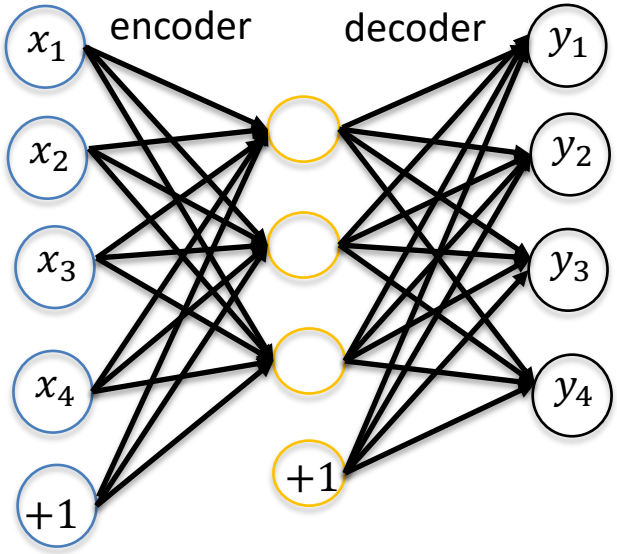
Chapter 10

Autoencoders

Neural networks and deep learning

Autoencoders



An autoencoder is a neural network that is trained to attempt to copy its input to its output. It's hidden layer describes a *code* that represents the input.

The network consists of two parts: an encoder and decoder.

Autoencoders

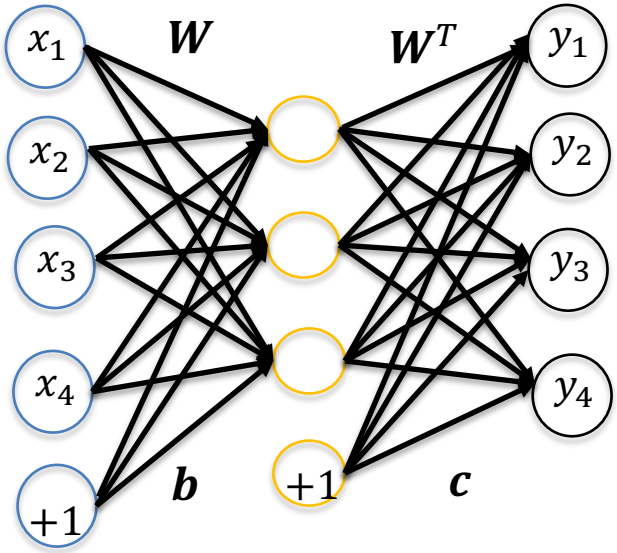
Given an input $\boldsymbol{x}$, the hidden-layer performs the encoding function $\boldsymbol{h} = \phi(\boldsymbol{x})$ and the decoder φ produces the reconstruction $\boldsymbol{y} = \varphi(\boldsymbol{h})$.

If the autoencoder succeeds:

$$\boldsymbol{y} = \varphi(\boldsymbol{h}) = \varphi\big(\phi(\boldsymbol{x})\big) = \boldsymbol{x}$$

In order to be useful, autoencoders are designed to be unable to copy exactly and enable to copy only inputs that resembles the training data. Since the model is forced to prioritize which aspects of the input should be copied, the hidden-layer often learns useful properties of the data.

Autoencoders



Reverse mapping from the hidden layer to the output can be optionally constrained to be the same as the input to hidden-layer mapping.
That is, if encoder weight matrix is $\boldsymbol{W}$, the decoder weigh matrix is $\boldsymbol{W}^T$.

Hidden layer and output layer activation can be then written as

$$\boldsymbol{h} = f(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b})$$
$$\boldsymbol{y} = f(\boldsymbol{W}\boldsymbol{h} + \boldsymbol{c})$$

f is usually a sigmoid.

Training autoencoders

The cost function of reconstruction can be measured by many ways, depending on the appropriate distributional assumptions on the inputs.

Learning of autoencoders is unsupervised as no specific targets are given.

Given a training set $\{\boldsymbol{x}_p\}_{p=1}^{P}$.

The *mean-square-error* cost is usually used if the data is assumed to be continuous and Gaussian distributed:

$$J_{mse} = \frac{1}{P}\sum_{p=1}^{P}\|\boldsymbol{y}_p - \boldsymbol{x}_p\|^2$$

where $\boldsymbol{y}_p$ is the output for input $\boldsymbol{x}_p$ and $\|\cdot\|$ denotes the magnitude of the vector.

Training autoencoders

If the inputs are interpreted as bit vectors or vectors of bit probabilities, *cross-entropy* of the reconstruction can be used:

$$J_{cross-entropy} = -\sum_{p=1}^{P} \left(\boldsymbol{x}_p \log \boldsymbol{y}_p + \left(1 - \boldsymbol{x}_p\right) \log\left(1 - \boldsymbol{y}_p\right)\right)$$

Learning are done by using gradient descent:

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \nabla_{\boldsymbol{W}} J$$
$$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha \nabla_{\boldsymbol{b}} J$$
$$\boldsymbol{c} \leftarrow \boldsymbol{c} - \alpha \nabla_{\boldsymbol{c}} J$$

Example 1

Given input patterns:
$(1,0,1,0,0), (0,0,1,1,0), (1,1,0,1,1)$ and $(0,1,1,1,0)$

Design an autoencoder with 3 hidden units.
Show the representations of inputs at the hidden layer upon convergence.

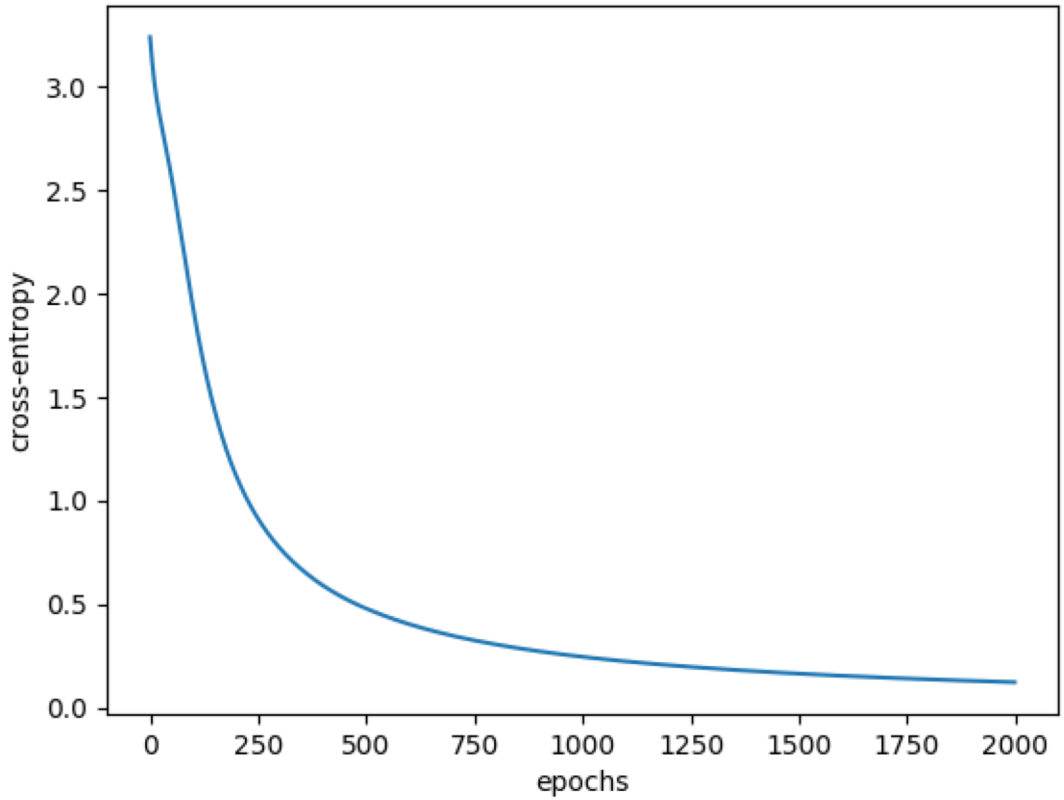
$$\boldsymbol{H} = f(\boldsymbol{XW} + \boldsymbol{B})$$
$$\boldsymbol{Y} = f(\boldsymbol{HW}^T + \boldsymbol{C})$$

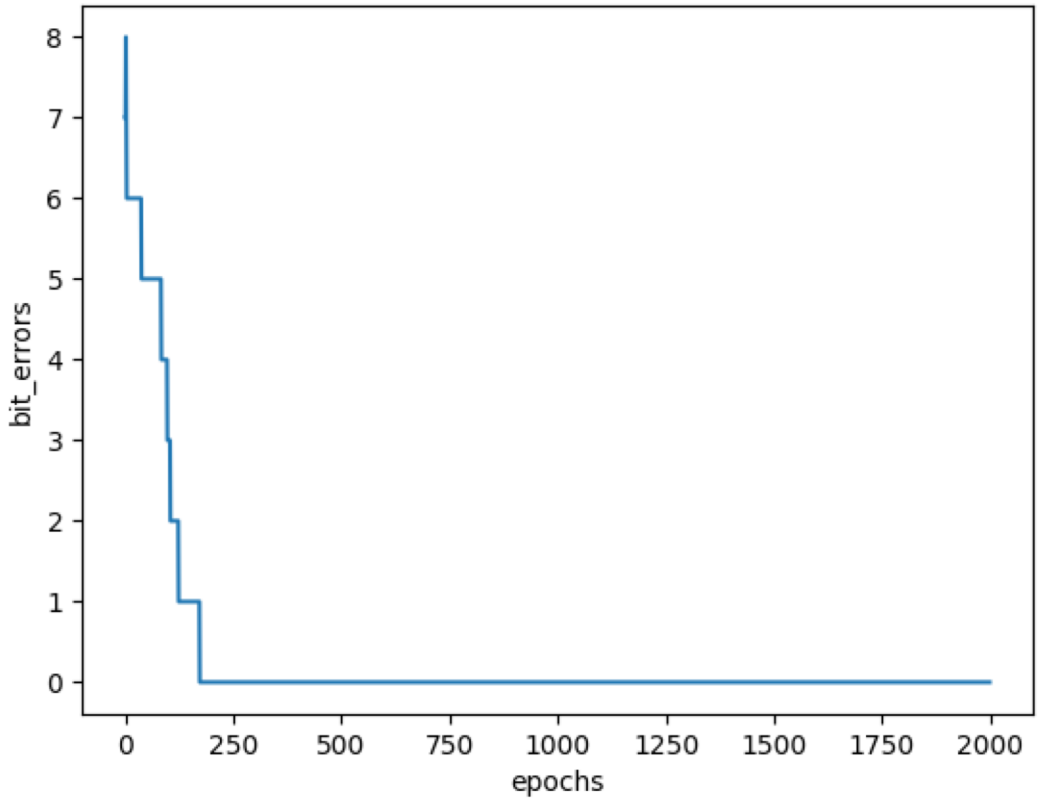
$$\boldsymbol{O} = 1(\boldsymbol{Y} > 0.5)$$

