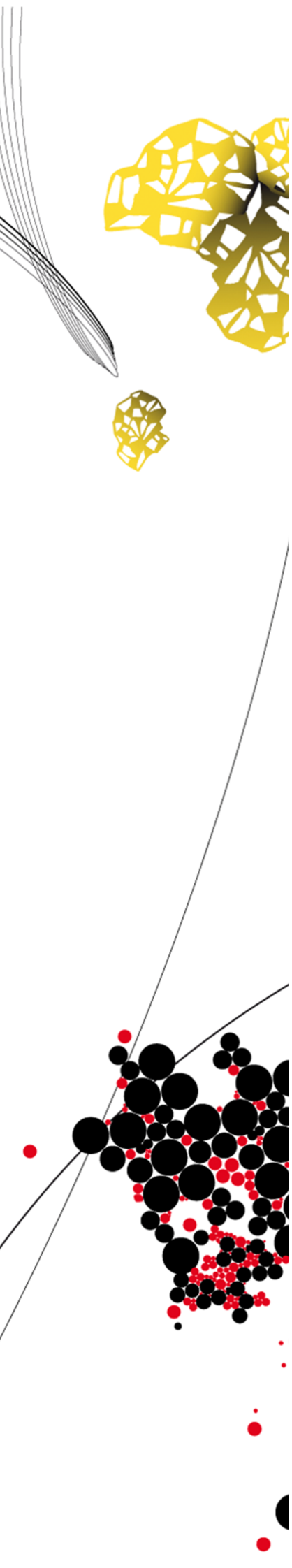

UNIVERSITY OF TWENTE.

**Faculty of Electrical Engineering,
Mathematics & Computer Science**

Autoencoder-based cleaning of non-categorical data in probabilistic databases

F.P.J. Nijweide

B.Sc. Thesis
August 2020

Supervisors:
dr.ir. M. van Keulen
N. Bouali M.Sc.
dr. H.K. Hemmes

Data Management & Biometrics Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

This report investigates the use of autoencoders to remove noise from non-categorical data in probabilistic databases. Previous research has shown that this is possible for categorical data, but a new solution is needed to do this for continuous or discrete distributions. The approach chosen was to approximate the data using discrete sampling. After training the autoencoder, we measured the difference between "cleaned" data and the original data using the Jensen-Shannon divergence. We concluded that the most effective solution was to use semi-supervised learning. This solution is quite effective at low sampling densities, reducing 99.54% of noise in a probabilistic database, while its performance at higher sampling densities is slightly lower, leading to an 86.99% reduction in the amount of noise.

Keywords— Autoencoder, probabilistic databases, machine learning, neural networks, data integration, data science, data cleaning, data cleansing

Contents

1 Introduction

1.1 Motivation & Goals

Data collected from real-world activities (such as experiments or sensor input) contains uncertainty and noise, which is often undesirable. Taking larger sample sizes could lead to a reduction in noise, but the resulting accuracy may not be sufficient. Probabilistic databases (PDB) allow for the storing of such data.

Removing noise from a PDB requires techniques like data integration [1], which involves manual work by domain experts. Automating this (using techniques such as machine learning) would reduce the cost of maintaining PDBs while improving their efficacy.

There is very little research on using machine learning for cleaning PDB data, but existing research shows that denoising autoencoders are effective at improving the accuracy of categorical data in PDBs [2]. However, research is lacking on how to apply this to cleaning non-categorically distributed data, such as numerical data. Furthermore, other neural network types might also prove useful for this purpose, but there is currently no research on this.

Thus, this research aims to identify methods for using autoencoders in cleaning non-categorical data in probabilistic databases and identifying which neural network structures are effective at meeting this goal.

1.2 Research questions

- How can autoencoders be used for cleaning non-categorical probabilistic data?
  - Which models of non-categorical PDB data are compatible as input into a neural network?
  - What autoencoder structures perform best for cleaning non-categorical PDB data?
  - What network hyperparameters led to the best results?
  - How does the solution perform on different types of data and different distributions?
  - How can other neural network techniques (as described in section 2.2.7) improve the results of this approach?

2 Literature review

2.1 Probabilistic databases

Various models for probabilistic databases exist. For this proposal, we chose to focus on a relational database model.

Figure 1 shows that categorical data in a PDB can be represented by a set of probabilities (virtual evidence) for each attribute, with the probabilities of the values for each attribute summing to 1 [3]. Extensions to the discrete categorical distribution model exist, allowing for the use of probabilities with a continuous distribution [4]. An example of an attribute in a PDB could be a simple property such as ”color of the car”. With the value of the property being ”red” or ”blue”. This is how categorical PDB functions. For a ”continuous” PDB, we choose to use a similar system, where each possible value has a different category. Categorization is explained further in section 2.4 and 3.2.

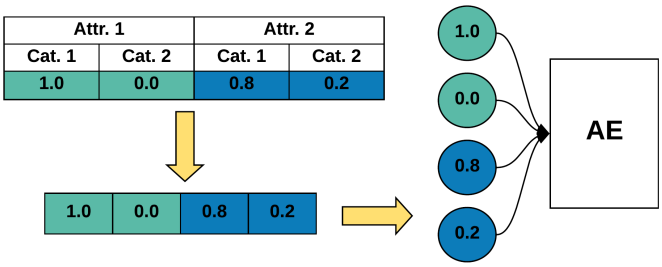

Figure 1: Example of categorical data in a PDB, and how it could be used as an input for an autoencoder [2]

A Probabilistic Inference Bayesian Network (PIBN) [5] can remove noise from a PDBs, as explained in section 2.4. Other conventional methods for PDB data cleansing focus on outlier detection, removal of duplicates, and the use of statistical [6] or rule-based techniques [7], and conditional functional dependencies [8].

Several implementations exist for representing uncertain data. Examples include IMPrECISE [9], for representing semi-structured data in XML, and the PostgreSQL-based mayBMS [10], for relational data. It is also possible to simulate a PDB in a standard database by ensuring all operations adhere to the following constraints:

A PDB with C_{total} attributes have N rows, defined by:

$$N = \sum_{c=1}^{C_{total}} n_c \tag{1}$$

Where n_c is the amount of categories for attribute c. Then care must be taken to make sure all entries are between 0 and 1, and all rows for attribute c sum to 1:

$$\sum_{j=\left(\sum_{k=1}^{k=c-1} n_k\right)+1}^{\sum_{k=1}^{k=c} n_k} PDB_{i,j} = 1 \tag{2}$$

We used the above approach for this paper, as we often stored data in non-probabilistic formats such as Pandas DataFrames, NumPy arrays, and TensorFlow tensors.

2.2 Neural networks

2.2.1 Theoretical background

A neural network is a system that is well-suited to learning how to construct the desired output from an input. Modern neural networks often consist of stacked layers of artificial neurons (although sometimes layers are not used [12]). The mechanisms of artificial neurons take inspiration from theories from neuroscience [13]. A neuron creates a new output by applying an activation function to a weighted sum of the outputs of neurons from previous layers. By repeating this process many times, a neural network can solve highly complex and non-linear problems.

The calculation to compute the output of a neuron $x_j^{(i)}$, the j^{th} neuron in layer i, is as follows:

$$x_j^{(i)} = \phi\left(\sum_k \left(w_{k,j}^{(i-1)} x_k^{(i-1)}\right) + b^{(i-1)}\right) \tag{3}$$

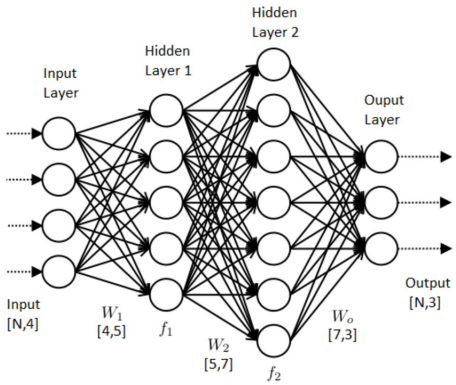

Figure 2: An example of a neural network [11]

Where $w_{k,j}^{(i-1)}$ is the weight of the connection from neuron k in layer $i-1$ to neuron j in layer i, and $b^{(i-1)}$ is the bias neuron of layer $i-1$, a neuron that always outputs 1. Furthermore, ϕ is the activation function of the neuron. It is possible to adapt this equation to use vectors:

$$x_j^{(i)} = \phi\left(\begin{bmatrix} w_{0,j}^{(i-1)} & \cdots & w_{n_{i-1},j}^{(i-1)} \end{bmatrix} \cdot \begin{bmatrix} x_0^{(i-1)} \\ \vdots \\ x_{n_{i-1}}^{(i-1)} \end{bmatrix}\right) \tag{4}$$

Where $x_0^{(i-1)}$ is the bias neuron of layer $i-1$, and n_i is the amount of non-bias neurons in layer i. By combining all the weight vectors into a single matrix, we get the output neurons of layer i as a vector:

$$\begin{bmatrix} x_1^{(i)} \\ \vdots \\ x_{n_i}^{(i)} \end{bmatrix} = \phi\left(\begin{bmatrix} w_{0,1}^{(i-1)} & \cdots & w_{n_{i-1},1}^{(i-1)} \\ \vdots & \ddots & \vdots \\ w_{0,n_i}^{(i-1)} & \cdots & w_{n_{i-1},n_i}^{(i-1)} \end{bmatrix} \cdot \begin{bmatrix} x_0^{(i-1)} \\ \vdots \\ x_{n_{i-1}}^{(i-1)} \end{bmatrix}\right) \tag{5}$$

In the compact form:

$$x^{(i)} = \phi(W^{(i-1)} \cdot x^{(i-1)}) \tag{6}$$

Since the design of modern computer components (especially the GPU) allows for high-speed matrix multiplications, calculating the output of a small neural network only takes a few cycles. At the speed of modern components, this is between 10^{-6} and 10^{-3} seconds.

Choosing the right hyperparameters, network topology, activations functions, and loss function is essential for creating an efficient network. However, most networks must *learn* to solve the tasks they were made for, although some neural networks exist that do not require training [12]. This happens by adapting their weights using algorithms such as backpropagation and stochastic gradient descent, to improve the quality of their output [14].

2.2.2 Activation functions

Each neuron may use one of many different activation functions.

Traditionally [13], the sigmoid function was used most:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{7}$$

However, the ReLU (rectified linear unit) function is commonly used now due to its quick learning capabilities in deep networks [14].

$$ReLU\,(x) = max(0, x) \tag{8}$$

Swish, an activation function recently developed at Google [15], seems to perform better than ReLU in many cases.

$$Swish\,(x) = x \cdot \sigma(x) \tag{9}$$

Another activation function that could prove beneficial for probabilistic data cleaning is the softmax function, which makes the output of a layer adhere to the constraints of a probabilistic distribution. For some i for which $1 \leq i \leq n$, where n is the number of neurons in the layer:

$$softmax\,(x)_i = e^{x_i} \sum_{j=1}^{n} e^{x_j} \tag{10}$$

Where x is a vector containing the outputs x_1 up to x_n of the neurons of that layer, without an activation function applied to them yet.

Most other activation functions implement well-known mathematical functions, such as cos, sin, linear, tanh, or step functions.

Different activations functions are well-suited to different tasks. Linear classification problems can use linear or semi-linear functions such as ReLU, while more complex problems might require trigonometric functions. Regression problems are best solved using an output layer with only one neuron, and a linear activation function. Binomial classification problems also require one output neuron, with a sigmoid or step activation function. For multinomial classification problems, an output layer with a softmax activation function is the best solution, resulting in proper one-hot encoding.

2.2.3 Loss functions

The goal of training the network is to minimize the distance between the output values and the intended values.

This distance is defined using a loss function. One of the most popular loss functions is the Euclidean distance, or the mean square error (MSE):

$$MSE(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\|^2 \tag{11}$$

$$MSE\,(\mathbf{x}, \mathbf{y})_i = (\mathbf{x}_i - \mathbf{y}_i)^2 \tag{12}$$

Where $\mathbf{x}$ is the network's output (the prediction), while $\mathbf{y}$ is the target output.

Instead of using the square of the error, one could also use the absolute error. However, this approach is not suitable for this paper, as we want to prioritize correcting larger errors (one error of 0.5 should be more important than five errors of 0.1).

