AUTOENCODER

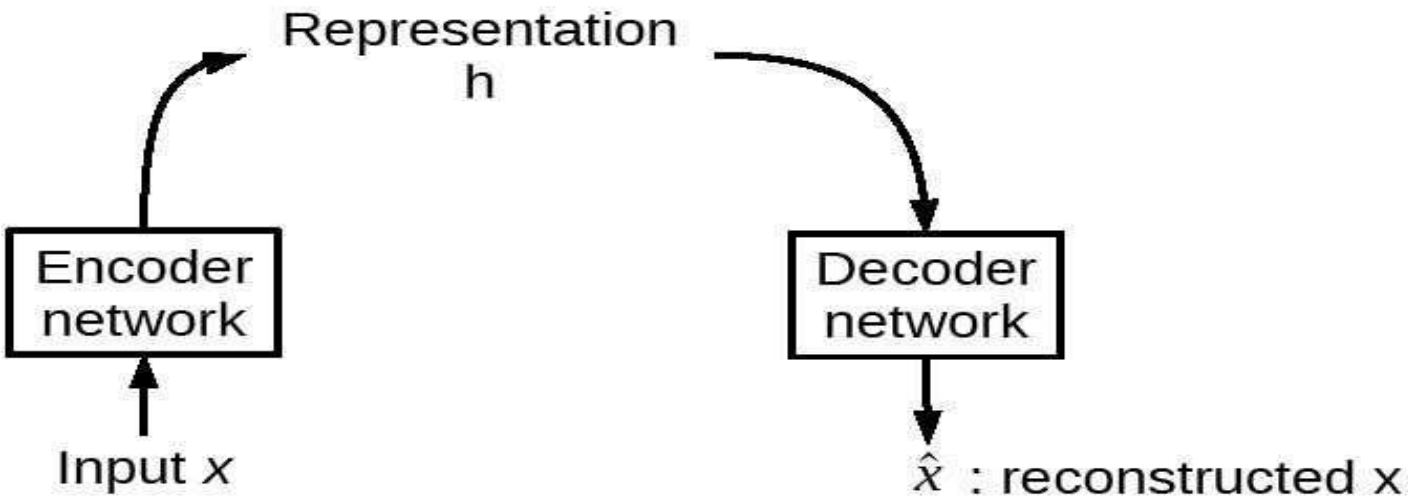
Autoencoders

- Autoencoders (AE) are a specific type of feedforward neural network where the input is the same as the output.
- AEs compress the input into a lower-dimensional *code* or *representation* and then reconstruct the output from this *representation*.
- The code is a compact “summary” or “compression” of the input, also called the *latent-space representation.*
- AEs were first introduced in 1986 by Hinton to address the problem of backpropagation algorithm with unlabelled data.

Autoencoder

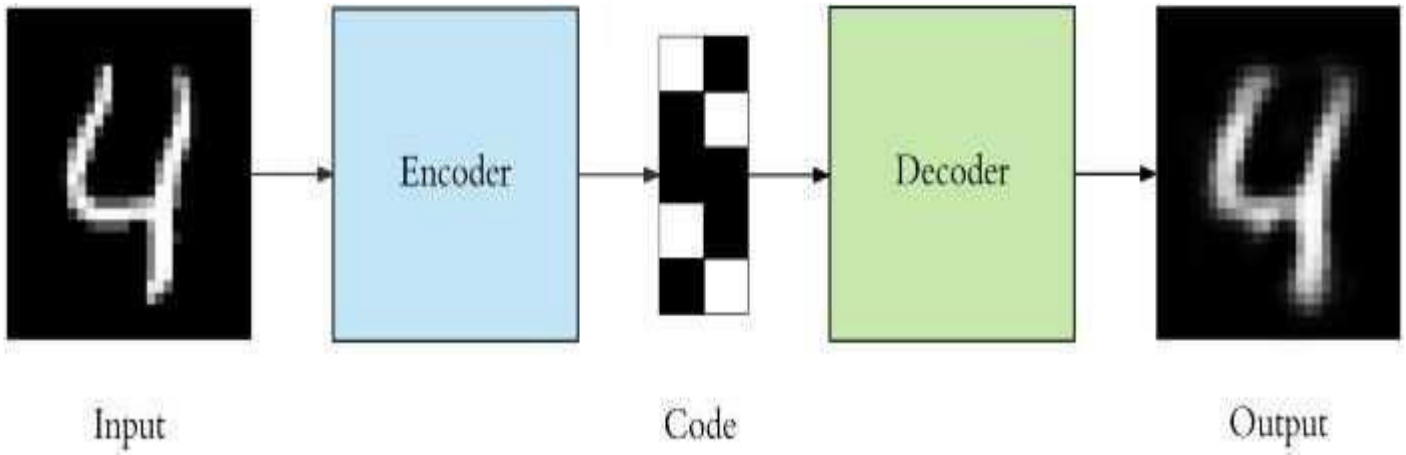
- An autoencoder consists of 3 components: encoder, code and decoder.
- The objective of an AE training process is to minimize the reconstruction error which is either Mean squared error or cross entropy error between original input and the reconstructed input.
- Autoencoders are mainly a dimensionality reduction (or compression)

