in input space, $z = g_{\theta'}(y)$. This mapping $g'_\theta : M^h \mapsto M^d$ is called the decoder. Its typical form is again an affine mapping optionally followed by a squashing non-linearity, that is, either $g_{\theta'}(y) = W'y + b'$ or $g_{\theta'} = s(W'y + b')$ with appropriately sized parameters $\theta' = \{W', b'\}$.

In general z is not to be interpreted as an exact reconstruction of x, but rather in probabilistic terms as the parameters (typically the mean) of a distribution $p(X|Z = z)$ that may generate x with high probability, [9]. See Figure 2.2 for a schematic representation of the procedure. The associated reconstruction error to be optimized:

$$\begin{aligned} L(x, z) &\propto -\log p(x|z) \\ \text{with } L &: M^d \times M^d \mapsto \mathbb{R} \end{aligned} \tag{2.9}$$

Common choices for $p(x|z)$ and associated loss function $L(x, z)$, as proposed in [9], include:

- For $x \in \mathbb{R}^d : X|z \sim \mathcal{N}\left(z, \sigma^2\right)$, that is $X_j|\mathbf{z} \sim \mathcal{N}\left(\mathbf{z}_j, \sigma^2\right)$. This yields $L(x, z) = L(x, z)_2 = \|\mathbf{x} - \mathbf{z}\|^2$. This is the squared error objective found in most traditional autoencoders. In this setting, due to the Gaussian interpretation, it is more natural not to use a squashing nonlinearity in the decoder.
- For $\mathbf{x} \in \{0,1\}^d : X|\mathbf{z} \sim \mathcal{B}(\mathbf{z})$, that is $, X_j|\mathbf{z} \sim \mathcal{B}\left(\mathbf{z}_j\right)$. In this case, the decoder needs to produce a $z \in [0,1]^d$. So a squashing nonlinearity such as a sigmoid s will typically be used in the decoder. This yields $L(x, z) = L_H(x, z) = -\sum_j \left[\mathbf{x}_j \log \mathbf{z}_j + \left(1 - \mathbf{x}_j\right) \log\left(1 - \mathbf{z}_j\right)\right] = H(\mathcal{B}(\mathbf{x}) \| \mathcal{B}(\mathbf{z}))$ which is termed the cross-entropy loss because it is seen as the cross-entropy between two independent multivariate Bernoullis, the first with mean x and the other with mean z. This loss can also be used when x is not strictly binary but rather $x \in [0,1]^d$.

The learning process is described, in [17], as simply minimizing a loss function $L(x, g_{\theta'}(f_\theta(x))) = L(x, z)$. That is

as defined in [9], carrying the following optimization:

$$\arg\min_{\theta,\theta'} \mathbb{E}_{q^0(X)}[L(X, Z(X))]$$

where $Z(X)$ emphasizes the fact that Z is a deterministic function of X, since Z is obtained by composition of deterministic encoding and decoding. Making this explicit and using our definition of loss L from Equation 2.9 this can be rewritten as:

$$\arg\max_{\theta,\theta'} \mathbb{E}_{q^0(X)}[\log p(X|Z = g_{\theta'}\left(f_\theta(X)\right)\Big)\Big]$$

or equivalently

$$\arg\max_{\theta,\theta'} \mathbb{E}_{q^0(X)}\left[\log p\left(X|Y = f_\theta(X); \theta'\right)\right]$$

The last line corresponds to Equation 2.8, the maximization of a lower bound on the mutual information between X and Y, as described in 2.1.9.

In summary, as described in [10] , basic auto-encoder training consists in finding a value of parameter vector θ minimizing reconstruction error

$$\mathcal{J}_{\mathrm{AE}}(\theta) = \sum_t L\left(x^{(t)}, g_\theta\left(f_\theta\left(x^{(t)}\right)\right)\right) \tag{2.10}$$

where $x^{(t)}$ is a training example. This minimization is usually carried out by stochastic gradient descent see section 2.2.6 for a detailed discussion.

It can thus be said that training an autoencoder to minimize reconstruction error amounts to maximizing a lower bound on the mutual information between input X and learnt representation Y [9]. Intuitively, if a representation allows a good reconstruction of its input, it means that it has retained much of the information that was present in that input.

As denoted in [17] and [9] if the encoder and decoder are allowed too much capacity, the autoencoder can learn to perform the copying task without extracting useful information about the distribution of the data, i.e. mutual information can be trivially maximized by setting $Y = X$. Similarly, an ordinary autoencoder where Y is of the same dimensionality as X (or larger) can achieve perfect reconstruction simply by learning an identity mapping. Without any other constraints, this criterion alone is unlikely to lead to the discovery of a more useful representation than the input. The use of “tied weights” can also change the solution: forcing encoder and decoder matrices to be symmetric and thus have the same scale can make it harder for the encoder to stay in the linear regime of its nonlinearity without paying a high price in reconstruction error [9].

Generally, an auto-encoder network is constructed by stacking multiple one layer auto-encoders. That is, the hidden representation of the previous one-layer auto-encoder is fed as the input of the next one. [13]

There are different types of autoencoders mostly described by the type of restriction they use:

- Undercomplete Autoencoders
- Sparse Autoencoders
- Denoising Autoencoders
- Contractive autoencoder
- Stochastic Encoders and Decoders

In the following section some of these types will be discussed in detail, but not all. Stochastic Encoders and Decoders will no be described.

2.2.1 Undercomplete autoencoders

A way to obtain useful features from the autoencoder, as proposed in [17] and [9], is the bottleneck constrain, i.e. $d' < d$. An autoencoder whose code dimension is less than the input dimension is called undercomplete. Learning an undercomplete representation forces the autoencoder to capture the most salient features of the training data, to attempt to separate useful information (to be retained) from noise (to be discarded). This will naturally translate to non-zero reconstruction error. The resulting lower-dimensional Y can thus be seen as a lossy compressed representation of X [9].

2.2.2 Sparse Autoencoders

To make the autoencoder learn useful information when the encoded data, y, is allowed to have dimension equal or greater, overcomplete, than the input, sparse autoencoders use a loss function that encourages the model to have other properties besides the ability to copy its input to its output. A sparse autoencoder is as described in [17], simply an autoencoder whose training criterion involves a sparsity penalty $\Omega(y)$ on the code layer y, in addition to the reconstruction error: $L(x, g'_\theta(f_\theta(x))) + \Omega(y)$.

Sparse autoencoders are typically used to learn features for another task, such as classification. Then as denoted by [9], a sparse over-complete representations can be viewed as an alternative "compressed" representation: it has implicit straightforward compressibility due to the large number of zeros rather than an explicit lower dimensionality. An autoencoder that has been regularized to be sparse must respond to unique statistical features of the dataset it has been trained on, rather than simply acting as an identity function. In this way, training to perform the copying task with a sparsity penalty can yield a model that has learned useful features as a by-product

2.2.3 Denoising autoencoders

In [1] and [9] the so-called Stacked Denoising Auto-Encoder (SDAE) is proposed. Using denoising, see section 2.1.2, auto-encoders as a building block for unsupervised modelling. The denoising auto-encoder is a stochastic variant of the ordinary auto-encoder with the distinctive property that even with a high capacity model, it cannot learn the identity mapping. A denoising autoencoder is explicitly trained to denoise a corrupted version of its input, making the whole mapping "robust", i.e., insensitive to small random perturbations as denoted in [10].

Learning the identity is no longer enough, as described in [10] the learner must capture the structure of the input distribution in order to optimally undo the effect of the corruption process, with the reconstruction essentially being a nearby but higher density point than the corrupted input. Figure 2.4 illustrates that the Denoising Auto-Encoder (DAE) is learning a reconstruction function that corresponds to a vector field pointing towards high-density regions (the manifold where examples concentrate). Training consist now of reconstructing a clean "repaired" input from a corrupted, partially destroyed one. This is done,as described in [1] by first corrupting the initial input x to get a partially destroyed version $\tilde{x}$ by means of a stochastic mapping $\overline{\mathbf{x}} \sim q_{\mathcal{D}}(\overline{\mathbf{x}}|\mathbf{x})$. The corrupted input $\tilde{x}$ is then mapped, as with the basic autoencoder, to a hidden representation $y = f_\theta(\tilde{x}) = s(W\tilde{x} + b)$ from which we reconstruct a

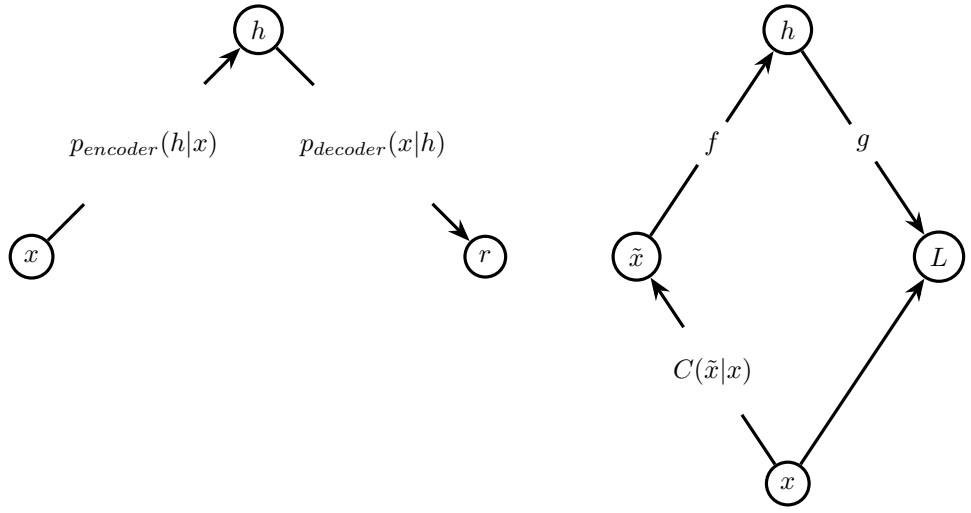

Figure 2.2: Information flow in a Normal- and a Denosing- Autoencoder

$z = g_{\theta'}(y) = s(W'y + b')$. Formally, the objective optimized by a DAE is [10]:

$$J_{\mathrm{DAE}} = \sum_t \mathbb{E}_{q(\bar{x}|x^{(t)})} \left[L\left(x^{(t)}, g_\theta\left(f_\theta(\bar{x})\right)\right)\right] \tag{2.11}$$

where $\mathbb{E}_{q(\bar{x}|x^{(t)})}[\cdot]$ averages over corrupted examples $\bar{x}$ drawn from corruption process $q(\bar{x}|x^{(t)})$. In practice this is optimized by stochastic gradient descent, where the stochastic gradient is estimated by drawing one or a few corrupted versions of $x(t)$ each time $x(t)$ is considered. Corruptions considered in [9] include additive isotropic Gaussian noise, salt and pepper noise for gray-scale images, and masking noise (salt or pepper only), see section 3.4 for more details. Qualitatively better features are reported with denoising, resulting in improved classification, and DAE features performed similarly or better than RBM features [10].

Denoising training forces f_θ and g'_θ to implicitly learn the structure of $p_{data}(x)$, as mentioned in [17]. Denoising autoencoders therefore provide another example of how useful properties can emerge as a by-product of minimizing reconstruction error.

2.2.4 Contractive autoencoder

Another strategy for regularizing an autoencoder described in [17], is to use a penalty Ω, as in sparse autoencoders, $L(x, g(f(x))) + \Omega(y, x)$, but with a different form of Ω: $\Omega(y, x) = \lambda \sum_i ||\nabla_x h_i||^2$, the squared Frobenius norm of the Jacobian matrix. In effect reducing the number of effective degrees of freedom of the representation (around each point) by making the encoder contractive, i.e., making the derivative of the encoder small (thus making the hidden units saturate). This forces the model to learn a function that does not change much when x changes slightly, in other words as described in [10] as making the representation as “constant” (insensitive) as possible with respect to changes in input . Because this penalty is applied only at training examples, it forces the autoencoder to learn features that capture information about the training distribution. An autoencoder regularized in this way is called a contractive autoencoder, or CAE. This approach has theoretical connections to denoising autoencoders, manifold learning, and probabilistic modelling For a more detailed explanation see chapter 14.7 of [17].

2.2.5 Tied weights

When an autoencoder is symmetrical a common technique, as described in [28], is to tie the weights of the decoder layers to the weights of the encoder layers. This halves the number of weights in the model, speeding up training and limiting the risk of overfitting. Specifically, if the autoencoder has a total of N layers (not counting the input layer), and W_L represents the connection weights of the L -th layer (e.g., layer 1 is the first hidden layer, layer N_{enc} is the coding layer, and layer N is the output layer), then the decoder layer weights can be defined simply as: $W_{N-L+1} = W_L^T$ (with $L = 1, 2, \cdots, N_{enc}$), i.e. $W' = W^T$ with W the weights of the decoder and W' of the decoder.

2.2.6 Training

In virtually all instances of deep learning, the objective function is a highly non-convex function of the parameters, with the potential for many distinct local minima in the model parameter space. The principal difficulty is that not all of these minima provide equivalent generalization errors and, in [26] is suggested that for deep architectures, the standard training schemes (based on random initialization) tend to place the parameters in regions of the parameters space that generalize poorly as was frequently observed empirically but rarely reported.

There are two big schools of thoughts concerning hot to train deep networks, and specially autoencoders one considering Pre-Training a must and the ones augmenting better results with no pre-Training.

Pre-training

Training a deep network to directly optimize only the supervised objective of interest by gradient descent, starting from random initialized parameters, does not work very well is the argumentation of the pro pre-training sector as in [9], [1] and [10]. What works much better is to initially use a local unsupervised criterion to (pre)train each layer in turn, with the goal of learning to produce a useful higher-level representation from the lower-level representation output by the previous layer. From this starting point on, gradient descent on the supervised objective leads to much better solutions in terms of generalization performance, this process is also called greedy layer-wise training. Deep layered networks trained in this fashion have been shown empirically, in [9], to avoid getting stuck in poor solutions typically reached with only random initializations. In addition to the supervised criterion relevant to the task, what appears to be key is using an additional unsupervised criterion to guide the learning at each layer. But as reflected in [9] there is yet no clear understanding of what constitutes “good” representations for initializing deep architectures or what explicit unsupervised criteria may best guide their learning.

In [26] is argument that the central challenge in training deep architectures is dealing with the strong dependencies that exist during training between the parameters across layers. The difficulty of the problem arises from simultaneously:

1. adapting the lower layers in order to provide adequate input to the final (end of training) setting of the upper layers.
2. adapting the upper layers to make good use of the final (end of training) setting of the lower layers.

The second problem is easy on its own (i.e., when the final setting of the other layers is known). But it it is not clear how difficult first one is, and in [26] it is speculated that a particular difficulty arises when both sets of layers must be learned jointly, as the gradient of the objective function is limited to a local measure given the current setting

of other parameters. Furthermore, because with enough capacity the top layers can easily overfit the training set, training error does not necessarily reveal the difficulty in optimizing the lower layers.

Joint training (pre-training)

In [20] it is argumented that the greedy layer-wise training focuses on the local constraints introduced by the learning algorithm (such as in autoencoders), but loses sight of the original data distribution when training higher layers. To compensate for this disadvantage, all the levels of the deep architecture should be trained simultaneously. But this type of joint training can be very challenging and if done naively, will fail to learn. This can also be viewed as a generalization of single- to multi-layer auto-encoders.

Following the same idea in [10] it was argumented that exploiting large quantities of labelled data and with proper initialization and choice of activation function (non-linearity), also proposed in [14], very deep purely supervised networks can be trained successfully without any layer wise pre-training. This reinforces the hypothesis presented in [10] that unsupervised pre-training acts as a prior, which may be less necessary when very large quantities of labelled data are available. The hypothesis is further affirmed in [20] ,by showing that the joint training method not only learned better data models, but also learned more representative features for classification as compared to the layerwise method, which highlights its potential for unsupervised feature learning (and to learn the prior without any pre-training.).

Further in [20] a join training method is proposed aimed to solve the drawbacks of layerwise training. By tuning all the parameters simultaneously toward the reconstruction error of the original input, the optimization is no longer local for each layer. In addition, the reconstruction error in the input space makes the training process much easier to monitor and interpret.

2.2.7 PCA equivalence

As denoted in [5] a kind of common rule is that superficially high-dimensional and complex phenomena can actually be dominated by a small amount of simple variables in most situations, therefore dimensionality reduction is of interest, i.e. looking for a projection method that maps the data from high feature space to low feature space.

This can be done using a linear auto-encoder (linear encoder and decoder) without any nonlinearity and a squared error loss; then the autoencoder essentially performs principal component analysis (PCA) [9]. When a nonlinearity such as a sigmoid is used in the encoder, things become a little more complicated, as described in [9]: obtaining the PCA subspace is a likely possibility since it is possible to stay in the linear regime of the sigmoid, but arguably not the only one. Also when using a cross-entropy loss rather than a squared error the optimization objective is no longer the same as that of PCA and will likely learn different features. The use of “tied weights” makes it not possible to learn the same subspace because W cannot be forced into being small and W' large to achieve a linear encoder [10].