Gradient descent on entropy:

$$J = -\sum_{p=1}^{4} \left(\boldsymbol{x_p} \log \boldsymbol{y}_p + \left(1 - \boldsymbol{x}_p\right) \log\left(1 - \boldsymbol{y}_p\right)\right)$$

$$bit\ errors = \sum_{p=1}^{4} \sum_{k=1}^{5} 1\left(x_{pk} = o_{pk}\right)$$


cross-entropy
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
250
500
750
1000
1250
1500
1750
2000
epochs


bit_errors
epochs
0
1
2
3
4
5
6
7
8
0
250
500
750
1000
1250
1500
1750
2000

At convergence:

$$\boldsymbol{W} = \begin{pmatrix} 1.17 & -7.53 & -0.66 \\ -6.41 & 1.14 & -2.75 \\ 3.20 & 3.23 & 5.28 \\ -2.70 & 5.62 & -3.11 \\ -3.04 & -3.93 & -4.31 \end{pmatrix}, \boldsymbol{b} = \begin{pmatrix} 1.42 \\ -1.10 \\ 1.52 \end{pmatrix}, \boldsymbol{c} = \begin{pmatrix} 3.38 \\ 3.93 \\ -3.05 \\ 3.06 \\ 2.95 \end{pmatrix}$$

x	h	y	o
$(1, 0, 1, 0, 0)$	$(1.0, 0.05, 1.0)$	$(0.98, 0.01, 1.0\ , 0.06, 0.01)$	$(1, 0, 1, 0, 0)$
$(0, 0, 1, 1, 0)$	$(0.87, 1.0, 0.97)$	$(0.00, 0.00, 1.0\ , 0.96, 0.0)$	$(0, 0, 1, 1, 0)$
$(1, 1, 0, 1, 1)$	$(0.0, 0.0, 0.0)$	$(0.97\ 0.98, 0.05\ , 0.95, 0.95)$	$(1, 1, 0, 1, 1)$
$(0, 1, 1, 1, 0)$	$(0.01, 1.0, 0.71)$	$(0.01, 0.95\ 0.98\ , 1.0, 0.01)$	$(0, 1, 1, 1, 0)$

Code – the latent representations of inputs

Denoising Autoencoders (DAE)

The *denoising autoencoders* (DAE) receives corrupted data points as inputs and is trained to predict the original uncorrupted data points as its output.

The idea of DAE is that in order to force the hidden layer to discover more robust features and prevent it from simply learning the identity. We train the autoencoder to reconstruct the input from a corrupted version of it.

In other words, DAE attempts to encode the input (preserve the information about input) and attempts to undo the effect of corruption process applied to the input of the autoencoder.

DAE

For training DAE:

- First, input data is corrupted to mimic the noise in the images: $\boldsymbol{x} \rightarrow \tilde{\boldsymbol{x}}$

  $\tilde{\boldsymbol{x}}$ is the corrupted version of data. The corruption process simulates the distribution of data

- The network is trained to produce uncorrupted data: $\boldsymbol{y} \rightarrow \boldsymbol{x}$

For a set $\{\boldsymbol{x}_p\}$ of training patterns:

$$\boldsymbol{x}_p \leftarrow \widetilde{\boldsymbol{x}_p}$$
$$\boldsymbol{d}_p \leftarrow \boldsymbol{x}_p$$

DAE: Corrupting inputs

To obtain corrupted version of input data, each input x_i of input data is added with additive or multiplicative noise.

Additive noise:

$$\widetilde{x_i} = x_i + \varepsilon$$

where noise ε is Gaussian distributed: $\varepsilon \sim N(0, \sigma^2)$
And σ is the standard deviation that determines the S/N ratio. Usually used for continuous data.

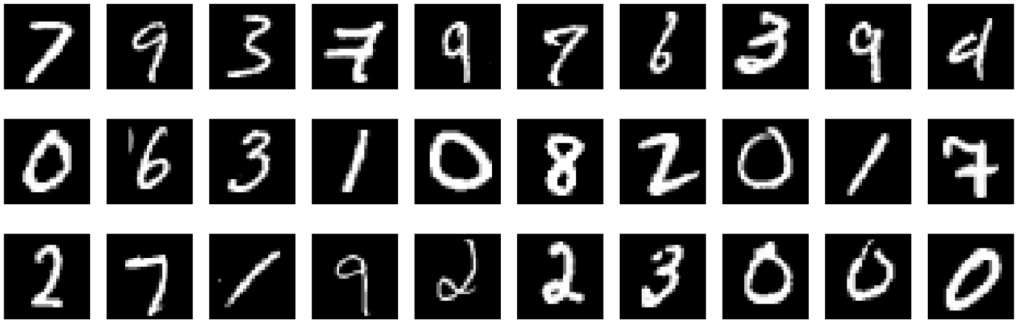
Multiplicative noise:

$$\widetilde{x_i} = \varepsilon x_i$$

where noise ε could be Binomially distributed: $\varepsilon \sim Binomial(p)$
And p is the probability of ones and $1 - p$ is the probability of zeros (noise).
Usually, used for binary data.

Example 2: Denoising autoencoders

The MNIST database of gray level images of handwritten digits:
http://yann.lecun.com/exdb/mnist/

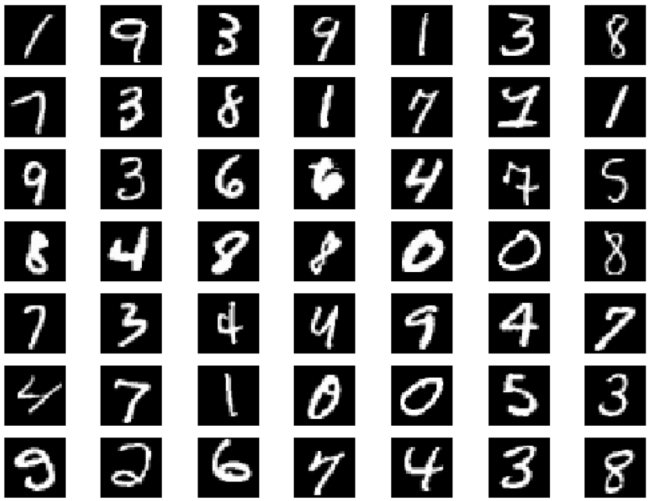


Each image is 28x28 size.
Intensities are in the range [0, 255] and normalized to [0, 1].
Training set: 60,000, Test set: 10,000

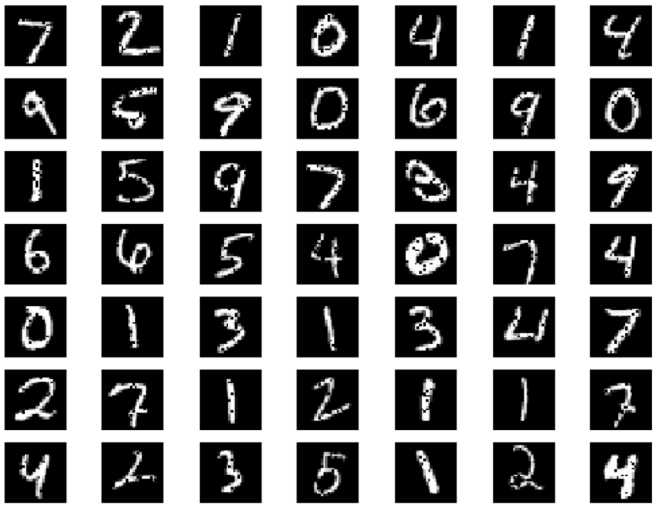
To build a DAE with 500 hidden units.