Autoencoders



Minimize $||x - \hat{x}||_2^2$

Autoencoders

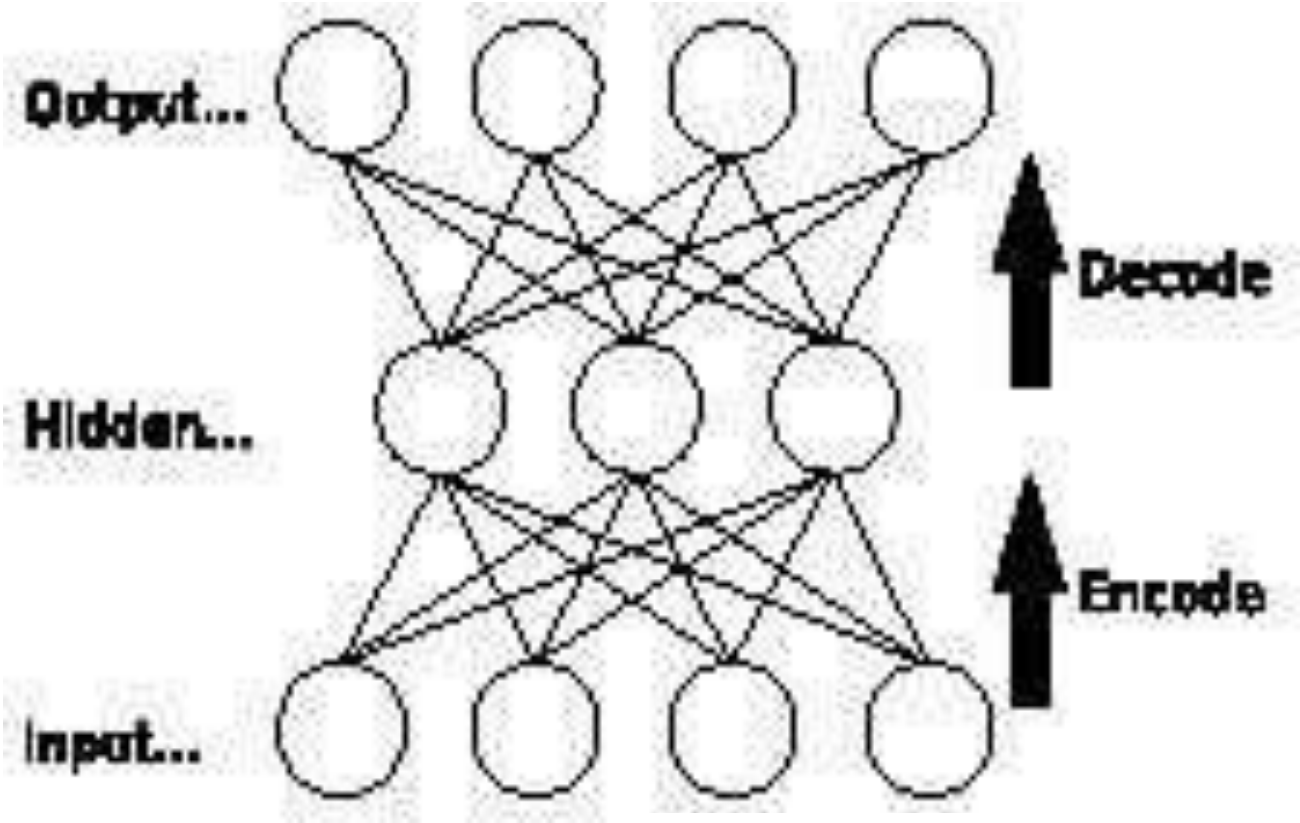


Latent representation, h

An autoencoder performs: encoding, decoding, and a loss function used to compare the output with the target.

Property

- AEs are different than a standard data compression algorithm.
- The output of the autoencoder is not exactly the same as the input, it will be a close but degraded representation.
- It is a Lossy Compression technique.
- Autoencoders are considered an *unsupervised* learning technique since they don’t need explicit labels to train on.
- Both the encoder and decoder are fully-connected feedforward neural networks.



$$\epsilon = \frac{1}{m}\sum_{i=1}^{m}(X_i' - X_i)^2 \qquad \epsilon = -\frac{1}{m}\sum_{i=1}^{m}(X_i \log(X_i') + (1 - X_i)\log(1 - X_i'))$$

where, X_i' is the reconstruction of input X_i,

- m represents the number of training samples.