While MSE is very useful for problems such as regression, which do not have probabilistic distributions as output, other functions might prove to be more useful for probabilistic data. For example, binary cross-entropy (BCE) is useful for binomial classification problems:

$$BCE\,(x, y) = y \cdot -\log(x) + (1 - y) \cdot -log(1 - x) \tag{13}$$

Furthermore, multinomial classification problems make use of categorical cross-entropy (CCE).

$$CCE\,(\mathbf{x}, \mathbf{y}) = -\sum_i \mathbf{y}_i \cdot log(\mathbf{x}_i) \tag{14}$$

Another way to measure the difference between probabilistic distributions is the Kullback–Leibler divergence (KLD) [16]:

$$KLD\left(\mathbf{x},\mathbf{y}\right)=\sum_{i}\mathbf{y}_{i}\cdot log(\frac{\mathbf{y}_{i}}{\mathbf{x}_{i}}) \tag{15}$$

KLD might be an interesting metric to use as a loss function, but it is not symmetric, as $KLD\left(\mathbf{x},\mathbf{y}\right)\neq KLD\left(\mathbf{y},\mathbf{x}\right)$. The Jensen-Shannon divergence (JSD) [16] might be an adequate, symmetric replacement:

$$JSD(\mathbf{x},\mathbf{y})\ =\frac{KLD(\mathbf{x},\mathbf{m})}{2}\ +\frac{KLD(\mathbf{y},\mathbf{m})}{2} \tag{16}$$

Where $\mathbf{m}=\frac{\mathbf{x}+\mathbf{y}}{2}$.

2.2.4 Backpropagation

The best value for the weights is found using stochastic gradient descent. This involves finding the ”direction” the weight vector should be moved in to decrease the loss of the output, by finding the derivative (gradient) of the loss function in terms of the weights. Small values are then added to the weights along the gradient, in an attempt to minimize the loss value. This loss can be found by backpropagating the error:

$$\frac{\delta Loss(\mathbf{x}_{j}^{(i)},\mathbf{y}_{j}^{(i)})}{\delta w_{k,j}^{(i-1)}}=\frac{\delta Loss(\mathbf{x}_{j}^{(i)},\mathbf{y}_{j}^{(i)})}{\delta\mathbf{x}_{j}^{(i)}}\cdot\frac{\delta\mathbf{x}_{j}^{(i)}}{\delta\phi\left(w_{k,j}^{(i-1)}x_{k}^{(i-1)}\right)}\cdot\frac{\delta\phi\left(w_{k,j}^{(i-1)}x_{k}^{(i-1)}\right)}{\delta w_{k,j}^{(i-1)}x_{k}^{(i-1)}}\cdot\frac{\delta w_{k,j}^{(i-1)}x_{k}^{(i-1)}}{\delta w_{k,j}^{(i-1)}} \tag{17}$$

In the case where our loss function is MSE, and the activation function is the sigmoid function $\sigma(x)$, of which the derivative is $\sigma\left(x\right)\left(1-\sigma\left(x\right)\right)$:

$$\frac{\delta Loss(\mathbf{x}_{j}^{(i)},\mathbf{y}_{j}^{(i)})}{\delta w_{k,j}^{(i-1)}}=2\mathbf{x}_{j}^{(i)}\cdot1\cdot\mathbf{x}_{j}^{(i)}(1-\mathbf{x}_{j}^{(i)})\cdot x_{k}^{(i-1)} \tag{18}$$

Thus, when the activations functions are easily differentiable, it is easy to calculate the updated weights. Just like with forward propagation, matrix multiplication can result in speed increases [17].

2.2.5 Hyperparameters and sparsity constraints

Many different parameters influence the weight updates. For example, the step size η determines how large the update to the weight should be for each training epoch, while the decay rate α determines how much the old weight update will contribute to the new weight:

$$\mathbf{w}_{new}=\mathbf{w}\ -\ \eta\nabla_{\mathbf{w}}\mathbf{Loss}+\ \alpha\Delta\mathbf{w} \tag{19}$$

Where $\Delta\mathbf{w}$ is the previous update of the weight, by making α large, the weight updates gain ”momentum” [18], becoming less prone to oscillations. Like a ball rolling down a hill, the weight following the gradient will oscillate less if the weigh has more momentum. If η is too large, the weight might overshoot and skip an optimal value. If it is too low, the training time will increase.

The optimizer ”Adam” calculates first and second-order moments of the gradients to adapt the learning rate. This has a better performance than stochastic gradient descent [19].

2.2.6 Convolutional layers

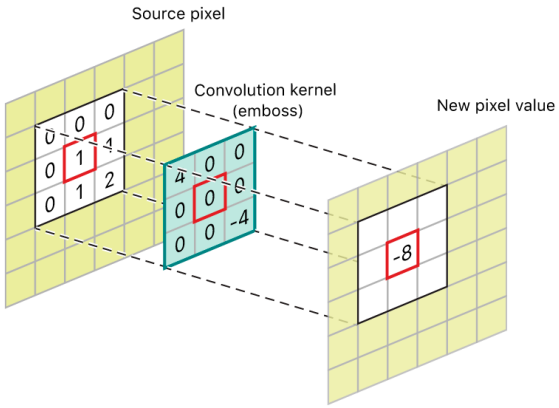

Figure 3: An example of a convolution matrix (sometimes called a kernel) being applied to an image [20]

Because of the combination of continuous variables and continuous probability distributions for those variables seen as a large set of pixels, image processing techniques may prove useful for this research. An example of machine learning in image processing is a Convolutional Neural Network (CNN). CNNs use techniques such as convolutional layers (which perform symmetric convolutions between a kernel and every pixel of the previous layer, of which an example is given in equation 20), and pooling layers (which reduce the size of the input layer while also being able to combine features from different convolution layers [14]). These features can also be used in autoencoders [21].

$$\left(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \right)_{2,2}$$

$$= (i \cdot 1) + (h \cdot 2) + (g \cdot 3) + (f \cdot 4) + (e \cdot 5) + (d \cdot 6) + (c \cdot 7) + (b \cdot 8) + (a \cdot 9) \tag{20}$$

Such an approach is useful to detect features in images. It might also be useful for detecting features in probability distributions.

2.2.7 Other layers

So far, the methods covered are related to neural networks with simple, densely connected layers. However, there are many more types of layers that make up a network. For example, preprocessing of data might be needed to identify specific features, such as taking the logarithm of a highly skewed dataset. This is achieved by adding a layer that applies such a function.

A more complex example of such a function is the radial basis function (also called a Gaussian kernel). Instead of classifying an input based on the distance from a line or polynomial, the network can learn to classify based on the squared Euclidean (radial) distance from certain "landmarks". Such an approach allows the network to learn much higher-dimensional features than simple dense layers and polynomial functions. Equation 21 describes the output of an RBF kernel for two vectors [22].

$$K(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\sigma^2} \right) \tag{21}$$

Where σ is a scaling factor.

Other interesting layers include blackout and gaussian blackout layers, which randomly set weights to 0 during training (a sparsity constraint). Such a constraint encourages the network to learn useful features while keeping fewer neurons active, increasing the likelihood of the network learning useful features. It also reduces the likelihood of overfitting, where a network achieves high performance on its training set, but a low performance on testing datasets. An analogy for this would be a high-degree polynomial appearing to be a good fit for the line $y = x$ at small values (the training set), but being inaccurate at high values (the testing dataset). A better fit would be a low-degree polynomial. Other sparsity constraints include L1 and L2 regularization, where the absolute values or the squares of weights are added to the loss function. Layers that add Gaussian noise to the data (only during training) are sometimes used to combat overfitting, like in denoising autoencoders.

2.3 Autoencoders

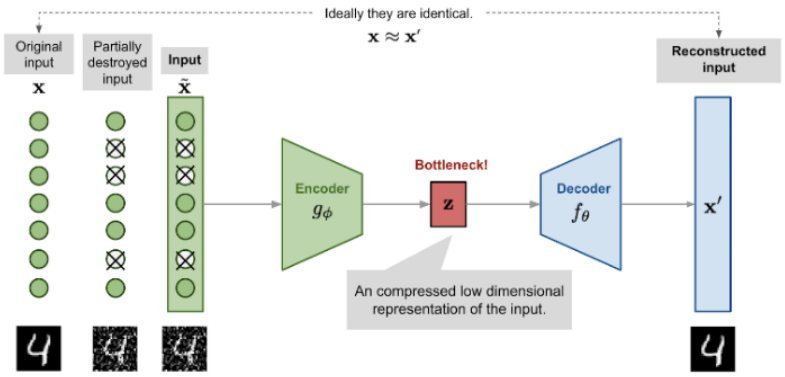

Figure 4: An example of how a denoising autoencoder can remove artificial noise from data [23]

An autoencoder is a neural network trained to minimize a loss function between input and output to learn efficient encodings of data. As this does not require labeled inputs, autoencoders are compatible with unsupervised learning [24].

Other uses for autoencoders include anomaly detection, where the input is determined to be an anomaly if the network is unable to reconstruct the input [25].

There is a risk of the autoencoder just learning the identity function, a case of extreme overfitting. Several types of autoencoders exist with designs that mitigate this problem. These types of autoencoders are not mutually exclusive, and they may be combined [26].

- An undercomplete autoencoder has a hidden, middle layer (called the feature space or code layer) with a lower dimensionality than the input or output spaces. Thus, the autoencoder must learn a compressed representation of the input data and how to decompress it. This compressed representation potentially captures only the essential features of the input and disregards noise.

- A denoising autoencoder has noise added to the input data before being fed to the network. The autoencoder then has to learn how to remove this noise because the loss function still uses the original, clean input. Preventing overfitting and has an added benefit of noise reduction [24].

- A sparse autoencoder adds a sparsity penalty to the training criterion. The autoencoder is now also penalized on the number of active neurons in the code (middle) layer. This constraint encourages the autoencoder to retain a more meaningful representation of the data in the code layer.

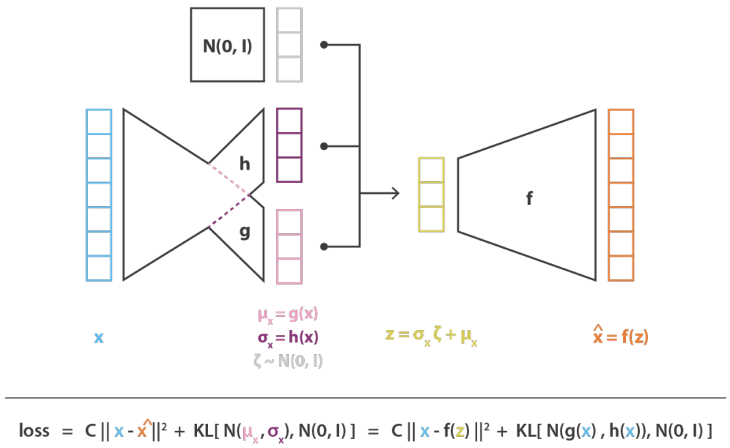

Figure 5: An example of a variational autoencoder [27]

- A variational autoencoder (VAE) has the encoder section output a tensor of means and a tensor of standard deviations (instead of deterministic variables as is usually the case). The KL-divergence between this distribution and a standard normal distribution is added to the loss function. A sample is taken from this distribution and fed to the decoder section [28]. The VAE ensures that the latent space representation learned is continuous (meaning that neighboring data points should lead to similar outputs) and complete (all inputs should lead to a sensible output). Thus, the basis learned in the latent space should be orthogonal, like in principal component analysis (ideally, the perfect autoencoder would perform a non-linear multilayer PCA).

  A latent space that adheres to these constraints is usable for generative purposes; data points are fed to the decoder data points are read by the decoder section, which will produce a sensible output. An example of a VAE is shown in Figure 5.

2.4 Related work: autoencoders for probabilistic data

In previous research [2], the data in a PDB (D_{PDB}) can is modeled as a corrupted representation of a ground truth (D_{GT}), which is sampled from an underlying distribution $P(D_{GT})$. $P(D_{GT})$ can be modeled as a Bayesian network [5], with relations such as the probability of a car observed as having a green paint, P(OG), being dependent on the probability of the car being green, P(G). D_{GT} is a deterministic sample drawn from this distribution, with values such as P(G)=1 and P(OG)=0. D_{PDB} then adds noise to these values or removes them altogether, resulting in values such as P(G)=0.5 and P(OG) = 0.1. Using a Probabilistic Inference Bayesian Network (PIBN) [5] based on $P(D_{GT})$, the values are recalculated, and the noise is removed. However, because this approach requires supervised learning, which requires D_{GT} and $P(D_{GT})$, since, in most cases of PDBs, the values of D_{GT} and $P(D_{GT})$ are unknown, it cannot be applied to real-world PDBs.