Example 2a: DAE (multiplicative noise)

Original data

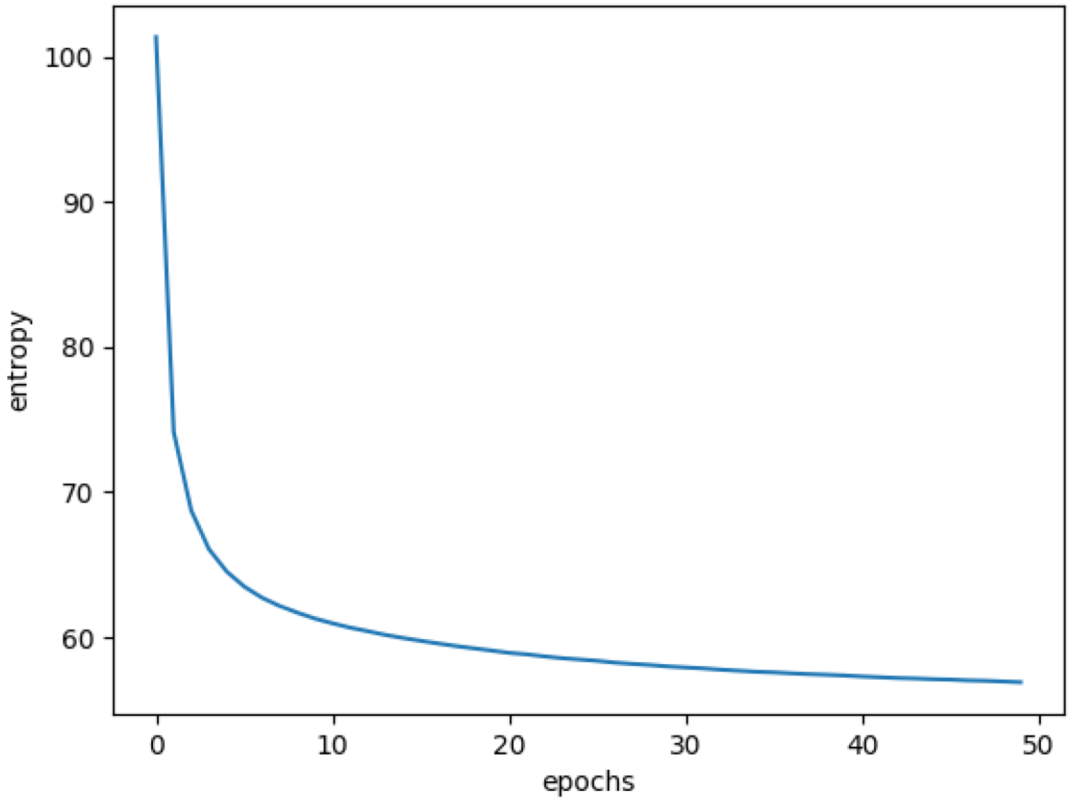


Corrupted data



DAE attempts to predict the input data from corrupted input data with multiplicative noise with binomial distribution.

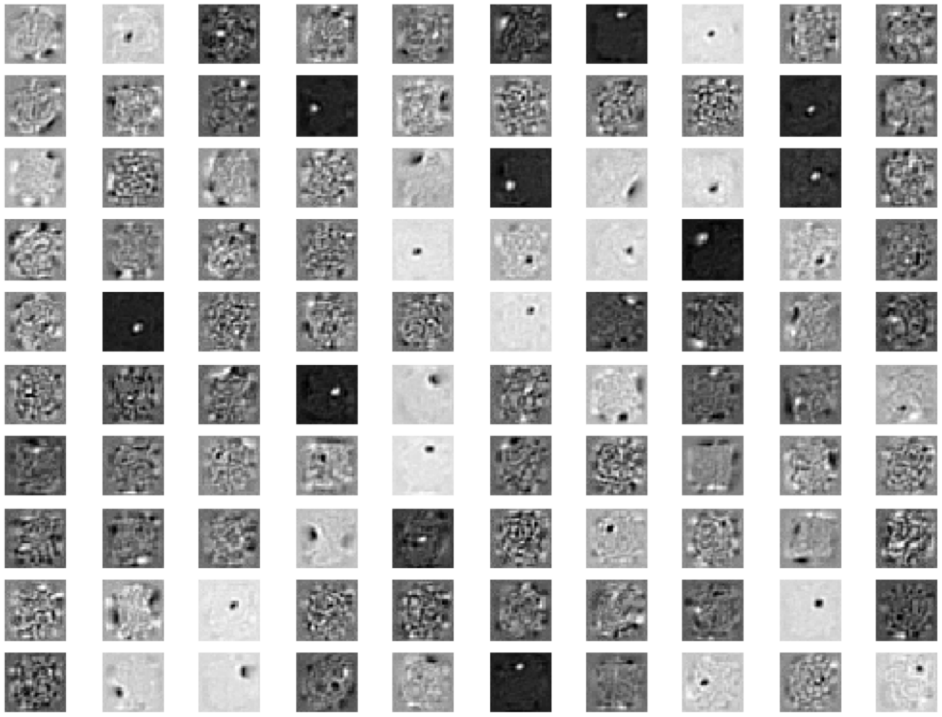
Corruption level $1 - p$ = 10%



500 hidden units

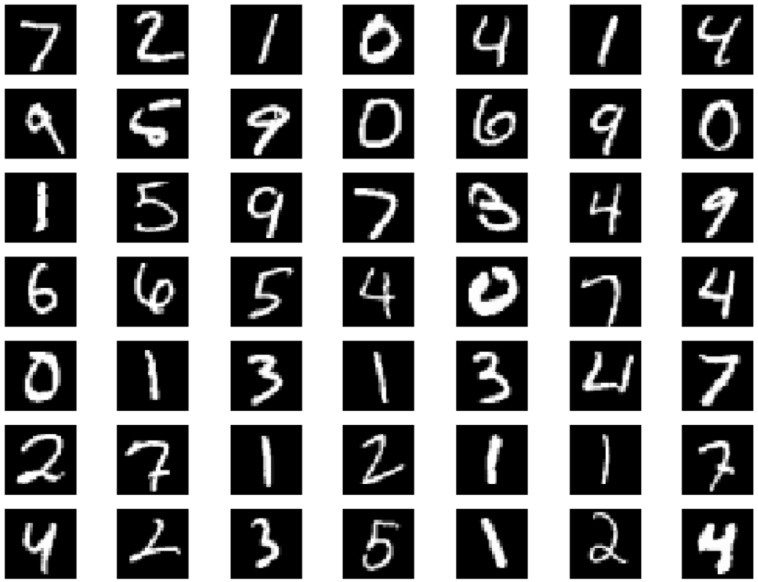
$\alpha = 0.1$

Features (weights) learned by the hidden-layer

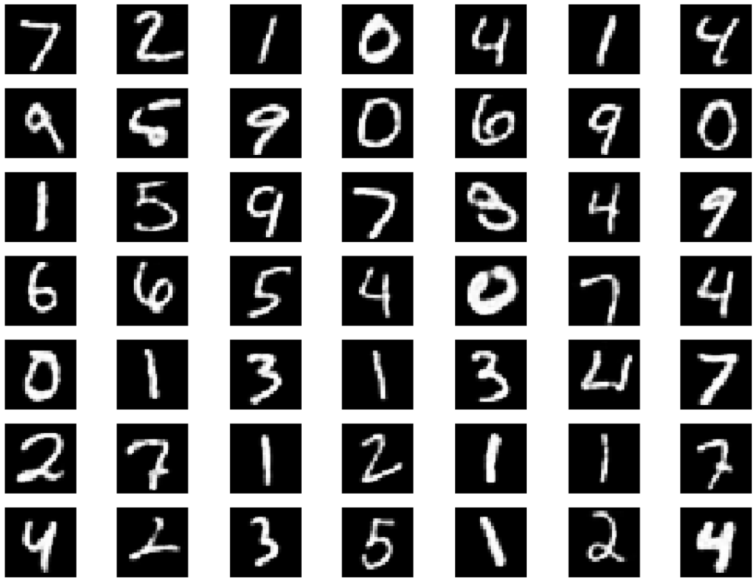


Sample of reconstructed test images

Input data

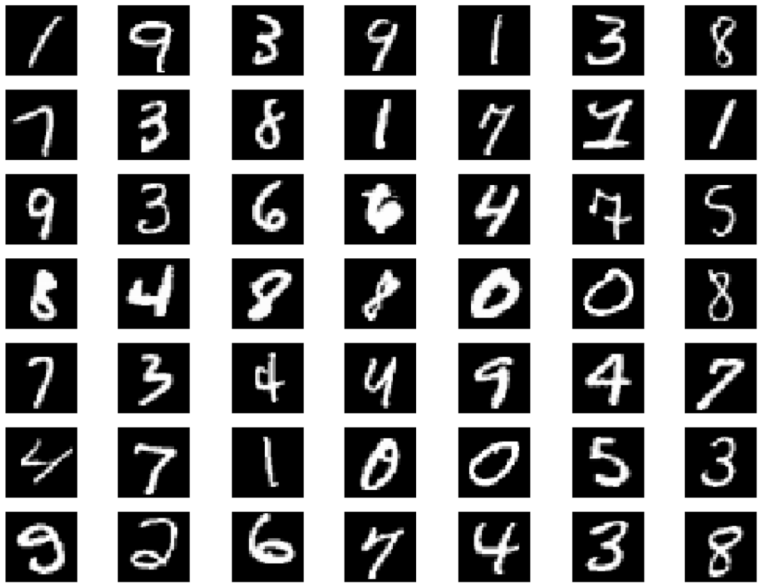


Reconstructed (noise-filtered) data

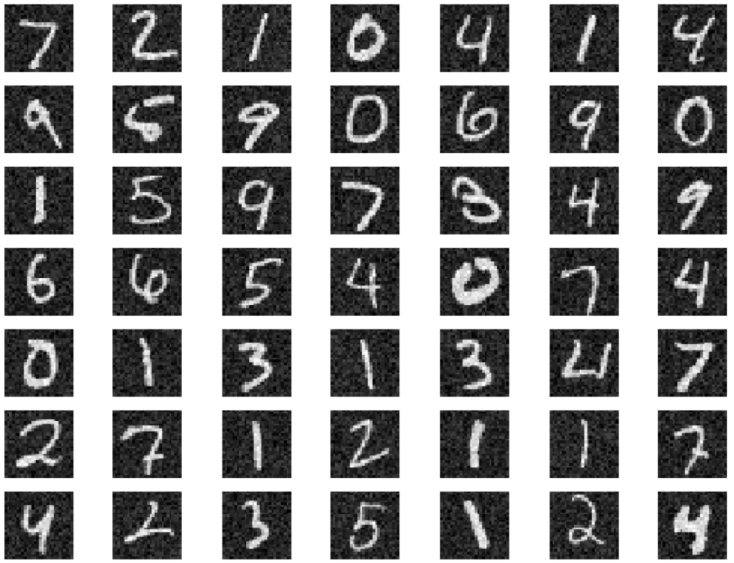


Example 2b: DAE (additive noise)