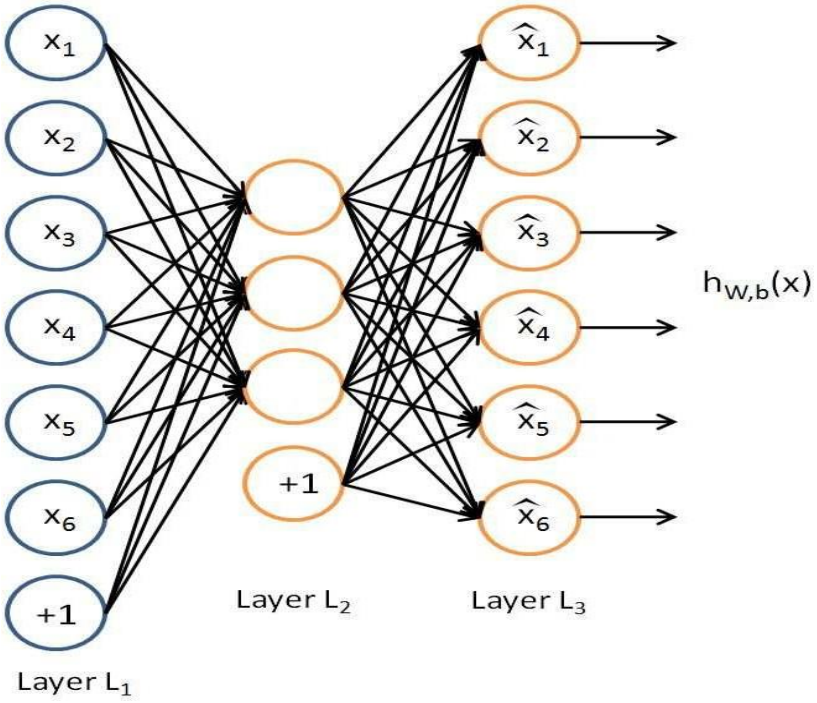
Error Function

- The choice of error function depends on the model.
- If we consider a probabilistic model where the output layer is implemented by sigmoid or softmax activation function, then CEE is a better choice compared to MSE.
- Similarly, if we assume the target data to be continuous and normally distributed, MSE is preferred.
- Optimization techniques like gradient descent may be used to minimize the reconstruction error.

Hyperparameter

- The number of nodes in the code layer (code size) is a *hyperparameter* that we set before training the autoencoder.
- Number of layers: the autoencoder can be as deep as we like.
- The number of nodes per layer decreases with each subsequent layer of the encoder, and increases back in the decoder.
- The decoder is symmetric to the encoder in terms of layer structure.
- Autoencoders are trained the same way as ANNs via backpropagation.

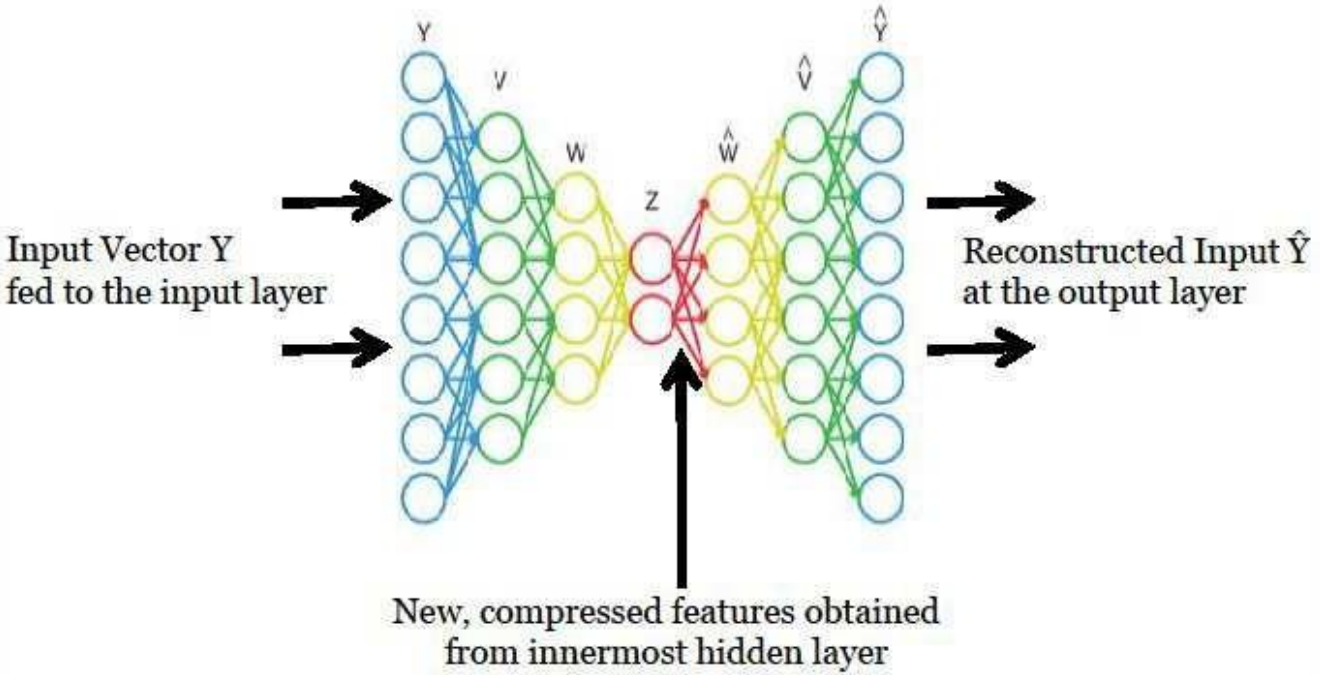
Autoencoder



- Input Images of size n×n and the latent space where m < n × n.
- Latent space is not sufficient to reproduce all images.
- Needs to learn an encoding that captures the important features in training data, sufficient for approximate reconstruction.

- The autoencoder tries to learn a function $h_{W,b}(x) \approx x$
- An approximation to the identity function, so as to output is similar to *input.*

Deep AE Architecture



Bottlenecks

- If the inputs are completely random—say, each x_i comes from an IID Gaussian independent of the other features—then this compression task would be very difficult.
- But if there is structure in the data, for example, if some of the input features are correlated, then this algorithm will be able to discover some of those correlations.
- In fact, this simple autoencoder often ends up learning a low-dimensional representation very similar to PCA's.