The PCA equivalence can be used to illustrate the probabilistic, auto-encoder and manifold views of representation learning. PCA learns a linear transformation $h = f(x) = W^T x + b$ of input $x \in \mathbb{R}^{d_x}$, where the columns of $d_x \times d_h$ matrix W form an orthogonal basis for the d_h orthogonal directions of greatest variance in the training data. The result is d_h features (the components of representation h) that are decorrelated. The three interpretations of PCA are the following, according to [10]:

a. it is related to probabilistic models such as probabilistic PCA, factor analysis and the traditional multivariate Gaussian distribution (the leading eigenvectors of the covariance matrix are the principal components);

b. the representation it learns is essentially the same as that learned by a basic linear auto-encoder;

c. it can be viewed as a simple linear form of linear manifold learning , i.e., characterizing a lower-dimensional region in input space near which the data density is peaked.

PCA can be given a natural probabilistic interpretation as factor analysis [10]:

$$p(h) = \mathcal{N}\left(h; 0, \sigma_h^2 \mathbf{I}\right)$$
$$p(x|h) = \mathcal{N}\left(x; Wh + \mu_x, \sigma_x^2 \mathbf{I}\right)$$

where $x \in \mathbb{R}^{d_x}, h \in \mathbb{R}^{d_h}, \mathcal{N}(v; \mu, \Sigma)$ is the multivariate normal density of v with mean μ and covariance Σ, and columns of W span the same space as leading d_h principal components, but are not constrained to be orthonormal.

For example in [7] a two-dimensional deep autoencoder is proposed that produced a better visualization of the data than did the first two principal components, see figure 2.3.

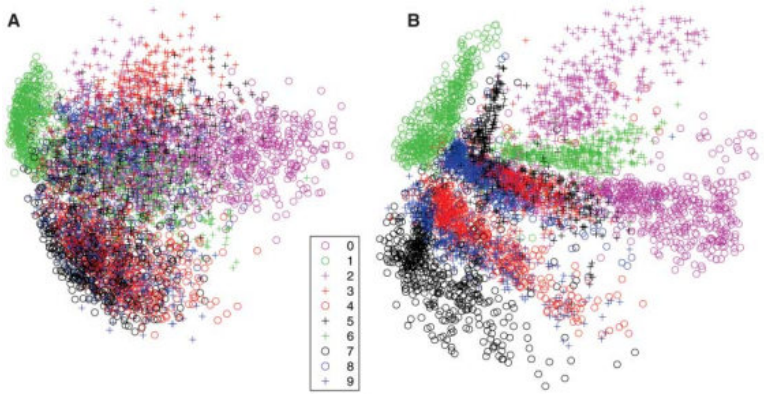

Figure 2.3: (A) The two-dimensional codes for 500 digits of each class produced by taking the first two principal components of all 60,000 training images. (B) The two-dimensional codes found by a 784-1000-500-250-2 autoencoder.[7]

2.2.8 Manifold learning

The process of denoising, see section 2.1.2, can be given an intuitive geometric interpretation under the so-called manifold assumption see section 2.1.1. As described in [9] natural high dimensional data concentrates close to a non-linear low-dimensional manifold. This prior is particularly well suited, as assured in [10], for AI tasks involving images, sounds or text, among others, for which most uniformly sampled input configurations are unlike natural stimuli. As soon as there is a notion of “representation” then it can be thought of a manifold by considering the variations in input space, which are captured by or reflected (by corresponding changes) in the learned representation [10].This is illustrated in Figure 2.4.

During denoising training, a stochastic operator $p(X|\tilde{X})$ that maps a corrupted $\tilde{X}$ back to its uncorrupted X is learned. As denoted in [9], corrupted examples are much more likely to be outside and farther from the manifold than the uncorrupted ones. Thus stochastic operator $p(X|\tilde{X})$ learns a map that tends to go from lower probability

points $\tilde{X}$ to nearby high probability points X, on or near the manifold. Note that when $\tilde{X}$ is farther from the manifold, $p(X|\tilde{X})$ should learn to make bigger steps, to reach the manifold. Successful denoising implies that the operator maps even far away points to a small region close to the manifold.

The denoising autoencoder can hence be seen as a way to define and learn a manifold. In particular, if the dimension of Y is constrained to be smaller than the dimension of X, then the intermediate representation $Y = f_\theta(X)$ may be interpreted as a coordinate system for points on the manifold. More generally, $Y = f(X)$ is a representation of X which is well suited to capture the main variations in the data along the manifold [9].

The process of mapping a corrupted example to an uncorrupted one can be visualized in Figure 2.4, with a low-dimensional manifold near which the data concentrate. We learn a stochastic operator $p(X|\tilde{X})$ that maps an $\tilde{X}$ to an X, $p(X|\tilde{X})$. The corrupted examples will be much more likely to be outside and farther from the manifold than the uncorrupted ones. Hence the stochastic operator $p(X|\tilde{X})$ learns a map that tends to go from lower probability points $\tilde{X}$ to high probability points X, generally on or near the manifold. Note that when $\tilde{X}$ is farther from the manifold, $p(X|\tilde{X})$ should learn to make bigger steps, to reach the manifold. At the limit we see that the operator should map even far away points to a small volume near the manifold.

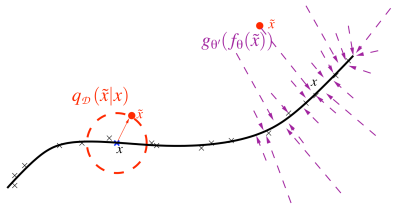

Figure 2.4: Manifold learning perspective. Suppose training data (x) concentrate near a low-dimensional manifold. Corrupted examples $(.)$ obtained by applying corruption process $q_{\mathcal{D}}(\tilde{x}|X)$ will generally lie farther from the manifold. The model learns with $p(X|\tilde{x})$ to "project them back" (via autoencoder $g'_\theta\left(f_\theta(\cdot)\right)$) onto the manifold. Intermediate representation $Y = f_\theta(X)$ may be interpreted as a coordinate system for points X on the manifold.[9]

The following is a very quick introduction of four representative dimensionality reduction methods as described in [5]:

PCA: Given a dataset on $\mathbb{R}^n$, PCA aims to find a linear subspace of dimension d lower than n which attempts to maintain most of the variability of the data.

LDA: The basic idea of Linear discriminant analysis is to ensure the samples after projection to have maximum-between-cluster distance and minimum-in-cluster-distance in the new subspace.

LLE: Locally linear embedding is a nonlinear approach to reduce dimensionality by computing low-dimensional, neighbourhood preserving embedding of high-dimensional data. A dataset of dimensionality n, which is assumed to lie on or near a smooth nonlinear manifold of dimensionality $d \leq n$, is mapped into a single global coordinate system of lower dimensionality d. The global nonlinear structure is recovered by locally linear fits.

isomap: Isomap is a nonlinear generalization of classical multidimensional scaling. The main idea is to perform multidimensional scaling, not in the input space, but in the geodesic space of the nonlinear data manifold. The geodesic distance represents the shortest paths along the curved surface of the manifold measured as if the surface were flat. This can be approximated by a sequence of short steps between neighbouring sample points.

T-Sne: [2] "t-SNE" [29] is a technique that visualizes high-dimensional data by giving each datapoint a location in a two or three-dimensional map. The technique is a variation of Stochastic Neighbour Embedding that is much easier to optimize, and produces significantly better visualizations by reducing the tendency to crowd points together in the center of the map. t-SNE is better than existing techniques at creating a single map that reveals structure at many different scales. This is particularly important for high-dimensional data that lie on several different, but related, low-dimensional manifolds, such as images of objects from multiple classes seen from multiple viewpoints.

[2] Not mentioned in [5] but used to visualize the results of this thesis.

3 Experiments

3.1 Related work

Papers and works about autoencoders and autoencoder Denoising are very abundant, some are used to research dimensionality reduction, as [5], [6], and [7], others to analyse and predict machinery faults [8] and multiples are concentrated in extracting features and representations, as [1], [9], [10], [11], and [12], or clustering data [13]. But works that are concentrated in using this iteratively are but a few.

One such work is [14] where a model for the rating prediction task in recommender systems which significantly outperforms previous state-of-the art models on a time-split Netflix data set is proposed. A deep autoencoder with 6 layers is used an trained end-to-end without any layer-wise pre-training. Further it is demonstrated that:

- deep autoencoder models generalize much better than the shallow ones
- non-linear activation functions with negative parts are crucial for training deep models
- heavy use of regularization techniques such as dropout is necessary to prevent overfitting

Further an algorithm based on iterative output re-feeding is proposed to overcome natural sparseness of collaborate filtering. In contrast to the case analysed in this thesis they start from a sparse vector that is converted to a dense vector and then this one is used in the next iteration step. Significant is the conclusion that just applying output re-feeding does not have significant impact on the model performance. However, in conjunction with a higher learning rate, it does significantly increase the model performance [14].

Another related work, although using convolutional neural networks, is [30] where inspired by the combination of feedforward and iterative computations in the visual cortex, and taking advantage of the ability of denoising autoencoders (DAE) to estimate the score of a joint distribution, an approach to iterative inference for capturing and exploiting the complex joint distribution of output variables conditioned on some input variables is proposed. The DAE is trained by means of stochastic gradient descent with RMSprop. While training, a zero-mean Gaussian noise ($\sigma = 0 : 1$ or $\sigma = 0 : 5$) is added to the DAE input. An the models are trained for a maximum of 500 epochs while monitoring the validation reconstruction error to early stop the training using a patience of 100 epochs.

3.2 Idea

The hypothesis to be tested is that iterative evaluation, denoising of signals, with deep autoencoder brings better results than just evaluating it ones. For this the output of the autoencoder, decoded data, is used as input data for the same autoencoder for the next iteration. The MNIST [31] dataset was used to represent the signals. These where corrupted with different strengths of noise and the objective was to get the best possible reconstruction.

Different architectures, targets, batch sizes, and noise types where tested. The effect of iterative training was only superficially analysed.

To further examine the feature representation learned by the autoencoder the manifold of the encoded data was visualized and the confusion matrix of a classifier also trained on the encoded data was analysed.

3.3 Fixing of Hyper-parameters

The hyperparameters are as described in [17], settings that can be used to control the behaviour of Neural Networks. The values of hyperparameters are not adapted by the learning algorithm itself, but through testing.

As first test the results of Section 4.1.2 AUTOENCODER LEARNING of Clevert et.al [21] were corroborated using the proposed autoencoder. The encoder part consisted of four fully connected hidden layers with sizes $1000 - 500 - 250 - 30$, respectively. The decoder part was symmetrical to the encoder. Concluding that the ELU activation function retuned the best results. For the initial testing phase and hyperparameters fixing this autoencoder structure was used. As noted in [14] it is very important for the activation function f in hidden layers to contain non-zero negative part.

I decided to not use pre-training but to train all layers at once following the argument in section 2.2.6, since good results where observed.

Then different parametrizations of the MNIST [31] dataset were tested, the standard ($[0, 1]$) and a negative ($[-1, 1]$) one. The idea being that the negative parametrization could retain more information. The results showed that the $[-1, 1]$ parametrizations delivered far better results on iterative testing.

Following these tests I decided to use the ($[-1, 1]$) parametrizations in the further testing. Which means that all binary activation- and loss-functions, as binary cross entropy, relu, and sigmoid can't be used for the autoencoders. Therefore and using the results of the first tests I decided to use ELU as activation for the hidden layers of the autoencoders and $tanh$ as activation function of the output layer, since the output range and the data parametrization are identical.

Finally multiple optimizers, RMSProp and ADAM [23], where tested. Concluding that ADAM presented better results and is going to be used for the subsequent testing.

Furthermore it was observed that the batch size had a big influence on the results, accordingly I decided to continue the test using multiple batch sizes, $\{64, 128, 256, 512\}$.

Early stopping was used to minimize the training time and also taking advantage of its regulating property as described in section 2.1.6. The epochs where set to maximal 250 and the "keras.callbacks.EarlyStopping" function, [32], was used to monitor the validation loss in the case of the autoencoders and the validation accuracy in the case of the classifiers with a patience of 30 and 10 steps respectively. In other words if the monitored value does not improve over patience steps, the training is stopped and the weight values are reverted to the ones that achieved the best solution, patience steps ago.

3.4 Noise-Types

Different types of noise where tested not only for the evaluation but also for training. For the evaluation phase Gaussian (GS) noise and Salt-and-Pepper (SP) noise where compared resulting in very similar results, although SP returned slight worse results and I decided to use it in further testing.

The SP noise is described as $\tilde{x} \sim \mathcal{U}(-1,1) \cdot \mathcal{B}(prob)$. With $prob$ a constant that defines how severe the corruption is, i.e. how probable is that a pixel is corrupted. The equation can be interpreted as a Uniform distribution that defines if the value is saturated to ± 1 or if no noise is applied multiplied with a Bernoulli trial that also defines if noise should be applied, i.e. a random subset of x is set to 1 or -1 and $prob$ determines how big this subset is(technically when the Uniform distribution returns 0 no noise is applied this avoids that the image are completely destroyed when $prob = 1$). With $prob$ ranging from 0, no corruption, to 1, almost total destruction of the original image. See Appendix A.1 for the Python implementation.

Denoising autoencoders where only slightly tested under the idea that iterations would produce clean data an therefore better to train a normal autoencoder. For training these autoencoders the elements of the MNIST dataset where replaced by random through SP, with random $prob \in [0,1]$, corrupted image, selected respecting the label of each element, i.e. for each number the same one but corrupted was used.

Although unnoticed at first a kind of denoising autoencoder was used when using the $10\_Targets$ targets for training, see detailed explanation in 3.7. These target set consist of one image for all ones, one for all twos and so on. It can be interpreted as some unknown type of corruption that converts all different images of number to a specific image, respecting the corresponding labels.

3.5 Architectures Autoencoders

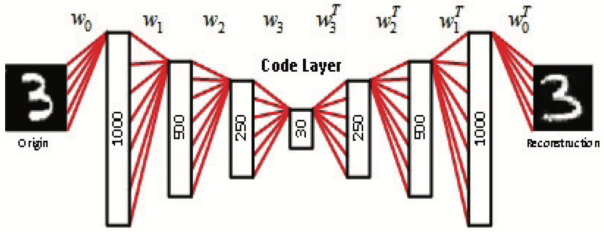

Figure 3.1: Autoencoder proposed in [6]

3.5.1 Initial architectures and research

Multiple shallow and deep structures, where the constraint $d_{l+1} < d_l$ is satisfied, where tested starting from $1000 - 500 - 250 - 30$ see 3.1, that is proposed in [21] , and similarly $1000 - 250 - 50 - 10$ proposed in [13].

Structures where the constraint wasn't always respected as the structure of $784 - 1000 - 500 - 250 - 30$ proposed in [6] were also researched.

Furthermore it was also considered what would happen if the weights W of the decoder and the weights W' of the decoder are tied, i.e $W' = W^T$.

3.5.2 Selected architectures

From the multiple structures I decided to select two that are most representative, one shallow and one deep.

The shallow architecture is $Normal\_dim$, see Figure 3.2 right, consisting of two layers with a encoded dimension of 30 and a structure $784 - 100 - 30 - 100 - 784$, each layer of the encoder has a smaller dimension than the previous one. See python implementation in A.5. And a version with tied weights $Normal\_dim\_tied$, see python implementation in A.6.

And $Over\_dim$ the deep architecture, see Figure 3.2 left, consisting of 4 layers with a encoded dimension of 36 and a structure $784 - 1024 - 529 - 256 - 36 - 256 - 529 - 1024 - 784$. This is very similar to the structure in [6] but quadratic values delivered better results, i.e. $28^2 - 32^2 - 23^2 - 16^2 - 6^2 - 16^2 - 23^2 - 32^2 - 28^2$. See python implementation in A.2. And a version with tied weights $Over\_dim\_tied$, see python implementation in A.3.

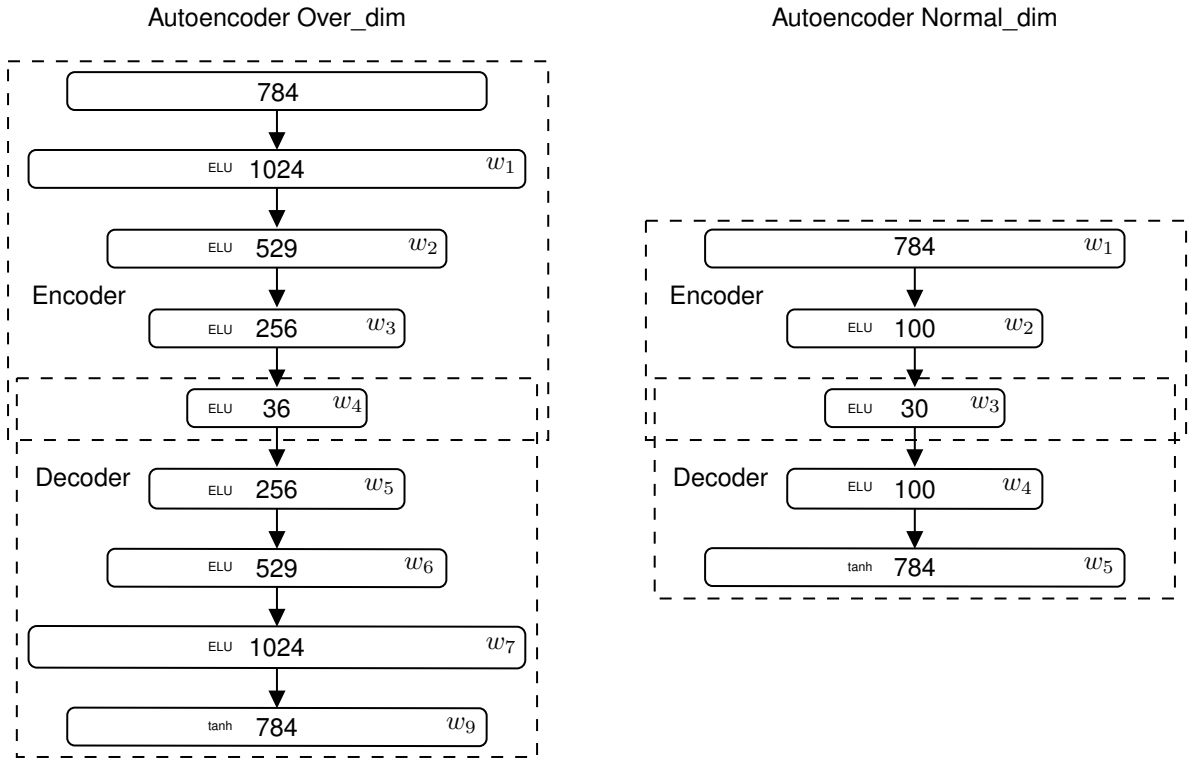

Figure 3.2: Selected Autoencoder structures witch activation functions. Left: Over_dim model, right: normal_dim model

3.6 Classifiers

A 3 layer classifier with $relu$ activation and $softmax$ output functions was trained on the encoded data, the output of the encoder prediction, with $Categorical - crossentropy$ loss function to predict the class, label, of each encoded image. See Figure 3.3 and python implementation in A.7. The classifier was then also used in the iterative evaluation to try to examine the, classification, changes in the encoded data.