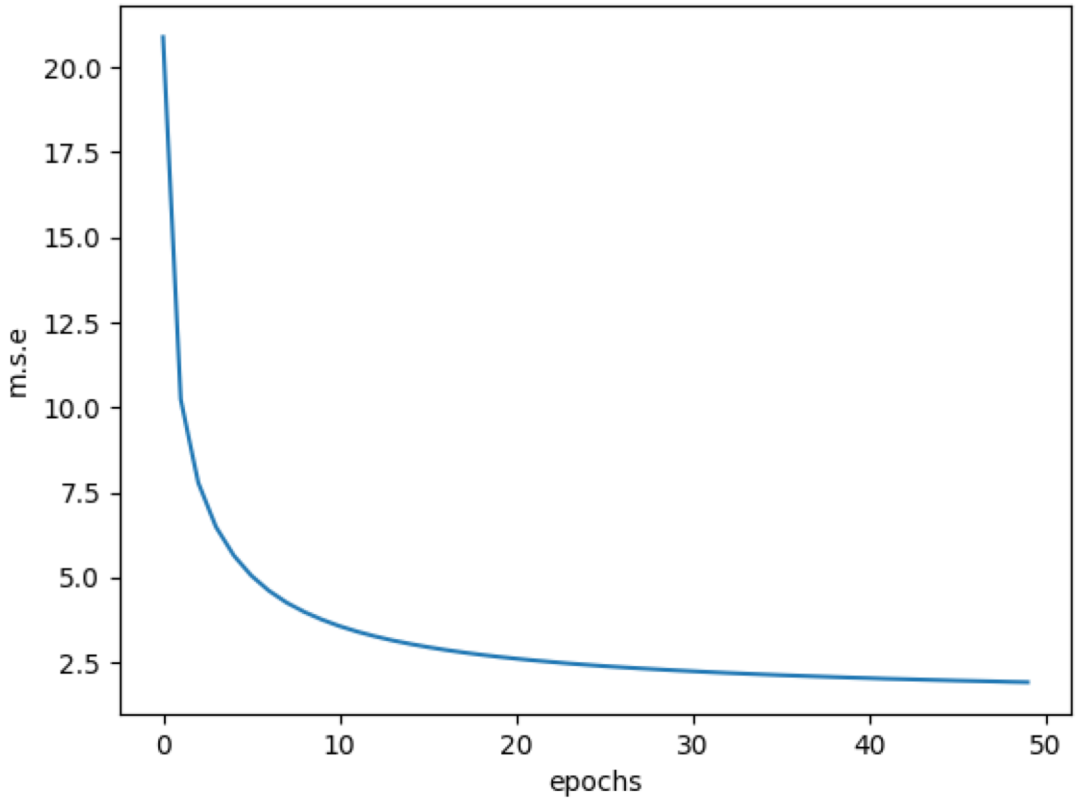
Original data



Corrupted data



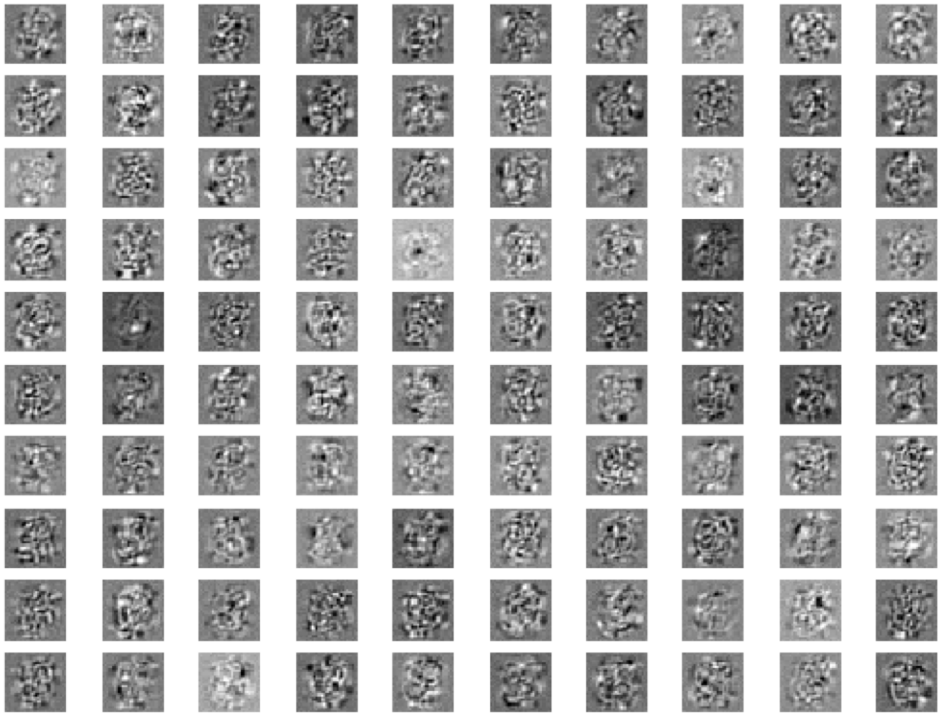
Additive noise with gaussian distribution.
Data is scaled between $[0, 1]$, $noise$ s.d = 0.1



500 hidden units

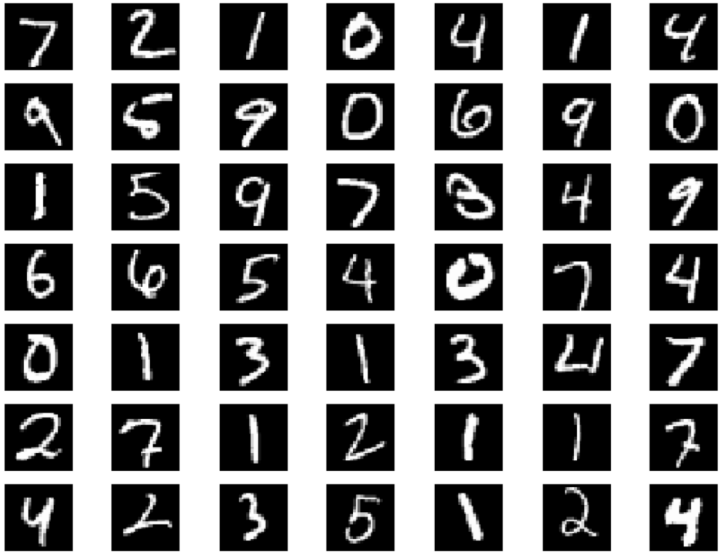
$\alpha = 0.1$

Features (weights) learned by the hidden-layer

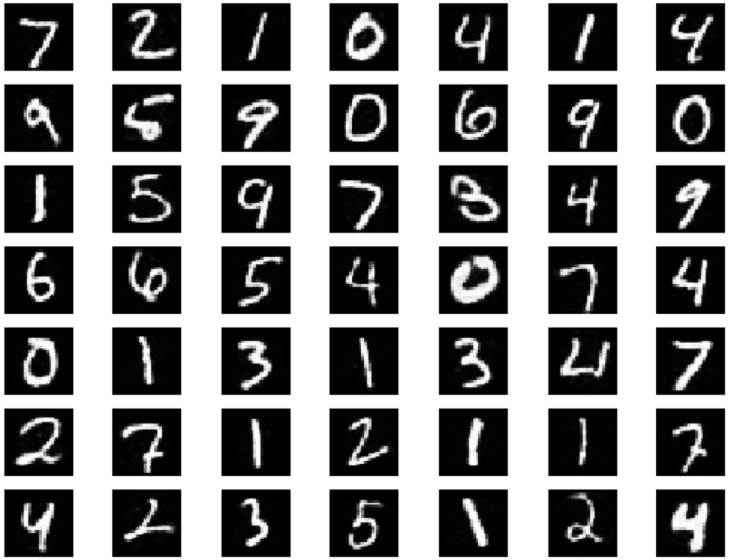


Sample of reconstructed images

Input data



Reconstructed (noise-filtered) data



Autoencoders

Input dimension n and hidden dimension M:

If $M < n$, *undercomplete* autoencoders

If $M > n$, *overcomplete* autoencoders

Undercomplete autoencoders

In undercomplete autoencoders, the hidden-layer has a lower dimension than the input layer.

By leaning to approximate an n-dimensional inputs with M ($< n$) number of hidden units, we obtain *a lower dimensional representation* of the input signals. The network reconstructs the input signals from the hidden reduced dimensional representation.

Learning an undercomplete representation forces the autoencoder to capture the most salient features. By limiting the number of hidden neurons, interesting structures of input data can be inferred from autoencoders. Autoencoders are thus capable of learning hidden structures of the data: for example, correlations among input variables, learning principal components of data, etc.

Overcomplete autoencoders

In overcomplete autoencoders, the hidden-layer has a higher dimension than the dimension of the input.

In order to learn useful information from overcomplete autoencoders, it is necessary use some constraints on its characteristics. Even when the hidden dimensions are large, one explore interesting structures of inputs by introducing other constraints such as ‘sparsity’ of input data.

Regularizing autoencoders

Both undercomplete and overcomplete autoencoders fail to learn anything useful if the encoder and decoder are given too much capacity. Some form of constraints are needed in order to make them useful. Deep autoencoders are only possible when some constraints are imposed on the cost function.

Regularized autoencoders incorporate a penalty to the cost function to learn interesting features from the input.

Regularized autoencoders provide the ability to train any autoencoder architecture successfully by choosing suitable code dimension and the capacity of the encoder and decoder. With regularized autoencoders, one can use larger model capacity (for example, deeper autoencoders) and large code size.

Regularizing autoencoders

Regularized autoencoders add an appropriate penalty function Ω to the cost function:

$$J_1 = J + \beta\Omega(\boldsymbol{h})$$

where β is the penalty or regularization parameter. The penalty is usually imposed on the hidden activations.