An alternative is to use autoencoders. Previous research [2, 16] has shown that the data in a PDB, $D_{PDB,}$ is suitable for training an autoencoder. Autoencoders do not require

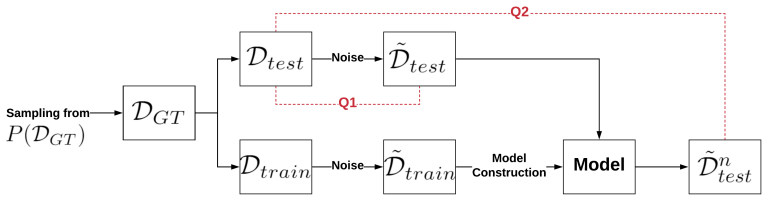

Figure 6: A schematic representation of how to train an autoencoder to clean categorical PDB data [2]

access to D_{GT} or $P(D_{GT})$, and the resulting network is still able to reduce the differences between D_{PDB} and D_{GT} in a controlled setting.

In such a controlled setting, D_{GT} is corrupted to produce a D_{PDB}, and split into training and testing datasets, as shown in Figure 6. After training, the output of the testing data can be compared to D_{GT} to verify the accuracy of this approach. In a real-world setting, the database operator would not have access to D_{GT}, and just train/test on D_{PDB} directly without being able to verify the accuracy of the results.

Categorical data lends itself well to this approach, as a database entry can easily be concatenated and used as input for a neural network (see Figure 1), it is not possible for numerical data with a continuous distribution. This approach can be modified to work for numerical data. A possible solution explored in later sections involves constructing a histogram with N bars, effectively changing the attribute into a categorical distribution with N categories.

3 Methodology & Approach

3.1 Instruments

For the experiments, we chose to use the Python programming language and several related libraries, because they are open-source and well-established tools for data science. These include:

- NumPy & SciPy for useful data structures and statistical methods.
- pyAgrum for modeling Bayesian networks.
- TensorFlow and Keras for their useful machine learning tools. Unless mentioned explicitly, the default hyperparameters are used.
- Pandas for representing and operating on databases.
- Plotly for plotting results.
- Jupyter for showing results, code, and text in the same document.

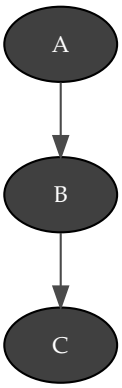

Figure 7: The Bayesian network used in most of the experiments

The source code and data used for this research are open-source and can be found at [29].

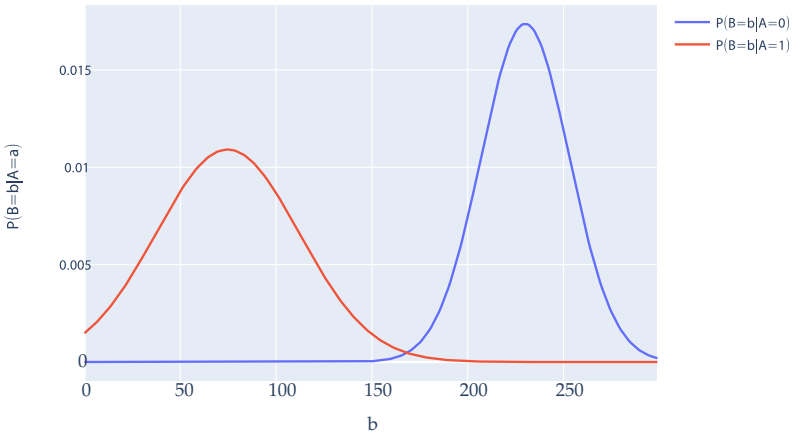

Figure 8: Distribution of P(B=b)

3.2 Sample

In most research, a sample population is selected based on the group's characteristics, how representative this sample is for the global population, and the ability to control variables in this population. Within this experiment, the data used can be easily controlled and exchanged for other datasets.

We used Bayesian networks (like the one in Figure 6) to generate the data for the experiments. We generate these networks from scratch and apply the steps from section 2.4 (which allows us to control some independent variables such as the standard deviation of some distribution).

The networks used in this experiment have the following properties:

$$P(A) = 0.4 \tag{22}$$

$$P(\overline{A}) = 0.6 \tag{23}$$

The probability distributions for B and C can are shown in Figure 8 and Figure 9, respectively. By "measuring" these continuous functions (in the case of the figures, normal distributions skewed slightly to the left or right) at a specific sampling density, it is possible to approximate the continuous distribution using histogram-like "bins". This structure is similar to how discrete distributions are defined and resembles a very large-scale,

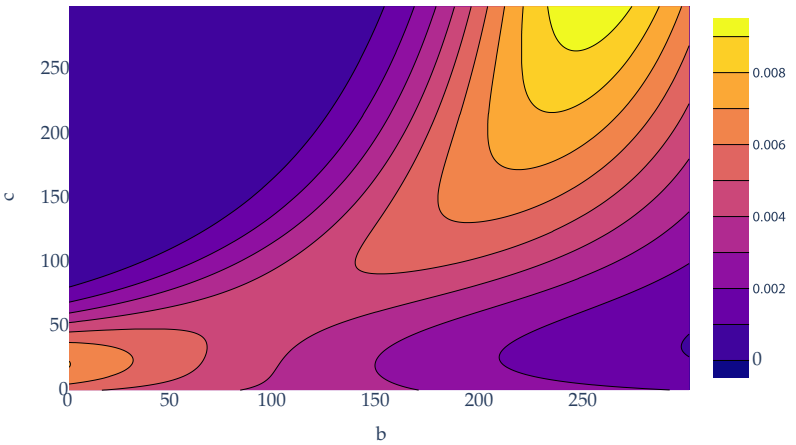

Figure 9: Distribution of P(C=c)

ordered generalization of a categorical distribution.

Then, samples are taken from the probabilities in this Bayesian network, and put into a database (an example in Table 1). We created a sparse matrix from these values, shown in Table 2. For each row in the original database, the new database is filled with zeroes, except for the column, representing the original value, which is set to 1. For instance, row 2 has B=2. Thus, on the second row of the sparse matrix, column 2 for variable B is set to 1. Afterward, we added Gaussian noise (with $\mu = 0$ and $\sigma = 0.01$) to all the entries (which is the most likely form of noise for continuous distributions) negative entries set to 0. The distributions are then normalized, to make sure they sum to 1. The result of this step is in Table 3.

Table 1: Example of sample database (hard evidence) at sampling density 4

	A	**B**	**C**
0	1	1	2
1	1	1	2
2	1	2	2
3	0	2	2
4	1	1	2
...	...	...	...
9995	0	2	2
9996	1	1	3
9997	1	1	2
9998	1	1	2
9999	1	1	0

Table 2: The same sample data from Table 1, converted to a sparse matrix, thus representing probabilities

Variable	**A**		**B**				**C**			
Value	**0**	**1**	**0**	**1**	**2**	**3**	**0**	**1**	**2**	**3**
0	0	1	0	1	0	0	0	0	1	0
1	0	1	0	1	0	0	0	0	1	0
2	0	1	0	0	1	0	0	0	1	0
3	1	0	0	0	1	0	0	0	1	0
4	0	1	0	1	0	0	0	0	1	0
...	...	...	...	...	...	...	...	...	...	...
9995	1	0	0	0	1	0	0	0	1	0
9996	0	1	0	1	0	0	0	0	0	1
9997	0	1	0	1	0	0	0	0	1	0
9998	0	1	0	1	0	0	0	0	1	0
9999	0	1	0	1	0	0	1	0	0	0

Table 3: The data from Table 2 with Gaussian noise added

Variable	**A**		**B**				**C**			
Value	**0**	**1**	**0**	**1**	**2**	**3**	**0**	**1**	**2**	**3**
0	0.000	1.000	0.017	0.962	0.006	0.016	0.000	0.000	1.000	0.000
1	0.004	0.996	0.019	0.981	0.000	0.000	0.000	0.009	0.971	0.020
2	0.000	1.000	0.005	0.008	0.986	0.000	0.006	0.008	0.986	0.000
3	1.000	0.000	0.000	0.000	0.977	0.023	0.000	0.000	1.000	0.000
4	0.000	1.000	0.000	0.995	0.005	0.000	0.014	0.000	0.984	0.003
...	...	...	...	...	...	...	...	...	...	...
9995	1.000	0.000	0.000	0.000	1.000	0.000	0.000	0.000	1.000	0.000
9996	0.009	0.991	0.008	0.992	0.000	0.000	0.017	0.000	0.001	0.981
9997	0.000	1.000	0.004	0.987	0.009	0.000	0.000	0.004	0.996	0.000
9998	0.000	1.000	0.013	0.967	0.002	0.018	0.005	0.000	0.995	0.000
9999	0.000	1.000	0.009	0.981	0.000	0.010	0.985	0.000	0.000	0.015

3.3 Data collection

The data, illustrated in Table 3, is used to train a neural network. Each row represents one data point and becomes a 1D input tensor. This input layer is then connected, through various hidden layers, to the output layer.

In the experiments for this paper, there are nine hidden layers. The middle layer (the latent space) has a dimensionality equal to the amount of Bayesian network variables, in this case, 3.

The layers between the input and middle layers decrease exponentially in size to match the dimensionality of both. The approach to use a dimensionality of 3 for the latent space encouraged the network to output one "best guess" for each variable A, B, C in the encoder section, like in a regression network. Then, the decoder section would hopefully learn the one-hot encoding of this number.

The output layer is a concatenation of multiple dense layers with a softmax activation function, the outputs of such a layer sum to 1, ensuring that the outputs are proper probability distributions.

A sample database is generated at a sampling density of 4, because training a network at low sampling densities is very fast, as the network has a low amount of trainable weights. We will test sampling densities once we have found a promising network structure.

The sample is split into 80% training and 20% testing data (although this is changed for some measurements), after which the network is trained and tested using various training methods (such as unsupervised and semi-supervised). The hyperparameters used are the default Keras/TensorFlow parameters, as explained in section 2.2. The batch size used is 32, and the amount of epochs is 100 (further explained in the Results section). The loss functions used in most measurements include categorical cross-entropy, which is commonly used for multinomial classification problems like this one, the Jensen-Shannon divergence, and the mean square error. From this basic setup, modifications will be added to the network to determine which network performs best on the task of removing noise. The networks with the highest performance will then be exposed to more complex datasets (with higher sampling densities, more variables, higher amounts of noise) to see if such networks generalize.

We started the research using ReLU to activate the hidden layers (as is common in deep learning). However, (due to disappointing performance) we soon switched to a combination of sin, cos, linear, ReLU and Swish activation functions, because they are better for non-linear problems such as those in the encoder section of the network (going from a noisy one-hot encoding to a regression output in the encoding layer). These activation functions are the "default" case mentioned in the rest of the text.

3.4 Data analysis

The primary way of checking the data quality is by comparing the output of the neural network to the clean data, D_{GT} (see section 2.4). There are several ways we do this, such as using the mean squared error between the two, comparing their Shannon entropy, or calculating their Jensen-Shannon divergence [16]. We can then determine how much the Jensen-Shannon divergence improved from this cleaning process, by subtracting the new JSD (which should be lower) from the original JSD.

$$Improvement = JSD_{old} - JSD_{new} \tag{24}$$

However, because this metric depends on the amount of noise added (a network that removes 90% of the noise will achieve a much higher improvement score for a noisy

dataset), we suggest the following metric instead:

$$Noise\ reduction\ in\ \% = 100 - (\frac{JSD_{new}}{JSD_{old}} \cdot 100) \tag{25}$$

This method should compensate for the size of the dataset, and the amount of noise added, showing the network's real improvement. The higher this number is, the better the network performed, with a maximum of 100% (meaning that all the noise was removed). If this number is below 0, the network was unable to remove noise and added noise to the dataset instead.

4 Results

In the figures of this section, several acronyms will be used to save space. SD stands for the sampling density used to generate the data. CCE, MSE and JSD are loss functions (as defined in section 2. JSD(5) means that 5 hidden layers were used instead of 9, with the JSD loss function. CCEu means that the CCE loss function was used, but training was unsupervised. In all other cases, training was semi-supervised (except for in Figure 11, where this is mentioned explicitly, and in Figure 12, where the differences between training methods such as supervised and unsupervised learning are being evaluated).

For each figure, a table with the values used to construct it can be found in the appendix.