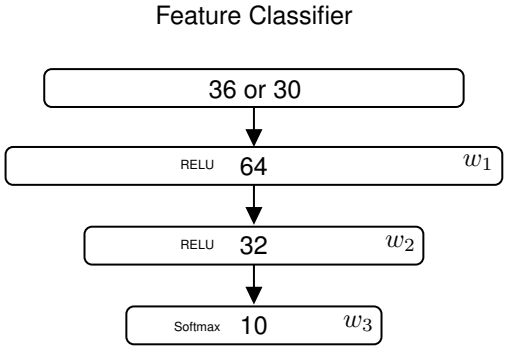

Figure 3.3: Feature Classifier with input dimension 36 or 30 depending on the autoencoder being used.

3.7 Datasets

The idea of this thesis is to test different architectures to denoise signals, to represent these the MNIST dataset [31] was chosen. As previously described SP noise (see 3.4) was used, and as evaluation data 11 "sets" where used applying the SP noise in 0.1 increments to create the categories. The function $SP(data, prob)$ returns $data$ corrupted with SP noise using a probability of $prob$, see python implementation in A.1, i.e. $evaluation\_set = \{SP(MNIST, 0), SP(MNIST, 0.1), SP(MNIST, 0.2), \cdots, SP(MNIST, 1)\}$ with the special condition than $MNIST$ represents the 10000 validation examples, see Figure 3.4.

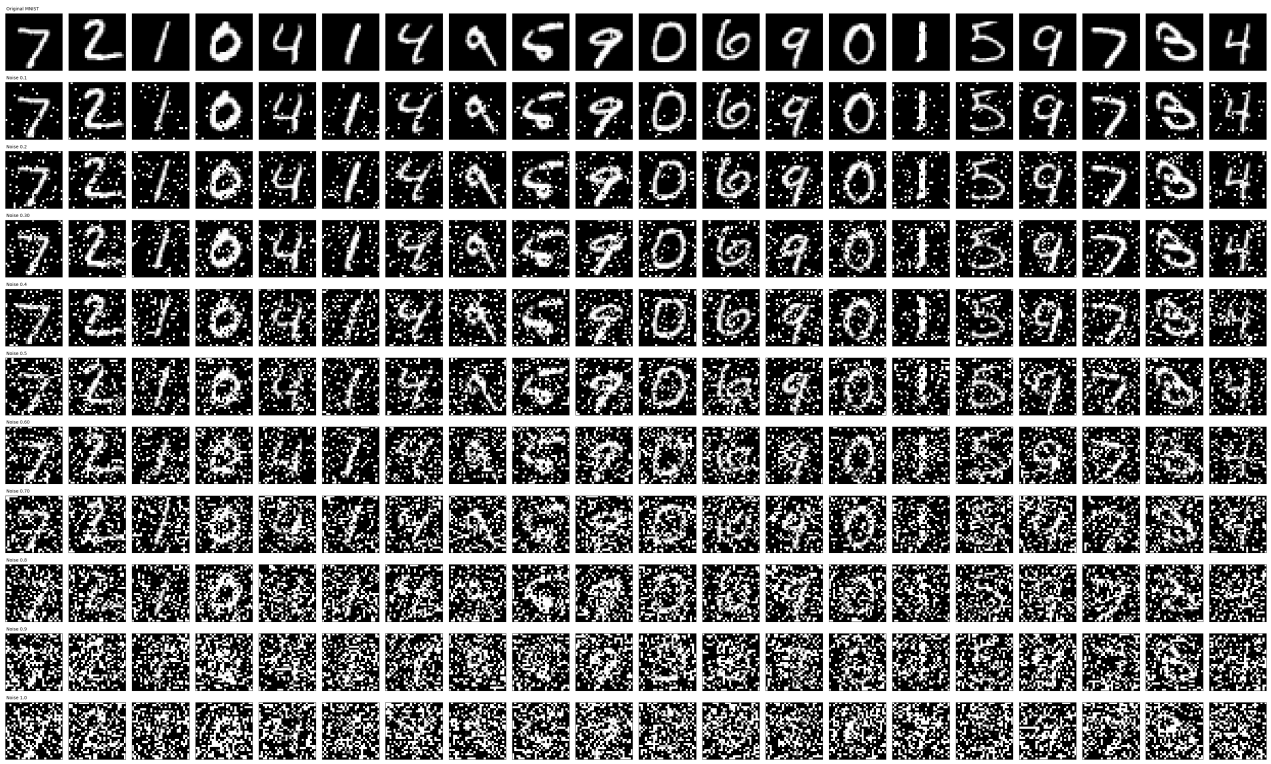

Figure 3.4: First 20 examples of the MNIST dataset, row-wise from top to bottom $SP(MNIST, 0)$, $SP(MNIST, 0.1)$, $SP(MNIST, 0.2)$, $\cdots$, $SP(MNIST, 1)$

For training the autoencoders following datasets where used:

MNIST The MNIST is a digit classification dataset by [31], containing 60,000 training and 10,000 validation examples of 28x28 handwritten digits in gray-scale. The original data ranges $[0, 255]$ as denoted in section 3.3 it was parametrized to $[-1, 1]$.

Noisy As denoted in section 3.4 each of the 60000 training examples was corrupted with $SP(example, prob)$ using a random $prob \in [0, 1]$.

10_Targets As described in 3.4 this target set consists of one image per number class, i.e. all ones are represented by a one, all twos by a two an so on. See Figure 3.5.

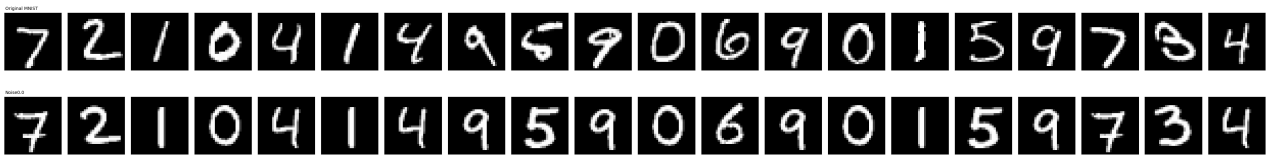

Figure 3.5: Upper row: 20 examples of the original MNIST dataset, Lower row: 20 examples of the 10_Targets dataset, it can be observed that for example the two different 9s in the upper row are represented by the same 9 in the lower row. Both dataset are not corrupted

3.8 Experiments

The experiments where conducted using the structures described in section 3.5.2 and the datasets denoted in section 3.7. Every experiment was conducted with $batch\_size = \{32, 64, 128, 256\}$, in other words for each row of the table 3.1 the experiment was conducted four times each time with different batch size. Following table 3.1 shows a overview of the experiments configuration as combinations of structure, training data and targets. For experiments with names ending in "_iteration" it means that the network was trained once normal and then without resetting the weights it was trained again using $Training\_data = autoencoder.predict(train\_data)$. The rest of the networks where trained only once.

Iteration

For each experiment a classifier was trained using the respective encoded training data as input, and the number labels as targets. the actual iterative denoising experiment was then performed by evaluating the iterative denoising capabilities of the trained autoencoder on the $evaluation\_set$. This was done as the pseudocodes of Algorithm 1 illustrates. The detailed results are presented in chapter 4.

3.9 Implementation and Frameworks

3.9.1 Network Training

The experiments were written in Python 3.7 [33], using the Keras [32] library for the high level neural network programming and training. The autoencoder and classifier networks were made using Keras sequential model

Name	Structure	Training dataset	Target dataset	Comment
Over _dim	Over _dim	MNIST	MNIST	
		Noisy	MNIST	
		MNIST	10_Targets	
Over_dim_iteration	Over _dim	MNIST	MNIST	
		Noisy	MNIST	Only $batch\_size = \{128, 256\}$
		MNIST	10_Targets	
Over_dim_tied	Over_dim_tied	MNIST	MNIST	For this structure the encoder and decoder weights are tied.
		MNIST	10_Targets	
Over_dim_tied_iteration	Over_dim_tied	MNIST	MNIST	For this structure the encoder and decoder weights are tied.
		MNIST	10_Targets	
Normal _dim	Normal _dim	MNIST	MNIST	
		Noisy	MNIST	
		MNIST	10_Targets	
Normal _dim_iteration	Normal _dim	MNIST	MNIST	
		MNIST	10_Targets	
Normal_dim_tied	Normal_dim_tied	MNIST	MNIST	For this structure the encoder and decoder weights are tied.
		MNIST	10_Targets	
Normal_dim_tied_iteration	Over_dim_tied	MNIST	MNIST	For this structure the encoder and decoder weights are tied.
		MNIST	10_Targets	

Table 3.1: Experiments configuration overview

Algorithm 1 Iterative Denoising Evaluation

Input:
$evaluation\_set = \{SP(MNIST, 0), SP(MNIST, 0.1), SP(MNIST, 0.2), \cdots, SP(MNIST, 1)\}$
$ae :=$ trained autoencoder according to experiment configuration.
$enc :=$ encoder of ae, i.e. the layers from input layer to the encoded one.
$cf :=$ trained classifier on the output of enc.
Output: None, data frames are saved to files.
Method:

1: **for** $signal$ **in** $evaluation\_set$ **do**
2: **repeat**
3: add $signal$ to $ae\_predictions$
4: add $ae.evaluate(signal)$ to $ae\_evaluations$
5: $enc\_pred \leftarrow enc.predict(signal)$ and add it to $enc\_predictions$
6: add $cf.predict(enc\_pred)$ to $cf\_predictions$
7: add $cf.eval(enc\_pred)$ to $cf\_evaluations$
8: $signal \leftarrow ae.predict(signal)$
9: **until** desired iterations are done, 7 iteration where used in this work
10: Save a DataFrame containing:
$\{ae\_predictions, ae\_evaluations, enc\_predictions, cf\_predictions, cf\_evaluations\}$
11: **end for**

and Dense layers. Tensorflow 2.0 [34] was used as backend, on a NVIDIA GeForce GTX 1060 GPU. Another mathematical library used was Numpy [35].

The Sacred [36] infrastructure was used to help configure, organize, log and reproduce experiments. This allows to create configuration for easy experimenting, in other words is always the same program with different configurations parameters to run all experiments.

One possibility that Sacred offers is to save the experiments, with it version system, and outputs as a so called Artifacts, in a MongoDb database. The prediction of the autoencoder, encoder and classifiers as well as the training and evaluation histories where stored as compressed DataFrames, as Artifacts, using the library Pandas

[37]. The trained models were saved as hdf5 [38] files using the save function of Keras.

3.9.2 Outcome visualisation

For the Outcome visualisation Jupyter notebook [39] were used with the IPython [40] and Matplotlib [41] libraries.

To load the experiments from the MongoDb and utilizing the same format as Sacred, the Incense [42] toolbox was used.

To visualize the underling manifold of the encoded data t-SNE-CUDA [43] was used, a GPU-accelerated implementation of t-distributed Symmetric Neighbour Embedding (t-SNE) for visualizing datasets and models. In [43] the authors claim that t-SNE-CUDA achieves over $50\times$ speed-up over state-of-the-art t-SNE implementations and over $650\times$ over the popular Scikit-learn library. Finally to create and display the confusion matrix of the classifier the Scikit-learn [44] library was used.

4 Discussion

Following the description in 3.8 for each of the eight experiments the reconstructions, the manifold of the encoded data, and the confusion matrix of the classifier were analysed for each $batch\_size$. Noticeable was the inconsistency between the MSE and, MAE errors in comparison to the reconstruction from the perspective of human vision, which could be explained from the point of view that MSE is not a good measure to evaluate the similarity between two images, as also denoted in [6]. No meaningful results were observable by interpreting the loss functions metrics over iterative evaluating.

To try and solve this for each experiment configuration and noise level, I classified the first five reconstructed images as acceptable or not, following if I could discern a clear number, and this corresponded to the label, or not. And from this 55 analysed images per experiment an Accuracy was calculated, i.e.

$$Acc = \frac{1}{55}\sum_{i=1}^{55}\left(apr = \begin{cases} 1 & \text{if acceptable} \\ 0 & \text{if discarted} \end{cases}\right)$$

4.1 Results comparison

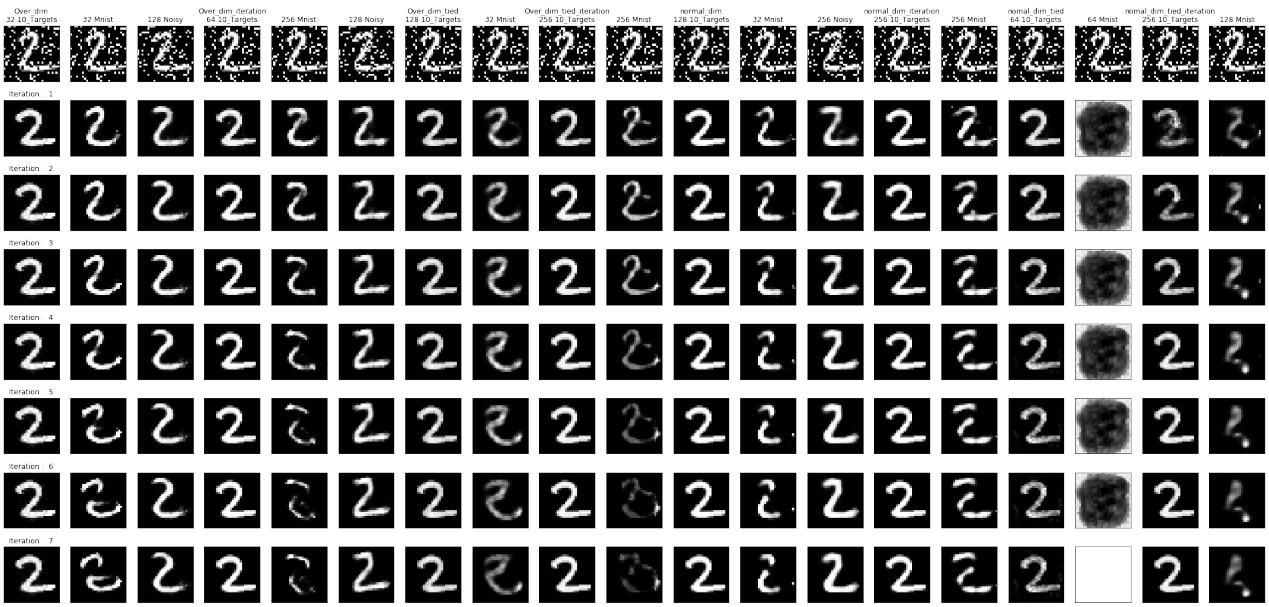

Figure 4.1: Prediction of the best network, for each architecture, batch_size and targets combination. First row: Input images for $SP(Data, 0.4)$, subsequent rows are the iterative reconstruction using the, output, of the above row as input

In this section the reconstruction and classifier accuracies are analysed.