A regularized autoencoder can be nonlinear and overcomplete but still learn something useful about the data distribution, even if the model capacity is great enough to learn trivial identity function. The regularized loss function encourages the model to have other properties besides the ability to copy its input to its output.

Sparse Autoencoders (SAE)

A sparse autoencoder (SAE) is simply an autoencoder whose training criterion involves the sparsity penalty $\Omega_{sparsity}$ at the hidden layer:

$$J_1 = J + \beta \Omega_{sparsity}(\boldsymbol{h})$$

The sparsity penalty term makes the features (weights) learnt by the hidden-layer to be sparse.

With the sparsity constraint, one would constraint the neurons at the hidden layers to be inactive for most of the time. We say that the neuron is “active” when its output is close to 1 and the neuron is “inactive” when its output is close to 0.

Sparsity constraint

For a set $\{\boldsymbol{x}_p\}_{p=1}^{P}$ of input patterns, the average activation ρ_j of neuron j at the hidden-layer is given by:

$$\rho_j = \frac{1}{P}\sum_{p=1}^{P} h_{pj} = \frac{1}{P}\sum_{p=1}^{P} f\left(\boldsymbol{x}_p^T \boldsymbol{w}_j + b_j\right)$$

where h_{pj} is the activation of hidden neuron j for p th pattern, and $\boldsymbol{w}_j$ and b_j are the weights and biases of the hidden neuron j.

We would like to enforce the constraint: $\rho_j = \rho$ such that the *sparsity parameter* ρ is set to a small value close to zero (say 0.05).

That is, most of the time, hidden neuron activations are maintained at ρ on average. By choosing a smaller value for ρ, the neurons are activated selectively to patterns and thereby learn sparse features.

Sparsity constraint

To achieve sparse activations at the hidden-layer, the Kullback-Leibler (KL) divergence is used as the sparsity constraint:

$$D(\boldsymbol{h}) = \sum_{j=1}^{M} \rho \log \frac{\rho}{\rho_j} + (1-\rho) \log \frac{1-\rho}{1-\rho_j}$$

where M is the number of hidden neurons and ρ is the sparsity parameter.

KL divergence measures the deviation of the distribution $\{\rho_j\}$ of activations at the hidden-layer from the uniform distribution of ρ.

The KL divergence is minimum when $\rho_j = \rho$ for all j.
That is, when the average activations are uniform and equal to very low value ρ.

Sparse Autoencoder (SAE)

The cost function for the sparse autoencoder (SAE) is given by

$$J_1 = J + \beta D(\boldsymbol{h})$$

where $D(\boldsymbol{h})$ is the KL divergence of hidden-layer activations:

$$D(\boldsymbol{h}) = \sum_{j=1}^{M} \rho \log \frac{\rho}{\rho_j} + (1-\rho) \log \frac{1-\rho}{1-\rho_j}$$

For gradient descent,

$$\nabla_{\boldsymbol{W}} J_1 = \nabla_{\boldsymbol{W}} J + \beta \nabla_{\boldsymbol{W}} D(\boldsymbol{h})$$

SAE

By chain rule:

$$\frac{\partial D(\boldsymbol{h})}{\partial \boldsymbol{w}_j} = \frac{\partial D(\boldsymbol{h})}{\partial \rho_j}\frac{\partial \rho_j}{\partial \boldsymbol{w}_j} \tag{A}$$

where

$$D(\boldsymbol{h}) = \sum_{j=1}^{M} \rho \log\frac{\rho}{\rho_j} + (1-\rho)\log\frac{1-\rho}{1-\rho_j}$$

and

$$\frac{\partial D(\boldsymbol{h})}{\partial \rho_j} = \rho\frac{1}{\frac{\rho}{\rho_j}}\left(-\frac{\rho}{{\rho_j}^2}\right) + (1-\rho)\frac{1}{\left(\frac{1-\rho}{1-\rho_j}\right)}\left(\frac{-(1-\rho)}{\left(\frac{1-\rho}{1-\rho_j}\right)^2}\right)$$

$$= -\frac{\rho}{\rho_j} + \frac{1-\rho}{1-\rho_j} \tag{B}$$

SAE

Note:

$$\rho_j = \frac{1}{P}\sum_{p=1}^{P} f\left(u_{pj}\right)$$

where $u_{pj} = \boldsymbol{x}_p^T \boldsymbol{w}_j + b_j$ is the synaptic input of the j th neuron due to p th pattern.

$$\frac{\partial \rho_j}{\partial \boldsymbol{w}_j} = \frac{1}{P}\sum_{p} f'\left(\left(u_{pj}\right)\right)\frac{\partial\left(u_{pj}\right)}{\partial \boldsymbol{w}_j} = \frac{1}{P}\sum_{p} f'\left(\left(u_{pj}\right)\right)\boldsymbol{x}_p \qquad \text{(c)}$$

Substituting (B) and (C) in (A):

$$\frac{\partial D(\boldsymbol{h})}{\partial \boldsymbol{w}_j} = \frac{1}{P}\left(\frac{\rho}{\rho_j} + \frac{1-\rho}{1-\rho_j}\right)\sum_{p} f'\left(\left(u_{pj}\right)\right)\boldsymbol{x}_p$$

SAE

Substituting in (A) , we can find:

$$\nabla_{\boldsymbol{W}} D(\boldsymbol{h}) = (\nabla_{\boldsymbol{w}_1} D(\boldsymbol{h}) \quad \nabla_{\boldsymbol{w}_1} D(\boldsymbol{h}) \quad \cdots \quad \nabla_{\boldsymbol{w}_M} D(\boldsymbol{h}))$$

The gradient of the constrained cost function:

$$\nabla_{\boldsymbol{W}} J_1 = \nabla_{\boldsymbol{W}} J + \beta \nabla_{\boldsymbol{W}} D(\boldsymbol{h})$$

Similarly, $\nabla_{\boldsymbol{b}} J_1$ and $\nabla_{\boldsymbol{c}} J_1$ can be derived.

Learning can be done as

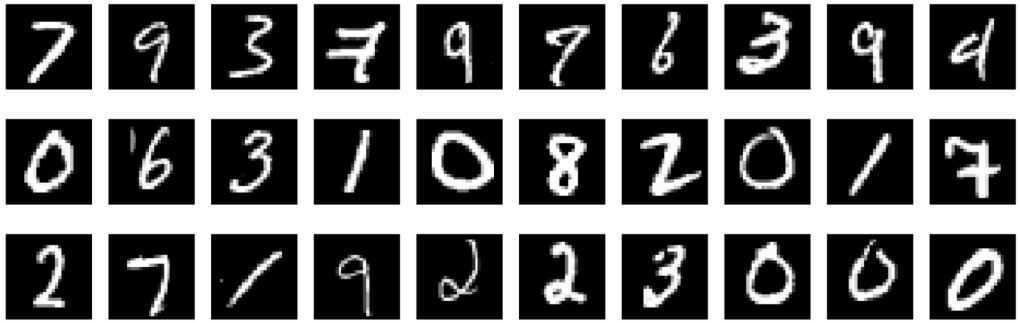
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \nabla_{\boldsymbol{W}} J_1$$
$$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha \nabla_{\boldsymbol{b}} J_1$$
$$\boldsymbol{c} \leftarrow \boldsymbol{c} - \alpha \nabla_{\boldsymbol{c}} J_1$$

Example 3

Design the following autoencoder to process MNIST images:

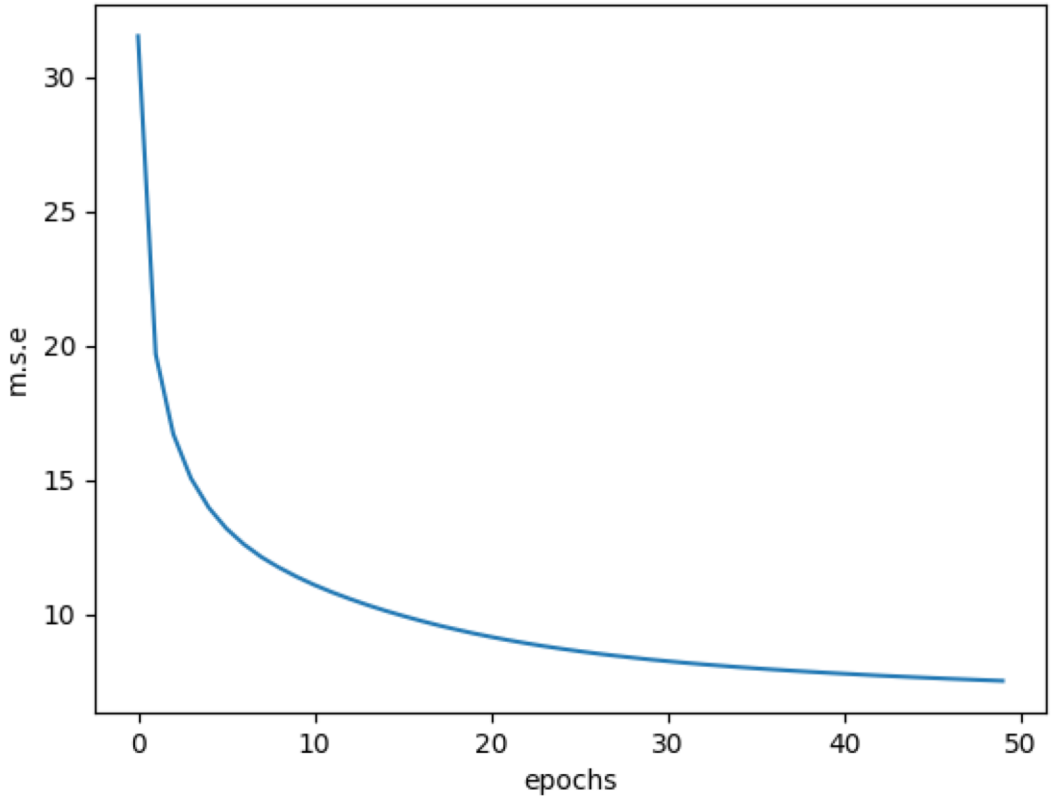
1. Undercomplete autoencoder with 100 hidden units
2. Overcomplete autoencoder with 900 hidden units
3. Sparse autoencoder with 900 hidden units. Use sparsity parameter = 0.5