4.1 Performance of various network architectures

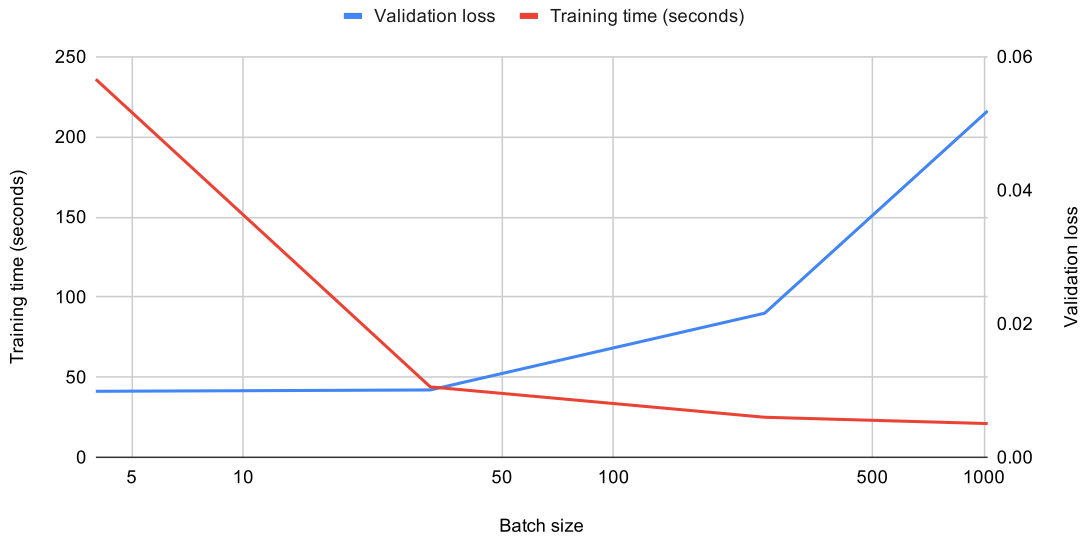

Figure 10: Influence of batch size on training convergence (for unsupervised learning with CCE loss function, 50 epochs). This figure was generated from the values in Table 4.

As shown in Figure 10, a batch size of 32 has a good compromise between convergence speed (lowest loss) and training time required. Thus, we used this parameter for all the experiments. We chose to set the number of epochs to 100, as most datasets seemed to converge by that point, while still not taking too long to train.

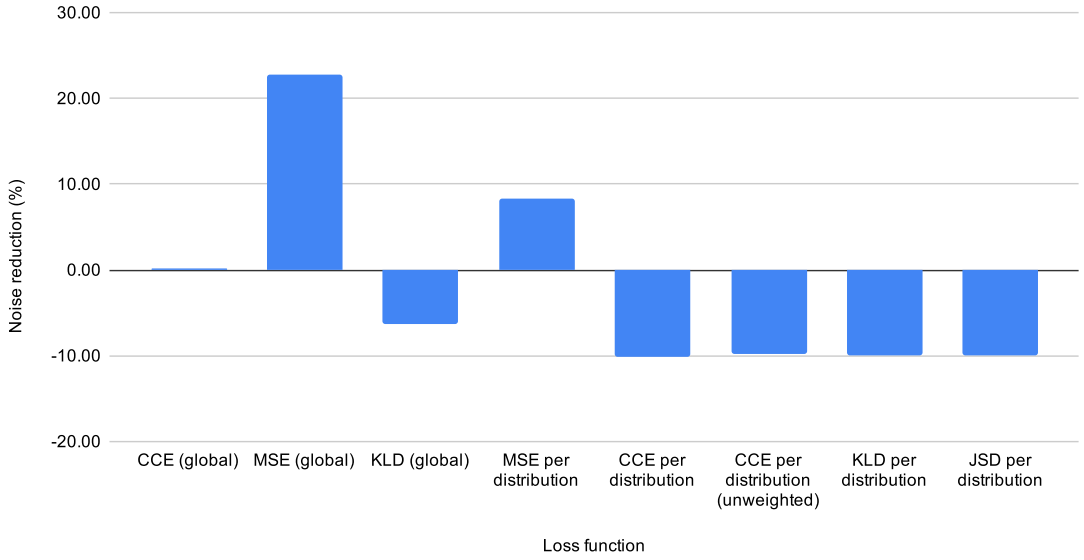

Figure 11: Effect of loss function on unsupervised learning performance at sampling density 4. ”Global” means that the loss function was applied to the entire output layer, while ”unweighted” means that the distribution size was not used in calculating the average loss. Due to the way the JSD loss function works, applying it globally was not possible. This figure was generated from the values in Table 5.

The first experiments (shown in Figure 11) were carried out with unsupervised learning as a training method. These showed that most loss functions were not suitable for reducing noise except for MSE and CCE (with CCE only achieving a tiny performance increase). Thus, the focus was shifted to these loss functions, as they were most promising.

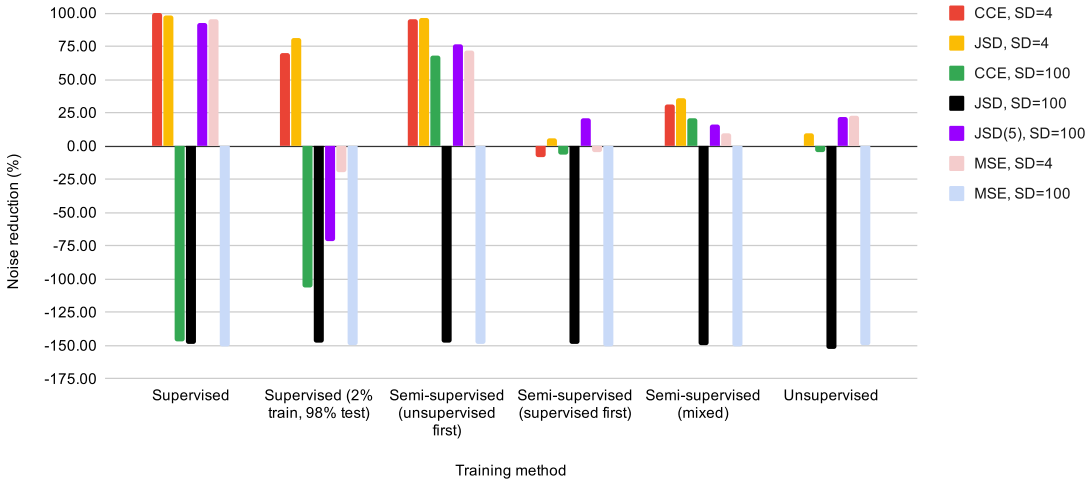

Figure 12: Effect of training methods on performance. This figure was generated from the values in Table 6.

Further experiments with unsupervised learning were not satisfactory, and although

small amounts of noise reduction were sometimes achieved, these were not consistently reproducible. However, experiments with different training methods (shown in Figure 12) showed that supervised learning resulted in an excellent performance at low sampling densities (97% noise reduction for JSD, 99% for CCE) for these loss functions. Because of this interesting result, we decided to investigate further. By changing the train/test split to 98% testing data and 2% training, we simulate a situation where the ground truth is available for a small part of the data. Performance in this situation was much better than in the unsupervised situation (for low sampling densities), and having only some of the data be labeled is more realistic than fully supervised learning, as data cleaning is not needed when the labels of the data are true values. Such a situation is quite similar to those encountered in existing techniques, such as probabilistic data integration [1]. The performance of this was still excellent, so we tried semi-supervised learning. We tried multiple combinations: unsupervised learning followed by supervised learning, the reverse, and a mixed-method where each training epoch switches between unsupervised and supervised. The method which had unsupervised learning followed by supervised learning performed best, achieving a noise reduction of ~95%, for the CCE and JSD loss functions at low sampling densities, and also achieving substantial noise reduction at higher sampling densities. This was not expected, as JSD performed so poorly during unsupervised learning. The performance of MSE was quite inconsistent in all cases except for unsupervised learning (frequently adding more noise to the data than removing it). Thus we decided not to investigate this loss function further. All these findings can be found in Figure 12.

Because unsupervised learning is an interesting baseline for the network's performance, its performance is also plotted in all the other figures. In most situations, it adds a tiny amount of noise, but for completely unviable network structures, it adds upwards of 100% of extra noise to the data. This makes it possible to see whether bad performance is caused by bad network structures or other factors.

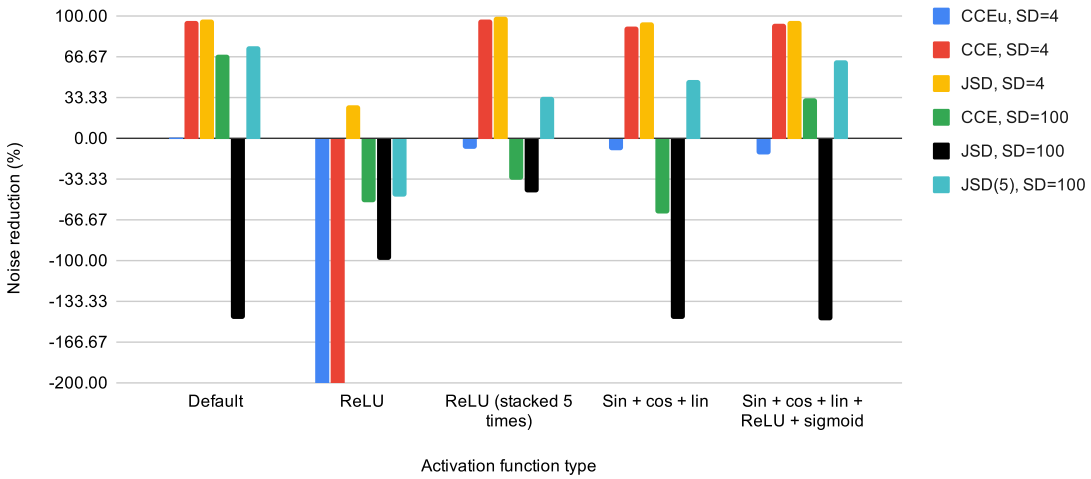

Figure 13: Effect of different activation functions on performance. ReLU preformance in some cases added ~800% of noise, which could not be shown properly in this figure. This figure was generated from the values in Table 7.

Figure 13 shows that in most cases, the noise reduction performance of the network using the ReLU function was disappointing, with the amount of noise after training frequently rising above 500% of the original noise levels. The activation function described in section 3.3 (referred to in Figure 13 as "Default") led to promising results, achieving a

high level of noise reduction, thus this was used as the standard activation function for further experiments. Figure 13 also shows that other activation function types (such as ReLU layers concatenated five times) performed similarly well. However, the default activation function still performed better than the others by a slight margin. Furthermore, it seems that ReLU activation functions perform much better with a JSD loss function. At a sampling density of 4 using the regular JSD setup, the network achieved 99.54% noise reduction, an impressive result. Sadly, this performance does not scale to higher sampling densities, leading to an increase in noise. However, regular JSD performance at a sampling density of 100 was much higher than the default case (but still negative) using the ReLU activation function stacked 5 times. While this activation led to JSD's positive results with 5 hidden layers, the default activation function led to the best results in that case.

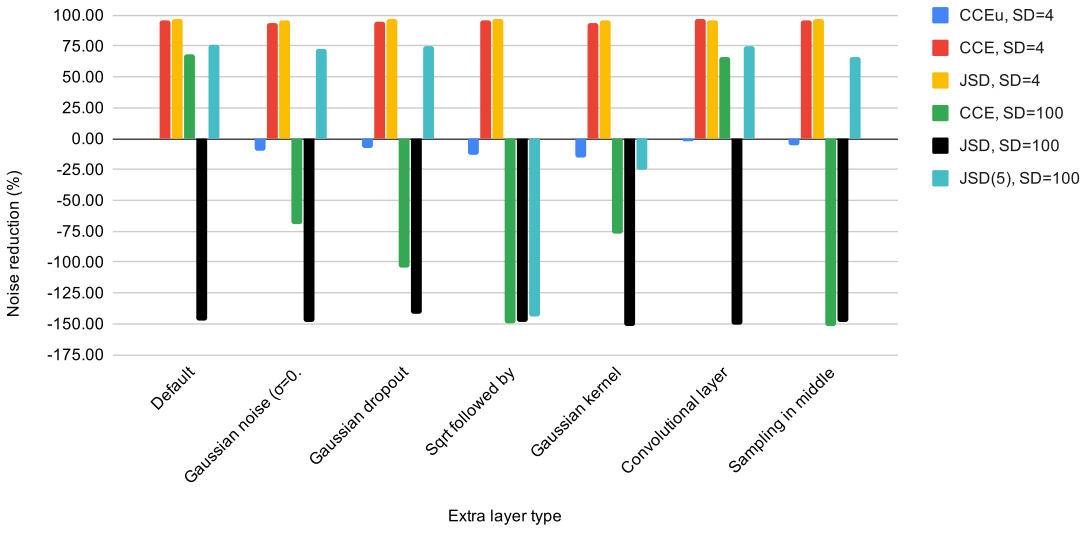

Figure 14: Effect of different layer types on performance. This figure was generated from the values in Table 8.