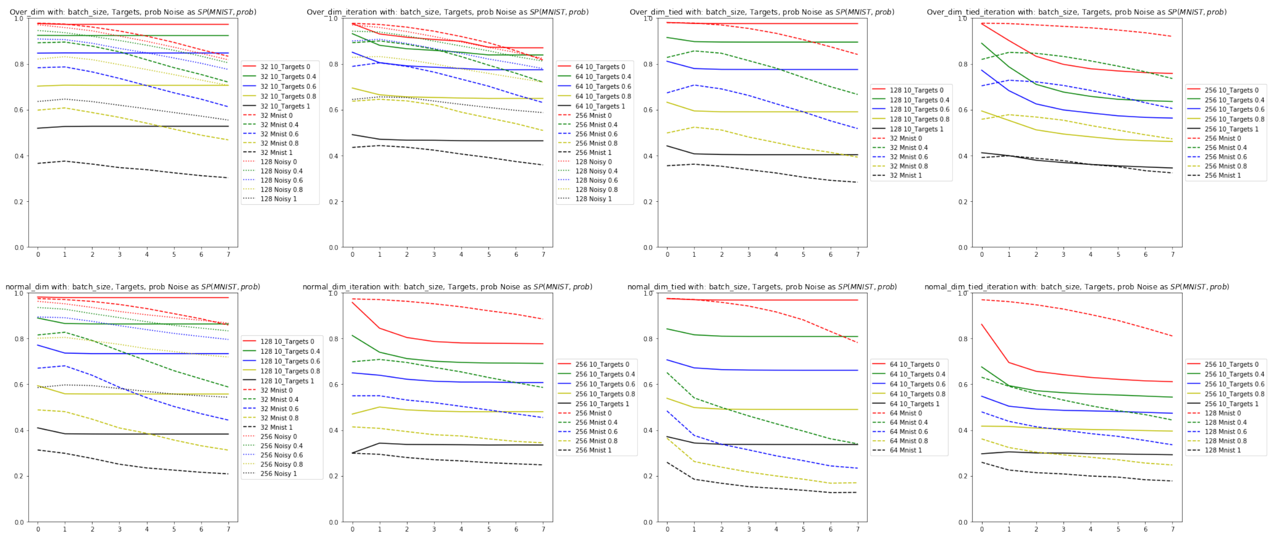

Figure 4.2: Classifier accuracy comparison over different noise levels

4.1.1 Over_dim

With the MNIST target dataset poor reconstructions results were shown. With the best network, $batch\_size = 32$, achieving $Acc = 20\%$. For the $10\_Targets$ dataset way better results were observed with the best network, $batch\_size = 32$ achieving $Acc = 87\%$. Lastly $Noisy$ training achieved almost as good results with $Acc = 80\%$ for $batch\_size = 128$, as shown in 4.3.

The resulting reconstructions for the example of a two and $evaluation\_set'$ $=\{SP(MNIST, 0), SP(MNIST, 0.4), SP(MNIST, 0.6), SP(MNIST, 0.8), SP(MNIST, 1)\}$ can be observed in the first three columns of figures 4.4, 4.1, 4.6, 4.9, 4.11. Showing that MNIST tends to produce a bad reconstruction and with iterations it becomes worse. $Noisy$ and $10\_Targets$ produce similar good results although when looking at figure 4.3 it can be observed that $10\_Targets$ tends to produces more accepted reconstructions. For some examples with MNIST targets, white images were reconstructed as shown in B.6.

Over_dim		1: Reconstruction of the example is good. 0: bad reconstruction																																																						Accuracy: sum of ones divided by 55, number of presented examples
Noise level ->		0%					10%					20%					30%					40%					50%					60%					70%					80%					90%					100%					
Targets	Batch Size	7	2	1	0	4	7	2	1	0	4	7	2	1	0	4	7	2	1	0	4	7	2	1	0	4	7	2	1	0	4	7	2	1	0	4	7	2	1	0	4	7	2	1	0	4	7	2	1	0	4	7	2	1	0	4	
10_Targets	256	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	1	1	1	0	0	0	0	0	1	0	0	0	80.00%
10_Targets	128	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	1	0	0	0	83.64%
10_Targets	64	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	1	1	1	1	0	1	1	1	1	0	1	1	1	0	0	0	0	0	1	0	0	0	78.18%
10_Targets	32	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	0	1	0	0	1	0	1	0	87.27%
MNIST	256	1	0	1	1	0	1	0	1	0	0	1	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	14.55%
MNIST	128	1	0	0	0	0	1	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	18.18%
MNIST	64	1	0	1	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	14.55%
MNIST	32	1	0	0	0	1	1	0	0	0	1	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	20.00%
Noisy	256	1	1	1	0	0	1	1	1	1	0	1	1	1	1	0	1	1	1	1	0	1	1	1	0	0	1	1	1	0	1	0	1	1	1	0	1	1	1	1	0	0	0	1	1	0	1	1	1	1	0	1	0	1	1	0	69.09%
Noisy	128	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1	1	1	1	1	0	0	1	1	0	0	1	0	0	80.00%
Noisy	64	0	1	1	0	0	0	1	1	0	0	0	1	1	0	0	0	1	1	0	0	1	0	1	1	0	1	1	1	0	0	0	1	1	0	0	1	0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0	38.18%
Noisy	32	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	23.64%

Figure 4.3: Visual evaluation of the reconstructions for over_dim

Figure 4.2, upper left corner, shows a comparison of the classification accuracy for each of the above selected networks for increasing noise over the $evaluation\_set'$. Showing that for all networks with increasing noise worse result are produced. For MNIST and $Noisy$, iterations makes the classification worse, but on the other hand for

$10\_Targets$ iterations tend to maintain the classification accuracy. Overall achieving the best classification for this experiment.

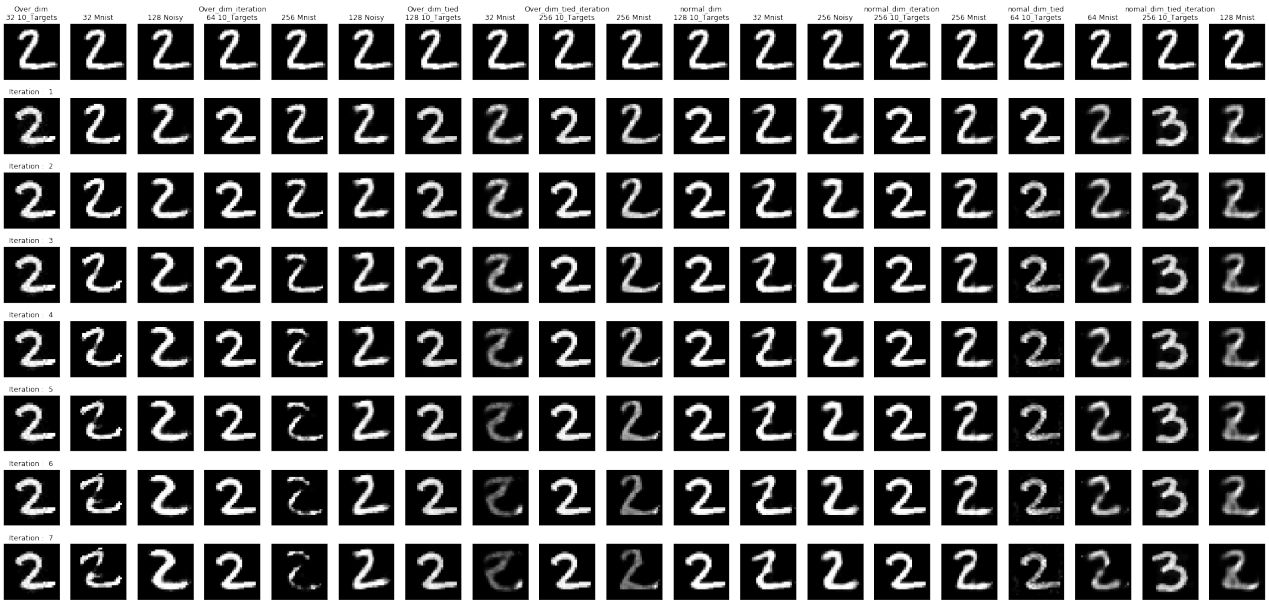

Figure 4.4: Prediction of the best network, for each architecture, batch_size and targets combination. First row: Input images for $SP(Data, 0)$, subsequent rows are the iterative reconstruction using the, output, of the above row as input

4.1.2 Over_dim_iteration

As described in 3.8 this network was trained with iteration. Again the MNIST target dataset shows poorer reconstructions results. With the best network, $batch\_size = 256$, achieving $Acc = 22\%$. For the $10\_Targets$ dataset better results were observed with the best network, $batch\_size = 64$ achieving $Acc = 82\%$. Lastly $Noisy$ training achieved intermediate results with $Acc = 74\%$, for both tested networks $batch\_size = 128, 256$ as shown in 4.3.

The resulting reconstructions for the example of a two and $evaluation\_set' = \{SP(MNIST, 0), SP(MNIST, 0.4), SP(MNIST, 0.6), SP(MNIST, 0.8), SP(MNIST, 1)\}$ are displayed in the fourth to sixth columns of figures 4.4, 4.1, 4.6, 4.9, 4.11. Showing that MNIST tends to produce a bad reconstruction and with iterations it becomes worse. $Noisy$ and $10\_Targets$ produce similar good results. For some examples with MNIST targets, white images were reconstructed.

Figure 4.2, second diagram on the first row, shows a comparison of the classification accuracy for each of the above selected networks for increasing noise over the $evaluation\_set'$. Showing that for all networks with increasing noise worse result are produced. For all three types iterations make the classification worse. For lower noise levels $Noisy$ trained networks achieve better classification, although with iterations it tends to fall quicker and for iteration 6 and 7 $10\_Targets$ obtains better results.

Over_dim_iteration — 1: Reconstruction of the example is good. 0: bad reconstruction

Noise level ->		0%					10%					20%					30%					40%					50%				
Targets	Batch Size	7	2	1	0	4	7	2	1	0	4	7	2	1	0	4	7	2	1	0	4	7	2	1	0	4	7	2	1	0	4
10_Targets	256	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	0	1	1	1	1	1	1	1
10_Targets	128	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
10_Targets	64	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
10_Targets	32	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
MNIST	256	1	0	0	1	1	0	0	0	0	1	1	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0
MNIST	128	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
MNIST	64	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
MNIST	32	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Noisy	256	1	0	1	1	1	1	0	1	1	1	1	0	1	1	1	1	0	1	1	1	1	0	1	1	1	1	0	1	1	1
Noisy	128	1	0	1	0	1	1	1	1	0	1	1	0	1	0	1	1	1	1	1	1	1	0	1	0	1	1	1	1	0	1

Noise level ->		60%					70%					80%					90%					100%					Accuracy: sum of ones divided by 55, number of presented examples
Targets	Batch Size	7	2	1	0	4	7	2	1	0	4	7	2	1	0	4	7	2	1	0	4	7	2	1	0	4	
10_Targets	256	1	1	1	1	1	1	1	0	1	1	1	1	0	0	1	1	0	0	0	0	0	1	0	0	0	76.36%
10_Targets	128	1	1	1	1	1	1	1	0	1	1	1	1	0	0	1	1	0	0	0	0	0	1	0	0	0	80.00%
10_Targets	64	1	1	1	1	1	1	1	1	0	1	1	1	0	1	1	1	0	0	0	0	0	1	0	0	0	81.82%
10_Targets	32	1	1	1	1	1	1	1	1	0	1	1	0	0	1	1	1	0	0	0	0	1	0	0	0	0	80.00%
MNIST	256	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	21.82%
MNIST	128	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	7.27%
MNIST	64	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	9.09%
MNIST	32	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3.64%
Noisy	256	1	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	0	0	1	1	1	0	1	0	0	74.55%
Noisy	128	1	1	1	1	1	1	1	1	1	1	0	0	1	1	0	1	1	1	1	1	0	0	1	1	0	74.55%

Figure 4.5: Visual evaluation of the reconstructions for Over_dim_iteration

Figure 4.6: Prediction of the best network, for each architecture, batch_size and targets combination. First row: Input images for $SP(Data, 0.6)$, subsequent rows are the iterative reconstruction using the, output, of the above row as input

4.1.3 Over_dim_tied

As described in 2.2.5 for this network the encoder and decoder weights are tied. The MNIST target dataset shows worse reconstructions results than before. With the best network, $batch\_size = 32$, achieving $Acc = 14\%$. For the $10\_Targets$ dataset better results were observed with the best network, $batch\_size = 128$ achieving $Acc = 80\%$, as shown in 4.7.

The resulting reconstructions for the example of a two and $evaluation\_set' = \{SP(MNIST, 0), SP(MNIST, 0.4), SP(MNIST, 0.6), SP(MNIST, 0.8), SP(MNIST, 1)\}$ are displayed in the seventh and eight columns of figures 4.4, 4.1, 4.6, 4.9, 4.11. Showing that MNIST tends to not produce acceptable reconstructions and with iterations it becomes worse. $10\_Targets$ produces significant better results. For some examples with MNIST targets, images of grey circles with white borders were reconstructed.

Figure 4.2, third diagram on the first row, shows a comparison of the classification accuracy for each of the above selected networks for increasing noise over the $evaluation\_set'$. Showing that for all networks with increasing

Figure 4.7: Visual evaluation of the reconstructions for Over_dim_tied

noise worse results are produced. For MNIST and $10\_Targets$ iteration makes the classification worse, but for $10\_Targets$ it tends to become constant after one iteration.

4.1.4 Over_dim_tied_iteration

As described in sections 2.2.5 and 3.8 for this network the encoder and decoder weights are tied and it also was trained with iteration. The MNIST target dataset shows slightly better reconstructions results than without iteration. With the best network, $batch\_size = 256$, achieving $Acc = 20\%$. For the $10\_Targets$ dataset significant better results were observed with the best network, $batch\_size = 256$ achieving $Acc = 74\%$, as shown in figure 4.8.

The resulting reconstructions for the example of a two and $evaluation\_set' = \{SP(MNIST, 0), SP(MNIST, 0.4), SP(MNIST, 0.6), SP(MNIST, 0.8), SP(MNIST, 1)\}$ are displayed in the ninth and tenth columns of figures 4.4, 4.1, 4.6, 4.9, 4.11. Showing that MNIST tends to not produce acceptable reconstructions and with iterations it becomes worse. $10\_Targets$ produces clearly better results. For some examples with MNIST targets, images of grey circles with white borders were reconstructed. And also some containing NaN what could be a sign for exploding gradient.

Figure 4.8: Visual evaluation of the reconstructions for Over_dim_tied_iteration

Figure 4.2, fourth diagram on the first row, shows a comparison of the classification accuracy for each of the above selected networks for increasing noise over the $evaluation\_set'$. Showing that for all networks with increasing noise worse results are produced. For MNIST and $10\_Targets$ iteration make the classification worse, in contrast to the previous experiments this time MNIST, tends to achieve better or equal classification accuracy than $10\_Targets$.

4.1.5 Normal_dim

With the MNIST target almost no useful reconstructions were shown. With the networks achieving $Acc = 4\%$. For the $10\_Targets$ dataset clearly better results were observed with the best network, $batch\_size = 128$ achieving

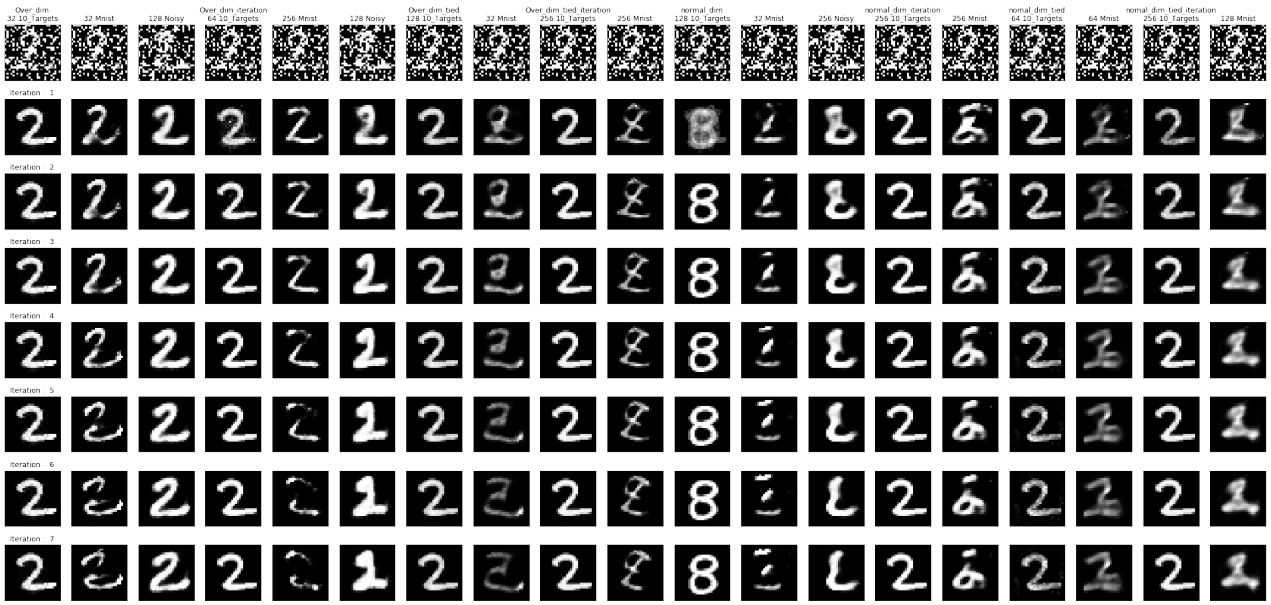

Figure 4.9: Prediction of the best network, for each architecture, batch_size and targets combination. First row: Input images for $SP(Data, 0.8)$, subsequent rows are the iterative reconstruction using the, output, of the above row as input

$Acc = 78\%$. Lastly $Noisy$ training achieved the best results with $Acc = 82\%$ for $batch\_size = 256$, as shown in 4.10.

The resulting reconstructions for the example of a two and $evaluation\_set' = \{SP(MNIST, 0), SP(MNIST, 0.4), SP(MNIST, 0.6), SP(MNIST, 0.8), SP(MNIST, 1)\}$ can be observed in the eleventh to thirteenth columns of figures 4.4, 4.1, 4.6, 4.9, 4.11. Showing that MNIST didn't produce any useful reconstruction. $Noisy$ and $10\_Targets$ produce similar good results, with $10\_Targets$ producing cleaner images but as observed figure 4.3 Noisy tends to produces more accepted reconstructions for higher noise levels.