Autoencoders: Applications

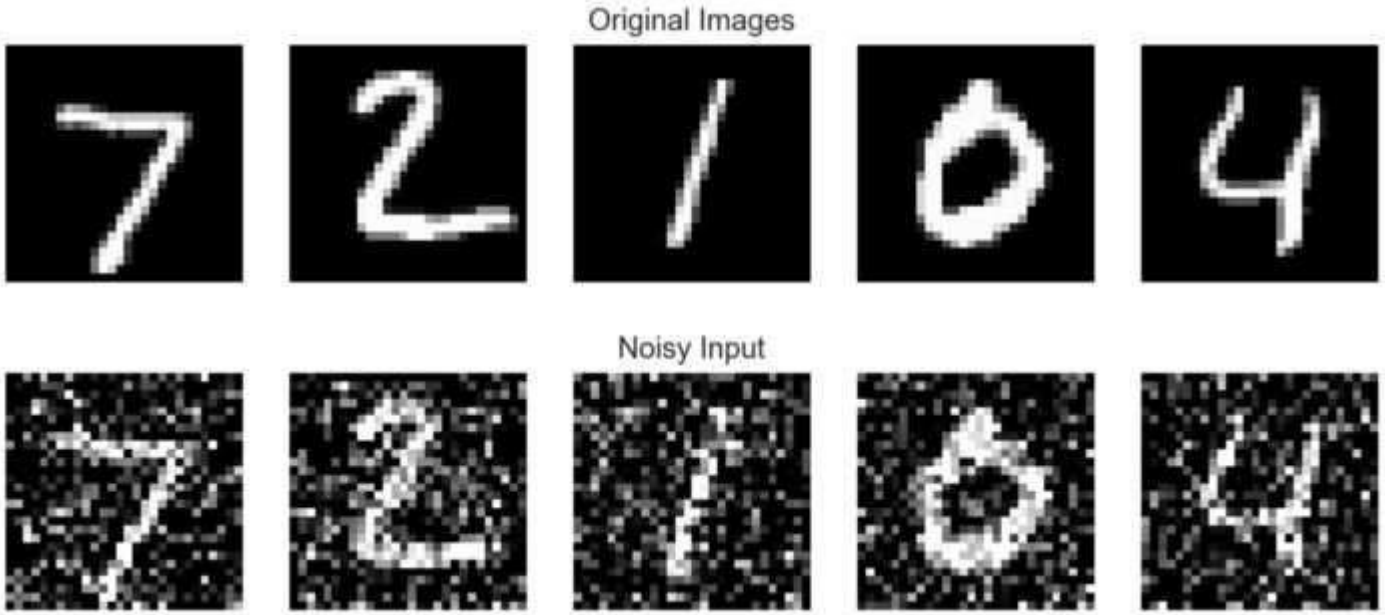
- Image colorization: input black and white and train to produce color images

https://www.edureka.co/blog/autoencoders-tutorial/

Denoising Autoencoders

- Keeping the code layer small forced the autoencoder to learn an intelligent representation of the data.
- Another way to force the autoencoder to learn useful features by adding random noise to its inputs and making it recover the original noise-free data.
- This autoencoder can’t simply copy the input to its output, called a *denoising autoencoder*.
- We add random Gaussian noise to them and the noisy data becomes the input to the autoencoder.

Denoising Autoencoder



- The autoencoder doesn’t see the original image at all.
- We expect the autoencoder to regenerate the noise-free original image.

Denoising Autoencoder

Autoencoders minimize $L(x, D(E(x)))$

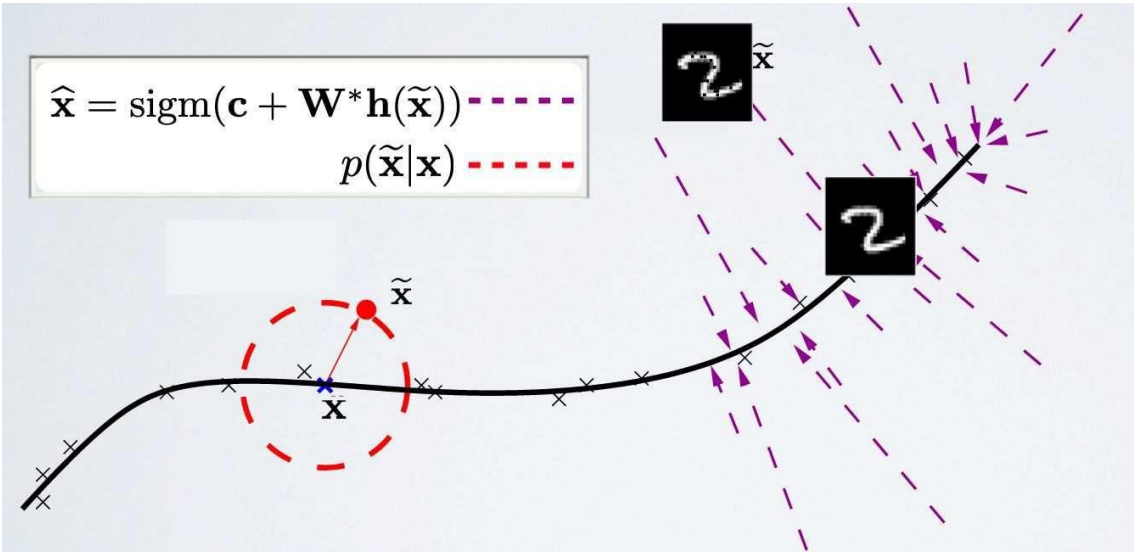
Denoising Autoencoders minimize $L(x, D(E(\tilde{x})))$

where $\tilde{x}$ is a copy of x that has been corrupted by some form of noise.