The MNIST database of gray level images of handwritten digits:
http://yann.lecun.com/exdb/mnist/

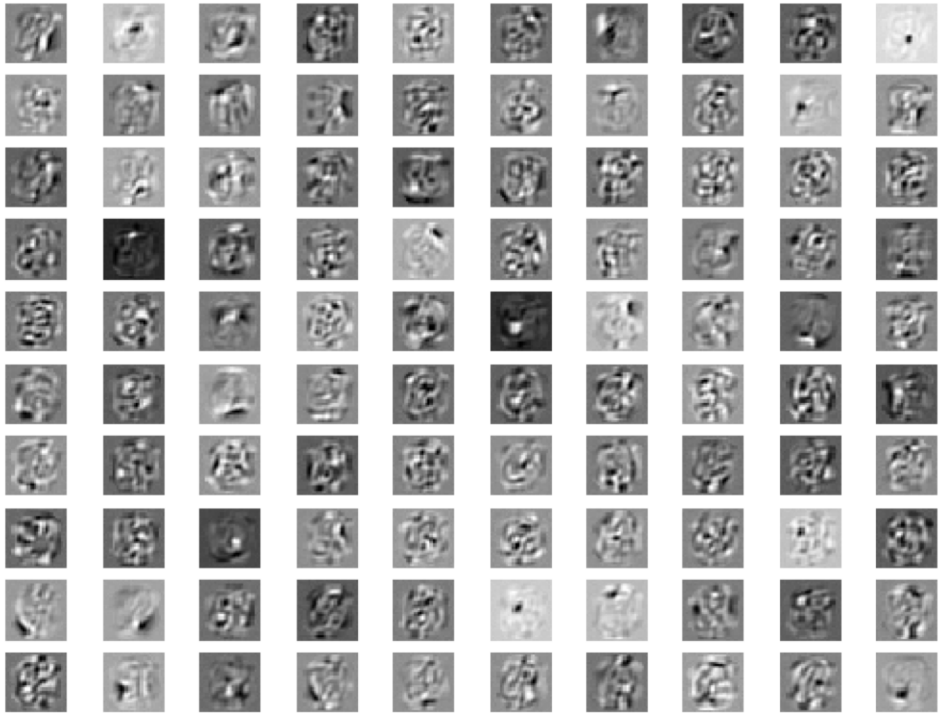


Each image is 28x28 size.
Intensities are in the range [0, 255].
Training set: 60,000
Test set: 10,000

Example 3a: Undercomplete autoencoder

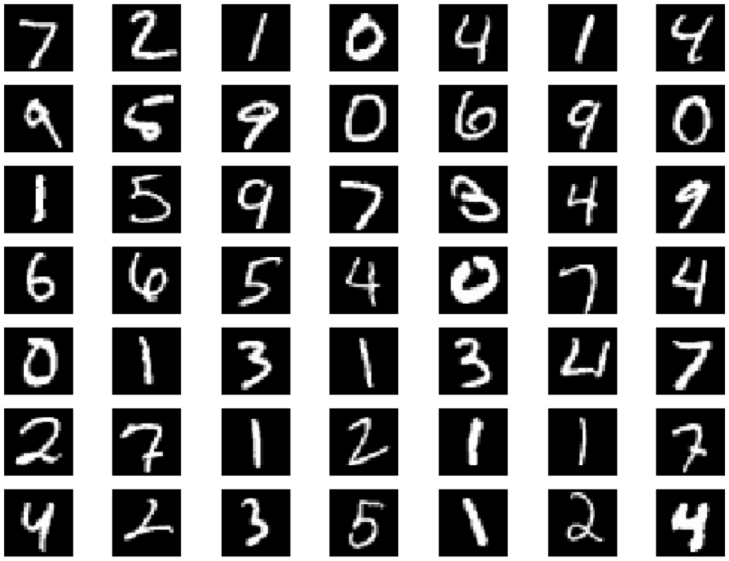


Learned weights

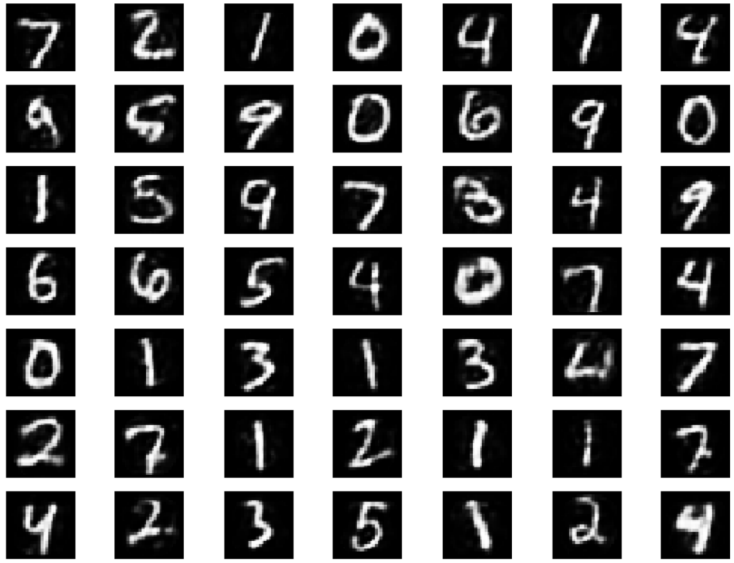


100 hidden units

Test inputs



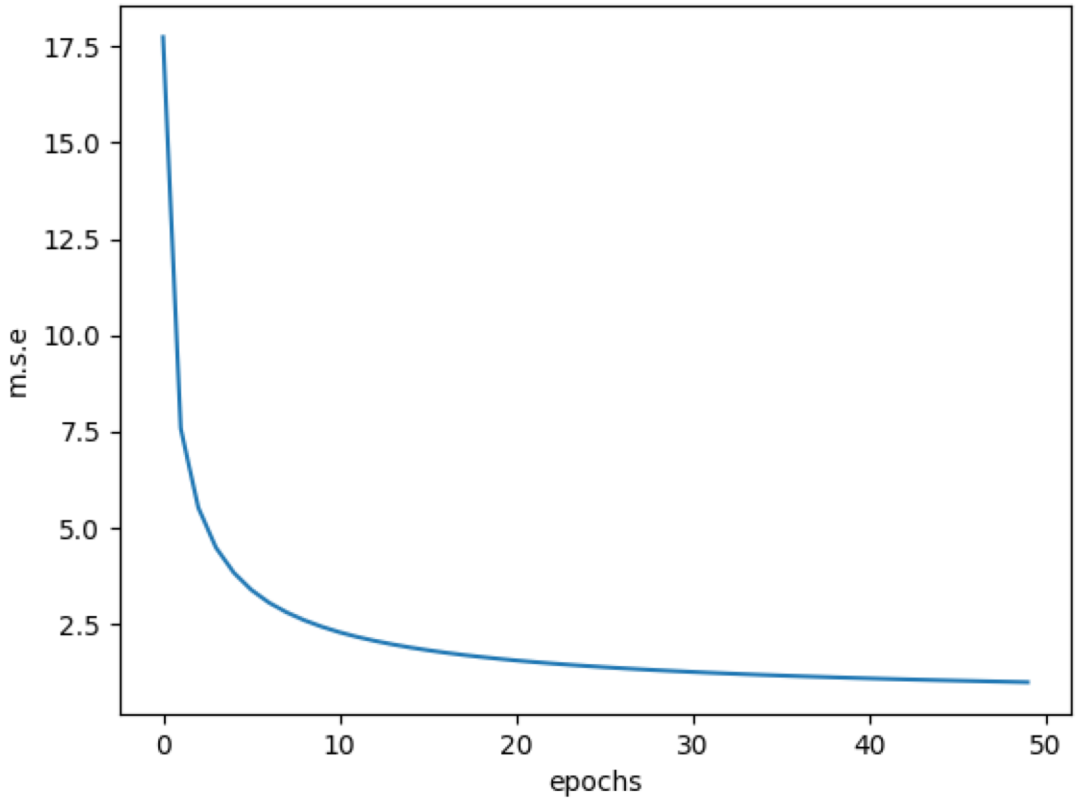
Reconstructed inputs



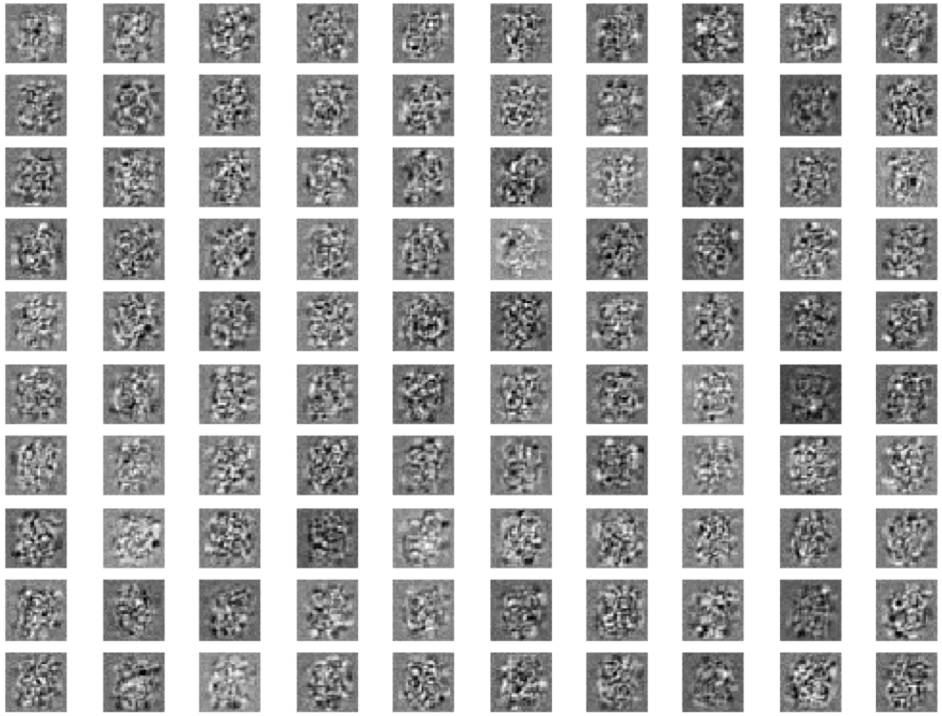
Hidden layer activations of 49 test patterns