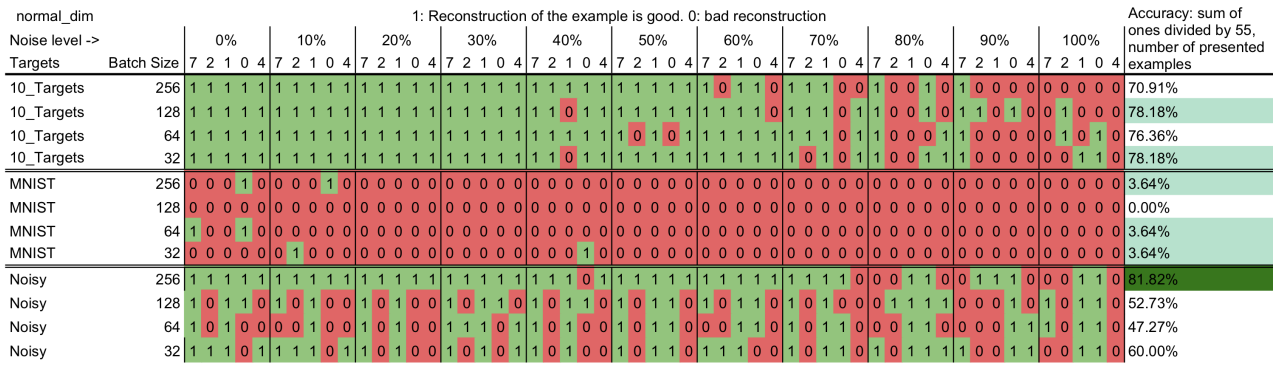

Figure 4.10: Visual evaluation of the reconstructions for Normal_dim

Figure 4.2, first diagram on the second row, shows a comparison of the classification accuracy for each of the above selected networks for increasing noise over the $evaluation\_set'$. Showing that for all networks with increasing noise worse results are produced. For MNIST and $10\_Targets$ iteration make the classification worse, but for $10\_Targets$ it tends to become constant after one iteration, retaining higher accuracy.

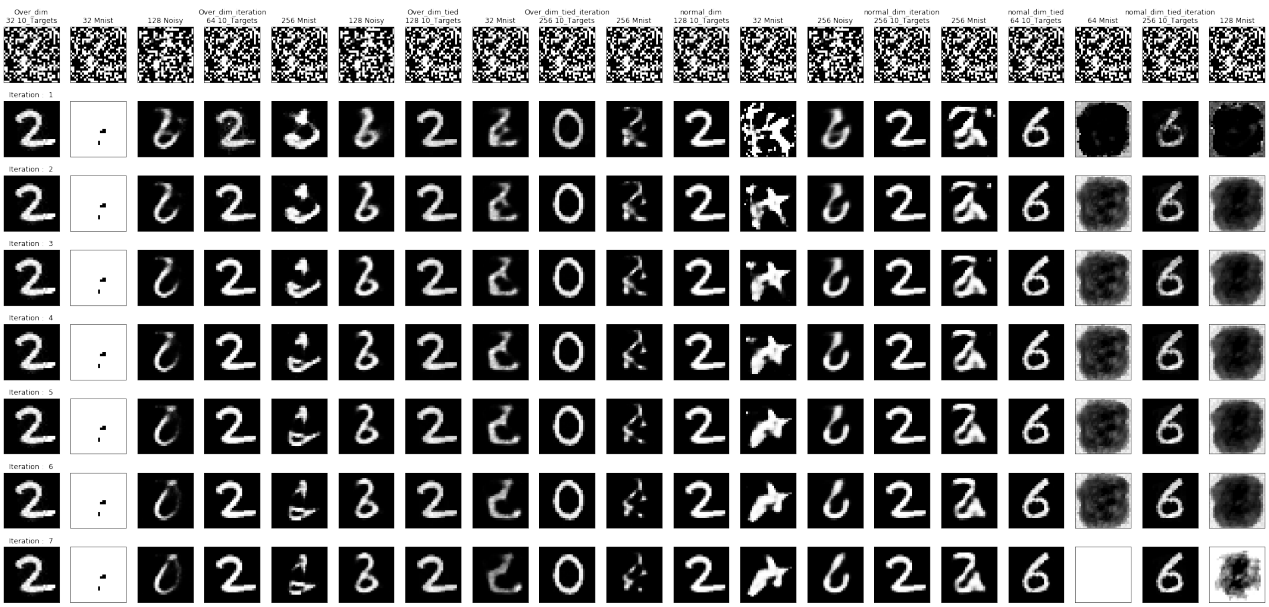

Figure 4.11: Prediction of the best network, for each architecture, batch_size and targets combination. First row: Input images for $SP(Data, 7)$, subsequent rows are the iterative reconstruction using the, output, of the above row as input

4.1.6 Normal_dim_iteration

As described in 3.8 this network was trained with iteration. The MNIST target dataset shows better reconstructions results than without iteration nevertheless achieving mediocre results. With the best network, $batch\_size = 256$, achieving $Acc = 11\%$. For the $10\_Targets$ dataset acceptable results were observed with the best network, $batch\_size = 256$ achieving $Acc = 71\%$ as shown in 4.12.

The resulting reconstructions for the example of a two and $evaluation\_set' = \{SP(MNIST, 0), SP(MNIST, 0.4), SP(MNIST, 0.6), SP(MNIST, 0.8), SP(MNIST, 1)\}$ are displayed in the thirteen and fourteen columns of figures 4.4, 4.1, 4.6, 4.9, 4.11.Showing that MNIST didn't produce any useful reconstruction. Contrary to this $10\_Targets$ produced good results.

Figure 4.12: Visual evaluation of the reconstructions for Normal_dim_iteration

Figure 4.2, second diagram on the second row, shows a comparison of the classification accuracy for each of the above selected networks for increasing noise over the $evaluation\_set'$. Showing that for all networks with increasing noise worse results are produced. For MNIST and $10\_Targets$ iterations make the classification worse. Similar to 4.1.4 MNIST, achieves better classification accuracy than $10\_Targets$ for no corruption. For higher noise

$10\_Targets$ obtains better and more constant results.

4.1.7 Normal_dim_tied

As described in 2.2.5 for this network the encoder and decoder weights are tied. The MNIST target dataset shows the same bad reconstructions results as without tying the weights, although for the inverted $batch\_sizes$, i.e where $Normal\_dim$ achieved $Acc = 0\%$ $Normal\_dim\_tied$ achieves $Acc = 4\%$. With the $batch\_size = 64$. For the $10\_Targets$ dataset slightly better results than without tied weights were observed with the best network, $batch\_size = 64$ achieving $Acc = 74\%$, as shown in 4.7.

The resulting reconstructions for the example of a two and $evaluation\_set' = \{SP(MNIST, 0), SP(MNIST, 0.4), SP(MNIST, 0.6), SP(MNIST, 0.8), SP(MNIST, 1)\}$ are displayed in the fifteenth and sixteenth columns of figures 4.4, 4.1, 4.6, 4.9, 4.11. Showing that MNIST tends to not produce any useful reconstructions. Contrarily $10\_Targets$ produces good results, although in comparison to previous experiments worse. For some examples with MNIST targets, images of grey circles with white borders were reconstructed as observed in 4.1.

Figure 4.13: Visual evaluation of the reconstructions for Normal_dim_tied

Figure 4.2, third diagram on the second row, shows a comparison of the classification accuracy for each of the above selected networks for increasing noise over the $evaluation\_set'$. Showing that for all networks with increasing noise worse results are produced. For MNIST and $10\_Targets$ iteration makes the classification worse, but for $10\_Targets$ it remains almost constant and produces significantly better classification.

4.1.8 Normal_dim_tied_iteration

As described in sections 2.2.5 and 3.8 for this network the encoder and decoder weights are tied and it also was trained with iteration. The MNIST target dataset shows no significant improvement in comparison to without iteration. With the best network, $batch\_size = 128$, achieving $Acc = 5\%$. For the $10\_Targets$ dataset better results were observed with the best network, $batch\_size = 256$ achieving $Acc = 69\%$, as shown in figure 4.14.

The resulting reconstructions for the example of a two and $evaluation\_set' = \{SP(MNIST, 0), SP(MNIST, 0.4), SP(MNIST, 0.6), SP(MNIST, 0.8), SP(MNIST, 1)\}$ are displayed in the last two columns of figures 4.4, 4.1, 4.6, 4.9, 4.11. Showing that MNIST tends to not produce acceptable reconstructions and with iterations it becomes worse. In comparison $10\_Targets$ produces good results, slightly better than without iteration. For some examples with MNIST targets, images of grey circles with white borders were reconstructed. And also NaN what could be a sign for exploding gradient.

Figure 4.14: Visual evaluation of the reconstructions for Normal_dim_tied_iteration

Figure 4.2, fourth diagram on the second row, shows a comparison of the classification accuracy for each of the above selected networks for increasing noise over the $evaluation\_set'$. Showing that for all networks with increasing noise worse result are produced. For MNIST and $10\_Targets$ iterations make the classification worse, for lower noise levels MNIST produces better classification but for higher noise levels $10\_Targets$ remains almost constant achieving better classification.

4.1.9 Overall

Overall no clear winning architecture can be found, nevertheless $10\_Targets$ definitively produces better reconstructions, with the best network, $Over\_Dim$ with $batch\_size = 32$ achieving $Acc = 87,27$ for all levels of noise combined. By limiting the corruption to 0.7 and 0.8 almost all reconstructions are successful achieving $Acc = 100\%$ and $Acc = 98\%$ for this network.

Although numerically these seems to show superiority, when looking at the reconstructions and manifolds produced using $10\_Targets$, no clear difference between $Over\_dim$ and other architectures nor $batch\_sizes$ is discernible

For further numbers reconstructions the figures in section B.2 of the appendix show examples for a seven.

4.2 Confusion matrix

As a form to corroborate how good the representation of the encoded dimension is, a classifier was trained using the 36 or 30 dimensional encoded data, corresponding to the outputs of the $Over\_dim$ and $Normal\_dim$ architectures see Section 3.5 for details. Figures 4.15, 4.16, and 4.17 present selected results for a subset of the $evaluation\_set$ and only some experiment configurations were selected as most representable to avoid showing all 74 experiments.

The subset $evaluation\_set' = \{SP(MNIST, 0), SP(MNIST, 0.7), SP(MNIST, 1)\}$ was chosen as the plots are very similar to one another for the other intermediate values of $SP(MNIST, \alpha), \alpha \in (0, 1)$.

For noise over $prob > 0.7$, figure 4.17, almost all classifiers deliver bad results, for less noise $prob \leq 0.7$, figures 4.15 and 4.16, with exception of $Over\_dim\_tied\_iteration$ with $batch\_size = 64$ and $10\_Targets$ which classifies all numbers as corresponding to class 1. Similar behaviour tendencies are observed for all networks for noise $prob = 1$.

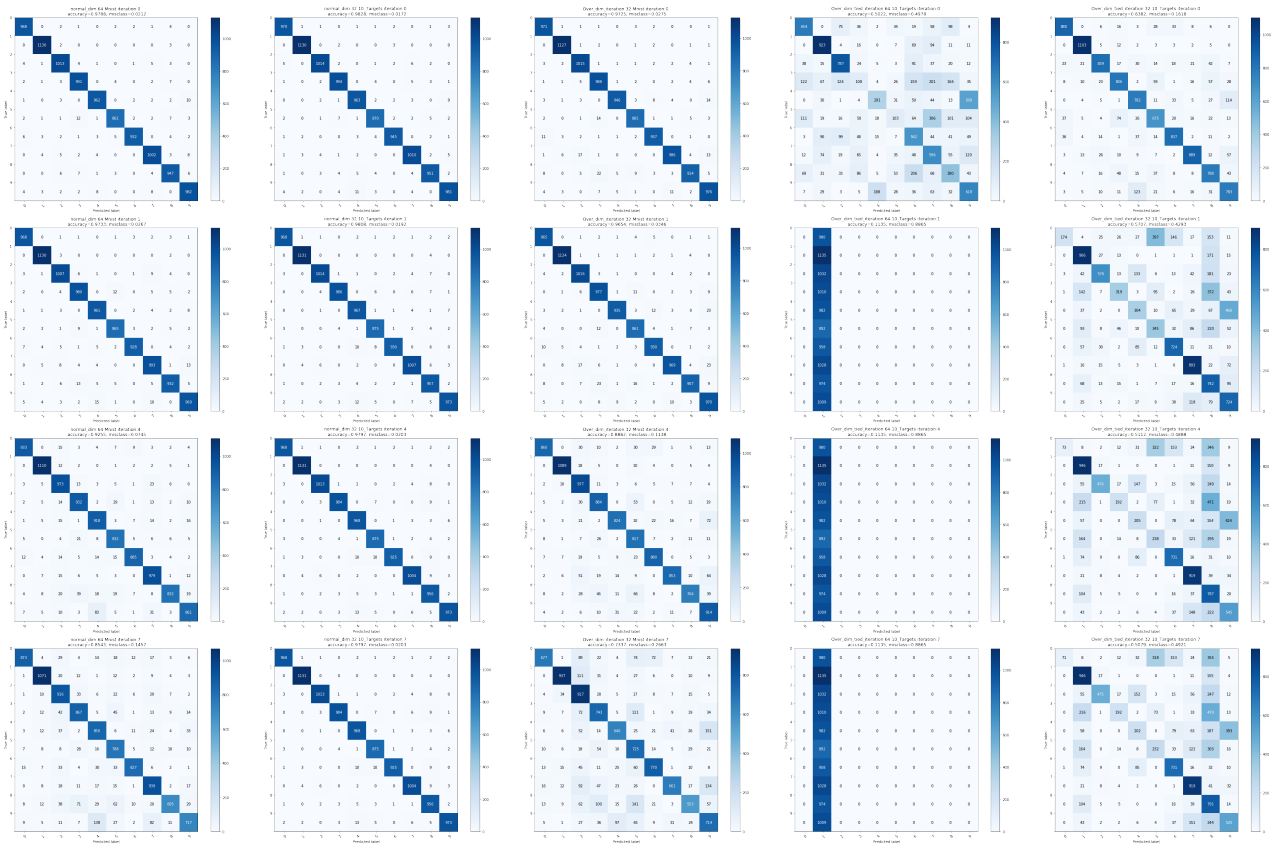

Figure 4.15: Selected confusion matrices of the feature classifier for $SP(MNIST, 0)$. First row: confusion matrix of of $class\_pred\_1 = classifier.predict(SP(MNIST, 0))$. Second row: confusion matrix of of $class\_pred\_2 = classifier.predict(class\_pred\_1)$. Third row: confusion matrix of of $class\_pred\_2 = classifier.predict(class\_pred\_1)$. Fourth row: confusion matrix of of $class\_pred\_2 = classifier.predict(class\_pred\_1)$.

4.3 Manifold

To analyse the underlying manifold generated from the 36 or 30 dimensional encoded data, corresponding to the outputs of the $Over\_dim$ and $Normal\_dim$ architectures, see Section 3.5 for details, t-sne (CUDA [43]) was used to reduce the dimensionality to a visualizable $2D$. As described in section 3.9.2. Figures 4.18, 4.19, and 4.20 present selected results for a subset of the $evaluation\_set$. The subset $evaluation\_set' = \{SP(MNIST, 0), SP(MNIST, 0.5), SP(MNIST, 1)\}$ was chosen then the plots are very similar to one another for the other intermediate values of $SP(MNIST, \alpha), \alpha \in (0, 1)$.

It can be observed that batch size doesn't seem to change the manifold significantly. Further for the experiments that were trained with iterations (experiments ending in "_iteration" see section 3.8 for details) a distinction in the manifold is observable, mostly rotation and the cluster formation becomes slightly worse through the iterations. Furthermore when using the Noisy and MNIST training datasets there is no appreciable difference.

Tied weights (experiments containing in "_tied" see section 3.8 for details) also seem to make the clustering worse. Over_dim_tied_iteration shows in contrast to Over_dim for the 10_Target target dataset significant worsening. This is a combination of the effects of tied weights and iterations.