Various layer types were tested to see if these would increase the performance further. In all cases, except for the sampling layer (which was placed in the middle layer to create a variational autoencoder, as described in section 2.3), these replaced the first dense layer of the network (at the input). The results of this can be found in Figure 14. None of these additions led to a substantial improvement in performance compared to the default case, although the convolutional layer increased the performance of the CCE network at sampling density 4 from ~95% to 97%.

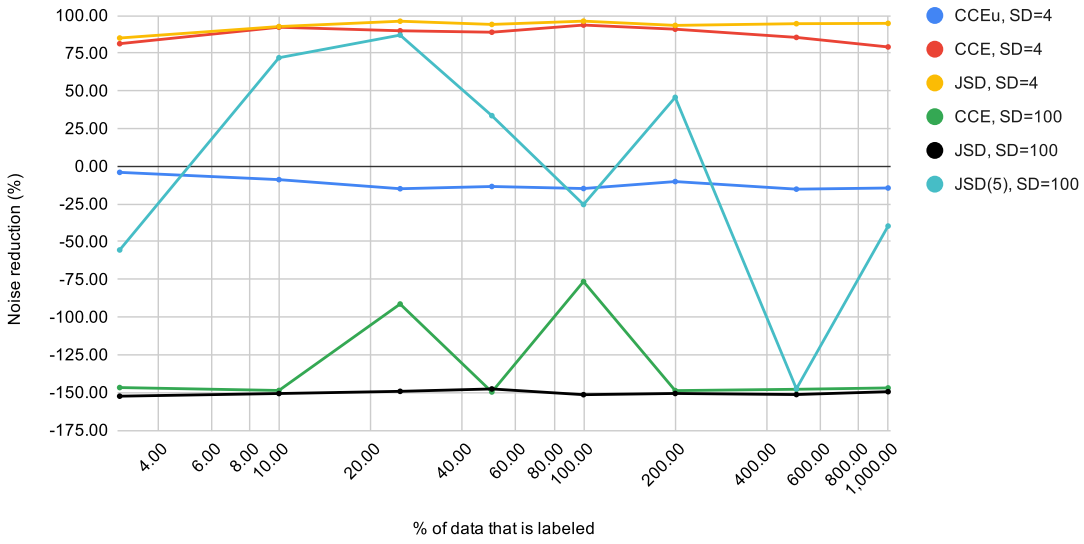

Figure 15: Effect of the amount of Gaussian kernel landmarks on performance. This figure was generated from the values in Table 9.

Figure 15 shows that the amount of landmarks of the Gaussian kernel layer (when it is used) does not affect performance much (having a slight peak at 100 landmarks) for all cases except JSD with 5 hidden layers. In the latter case, performance decreases as the number of landmarks is further away from 25. Remarkably, the performance of JSD(5) at 25 landmarks is higher than the standard case, reaching 87% noise reduction.

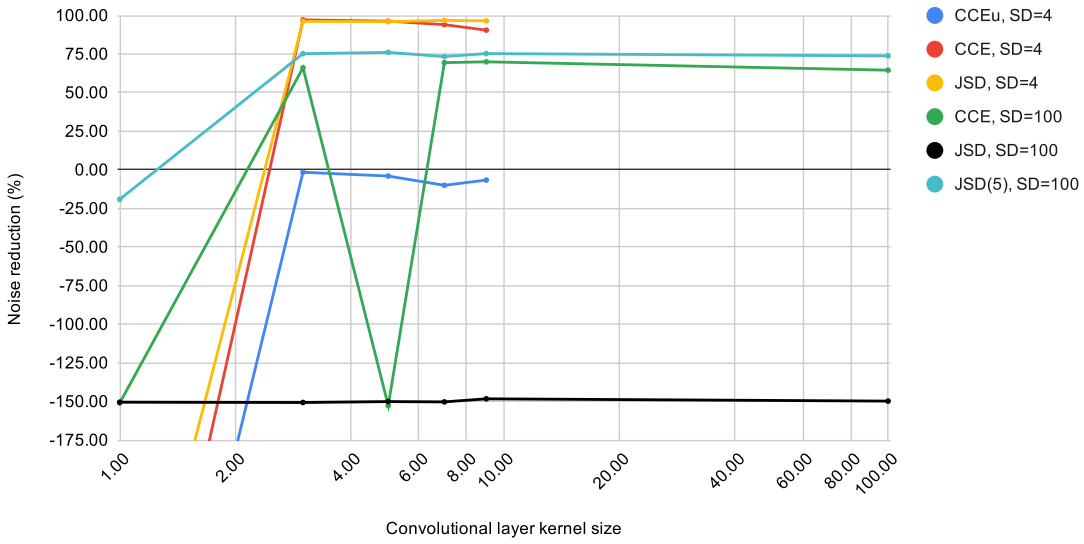

Figure 16: Effect of the convolutional layer kernel size on performance. This figure was generated from the values in Table 10.

The kernel size of the convolutional layers (shown in Figure 16) does not affect performance much either. A kernel size of 1 has a bad performance in all cases. Perhaps the network could have learned more interesting features when the kernel size matched the sampling densities of each variable (up to 100 for some datasets), which is why we decided to extend the plots for the datasets. This turned out not to be the case.

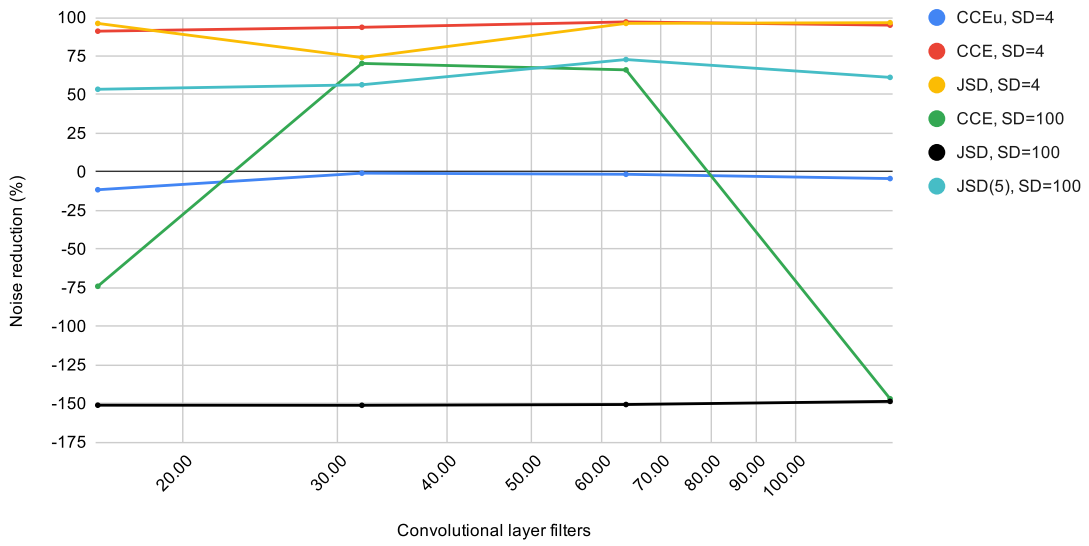

Figure 17: Effect of the amount of convolutional layer filters on performance. This figure was generated from the values in Table 11.

The number of filters for the convolutional layer does not profoundly affect performance in most cases except when the CCE loss function is used at sampling density 100. In that case, a low amount of filters leads to decreased performance (as the network can learn fewer features), while a high amount of filters also decreases performance. These findings can be found in Figure 17.

We also attempted to use more than one convolutional layer. We did not see any performance improvement for low sampling densities, and training these networks was abysmally slow. At higher sampling densities, one performance measurement would have taken upwards of 12 hours. Thus, we did not investigate the use of more than one convolutional layer any further.

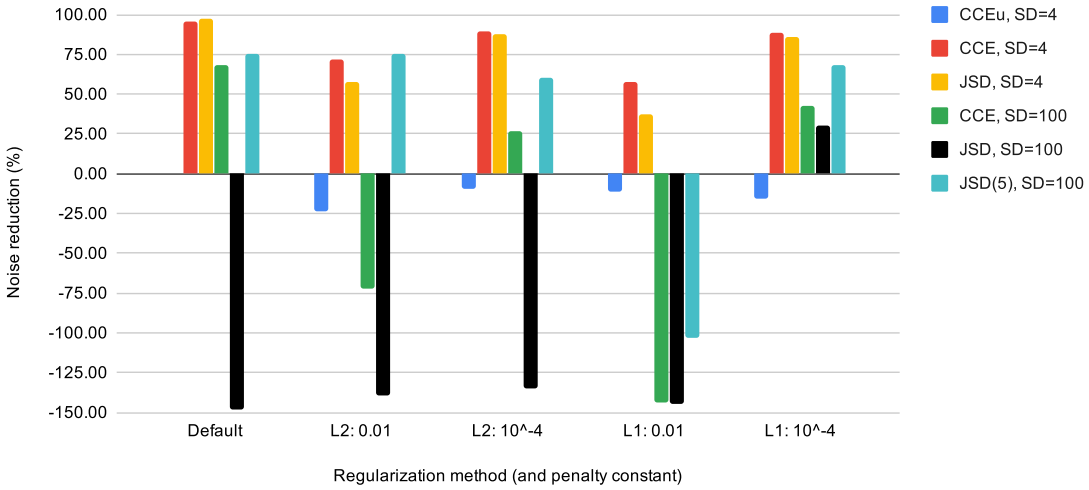

Figure 18: Effect of regularization on performance. This figure was generated from the values in Table 12.

Figure 18 shows that regularization constraints on the neurons' activity (how high the

absolute value of their output is) decreased performance in all cases, with one exception: Using the JSD loss function at sampling density 100, L1 regularization with a penalty constant of $10^{-}4$ caused the network to no longer add noise to the data.

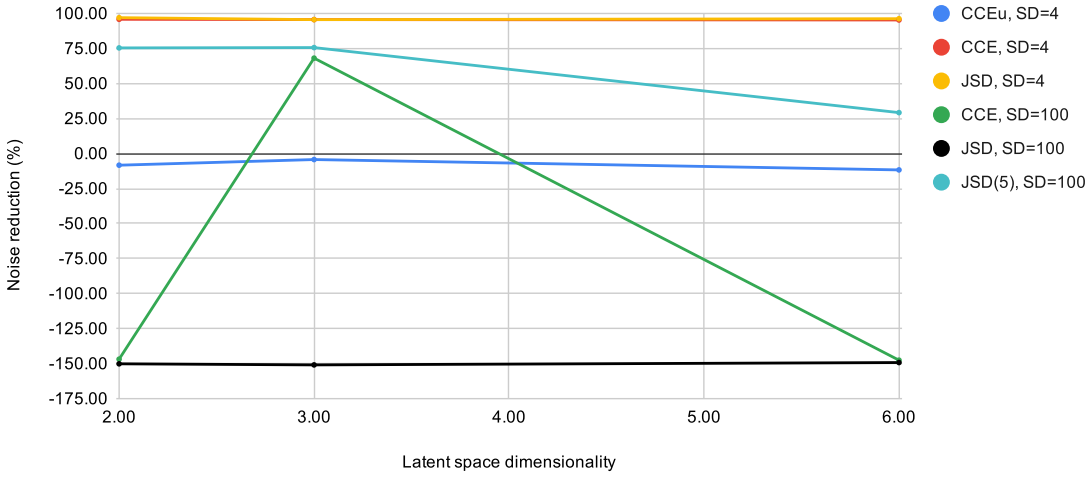

Figure 19: Effect of latent space dimensionality on performance. This figure was generated from the values in Table 13.

The dimensionality of the latent space did not seem to affect noise removal performance in most cases. However, when the JSD loss function is used at sampling density 100, performance is best at a dimensionality of 3. Furthermore, there is a slight linear decrease in network performance using the JSD loss function with 5 hidden layers. These findings are shown in Figure 19.