10x10 neurons

Example 3: Overcomplete autoencoder

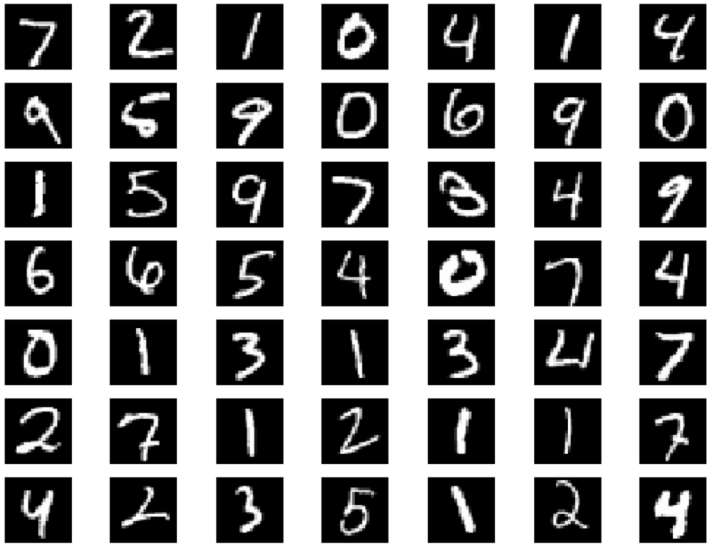


Learned weights

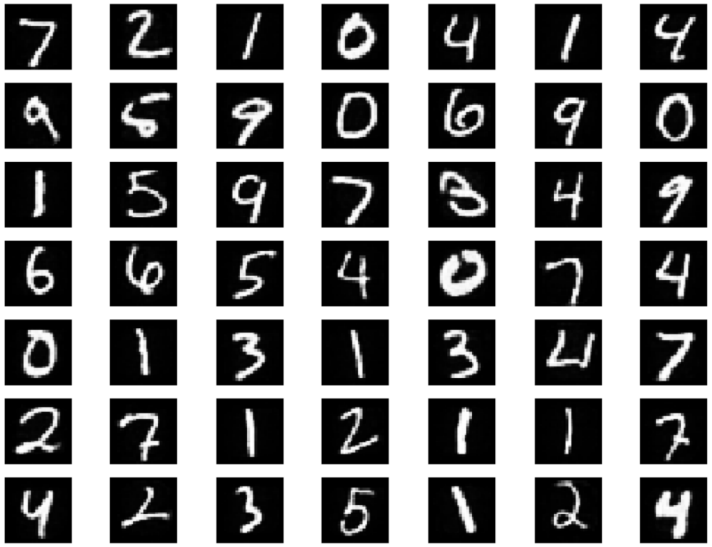


900 units

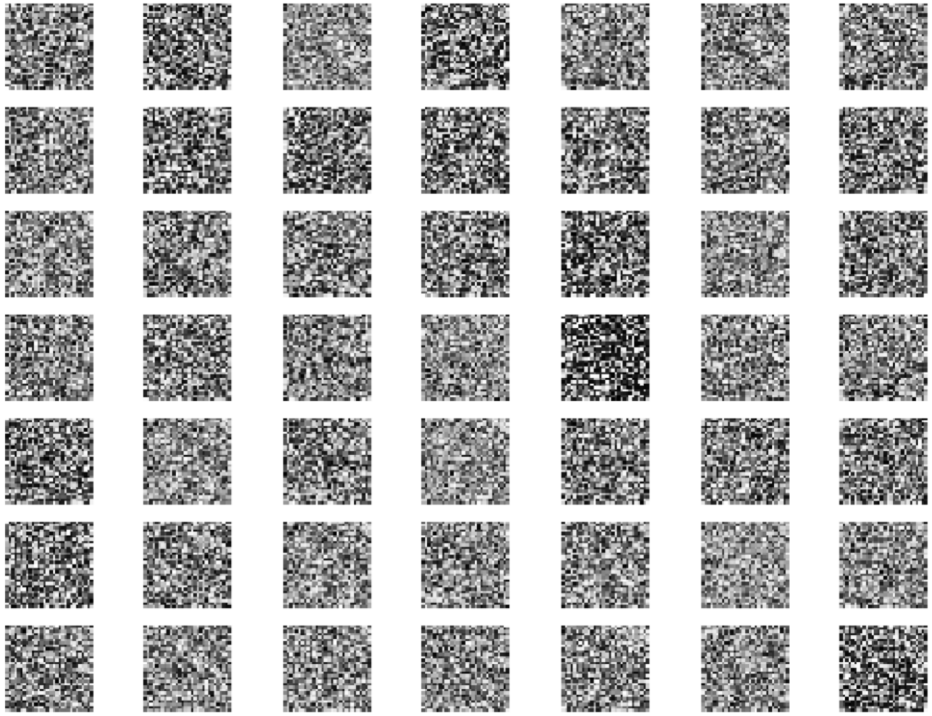
Test inputs



Reconstructed inputs

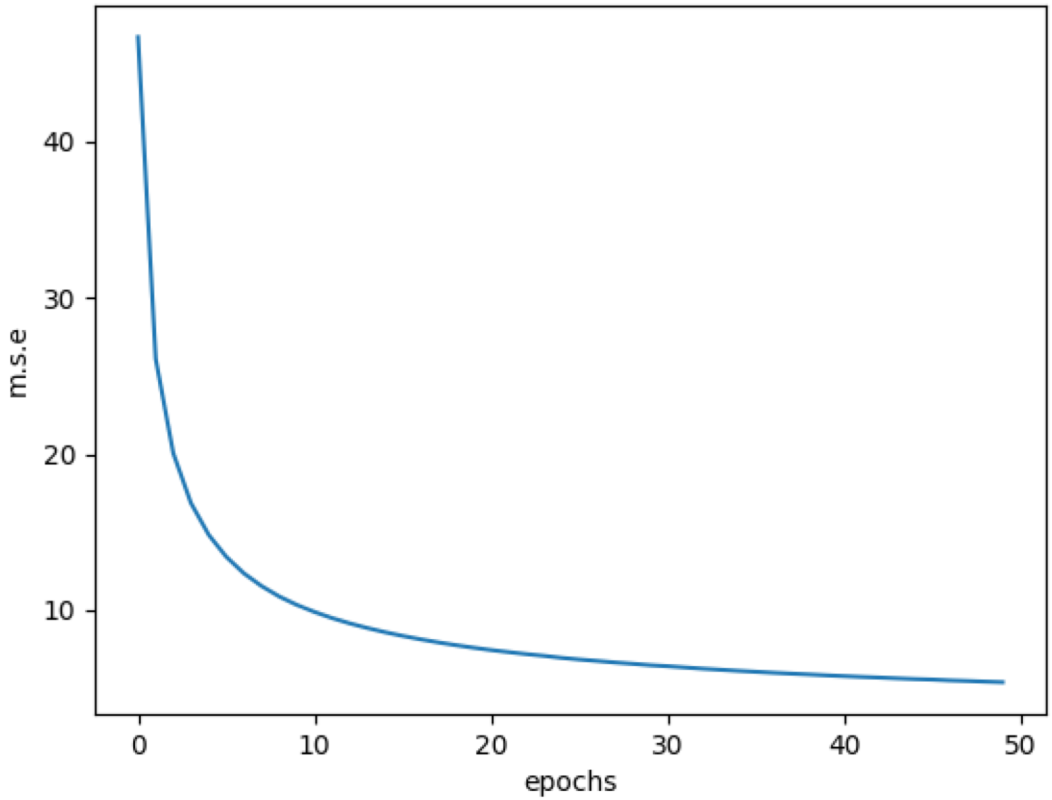


Hidden layer activations

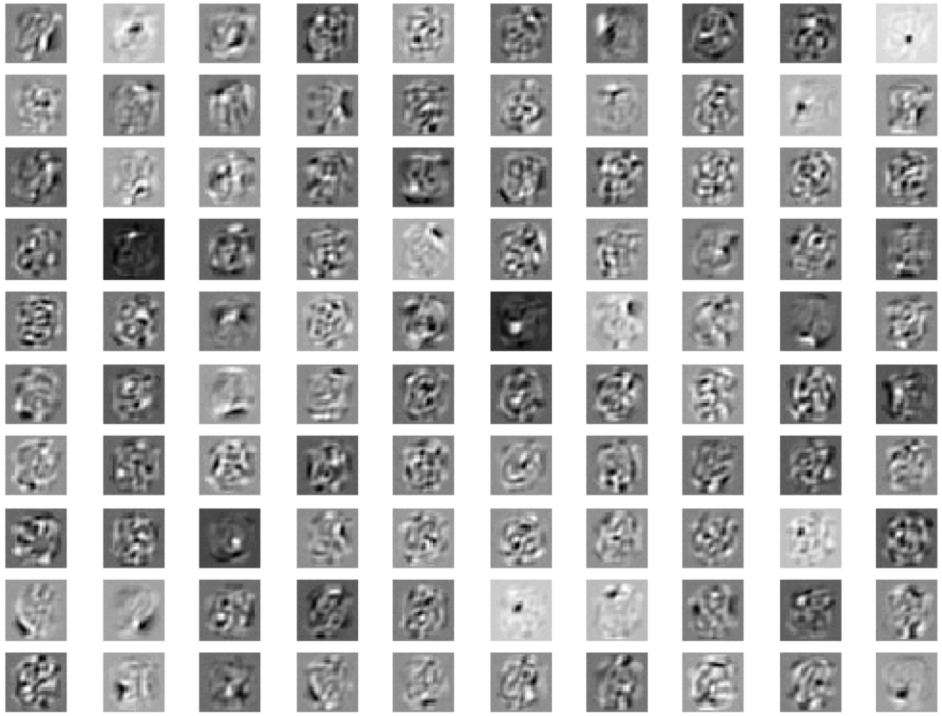


30x30 neurons

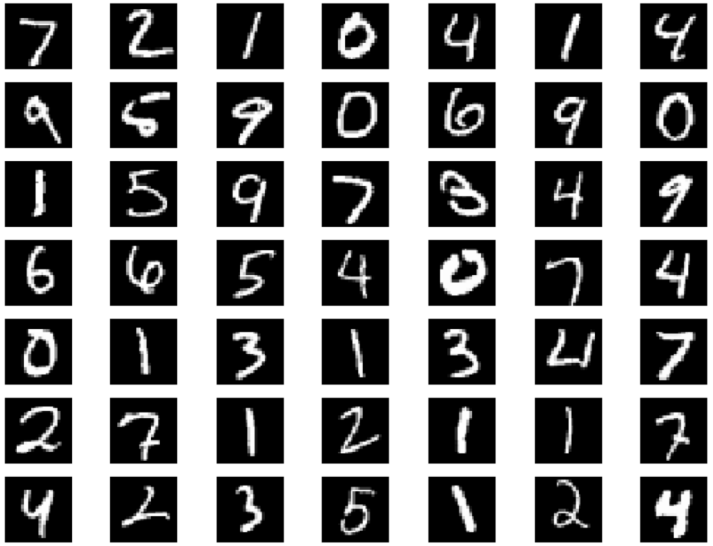
Example 3c: Sparse autoencoder



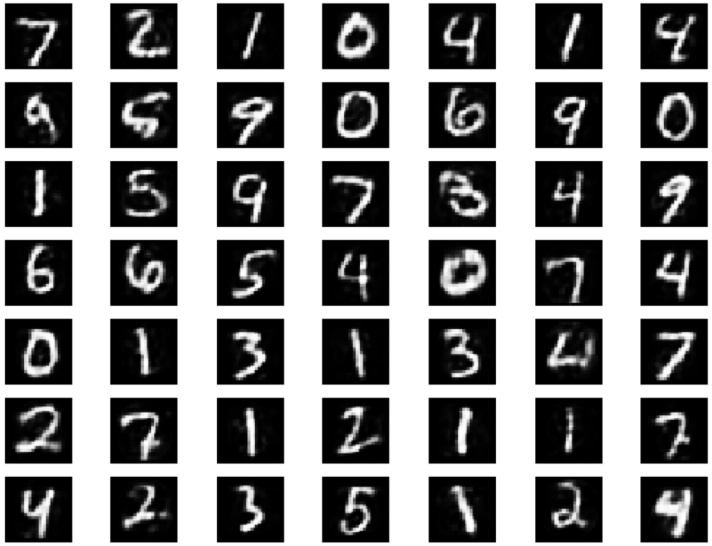
Learned weights



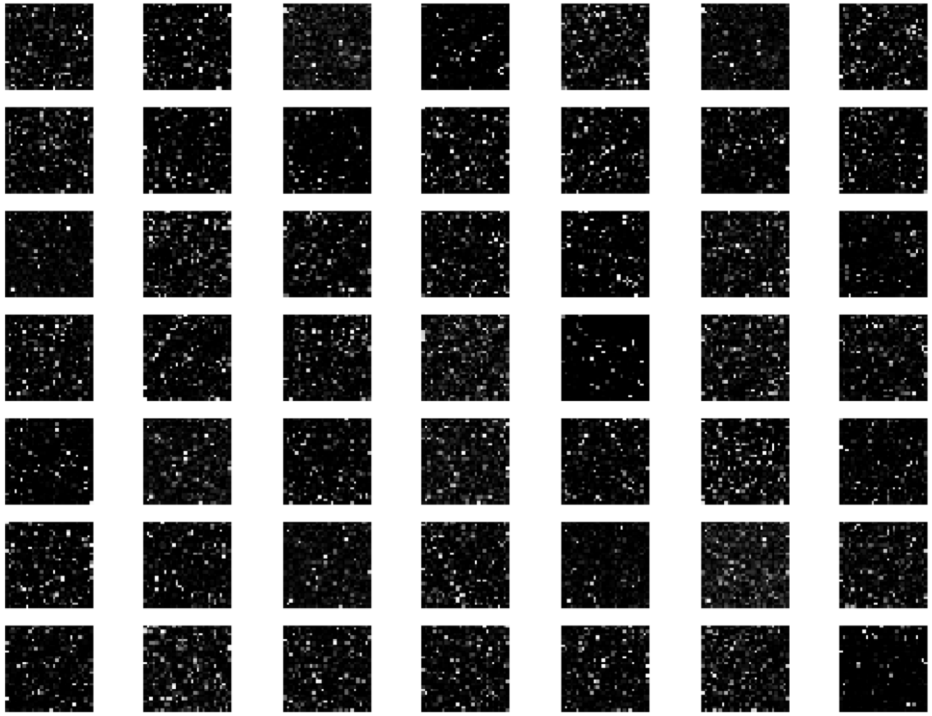
Test inputs



Reconstructed inputs



Hidden layer activations

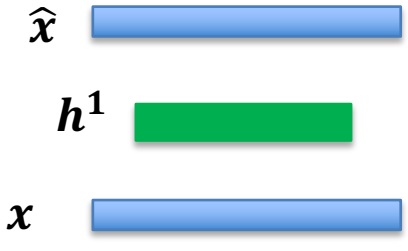


Deep stacked autoencoders

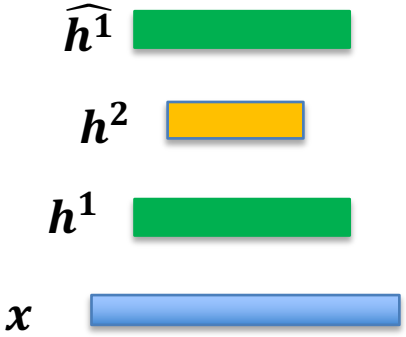
Deep autoencoders can be built by stacking autoencoders one after the other. Training of deep autoencoders is done in a step-by-step fashion on layer at time.

After training the first level of denoising autoencoder, the resulting hidden representation is used to train a second level of the denoising encoder. The second level hidden representation can be used to train the third level of the encoders. This process is repeated and deep stacked autoencoder can be realized.