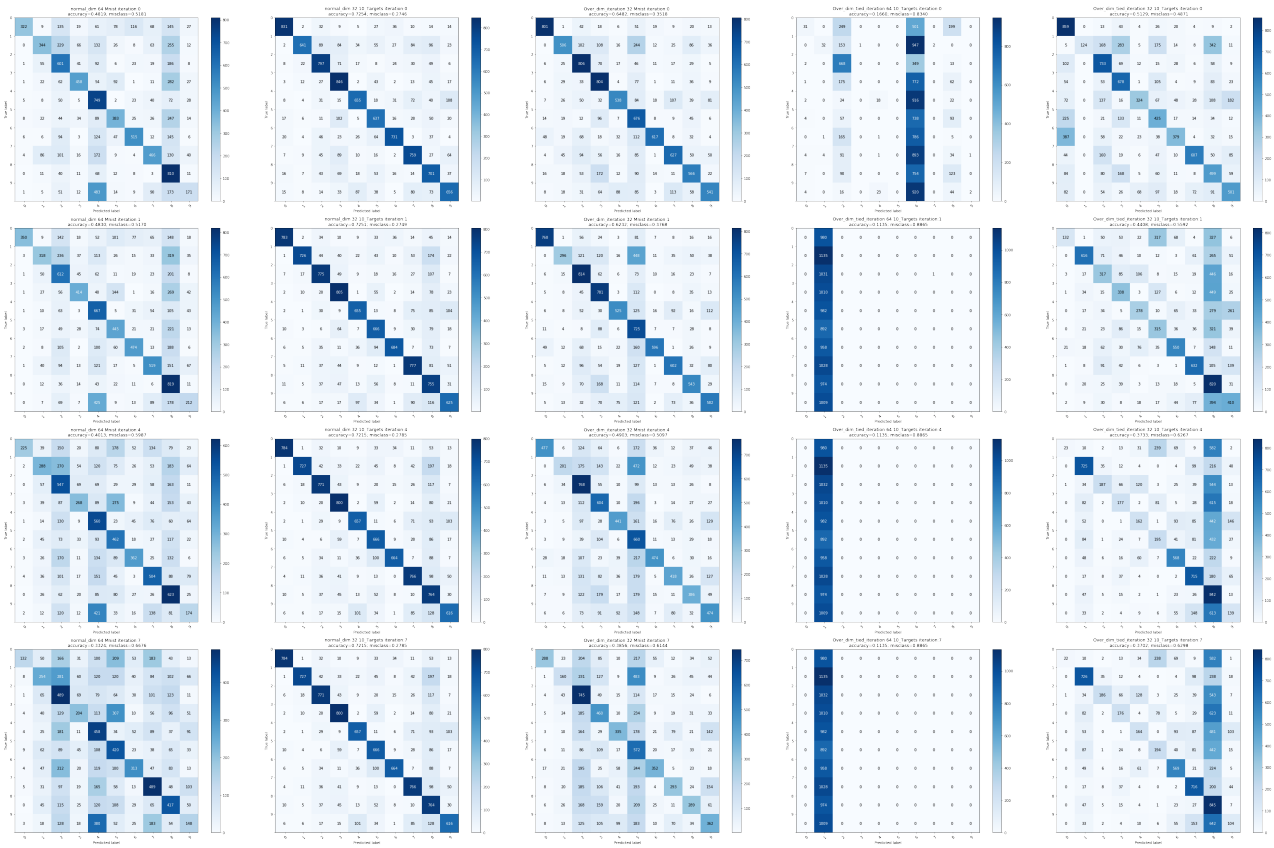

Figure 4.16: Selected confusion matrices of the feature classifier for $SP(MNIST, 0.7)$. First row: confusion matrix of of $class\_pred\_1 = classifier.predict(SP(MNIST, 0.7))$. Second row: confusion matrix of of $class\_pred\_2 = classifier.predict(class\_pred\_1)$. Third row: confusion matrix of of $class\_pred\_2 = classifier.predict(class\_pred\_1)$. Fourth row: confusion matrix of of $class\_pred\_2 = classifier.predict(class\_pred\_1)$.

For the Noisy and MNIST training dataset both architectures, Over_dim and Normal_dim, don't present a big difference. Although normal_dim produces marginal worse reconstructions.

Overall a big difference between the manifolds of MNIST (and Noisy) and 10_Targets can be observed. For the first case a kind of clustering is achieved, but iterations don't seem to improve the results on the contrary it seems to worsen, this is also reflected in the reconstruction error. The Noisy datasets manifold does not have an apparent structure but nevertheless the reconstruction are good. On the other hand 10_Targets produces a very defined separation of the data in clusters. But it does generate multiple separated clusters per class.

It can also be observed that for corrupted inputs up to approximately a noise $prob = 0.7 \implies 70\%$ good results are obtained. Figure 4.20 is included to show what happens when the input is almost totally corrupted multiple networks can't achieve useful representations. But surprisingly although the clusters aren't cleanly defined, multiple miss clustering, the $10\_targets$ networks produces surprisingly good representations.

Noteworthy is that the iterative trained networks produced NaN or inf values when evaluating after the fourth iteration, not training iteration but prediction iteration as described in Figure 4.18. This surprising behaviour also occurred in some networks with tied weights and $batch\_size = \{128, 256\}$. It could be an indicator of exploding gradient.

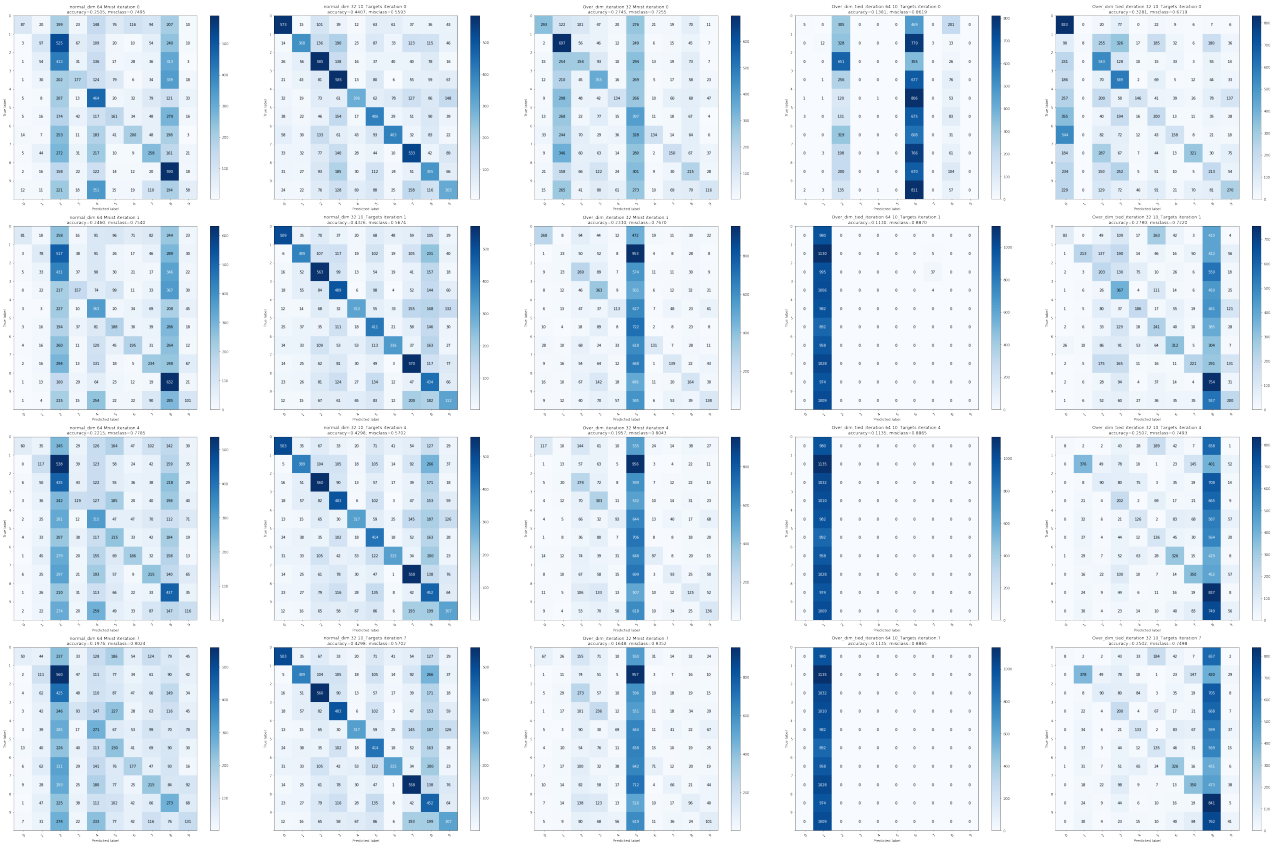

Figure 4.17: Selected confusion matrices of the feature classifier for $SP(MNIST, 1)$. First row: confusion matrix of of $class\_pred\_1 = classifier.predict(SP(MNIST, 1))$. Second row: confusion matrix of of $class\_pred\_2 = classifier.predict(class\_pred\_1)$. Third row: confusion matrix of of $class\_pred\_2 = classifier.predict(class\_pred\_1)$. Fourth row: confusion matrix of of $class\_pred\_2 = classifier.predict(class\_pred\_1)$.

For high corruption probabilities not all networks produced useful results see Appendix B.1, for surprisingly artistic visualisations.

4.4 Conclusion and experiences

As observed in the previous sections 3.8, 4.2, and 4.4 no one architecture seem to generate definitive better results. Nevertheless the targets-set $10\_Targets$, showed promising outcomes.

Concluding that without special constraints, in the form of:

- more complex architectures
- special loss functions
- noisy or corrupted training data

the iterative reconstruction of autoencoders offer only a slight advantage if at all against the same autoencoders applied only once. Further with each iteration the results seem to worsen.

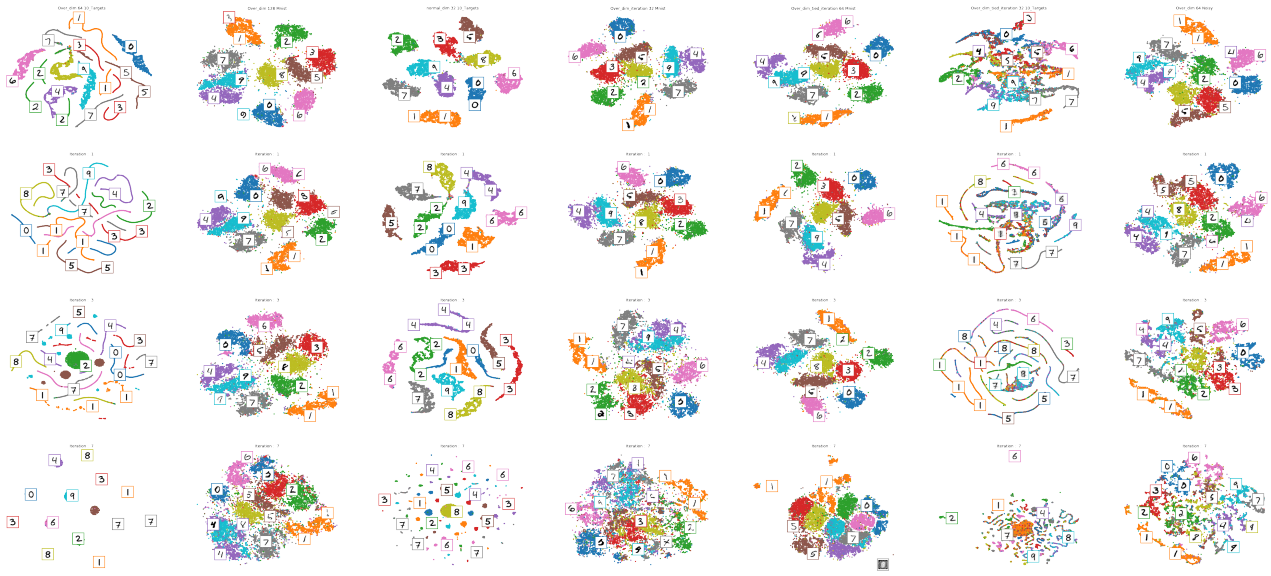

Figure 4.18: Selected manifold visualisation, with t-sne, for $SP(MNIST, 0)$. First row: the manifold produced with the encoded input data, i.e. t-sne of $enc\_pred\_1 = encoder.predict(SP(MNIST, 0))$. Second row: the manifold produced in the first iteration i.e.t-sne of $enc\_pred\_2 = encoder.predict(enc\_pred\_1)$. Third row: the manifold produced in the fourth iteration i.e.t-sne of $enc\_pred\_4 = encoder.predict(enc\_pred\_3)$. Fourth row: the manifold produced in the seventh, an last, iteration i.e.t-sne of $enc\_pred\_7 = encoder.predict(enc\_pred\_6)$.

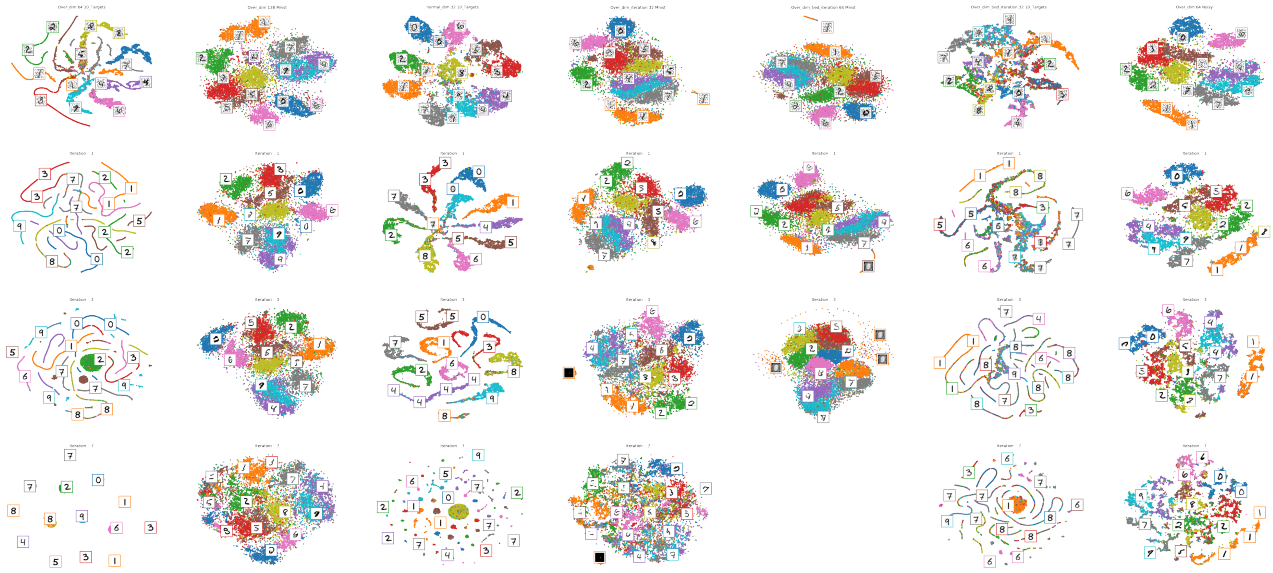

Figure 4.19: Selected manifold visualisation, with t-sne, for $SP(MNIST, 0, 5)$. First row: t-sne of $enc\_pred\_1 = encoder.predict(SP(MNIST, 0.5))$. Second row: t-sne of $enc\_pred\_2 = encoder.predict(enc\_pred\_1)$. t-sne of $enc\_pred\_4 = encoder.predict(enc\_pred\_3)$. Fourth row: t-sne of $enc\_pred\_7 = encoder.predict(enc\_pred\_6)$.

It could not be clearly shown whether iterations offer an improved solution, since the reconstructions behave very differently on examples, not only on distinct examples but also on the same example when applied iteratively, and do not follow any general criterion. Nevertheless under specific constraints encouraging behaviour can be observed.

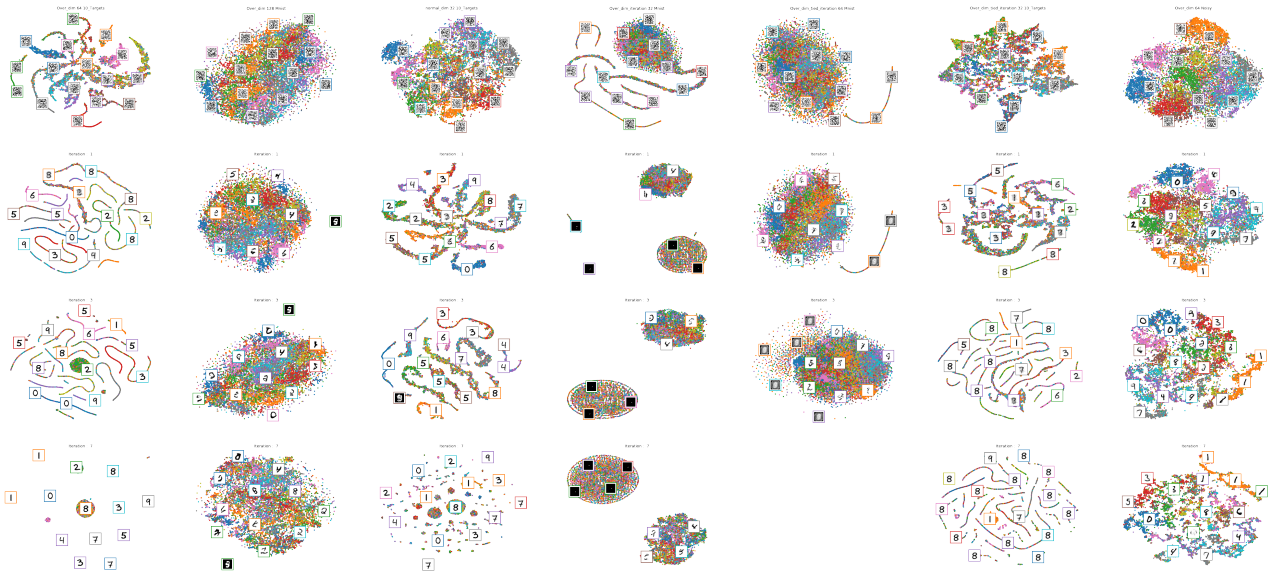

Figure 4.20: Selected manifold visualisation, with t-sne, for $SP(MNIST, 1)$. First row: t-sne of $enc\_pred\_1 = encoder.predict(SP(MNIST, 1))$. Second row: t-sne of $enc\_pred\_2 = encoder.predict(enc\_pred\_1)$. Third row: t-sne of $enc\_pred\_4 = encoder.predict(enc\_pred\_3)$. Fourth row:t-sne of $enc\_pred\_7 = encoder.predict(enc\_pred\_6)$.