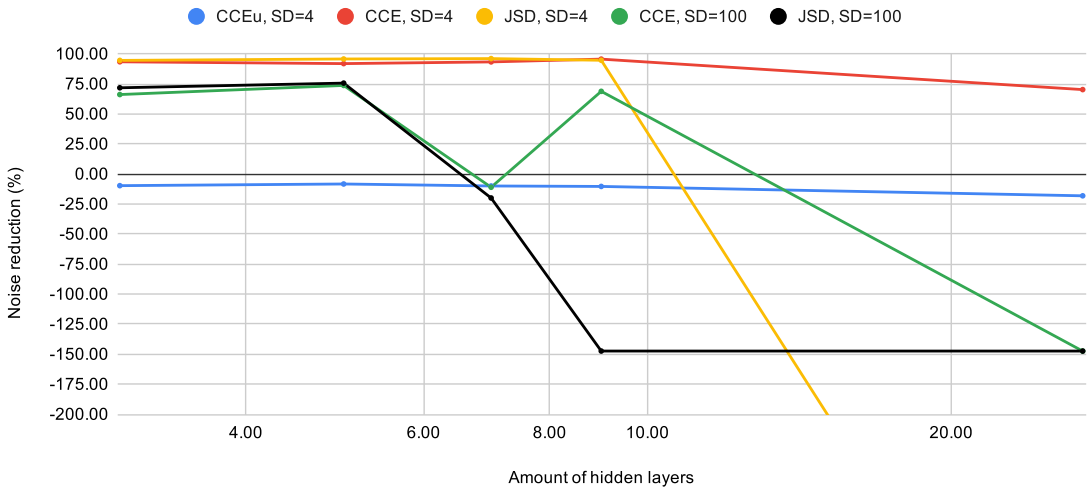

Figure 20: Effect of network deepness on performance. This figure was generated from the values in Table 14.

It seems that between 5 and 9 hidden layers are the best deepness for performance in most cases (Figure 20). Because there seemed to be a slight peak in performance at 9 hidden layers, most experiments were done with that configuration. We later discovered that networks with high sampling densities, and networks with the JSD loss function,

perform better with 5 hidden layers. This difference was not very large for the CCE loss function (only ~5%), but the difference was significant enough for the JSD loss function to justify redoing the experiments at high sampling densities with 5 hidden layers instead. This is the JSD(5) configuration shown in other figures.

4.2 Network performance under different conditions

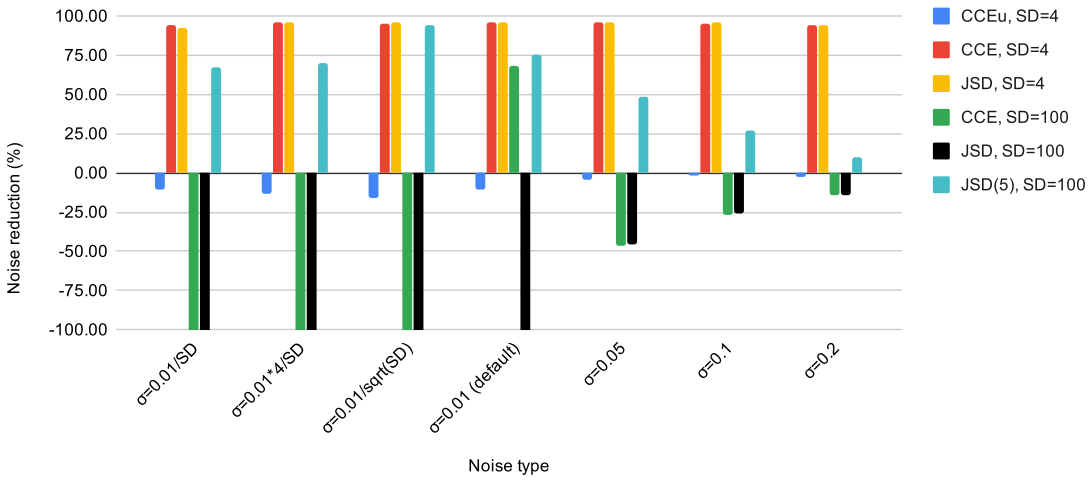

Figure 21: Effect of the standard deviation of noise on performance. Some configurations added between 600% and 2000% more noise to the data, which could not properly be shown in this figure. This figure was generated from the values in Table 15.

Because we were worried the noise generation might be causing the lower performance at high sampling densities (instead of the higher complexity of the data), we tried different noise generation methods, which are shown in Figure 21. The distribution of the noise does not significantly impact the performance at low sampling densities. At higher sampling densities, the performance seems to tend towards zero as more noise is added. There is a strange peak in performance at a standard deviation of 0.01 for the CCE loss function at sampling density 100. It is also quite apparent that using JSD as the loss function with 5 hidden layers is the only method that leads to a reliable noise reduction in all cases.

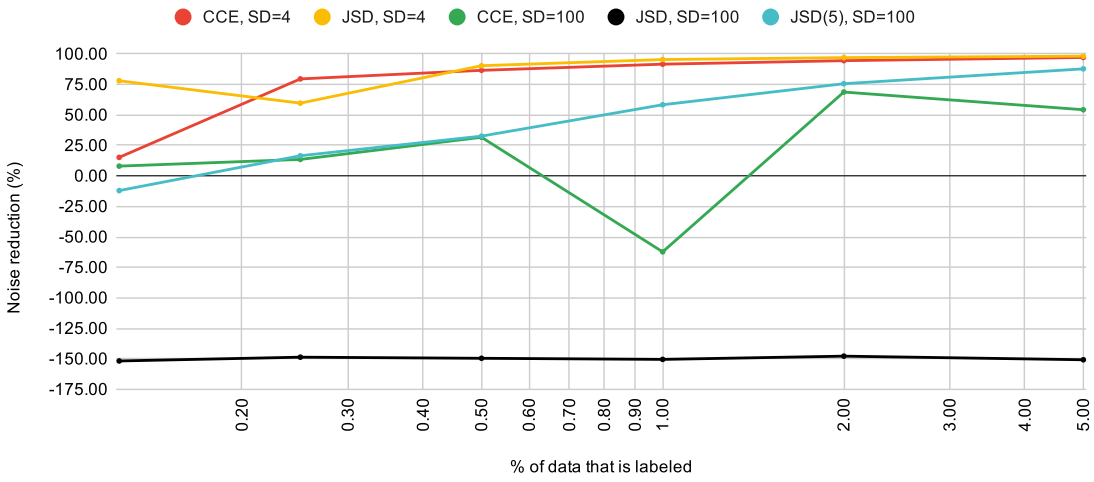

Figure 22: Effect of the amount of labeled data on performance. This figure was generated from the values in Table 16.

For semi-supervised training, the amount of labeled data had a relatively small effect on performance at low sampling densities (except for minimal percentages, in which performance went down), but a large effect at high sampling densities. In that case, it seems that labels for 2% or more of the data is necessary for reliable noise removal. The JSD loss function (at low sampling densities) performs exceptionally well when only a minuscule amount of data is labeled. It performs badly for all labeling percentages at high sampling densities with 9 hidden layers while steady linear growth in performance with 9 hidden layers. These findings are shown in Figure 22.

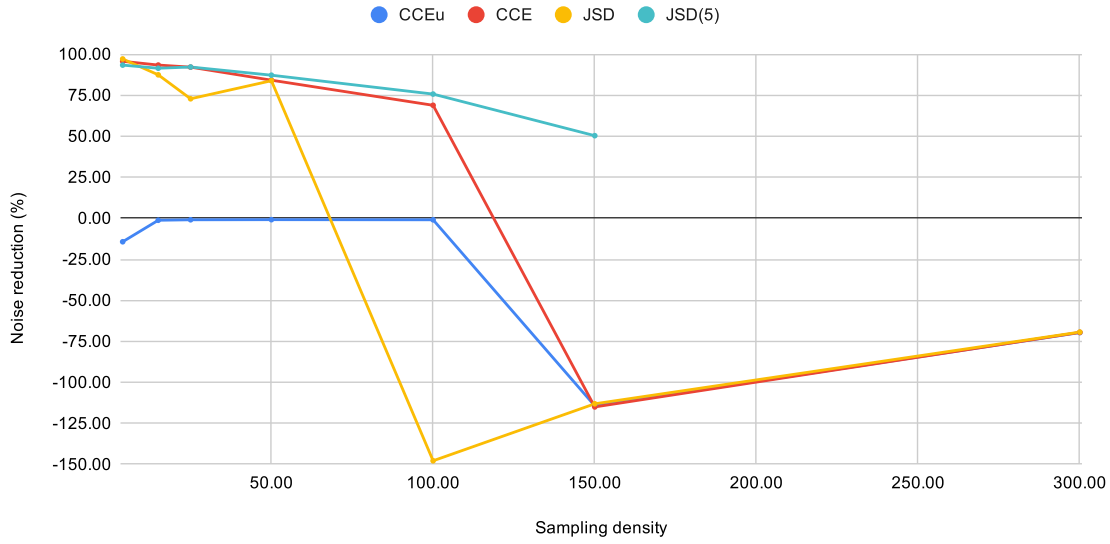

Figure 23: Effect of sampling density on performance. Despite multiple attempts, training does not converge to a network that produces proper probability distributions for JSD(5) at SD=300. This figure was generated from the values in Table 17.

CCE performance seems to worsen slightly for higher sampling densities up to 100,

after which noise reduction stops entirely (shown in Figure 23). While standard JSD outperforms CCE at low sampling densities, its performance diminishes faster, losing noise reduction capabilities around a sampling density of 100. JSD with 5 hidden layers has a much more linear decrease, still managing to remove half of the noise at sampling density 150. Unsupervised learning performance decreases at such sampling densities as well. Strangely, performance seems to rise again for higher sampling densities. In any case, because the sampling density of 100 seems to be the edge of stability for the system, and because a sampling density of 100 is an adequate approximation of continuous functions in most cases, we decided to use this sampling density in most other experiments as well. Performance at such a high sampling density is more sensitive to noise and more inconsistent. Sometimes the network does not converge to a configuration that reduces noise, requiring a restart of the training process.

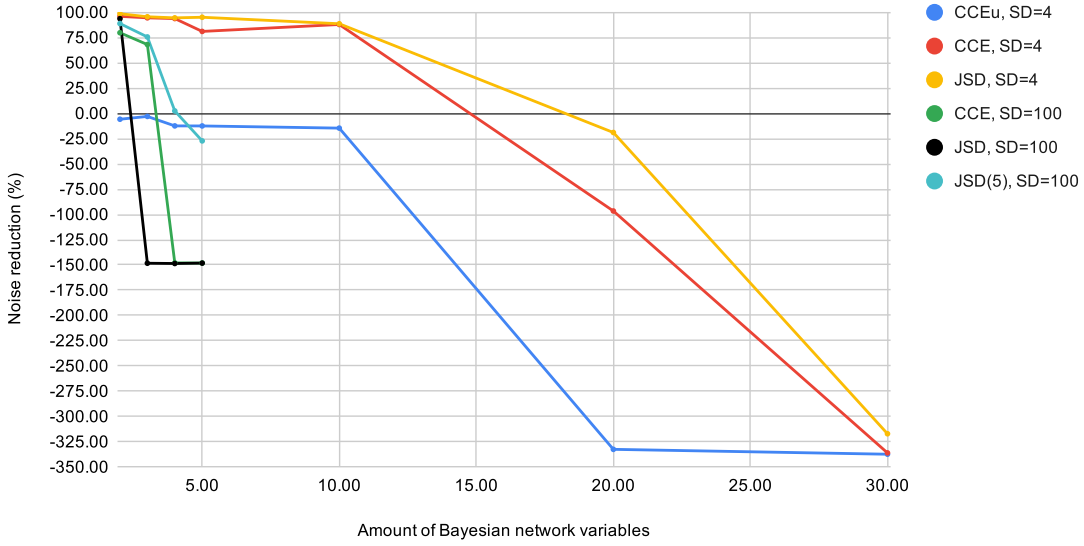

Figure 24: Effect of Bayesian network size on performance. Higher values for sampling density 100 would have taken several days to train, which was aborted due to low performance. This figure was generated from the values in Table 18.

The amount of variables in the Bayesian network seems to barely affect performance at low sampling densities, up to 10, after which performance becomes much worse. At high sampling densities, performance seems to be very bad for Bayesian networks of a size larger than 3 for CCE, and 2 for JSD with 9 hidden layers. However, JSD with 5 hidden layers has a much more linear decrease in performance, still removing 2.42% of noise at a Bayesian network size of 4. These findings are shown in Figure 24.