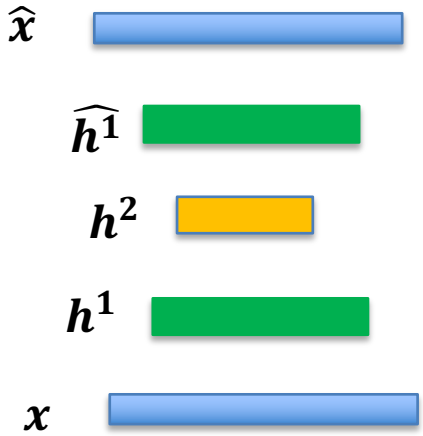
Deep stacked autoencoders



Training first hidden layer

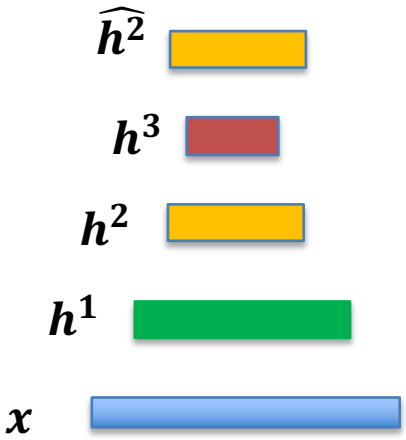


Training second hidden layer. Reconstructs first-hidden layer output

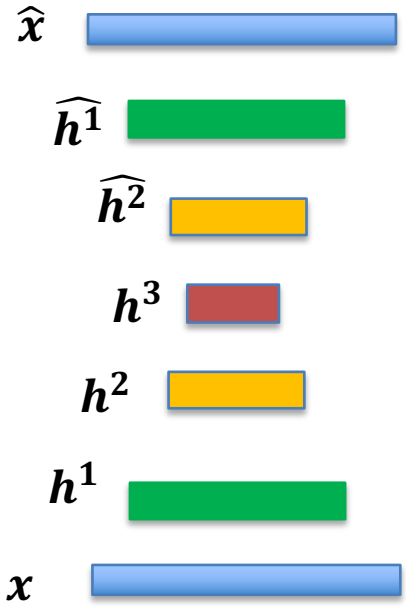


Autoencoder with two hidden layers

Deep stacked autoencoders

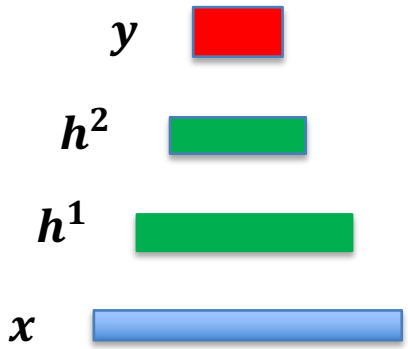


Training the third hidden layer. Reconstructs second hidden layer output



Autoencoder with three hidden layers

Fine tuning deep network for classification/regression



After training a stacked autoencoder, an output layer may be added on the top of the stacked hidden layers for classification/regression. The final layer is a supervised layer such as a softmax for classification or a linear layer for regression.

The parameters of the feedforward network can be fine-tuned to minimize the error in regressing/classifying the targets by supervised gradient descent learning.