To further investigate this, additional experiments with other types of neural networks, and more complex constraints are suggested in the next chapter 5.

Some of the initial theoretical assumptions were wrong and because of the long time consumption to train the networks not all newly discovered alternatives, parameters and architectures could be tested.

The corrupted training set was generated only once, it would be better to add a corruption layer to the network so that in the training each epoch would be different. But for the evaluation set it should be done only once as this is example data, i.e. all networks should become the same input while evaluating.

Autoencoders trained with noisy data were wrongly assumed as failing in the initial test and were not thoroughly tested, since the time cost to execute all the experiments was too high. Furthermore the test data for DAE was wrong in the code and it probably lead to $Noisy$ having worse results than it could have achieved.

Personally I was astonished by the time consumption and hardship of hyper-parameter testing. And I had to limit my search to only a couple of them, nevertheless making 74 experiments. This produced very much data, 130 GB and was very time consuming, every test of a configuration was at least 40 min. A big challenge was to keep an overview of this, personally I underestimate the time consumption of analysing all the results, and knowing what to do with them. Regardless, this thesis helped me to acquire much knowledge in a thematic that I find very interesting and was all in all a very fascinating research opportunity.

5 Future work

In this thesis approaches using "normal" autoencoder architectures were proposed but other more complex neural networks could deliver better results on iterative signal Denoising. For example convolutional auto-encoder (CAE) for unsupervised feature learning presented in [11] could be used instead of the autoencoder proposed. A stack of CAEs forms a convolutional neural network (CNN). Each CAE is trained using conventional on-line gradient descent without additional regularization terms. A max-pooling layer is essential to learn biologically plausible features. Initializing a CNN with filters of a trained CAE stack yields superior performance on a digit (MNIST) and an object recognition (CIFAR10) benchmark according to the results obtained by [11]. Similar approaches are also proposed in [45]

Further another more stochastic type of autoencoders proposed in [46] are the so called “adversarial autoencoder” (AAE), which is a probabilistic autoencoder that uses the recently proposed generative adversarial networks (GAN) to perform variational inference by matching the aggregated posterior of the hidden code vector of the autoencoder with an arbitrary prior distribution. Matching the aggregated posterior to the prior ensures that generating from any part of prior space results in meaningful samples. As a result, the decoder of the adversarial autoencoder learns a deep generative model that maps the imposed prior to the data distribution. Through this properties the effect of corruption could be less disruptive.

Another idea would be to try how Long Short-Term Memory (LSTM) autoencoders behave, since their common use is on sequences, and iteration could be considered a sequence.

Further more complex autoencoders structures could be tried, as a network with a classifier on the encoded dimension but not as an extra network, i.e. one input and two outputs. Then this network could be trained with two losses simultaneously one on classification and another on reconstruction. Also a third output could be present to represent the data manifold and an associated clustering or representation loss.

Lastly the effects of iterative training could be researched, for example different learning rates or data modification between iterations. Further more complex loss functions to get a richer feature representation or as proposed in [13] adapt the autoencoder for clustering by simultaneously considering data reconstruction and compactness. Another possibility is to see what effect different hidden layer dimensions, and dimension behaviour produce. For example what if the third hidden layer is smaller than all previous but then the fourth bigger than all before and then the fifth smaller again and so on.

Bibliography

[1] Vincent, P., Larochelle, H., Bengio, Y., Manzagol, P.A.: Extracting and composing robust features with denoising autoencoders. In: Proceedings of the 25th international conference on Machine learning - ICML '08, Helsinki, Finland, ACM Press (2008) 1096–1103

[2] Hinton, G.E., Osindero, S., Teh, Y.W.: A fast learning algorithm for deep belief nets. Neural Comput. **18** (2006) 1527–1554

[3] Bengio, Y., Lamblin, P., Popovici, D., Larochelle, H.: Greedy layer-wise training of deep networks. In: Proceedings of the 19th International Conference on Neural Information Processing Systems. NIPS'06, Cambridge, MA, USA, MIT Press (2006) 153–160

[4] Ranzato, M., Poultney, C., Chopra, S., LeCun, Y.: Efficient learning of sparse representations with an energy-based model. In: Proceedings of the 19th International Conference on Neural Information Processing Systems. NIPS'06, Cambridge, MA, USA, MIT Press (2006) 1137–1144

[5] Wang, Y., Yao, H., Zhao, S.: Auto-encoder based dimensionality reduction. Neurocomputing **184** (2016) 232–242

[6] Wang, J., He, H., Prokhorov, D.V.: A Folded Neural Network Autoencoder for Dimensionality Reduction. Procedia Computer Science **13** (2012) 120–127

[7] Hinton, G.E., Salakhutdinov, R.R.: Reducing the dimensionality of data with neural networks. Science **313** (2006) 504–507

[8] Shao, H., Jiang, H., Zhao, H., Wang, F.: A novel deep autoencoder feature learning method for rotating machinery fault diagnosis. Mechanical Systems and Signal Processing **95** (2017) 187–204

[9] Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., Manzagol, P.A.: Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. J. Mach. Learn. Res. **11** (2010) 3371–3408

[10] Bengio, Y., Courville, A., Vincent, P.: Representation Learning: A Review and New Perspectives. arXiv:1206.5538 [cs] (2014) arXiv: 1206.5538.

[11] Masci, J., Meier, U., Cireşan, D., Schmidhuber, J.: Stacked convolutional auto-encoders for hierarchical feature extraction. In Honkela, T., Duch, W., Girolami, M., Kaski, S., eds.: Artificial Neural Networks and Machine Learning – ICANN 2011, Berlin, Heidelberg, Springer Berlin Heidelberg (2011) 52–59

[12] Alain, G., Bengio, Y.: What Regularized Auto-Encoders Learn from the Data Generating Distribution. arXiv:1211.4246 [cs, stat] (2014) arXiv: 1211.4246.

[13] Song, C., Liu, F., Huang, Y., Wang, L., Tan, T.: Auto-encoder Based Data Clustering. In Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Ruiz-Shulcloper, J., Sanniti di Baja, G., eds.: Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications. Volume 8258. Springer Berlin Heidelberg, Berlin, Heidelberg (2013) 117–124

[14] Kuchaiev, O., Ginsburg, B.: Training Deep AutoEncoders for Collaborative Filtering. arXiv:1708.01715 [cs, stat] (2017) arXiv: 1708.01715.

[15] Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples (2014)

[16] Jaques, N., Taylor, S., Sano, A., Picard, R.: Multimodal autoencoder: A deep learning approach to filling in missing sensor data and enabling better mood prediction. In: 2017 Seventh International Conference on Affective Computing and Intelligent Interaction (ACII). (2017) 202–208

[17] Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016) `http://www.deeplearningbook.org`.

[18] Kächele, M., Schels, M., Thiam, P., Schwenker, F.: Fusion mappings for multimodal affect recognition. In: 2015 IEEE Symposium Series on Computational Intelligence. (2015) 307–313

[19] Schwenker, F., Böck, R., Schels, M., Meudt, S., Siegert, I., Glodek, M., Kächele, M., Schmidt-Wack, M., Thiam, P., Wendemuth, A., Krell, G. In: Multimodal Affect Recognition in the Context of Human-Computer Interaction for Companion-Systems. Springer International Publishing, Cham (2017) 387–408

[20] Zhou, Y., Arpit, D., Nwogu, I., Govindaraju, V.: Is Joint Training Better for Deep Auto-Encoders? arXiv:1405.1380 [cs, stat] (2015) arXiv: 1405.1380.

[21] Clevert, D.A., Unterthiner, T., Hochreiter, S.: Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). arXiv:1511.07289 [cs] (2016) arXiv: 1511.07289.

[22] Zhang, Z., Sabuncu, M.: Generalized cross entropy loss for training deep neural networks with noisy labels. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R., eds.: Advances in Neural Information Processing Systems 31. Curran Associates, Inc. (2018) 8778–8788

[23] Kingma, D.P., Ba, J.: Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs] (2017) arXiv: 1412.6980.

[24] Duchi, J., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. J. Mach. Learn. Res. **12** (2011) 2121–2159

[25] Tieleman, T., Hinton, G.: Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning (2012)

[26] Erhan, D., Bengio, Y., Courville, A., Manzagol, P.A., Vincent, P., Bengio, S.: Why does unsupervised pre-training help deep learning? J. Mach. Learn. Res. **11** (2010) 625–660

[27] Bengio, Y.: Practical recommendations for gradient-based training of deep architectures (2012)

[28] Géron, A.: Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems. 2nd edn. O'Reilly Media, Inc. (2019)

[29] van der Maaten, L., Hinton, G.: Visualizing data using t-SNE. Journal of Machine Learning Research **9** (2008) 2579–2605

[30] Romero, A., Drozdzal, M., Erraqabi, A., Jégou, S., Bengio, Y.: Image Segmentation by Iterative Inference from Conditional Score Estimation. arXiv:1705.07450 [cs] (2017) arXiv: 1705.07450.

[31] LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86** (1998) 2278–2324

[32] Chollet, F., et al.: Keras. `https://keras.io` (2015)

[33] Oliphant, T.E.: Python for scientific computing. Computing in Science & Engineering **9** (2007) 10–20

[34] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015) Software available from tensorflow.org.

[35] Walt, S.v.d., Colbert, S.C., Varoquaux, G.: The numpy array: A structure for efficient numerical computation. Computing in Science & Engineering **13** (2011) 22–30

[36] Klaus Greff, Aaron Klein, Martin Chovanec, Frank Hutter, Jürgen Schmidhuber: The Sacred Infrastructure for Computational Research. In Katy Huff, David Lippa, Dillon Niederhut, Pacer, M., eds.: Proceedings of the 16th Python in Science Conference. (2017) 49 – 56

[37] McKinney, W.: Data structures for statistical computing in python. In van der Walt, S., Millman, J., eds.: Proceedings of the 9th Python in Science Conference. (2010) 51 – 56

[38] The HDF Group: Hierarchical Data Format, version 5 (1997-2019) http://www.hdfgroup.org/HDF5/.

[39] Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J., Grout, J., Corlay, S., Ivanov, P., Avila, D., Abdalla, S., Willing, C., development team, J.: Jupyter notebooks ? a publishing format for reproducible computational workflows. In Loizides, F., Scmidt, B., eds.: Positioning and Power in Academic Publishing: Players, Agents and Agendas, IOS Press (2016) 87–90

[40] Pérez, F., Granger, B.E.: Ipython: A system for interactive scientific computing. Computing in Science & Engineering **9** (2007) 21–29

[41] Hunter, J.D.: Matplotlib: A 2d graphics environment. Computing in Science & Engineering **9** (2007) 90–95

[42] Busche, R., Meyer zu Driehausen, F., Rebstadt, J.: Incense (2018)

[43] Chan, D.M., Rao, R., Huang, F., Canny, J.F.: t-SNE-CUDA: GPU-Accelerated t-SNE and its Applications to Modern Data. arXiv:1807.11824 [cs, stat] (2018) arXiv: 1807.11824.

[44] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in python. J. Mach. Learn. Res. **12** (2011) 2825–2830

[45] Leng, B., Guo, S., Zhang, X., Xiong, Z.: 3d object retrieval with stacked local convolutional autoencoder. Signal Processing **112** (2015) 119 – 128 Signal Processing and Learning Methods for 3D Semantic Analysis.

[46] Makhzani, A., Shlens, J., Jaitly, N., Goodfellow, I., Frey, B.: Adversarial autoencoders (2015)

A Code snippets

In this appendix are some important Code snippets.

A.1 SP-noise

```
def salt_and_pepper(images,prob):
        images = images
                +10*np.random.randint(-1,2, size =images.shape)
                *np.random.choice(2, images.shape, p=[1-prob, prob])
        images = np.clip(images, -1., 1.) //Mantain max and min values
return(images)
```

Note that the multiplication by 10 is to saturate the value to either > 1 or < -1 and then trough the np.clip the value is set to 1 or -1 respectively.

A.2 Over_Dim

```
def deep_autoencoder_generator_over_dim():
        act = tf.keras.activations.elu
        encoded_dim = 36

        stacked_encoder = Sequential([
                layers.Dense(1024, input_shape=(784,), activation=act,
                        name='encode-1'),
                layers.Dense(529, activation=act, name='encode-2'),
                layers.Dense(256, activation=act, name='encode-3'),
                layers.Dense(encoded_dim, activation=act,
                        name='encoded-features-4'), ], name='encoder')

        stacked_decoder = Sequential([
                layers.Dense(256, activation=act, name='decode-3'),
                layers.Dense(529, activation=act, name='decode-2'),
                layers.Dense(1024, activation=act, name='decode-1'),
                layers.Dense(784, activation='tanh', name='Output')],
                        name='decoder')
```

```
19
20          deep_autoencoder = Sequential([stacked_encoder, stacked_decoder],
21                  name='deep_autoencoder')
22
23          return deep_autoencoder, stacked_encoder, encoded_dim
```

A.3 Over_Dim_tied

with the Transpose layer as defined in A.4

```
1  def deep_autoencoder_generator_over_dim_tied():
2          act = tf.keras.activations.elu
3          encoded_dim = 36
4
5          dense_1 = layers.Dense(1024, activation=act, name='encode-1')
6          dense_2 = layers.Dense(529, activation=act, name='encode-2')
7          dense_3 = layers.Dense(256, activation=act, name='encode-3')
8          dense_4 = layers.Dense(encoded_dim, activation=act,
9                  name='encoded-features-4')
10
11         tied_encoder = Sequential([keras.Input(shape=(784,)),
12                 dense_1, dense_2, dense_3, dense_4, ], name='encoder')
13
14         tied_decoder = Sequential([
15                 DenseTranspose(dense_4, activation=act),
16                 DenseTranspose(dense_3, activation=act),
17                 DenseTranspose(dense_2, activation=act),
18                 DenseTranspose(dense_1, activation=act)], name='decoder')
19
20         tied_autoencoder = Sequential([tied_encoder, tied_decoder],
21                 name='deep_autoencoder')
22
23         return tied_autoencoder, tied_encoder, encoded_dim
```

A.4 Transpose layer

As defined in Chapter 17 of Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition [28]:

```
1  class DenseTranspose(keras.layers.Layer):  # Hands-on ml2 chapter 17
2          def __init__(self, dense, activation=None, **kwargs):
3                  self.dense = dense
```

```
                self.activation = keras.activations.get(activation)
                super().__init__(**kwargs)

        def build(self, batch_input_shape):
                self.biases = self.add_weight(name="bias",
                        shape=[self.dense.input_shape[-1]],
                        initializer="zeros")
                super().build(batch_input_shape)

        def call(self, inputs):
                z = tf.matmul(inputs, self.dense.weights[0],
                        transpose_b=True)
                return self.activation(z + self.biases)
```

A.5 Normal_Dim

```
def deep_autoencoder_generator_normal_dim():
        act = tf.keras.activations.elu
        encoded_dim = 30

        stacked_encoder = Sequential(
                [
                layers.Dense(100, input_shape=(784,), activation=act,
                        name='encode-1'),
                layers.Dense(encoded_dim, activation=act,
                        name='encoded-features-2'),
                ], name='encoder')

        stacked_decoder = Sequential([
                layers.Dense(100, activation=act, name='decode-'),
                layers.Dense(784, activation='tanh', name='Output')
                ], name='decoder')

        deep_autoencoder = Sequential([stacked_encoder, stacked_decoder],
                name='deep_autoencoder')

        return deep_autoencoder, stacked_encoder, encoded_dim
```

A.6 Normal_Dim_tied

with the Transpose layer as defined in A.4

```
def deep_autoencoder_generator_normal_dim_tied():
        act = tf.keras.activations.elu
        encoded_dim = 30

        dense_1 = layers.Dense(100, activation=act, name='encode-1')
        dense_2 = layers.Dense(encoded_dim, activation=act,
                name='encoded-features-2')

        tied_encoder = Sequential([
                        keras.Input(shape=(784,)), dense_1, dense_2
                ], name='encoder')

        tied_decoder = Sequential([
                DenseTranspose(dense_2, activation=act),
                DenseTranspose(dense_1, activation=act)], name='decoder')

        tied_autoencoder = Sequential([tied_encoder, tied_decoder],
                name='deep_autoencoder')

        return tied_autoencoder, tied_encoder, encoded_dim
```

A.7 Classifier

```
def generate_classifiers(encoded_dim):
        act = tf.keras.activations.relu
        soft = tf.keras.activations.softmax

        feature_classifier = Sequential([
                layers.Dense(64, input_shape=(encoded_dim,), activation='relu'),
                layers.Dense(32, activation=act),
                layers.Dense(10, activation=soft)], name='encoded_classifier')

return feature_classifier
```