5 Discussion

The results were somewhat unexpected. While the fact that semi-supervised learning is a reliable technique for removing noise is not surprising, we were expecting to be able to achieve this with unsupervised learning, like in [2]. However, in hindsight, it is not surprising that an unsupervised approach does not work. Training a neural network to reproduce noisy data will not suddenly remove the noise (garbage in, garbage out). Perhaps the approach we were trying to reproduce was not fully supervised somehow (accidentally using the uncorrupted data in the loss function), or had unintentional bias (for example, datasets chosen for their good performance instead of at random).

It is still unclear why the MSE loss function works better than all the other loss functions for unsupervised learning, and why we can sometimes achieve some unsupervised noise reduction at high sampling densities using the JSD loss function with 5 hidden layers. Perhaps these methods are more well-suited to simply reducing errors than the other, more complicated configurations. It is strange that almost none of the modifications that we tried improved the performance for unsupervised learning.

Using 5 concatenated layers with ReLU activation functions leads to a much better performance than using just a regular layer with that activation function, suggesting that the increased dimensionality of the layers might cause the performance increase. This might be because ReLU activations functions lead to a faster convergence [14]. It is interesting to see that this combination of layers leads to the highest performance we observed at low sampling densities (99.54% noise reduction using the JSD loss function with 5 hidden layers compared to 96.66% with the default activation function), but that this does not generalize to higher sampling densities (where the default activation function performs better). Perhaps the data contains more non-linear relations at higher sampling densities, leading to increased performance for these non-linear activation functions.

It makes sense that none of the added constraints (L1/L2 regularization, dropout layers, forcing the latent space to be more regular using variational autoencoder techniques) had a substantial positive effect on performance, as the low dimensionality of the middle layer is already a sparsity constraint of sorts. Interestingly, adding a convolutional layer increased performance from 95.63% up to 97% for the CCE loss function at sampling density 4, while adding a Gaussian kernel (RBF) layer with 25 landmarks to the JSD loss function (with 5 hidden layers) leads to the highest performance we observed at sampling density 100: 86.99% noise reduction, up from 76.88% noise reduction without the Gaussian kernel layer. Perhaps convolutional layers are useful at low sampling densities because they are well-suited to learning low-dimensional features, while Gaussian kernels are often used for learning high-dimensional features.

However, convolutional layers led to a significant decrease in performance when a kernel of size 1 was used, which is explainable by the fact that such a kernel cannot learn any features (convolutions over 1 datapoint merely reduce the number of channels per pixel in most cases [30], which is not interesting for us). Using a large number of filters in the convolutional layer also decreased performance, probably because the network takes much longer to converge due to an increase in trainable weights.

The autoencoder did perform slightly better at higher code layer dimensionalities in some cases, but this improvement is so small that we are not sure if it is even statistically significant. We decided to keep using the solution of having the amount of the code layer dimensionality be equal to the number of variables in the Bayesian network (in most cases, 3) because it is easily explainable what features the layer represents in that case, and because there was a slight peak in performance at high sampling densities.

Very shallow networks do not perform as well as deeper networks, which is as expected. Deeper networks can learn more complex features [14]. However, very deep

networks seem to perform less adequately. Perhaps this is because it takes longer for the training process to converge, as there are far more trainable weights. This is even more visible for high sampling densities. Because those networks are much "wider" (each layer has many more nodes), there are additional weights to train.

We did not discover why reducing the amount of noise makes the network's performance worse in most cases. This performance reduction might be caused by the lack of accuracy for such low values when represented by 32-bit floating-point numbers. However, it does make sense that the performance tends towards zero as more noise is added to the data. The amount of noise removed or added by the network becomes more and more insignificant compared to the amount of noise in the data.

We expected that performance increases with the amount of labeled data, which is true for most cases. This trend is harder to identify for CCE at high sampling densities (the data seems very noisy). Perhaps large datasets such as those made from high sampling densities are more sensitive to the randomness in the (randomly shuffled) test/train splits of the dataset. JSD networks with 9 hidden layers and a sampling density of 4 perform very well at low amounts of labeled data, suggesting that the network's complexity can compensate for this low data availability (as such networks are already able to achieve small amounts of noise removal using unsupervised learning).

The autoencoder performing a ~70% or higher noise reduction on small Bayesian networks for a sampling density of 100 is very significant. Such a sampling density is an excellent approximation of continuous distributions and is almost indistinguishable from them in plots.

We cannot explain why exactly the performance decreases so much between a sampling density of 100 and 150. Such a high sampling density seems to be the edge of stability for the system in most cases. Thus, the bad performance for four or more Bayesian network variables at sampling density 100 is easily explainable. Adding more variables to the Bayesian network at sampling density 100 will vastly reduce the performance, appending at least 100 columns to the database. Perhaps there is just a maximum amount of columns that the autoencoder can learn to reproduce reliably.

It seems that for the JSD loss function (with 9 hidden layers), this already happens at between 200 columns (BN size of 2, sampling density 100) and 300 columns (BN size 3, sampling density 150), while for CCE this only happens after 300 columns. JSD is more well-suited to data that is less complex, while CCE is better for larger and more complex inputs. Changing the number of hidden layers to 5 results in the JSD loss function having a much more linear decrease in performance as the Bayesian network size or sampling density is decreased. This might be due to the reduced amount of trainable weights converging faster to a working noise-removing structure. Furthermore, while semi-supervised learning performance decreases at high sampling densities, so does unsupervised learning performance. This suggests that the low performance of semi-supervised learning in those cases might be related to the network's inability to work with such large datasets regardless of the learning method or goal.

Performance seems to tend towards zero for very high sampling densities, but this might be because the same amount of noise is added to every point of data, leading to the total amount of noise being more significant at high sampling densities. Thus, the noise added or removed by the networks themselves is comparatively smaller.

6 Conclusion

The approach outlined in this paper can remove significant amounts of noise from non-categorical probabilistic data, granted that ground truth values are available for a small percentage of this data. Its performance decreases as the sampling density increases, but it performs well in most cases. The histogram structure proposed for this generalization works as an appropriate way to model continuous variables allowing them to be possible inputs for neural networks.

A network with 5 (at high sampling densities) to 9 (at low sampling densities) hidden layers and a latent space dimensionality equal to the amount of Bayesian network variables seem to have the best performance. Adding sparsity constraints, layers that add noise, or trying to regularize the latent space does not improve the performance. The undercomplete autoencoder is an adequate structure, and its low latent space dimensionality is already a sparsity constraint in and of itself.

It seems that the default Tensorflow and Keras hyperparameters combined with the *Adam* optimizer work well. Furthermore, a batch size of 32, and 100 epochs for training led to quite a rapid convergence. Thus, not much tuning of hyperparameters was needed.

Using the Jensen-Shannon distance as loss function works best for removing noise. However, this requires 2% of the data to be labeled. In most cases, a concatenation of layers using sin, cos, linear, ReLU, and Swish activation functions led to the highest performance. However, at low sampling densities (of 4), using 5 concatenated ReLU layers lead to a staggering 99.54% noise reduction. At high sampling densities (of 100), adding a Gaussian kernel layer led to a 86.99% noise reduction. Unsupervised learning sometimes leads to a noise reduction of 22.10%, but these results are much more unreliable. Lastly, using 9 hidden layers instead of 5 at low sampling densities leads to high noise reduction capabilities for very low amounts of labeled data, still achieving 78.06% noise reduction while only 0.125% of data was labeled.

Further research should focus on the influence of different noise distributions on the performance of this solution. We used Gaussian noise, as it seemed like the best way to model the noise of continuous variables, but other distributions might be more realistic. Furthermore, statistical analysis of the data might be necessary to remove the inherent noisiness in the results that comes from training neural networks, as the network converges to slightly different weights each time.

Other neural networks that are closely related to autoencoders might also prove useful in future research. For example, autoencoders can be stacked [31] with restricted Boltzmann machines (which learn probability distributions over their inputs [32]) to produce deep belief networks [33].

Acknowledgements

I would like to thank dr.ir. M. van Keulen and N. Bouali M.Sc. for supervising this thesis, and for their helpful feedback during our frequent meetings. Even though this thesis was completed during a pandemic, which led to strange and unfamiliar situations, their frequent feedback made this a streamlined process and left me well-prepared for completing this thesis. Even though I was working from total isolation, it did not feel like it.

Next, I would also like to thank dr. H.K. Hemmes for interrupting his holiday to assist in the grading of this assignment. Without his help, I would not have been able to graduate on time and continue my future endeavors.

I would also like to thank dr. N. Strisciuglio and dr.ir. D.C. Mocanu for their feedback on interim presentations, and helpful suggestions on which autoencoder structures might increase performance.

Furthermore, I want to thank my housemates; Merijn Hofsteenge gave me excellent guidance on improving autoencoder performance, while Floris Breggeman and Andrew Heath provided very insightful feedback when proofreading this report.

Lastly, I would like to thank my family for their unwavering support.

References

[1] M. V. Keulen, "Probabilistic Data Integration," in *Encyclopedia of Big Data Technologies*. Springer International Publishing, 2018, pp. 1–9.

[2] R. R. Mauritz, "Improving data quality in a probabilistic database by means of an autoencoder," jan 2020. [Online]. Available: http://essay.utwente.nl/80505/

[3] D. Suciu, D. Olteanu, C. Ré, and C. Koch, *Probabilistic Databases*, 2011, vol. 3, no. 2.

[4] M. Grohe and P. Lindner, "Infinite Probabilistic Databases," apr 2019. [Online]. Available: http://arxiv.org/abs/1904.06766

[5] C. Huang and A. Darwiche, "Inference in belief networks: A procedural guide," *International Journal of Approximate Reasoning*, vol. 15, no. 3, pp. 225–263, oct 1996.

[6] A. Jadhav, D. Pramod, and K. Ramanathan, "Comparison of Performance of Data Imputation Methods for Numeric Dataset," *Applied Artificial Intelligence*, vol. 33, no. 10, pp. 913–933, aug 2019. [Online]. Available: https://www.tandfonline.com/doi/full/10.1080/08839514.2019.1637138

[7] I. F. Ilyas and X. Chu, *Data Cleaning*. Association for Computing Machinery, jul 2019.

[8] S. De, Y. Hu, M. V. Vamsikrishna, Y. Chen, and S. Kambhampati, "BayesWipe: A Scalable Probabilistic Framework for Cleaning BigData," pp. 1–14, 2015. [Online]. Available: http://arxiv.org/abs/1506.08908

[9] A. de Keijzer and M. van Keulen, "IMPrECISE: Good-is-good-enough data integration," in *2008 IEEE 24th International Conference on Data Engineering*. IEEE, apr 2008, pp. 1548–1551. [Online]. Available: http://ieeexplore.ieee.org/document/4497618/

[10] J. Huang, L. Antova, C. Koch, and D. Olteanu, "MayBMS: A probabilistic database management system," in *SIGMOD-PODS'09 - Proceedings of the International Conference on Management of Data and 28th Symposium on Principles of Database Systems*. New York, New York, USA: ACM Press, 2009, pp. 1071–1073. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1559845.1559984

[11] V. Araujo, A. Guimarães, P. Campos Souza, T. Rezende, and V. Araujo, "Using Resistin, Glucose, Age and BMI and Pruning Fuzzy Neural Network for the Construction of Expert Systems in the Prediction of Breast Cancer," *Machine Learning and Knowledge Extraction*, vol. 1, feb 2019.

[12] A. Gaier and D. Ha, "Weight Agnostic Neural Networks," jun 2019. [Online]. Available: http://arxiv.org/abs/1906.04358

[13] S. Russell and P. Norvig, *Artificial Intelligence A Modern Approach Third Edition*, 2010.

[14] Y. Lecun, Y. Bengio, and G. Hinton, "Deep learning," pp. 436–444, may 2015. [Online]. Available: http://www.nature.com/articles/nature14539

[15] P. Ramachandran, B. Zoph, and Q. V. Le, "Swish: a self-gated activation function," *arXiv preprint arXiv:1710.05941*, vol. 7, 2017.

[16] J. Lin, "Divergence Measures Based on the Shannon Entropy," *IEEE Transactions on Information Theory*, vol. 37, no. 1, pp. 145–151, 1991.

[17] H. Kamper, "Yet another introduction to backpropagation," 2017.

[18] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986. [Online]. Available: https://www.nature.com/articles/323533a0

[19] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, 2015.

[20] Apple, "Blurring an Image — Apple Developer Documentation." [Online]. Available: https://developer.apple.com/documentation/accelerate/blurring{_}an{_}image

[21] X. Guo, X. Liu, E. Zhu, and J. Yin, "Deep Clustering with Convolutional Autoencoders," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10635 LNCS. Springer Verlag, 2017, pp. 373–382.