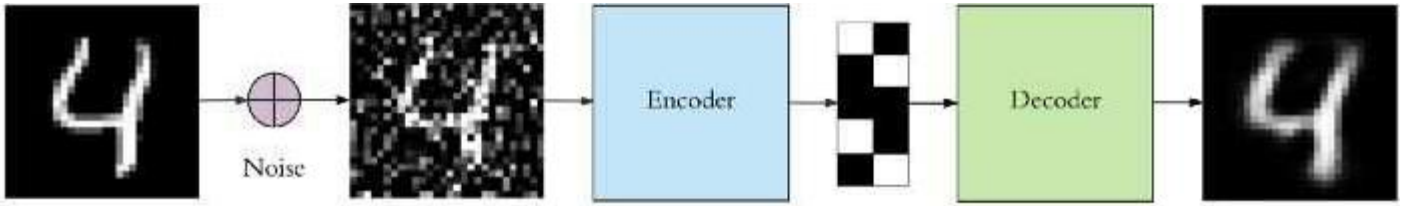
By corrupting x, the model is prevented from learning *E(D( ))* as identity.

Denoising autoencoders

- Denoising autoencoders can't simply memorize the input output relationship.
- Intuitively, a denoising autoencoder learns a projection from a neighborhood of our training data back onto the training data.



https://ift6266h17.files.wordpress.com/2017/03/14_autoencoders.pdf



Original Image | Noisy Input | Code | Output

Original Images

Noisy Input

Autoencoder Output

Sparse Autoencoder

- Sparse autoencoder learning algorithm automatically learns features from unlabeled data.
- The simple autoencoder tries to learn a function $h_{W,b}(x) \approx x$.
- In other words, it is trying to learn an approximation to the identity function, so as to output $\hat{x}$ that is similar to x.
- By placing constraints on the network, such as by limiting the number of hidden units, we can discover interesting structure about the data.

Sparsity Constraint

- Even when the number of hidden units is large (perhaps even greater than the number of input pixels), we can still discover interesting structure, by imposing other constraints on the network.
- Impose a sparsity constraint on the hidden units.
- Think of a neuron as being “active” (or as “firing”) if its output value is close to 1, or as being “inactive” if its output value is close to 0.
- We would like to constrain the neurons to be inactive most of the time.

Sparse Autoencoders

- We would construct loss function by penalizing **activations** of hidden layers so that only a few nodes are encouraged to activate when a single sample is fed into the network.
- This forces the autoencoder to represent each input as a combination of small number of nodes, and demands it to discover interesting structure in the data.
- This method works even if the code size is large, since only a small subset of the nodes will be active at any time.
- $a_j^{(2)}$ denotes the activation of hidden unit j at hidden layer (i.e. 2^{nd}) in the autoencoder.

Sparsity parameter

- The average activation of hidden unit *j* (averaged over the training set).

$$\hat{\rho}_j = \frac{1}{m}\sum_{i=1}^{m}[a_j^{(2)}(x^{(i)})]$$

$a_j^{(2)}(x)$ denotes the activation of hidden unit *j* when the network is given a specific input *x*.

- We would like to (approximately) enforce the constraint $\hat{\rho}_j = \rho$
- where ρ is a *sparsity parameter*, typically a small value close to zero (say $\rho = 0.05$), supplied by the user.
- In other words, we would like the average activation of each hidden neuron *j* to be close to 0.05 (say).

Sparsity Cost

- ***a*2** is a matrix containing the hidden neuron activations with one row per hidden neuron and one column per training example, then you can just sum along the rows of ***a2*** and divide by *m*.
- The result is a column vector with one row per hidden neuron.

$$\sum_{j=1}^{s_2} \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j}$$

Penalty Term

- To satisfy this constraint, the hidden unit's activations must mostly be near 0.
- To achieve this, we will add an extra penalty term to our optimization objective that penalizes $\hat{\rho}_j$ deviating significantly from ρ.

$$\sum_{j=1}^{s_2} \rho log \frac{\rho}{\hat{\rho_j}} + (1-\rho) log \frac{(1-\rho)}{(1-\hat{\rho_j})}$$

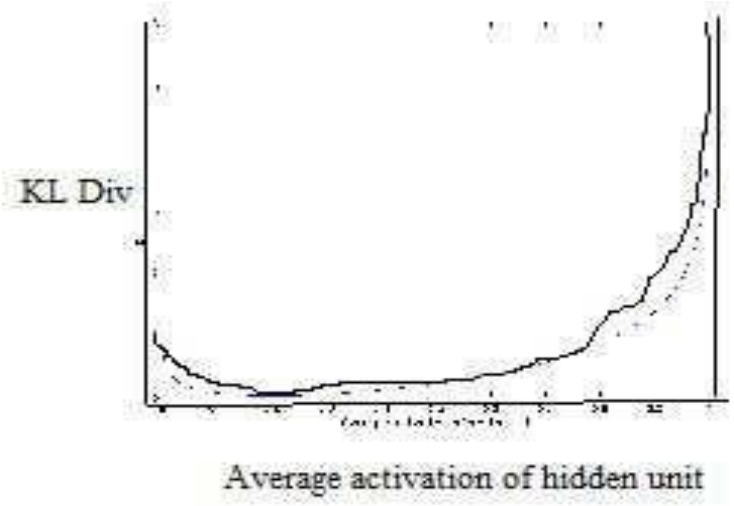
- s_2 is the number of neurons in the hidden layer. assuming a *sigmoid* activation function.
- If you are using a *tanh* activation function, then we think of a neuron as being inactive when it outputs values close to -1.