B Images

B.1 Manifold

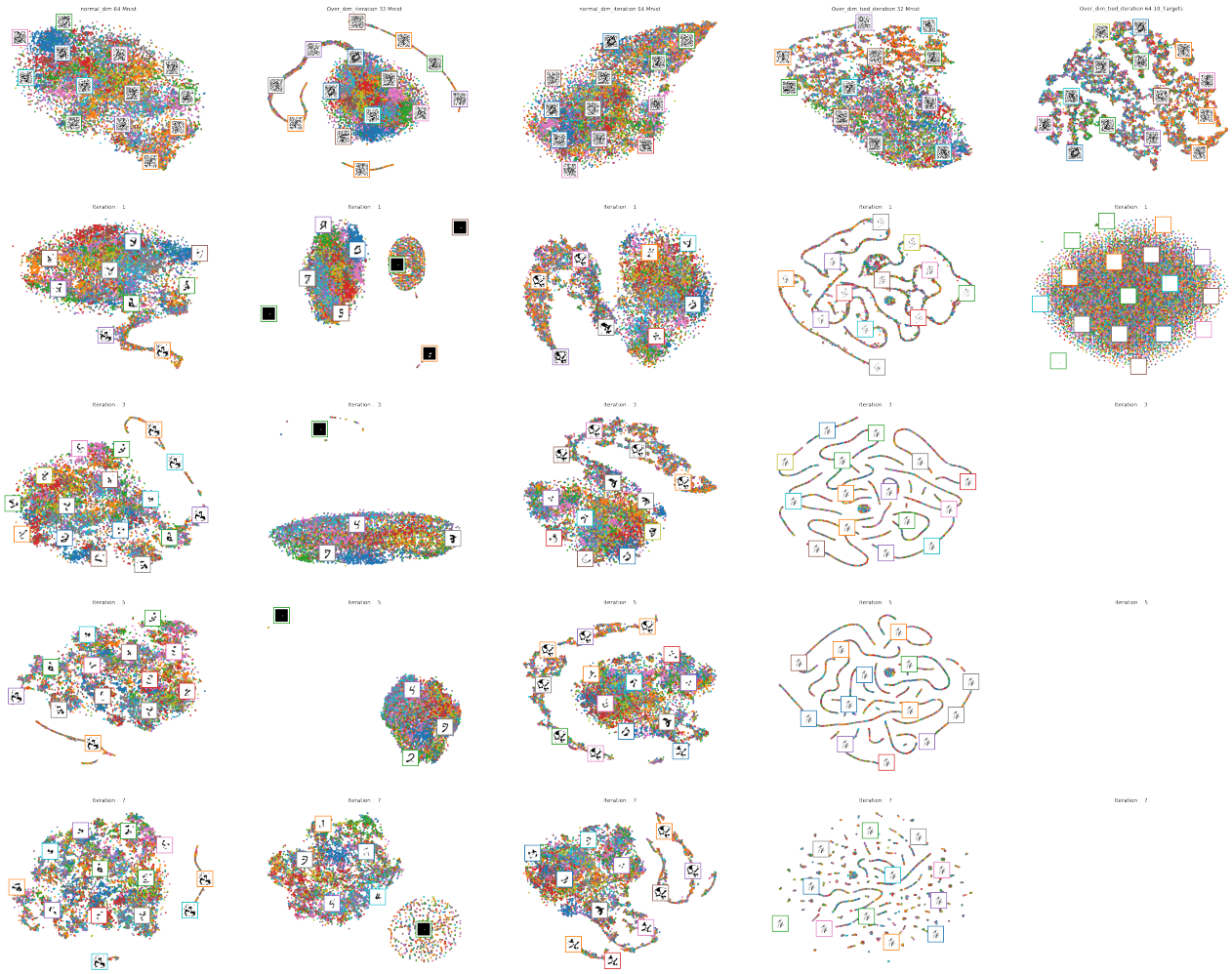

Figure B.1: Selected manifold visualisation, with t-sne, for $SP(MNIST, 0.9)$. First row: t-sne of $enc\_pred\_1 = encoder.predict(SP(MNIST, 0.9))$. Second row: t-sne of $enc\_pred\_2 = encoder.predict(enc\_pred\_1)$. Third row: t-sne of $enc\_pred\_4 = encoder.predict(enc\_pred\_3)$. Fourth row:t-sne of $enc\_pred\_7 = encoder.predict(enc\_pred\_6)$.

B.2 Reconstruction

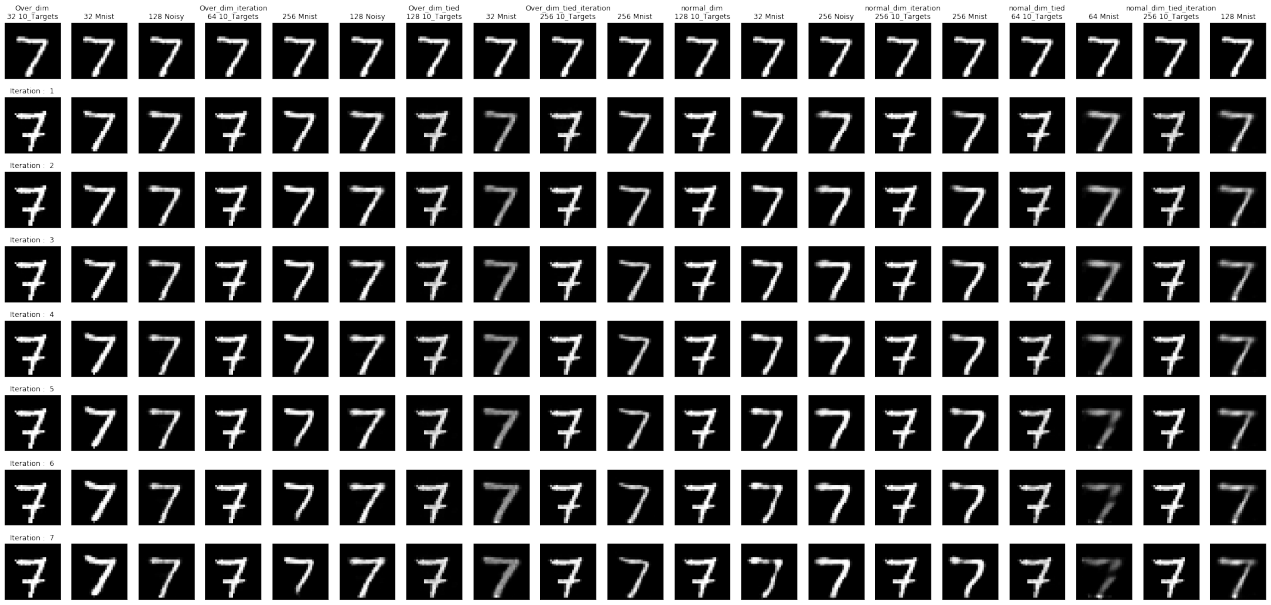

Figure B.2: Prediction of the best network, for each architecture, batch_size and targets combination. First row: Input images for $SP(Datat, 0)$, subsequent rows are the iterative reconstruction using the, output, of the above row as input

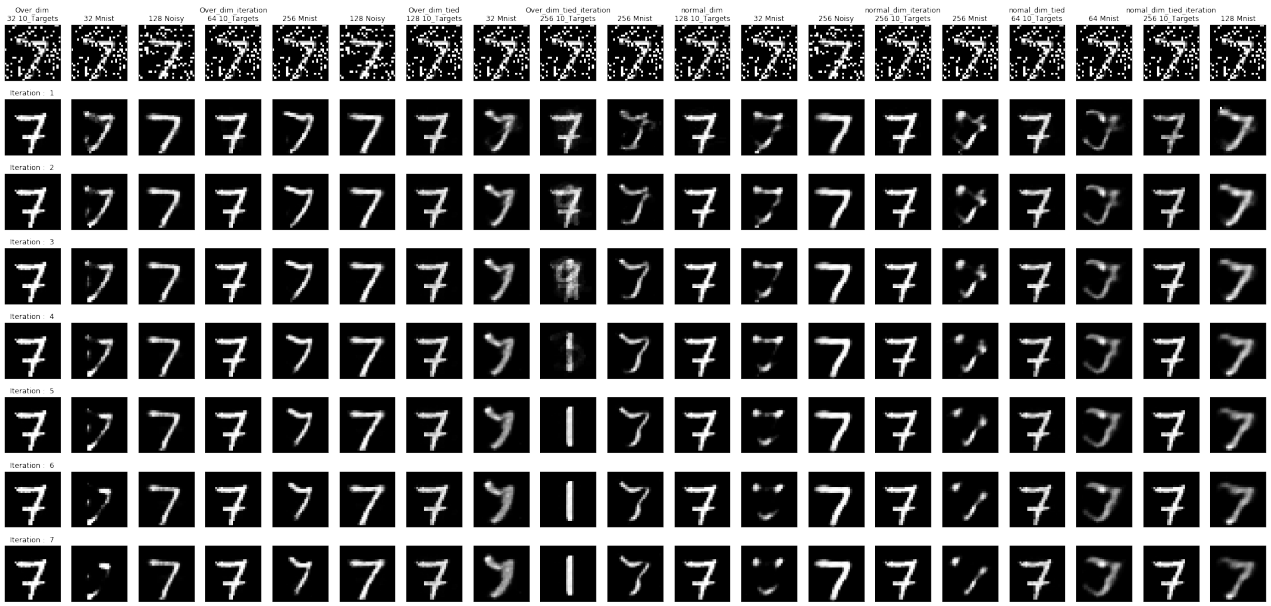

Figure B.3: Prediction of the best network, for each architecture, batch_size and targets combination. First row: Input images for $SP(Datat, 0.4)$, subsequent rows are the iterative reconstruction using the, output, of the above row as input

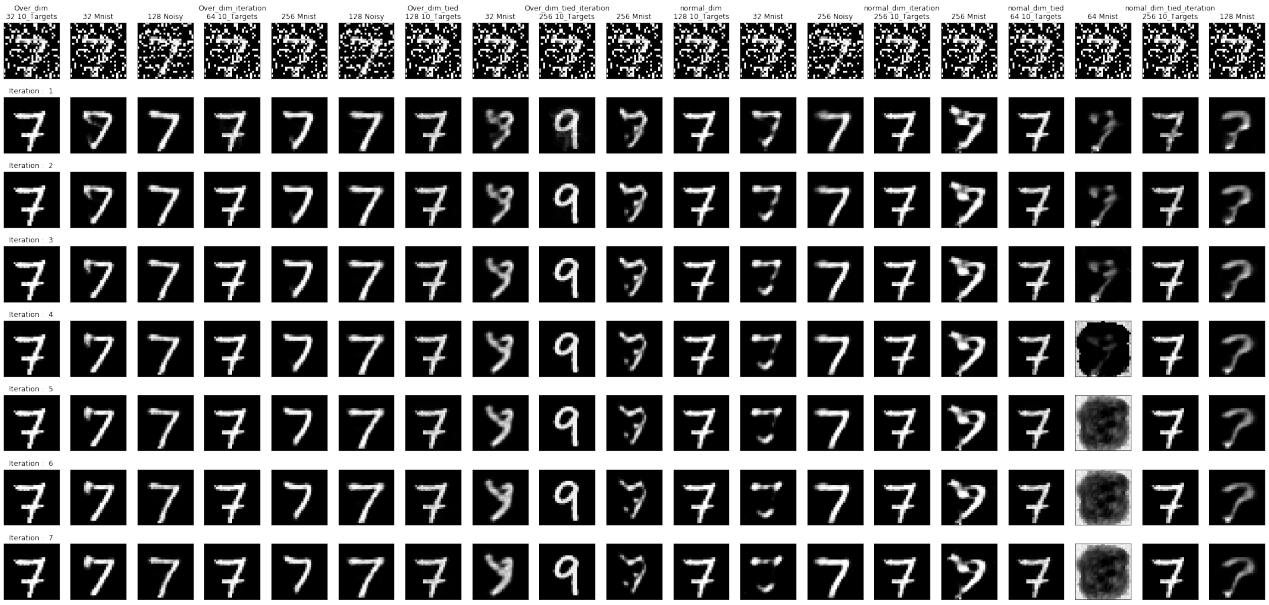

Figure B.4: Prediction of the best network, for each architecture, batch_size and targets combination. First row: Input images for $SP(Datat, 0.7)$, subsequent rows are the iterative reconstruction using the, output, of the above row as input

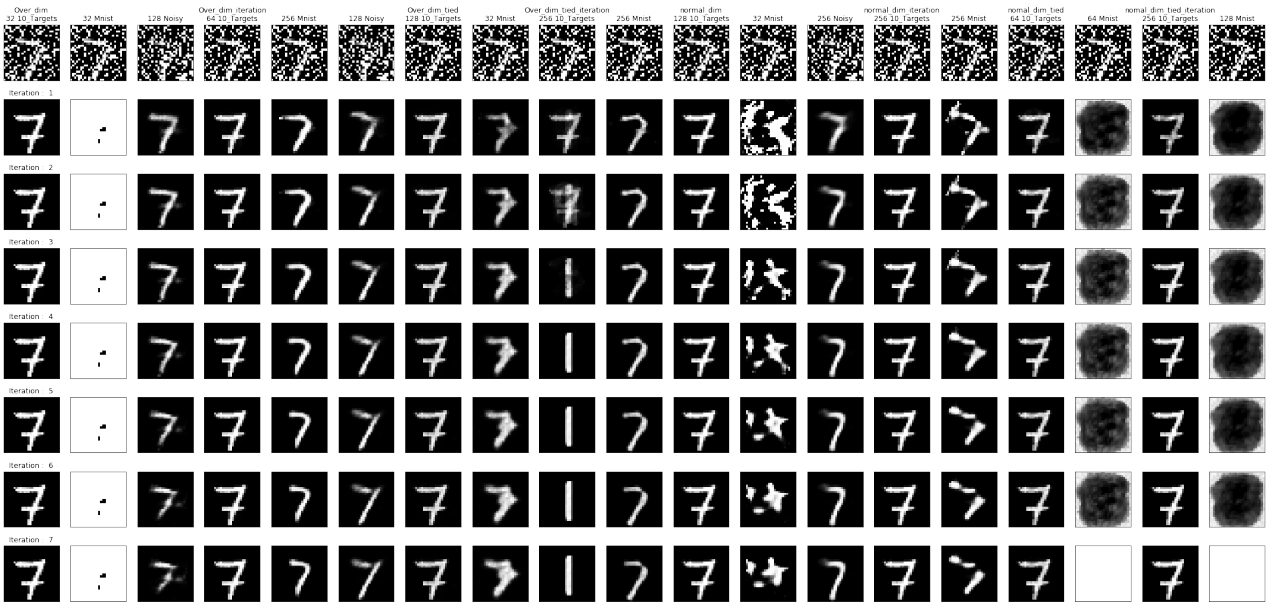

Figure B.5: Prediction of the best network, for each architecture, batch_size and targets combination. First row: Input images for $SP(Datat, 0.8)$, subsequent rows are the iterative reconstruction using the, output, of the above row as input

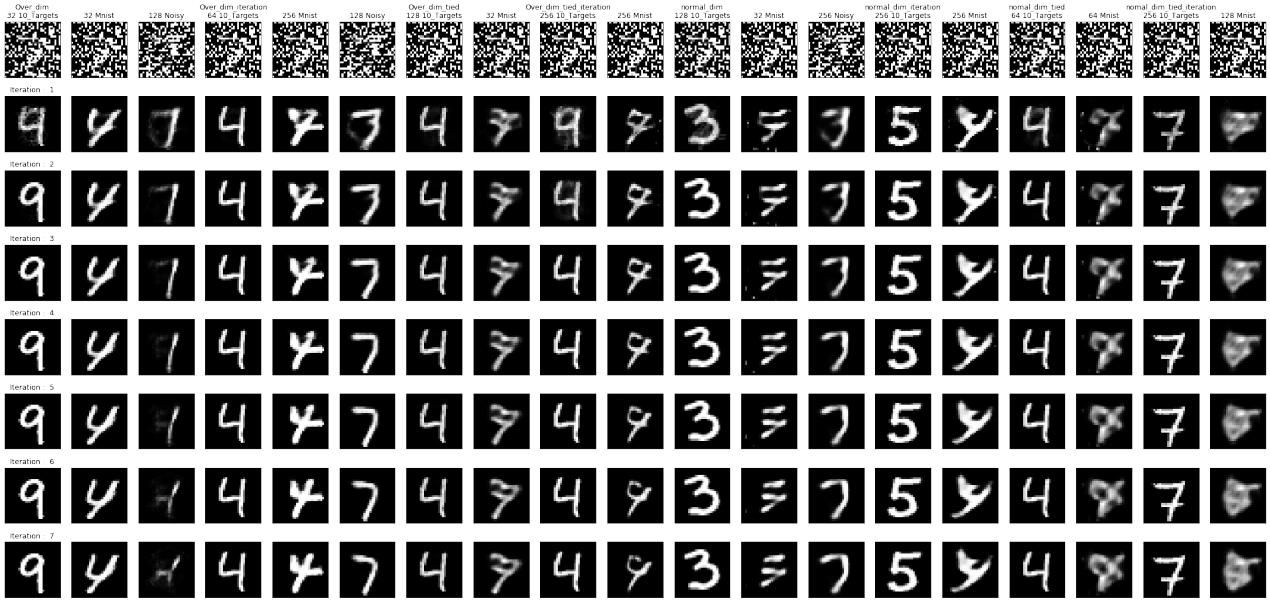

Figure B.6: Prediction of the best network, for each architecture, batch_size and targets combination. First row: Input images for $SP(Datat, 1)$, subsequent rows are the iterative reconstruction using the, output, of the above row as input

List of Figures

List of Tables

Name: Manuel Lautaro Hickmann Matriculation number:: 953591

Honesty disclaimer

I hereby affirm that I wrote this thesis independently and that I did not use any other sources or tools than the ones specified.

Ulm, ...

Manuel Lautaro Hickmann