[22] J.-P. Vert, K. Tsuda, and B. Schölkopf, "A primer on kernel methods," *Kernel methods in computational biology*, vol. 47, pp. 35–70, 2004.

[23] L. Weng, "From Autoencoder to Beta-VAE," aug 2018. [Online]. Available: https://lilianweng.github.io/lil-log/2018/08/12/from-autoencoder-to-beta-vae.html

[24] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P. A. Manzagol, "Stacked denoising autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion," *Journal of Machine Learning Research*, 2010.

[25] M. Sakurada and T. Yairi, "Anomaly detection using autoencoders with nonlinear dimensionality reduction," in *ACM International Conference Proceeding Series*, vol. 02-Decembe. Association for Computing Machinery, dec 2014, pp. 4–11.

[26] Z. Fan, D. Bi, L. He, M. Shiping, S. Gao, and C. Li, "Low-level structure feature extraction for image processing via stacked sparse denoising autoencoder," *Neurocomputing*, vol. 243, pp. 12–20, jun 2017.

[27] J. Rocca, "Understanding Variational Autoencoders (VAEs) ," sep 2019. [Online]. Available: https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73

[28] C. Doersch, "Tutorial on Variational Autoencoders," jun 2016. [Online]. Available: http://arxiv.org/abs/1606.05908

[29] F. Nijweide, "Autoencoder-based cleaning of non-categorical data in probabilistic databases," jun 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3911634

[30] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 07-12-June. IEEE Computer Society, oct 2015, pp. 1–9. [Online]. Available: https://arxiv.org/abs/1409.4842v1

[31] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, jul 2006.

[32] A. Fischer and C. Igel, "Training restricted Boltzmann machines: An introduction," *Pattern Recognition*, vol. 47, no. 1, pp. 25–39, jan 2014.

[33] G. Hinton, *Deep Belief Nets*. Boston, MA: Springer US, 2010, pp. 267–269. [Online]. Available: https://doi.org/10.1007/978-0-387-30164-8{_}208

Appendices

A Data used to generate figures

Table 4: Influence of batch size on training convergence (for unsupervised learning with CCE loss function, 50 epochs). These values were used to generate Figure 10.

Batch size	**4**	**32**	**256**	**1024**
Validation loss	0.0099	0.0101	0.0216	0.0519
Training time (seconds)	236	44	25	21

Table 5: Effect of loss function on unsupervised learning performance at sampling density 4. "Global" means that the loss function was applied to the entire output layer, while "unweighted" means that the distribution size was not used in calculating the average loss. Due to the way the JSD loss function works, applying it globally was not possible. These values were used to generate Figure 11.

Loss function	**Performance**
CCE (global)	0.22
MSE (global)	22.86
KLD (global)	-6.23
MSE per distribution	8.38
CCE per distribution	-10.07
CCE per distribution (unweighted)	-9.78
KLD per distribution	-10.00
JSD per distribution	-10.00

Table 6: Effect of training methods on performance. These values were used to generate Figure 12.

Training method	**CCEu, SD=4**	**CCE, SD=4**	**JSD, SD=4**	**CCE, SD=100**	**JSD, SD=100**	**JSD(5), SD=100**	**MSE, SD=4**	**MSE, SD=100**
Supervised	N/A	99.99	97.86	-146.94	-148.52	92.95	95.53	-150.84
Supervised (2% train, 98% test)	N/A	70.13	81.28	-106.05	-148.04	-71.42	-19.48	-149.98
Semi-supervised (unsupervised first)	N/A	95.63	96.66	68.12	-147.79	76.88	71.76	-149.11
Semi-supervised (supervised first)	N/A	-8.51	5.90	-5.98	-148.37	21.21	-4.27	-150.82
Semi-supervised (mixed)	N/A	31.01	35.94	21.06	-149.49	15.96	9.81	-150.36
Unsupervised	N/A	0.22	9.39	-4.41	-152.62	22.10	22.86	-149.87

Table 7: Effect of different activation functions on performance. These values were used to generate Figure 13.

Activation function	CCEu, SD=4	CCE, SD=4	JSD, SD=4	CCE, SD=100	JSD, SD=100	JSD(5), SD=100
Default	0.22	95.63	97.05	68.12	-147.79	75.65
ReLU	-791.58	-798.86	27.02	-52.61	-99.93	-47.46
ReLU (stacked 5 times)	-8.10	97.34	99.54	-33.36	-44.80	33.97
Sin + cos + lin	-10.24	91.96	94.39	-61.69	-147.68	47.28
Sin + cos + lin + ReLU + sigmoid	-13.60	94.09	96.28	32.79	-149.40	63.34

Table 8: Effect of different layer types on performance. These values were used to generate Figure 14.

Layer type	CCEu, SD=4	CCE, SD=4	JSD, SD=4	CCE, SD=100	JSD, SD=100	JSD(5), SD=100
Default	0.22	95.63	97.05	68.12	-147.79	75.65
Gaussian noise (σ=0.01)	-9.37	93.47	96.10	-68.97	-148.52	73.30
Gaussian dropout (σ=0.01)	-7.74	95.23	96.46	-104.72	-141.46	74.73
Sqrt followed by softmax	-12.57	95.48	96.96	-149.87	-148.10	-144.54
Gaussian kernel (100 landmarks)	-14.77	93.63	96.23	-76.45	-151.48	-25.37
Convolutional layer (64 filters, kernel size 3)	-1.55	97.12	96.16	66.07	-150.66	75.07
Sampling in middle (VAE)	-5.11	95.61	96.52	-152.29	-148.14	66.20

Table 9: Effect of the amount of Gaussian kernel landmarks on performance. These values were used to generate Figure 15.

Gaussian kernel landmarks	CCEu, SD=4	CCE, SD=4	JSD, SD=4	CCE, SD=100	JSD, SD=100	JSD(5), SD=100
3.00	-4.04	81.29	85.04	-146.75	-152.46	-55.52
10.00	-8.87	92.19	92.65	-148.67	-150.71	71.97
25.00	-14.90	89.85	96.19	-91.35	-149.25	86.99
50.00	-13.42	88.89	94.06	-149.69	-147.67	33.56
100.00	-14.77	93.63	96.23	-76.45	-151.48	-25.37
200.00	-10.15	90.89	93.42	-148.77	-150.67	45.69
500.00	-15.18	85.45	94.54	-147.89	-151.39	-147.54
1,000.00	-14.42	79.12	94.77	-147.00	-149.48	-39.66

Table 10: Effect of the convolutional layer kernel size on performance. These values were used to generate Figure 16.

Convolutional kernel size	CCEu, SD=4	CCE, SD=4	JSD, SD=4	CCE, SD=100	JSD, SD=100	JSD(5), SD=100
1.00	-489.76	-436.17	-363.77	-151.04	-150.47	-19.08
3.00	-1.55	97.12	96.16	66.07	-150.66	75.24
5.00	-4.01	96.29	95.96	-152.66	-150.08	76.08
7.00	-10.00	94.02	96.82	69.43	-150.28	73.35
9.00	-6.63	90.48	96.48	69.96	-148.28	75.29
100.00	N/A	N/A	N/A	64.55	-149.76	73.79

Table 11: Effect of the amount of convolutional layer filters on performance. These values were used to generate Figure 17.

CNN filters	CCEu, SD=4	CCE, SD=4	JSD, SD=4	CCE, SD=100	JSD, SD=100	JSD(5), SD=100
16.00	-11.57	91.12	96.16	-74.09	-151.11	53.49
32.00	-0.83	93.64	74.04	70.24	-151.18	56.40
64.00	-1.55	97.12	96.16	66.07	-150.66	72.75
128.00	-4.33	95.02	96.62	-146.93	-148.63	61.20

Table 12: Effect of regularization on performance. These values were used to generate Figure 18.

Regularization	CCEu, SD=4	CCE, SD=4	JSD, SD=4	CCE, SD=100	JSD, SD=100	JSD(5), SD=100
Default	0.22	95.63	97.05	68.12	-147.79	75.65
L2: 0.01	-23.83	72.03	57.70	-72.34	-139.32	74.90
L2: 10^-4	-9.65	89.18	87.59	26.83	-134.98	60.13
L1: 0.01	-11.04	57.94	37.66	-144.13	-145.08	-103.18
L1: 10^-4	-15.54	88.90	85.51	42.59	30.12	68.59

Table 13: Effect of latent space dimensionality on performance. These values were used to generate Figure 19.

Latent space dimensionality	CCEu, SD=4	CCE, SD=4	JSD, SD=4	CCE, SD=100	JSD, SD=100	JSD(5), SD=100
6.00	-11.69	95.35	96.27	-147.55	-149.22	29.21
3.00	-4.29	95.63	95.60	68.12	-150.91	75.65
2.00	-8.32	95.84	97.02	-146.84	-150.13	75.42

Table 14: Effect of network deepness on performance. These values were used to generate Figure 20.

Hidden layers	CCEu, SD=4	CCE, SD=4	JSD, SD=4	CCE, SD=100	JSD, SD=100	JSD(5), SD=100
3.00	-9.73	93.40	94.56	66.13	71.76	N/A
5.00	-8.34	91.89	95.74	73.71	75.65	N/A
7.00	-9.99	93.31	96.06	-11.05	-19.98	N/A
9.00	-10.37	95.63	94.67	68.82	-147.42	N/A
27.00	-18.17	70.25	-533.18	-147.47	-147.44	N/A

Table 15: Effect of standard deviation of noise on performance. These values were used to generate Figure 21.

Noise standard deviation	CCEu, SD=4	CCE, SD=4	JSD, SD=4	CCE, SD=100	JSD, SD=100	JSD(5), SD=100
σ=0.01/SD	-10.59	94.45	92.13	-2,123.44	-2,115.61	67.11
σ=0.01*4/SD	-13.46	95.63	95.91	-1,028.23	-1,013.25	69.76
σ=0.01/sqrt(SD)	-15.92	95.30	96.26	-608.25	-614.61	94.19
σ=0.01 (default)	-10.59	95.63	95.91	68.12	-149.76	75.65
σ=0.05	-4.47	96.12	95.88	-46.41	-45.30	48.89
σ=0.1	-1.75	95.50	96.24	-26.84	-25.50	27.12
σ=0.2	-2.99	94.60	93.88	-14.25	-14.18	10.23

Table 16: Effect of the amount of labeled data on performance. These values were used to generate Figure 22.

Amount of labeled data (%)	CCEu, SD=4	CCE, SD=4	JSD, SD=4	CCE, SD=100	JSD, SD=100	JSD(5), SD=100
5.00	N/A	97.16	98.24	54.31	-150.75	87.84
2.00	N/A	94.55	97.05	68.82	-147.79	75.65
1.00	N/A	91.67	95.48	-62.24	-150.45	58.40
0.50	N/A	86.63	90.41	31.72	-149.54	32.67
0.25	N/A	79.58	59.72	13.55	-148.63	16.57
0.125	N/A	15.24	78.06	8.07	-151.74	-12.01

Table 17: Effect of sampling density on performance. Despite multiple attempts, training does not converge to a network which produces proper probability distributions for JSD(5) at SD=300. These values were used to generate Figure 23.

Sampling density	CCEu	CCE	JSD	JSD(5)
4.00	-14.34	95.63	97.05	93.30
15.00	-1.24	93.34	87.38	91.40
25.00	-0.95	92.06	72.79	92.15
50.00	-0.88	84.12	83.80	87.16
100.00	-0.95	68.82	-147.79	75.65
150.00	-114.31	-114.98	-113.07	50.30
300.00	-69.68	-69.56	-69.38	N/A

Table 18: Effect of Bayesian network size on performance. Higher values for sampling density 100 would have taken several days to train, which was aborted due to low performance. These values were used to generate Figure 24.

Amount of BN variables	CCEu, SD=4	CCE, SD=4	JSD, SD=4	CCE, SD=100	JSD, SD=100	JSD(5), SD=100
2.00	-5.78	96.02	98.60	79.80	93.72	88.88
3.00	-3.01	94.55	95.60	68.12	-148.29	75.65
4.00	-12.39	93.82	94.52	-147.93	-148.55	2.42
5.00	-12.42	81.11	95.12	-148.11	-148.16	-27.22
10.00	-14.58	87.96	88.69	No convergence	No convergence	No convergence
20.00	-332.45	-96.60	-18.93	No convergence	No convergence	No convergence
30.00	-337.28	-335.97	-317.02	No convergence	No convergence	No convergence