Penalty term

- Penalty term is based on K-L divergence: $\sum_{j=1}^{s_2} KL(\rho||\hat{\rho}_j)$

$$KL(\rho||\hat{\rho}_j) = \rho log\frac{\rho}{\hat{\rho}_j} + (1-\rho) log\frac{1-\rho}{1-\hat{\rho}_j}$$

- KL divergence between a Bernoulli random variable with mean ρ and a Bernoulli random variable with mean $\hat{\rho}j$.

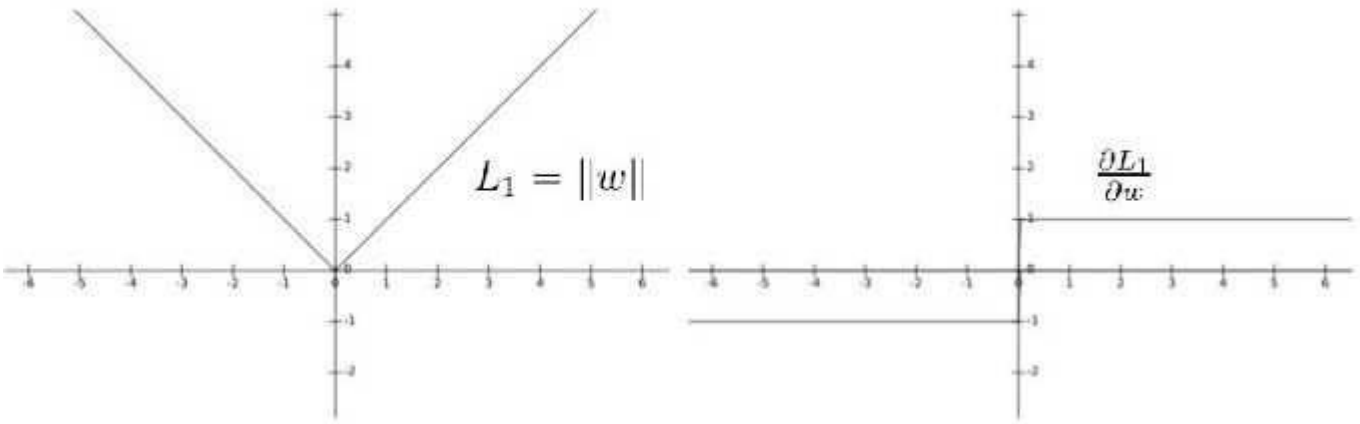


Say, $\rho = 0.2$

KL-divergence reaches its minimum to 0 at $\hat{\rho}_j = \rho$, and increases when $\hat{\rho}_j$ approaches 0 or 1.
Thus, minimizing
this penalty term has the effect of causing $\hat{\rho}_j$ close to ρ.

L1 Regularization Sparse

- There are two different ways to construct sparsity penalty: **L1 regularization** and **KL-divergence**.
- L1 regularization adds “absolute value of magnitude” of coefficients as penalty term.
- L1 regularization tends to **shrink the penalty coefficient to zero.**
- For L1 regularization, the gradient is either 1 or -1 except when w=0, which means that L1 regularization will always move w towards zero with **same step size** (1 or -1) regardless of the value of w.
- when w=0, the gradient becomes zero and no update will be made anymore.



$$J_{sparse} = J + L1 + \lambda \sum_i |a_i|_h$$

The third term which penalizes the absolute value of the vector of activations *a* in layer *h* for sample *i*.

Cost function: KL Divergence

$$J_{sparse}(W,b) = J(W,b) + \beta \sum_{j=1}^{s_2} KL(\rho || \hat{\rho}_j)$$

- β controls the weight of the sparsity penalty term.
- $\hat{\rho}_j$ depends on W, b; the average activation of hidden unit *j*
- In the second layer (l = 2), during backpropagation we compute,

$$\delta_i^{(2)} = (\sum_{j=1}^{s_3} W_{ji}^{(3)} \delta_j^{(3)}) f'(z_i^{(2)})$$

- Now compute:

$$\delta_i^{(2)} = (\sum_{j=1}^{s_3} W_{ji}^{(3)} \delta_j^{(3)}) f'(z_i^{(2)}) + \beta(-\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i})$$

Gradient Calculation

- To Compute $\hat{\rho}_i$ allow a forward pass on all the training examples first i.e. the average activations on the training set, before computing backpropagation on any example.

- Then you can use your precomputed activations to perform backpropagation on all your examples.

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x, y) = a_j^{(l)} \delta_i^{(l+1)}$$

The gradient for a *single weight value* relative to a *single training example*. This equation needs to be evaluated for every combination of j and i, leading to a matrix with same dimensions as the weight matrix. Then it needs to be evaluated for every training example, and the resulting matrices are summed.

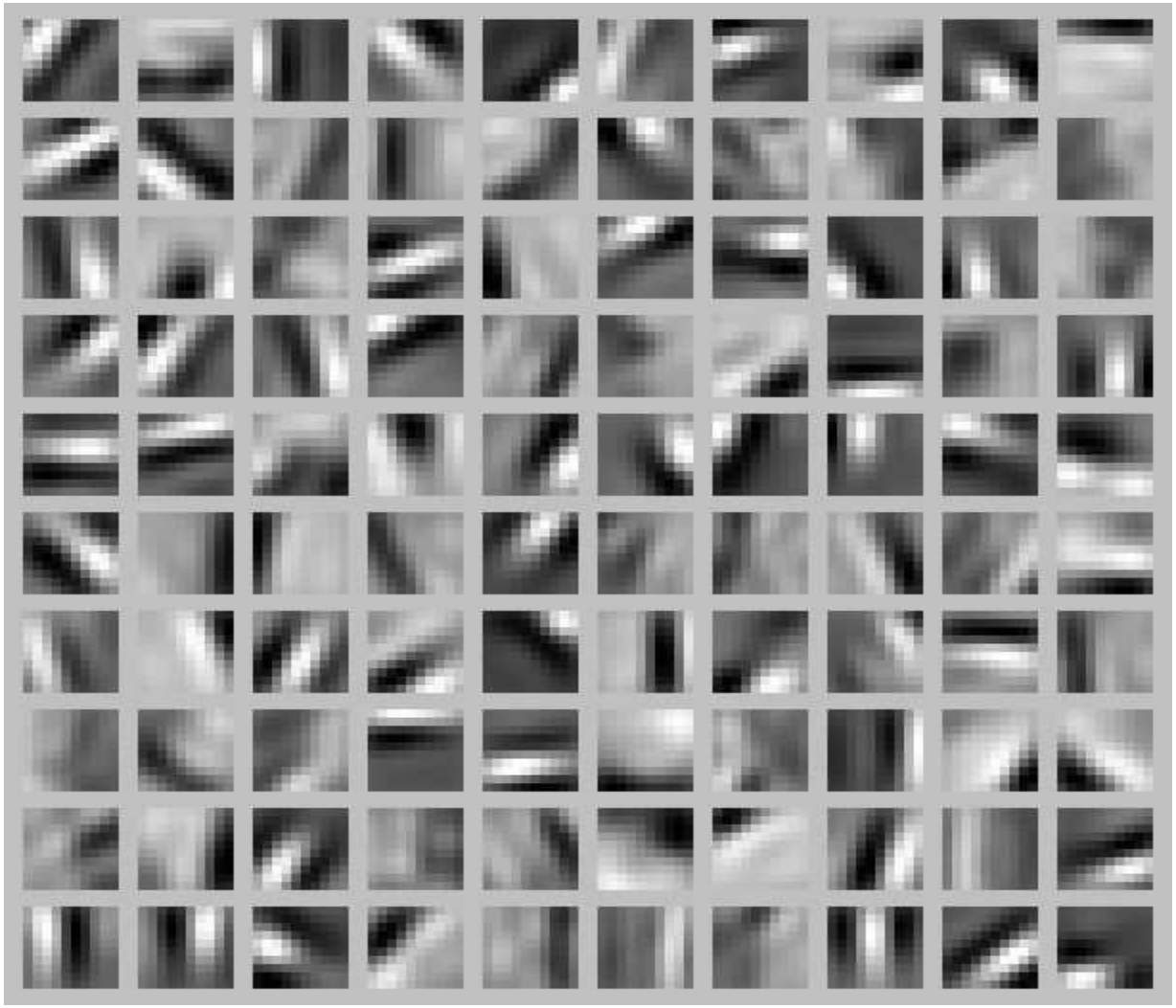
Visualization

- Having trained a (sparse) autoencoder, we would now like to visualize the function learned by the algorithm.
- Consider the case of training an autoencoder on 10×10 images.
- Each hidden unit *i* computes a function of the input:

$$a_i^{(2)} = f\left(\sum_{j=1}^{100} w_{ij}^{(1)} x_j\right)$$

If we have an autoencoder with 100 hidden units (say), then we our visualization will have 100 such images—one per hidden unit. By examining these 100 images.

Each square shows the (norm bounded) input image x
that maximally actives one of 100 hidden units. Different hidden units have learned to detect edges at different positions and orientations in the image.



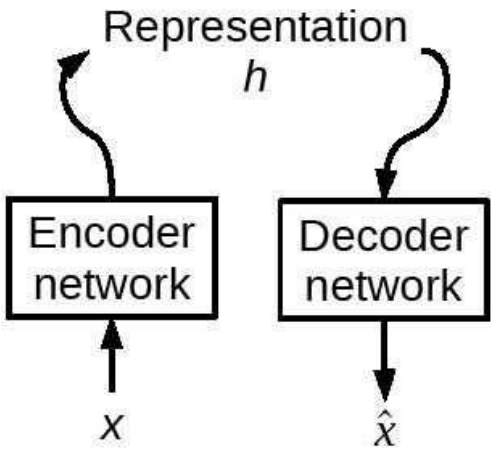
Variational Autoencoder

- The basic idea behind a variational autoencoder is that instead of mapping an input to fixed vector, input is mapped to a distribution.
- Thus, rather than building an encoder which outputs a single value to describe each latent state attribute, we'll formulate our encoder to describe a probability distribution for each latent attribute.
- Represent each latent attribute as a range of possible values.
- When decoding from the latent state, we'll randomly sample from each latent state distribution to generate a vector as input for our decoder model.

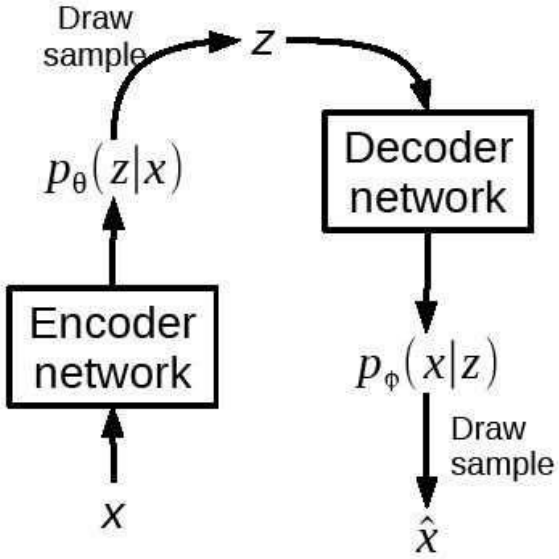
Variational Autoencoder

[Kingma and Welling (2013)]

Classical Autoencoder:

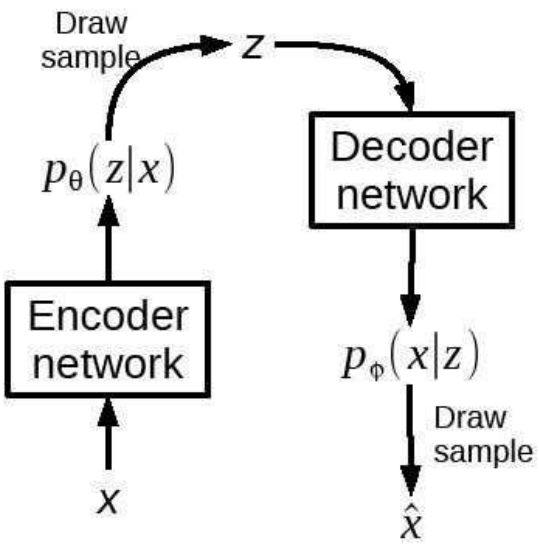


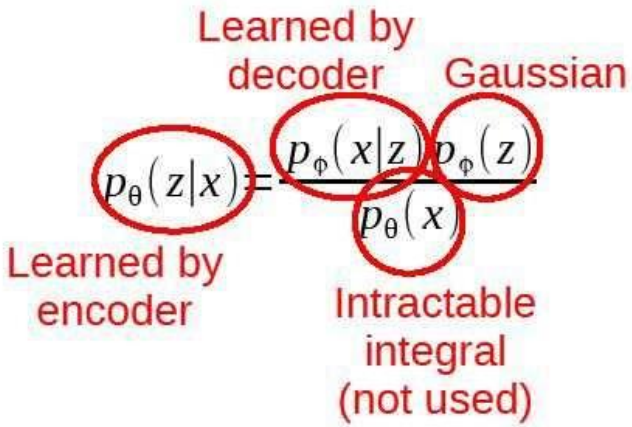
Variational Autoencoder:



Variational Autoencoder

[Kingma and Welling (2013)]





varitational inference

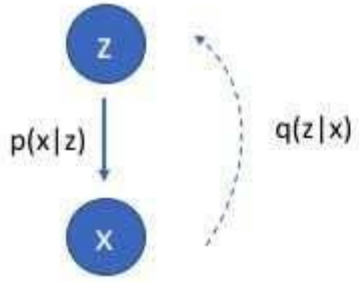
- Suppose that there exists some hidden variable z which generates an observation x.
- We can only see x, but we would like to infer the characteristics of z i.e. compute p(z|x).

$$p(z|x) = \frac{p(x|z)p(z)}{p(x)}$$

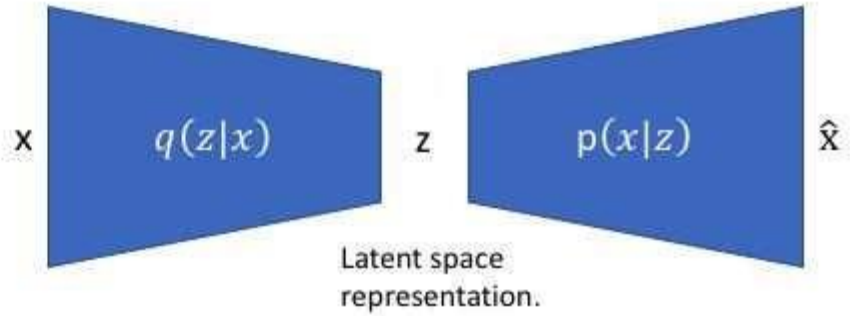
- Computing p(x) is quite difficult. $P(x) = \int p(x|z)\, p(z)\, dz$
- we can apply varitational inference to estimate this value.

- Let's approximate p($z|x$) by another distribution q(z|x).
- Define the parameters of q(z|x) such that it is very similar to p(z|x).
- KL divergence is a measure of difference between two probability distributions.
- We wanted to minimize the KL divergence (min[KL(q(z| x) || p(z | x)]) between the two distributions.
- we can minimize the above expression by maximizing
- $E_{q(z|x)}$ log p(x|z) - KL(q(z|x) || p(z|x)
- The first term represents the reconstruction likelihood and the second term ensures that our learned distribution q is similar to the true prior distribution p.

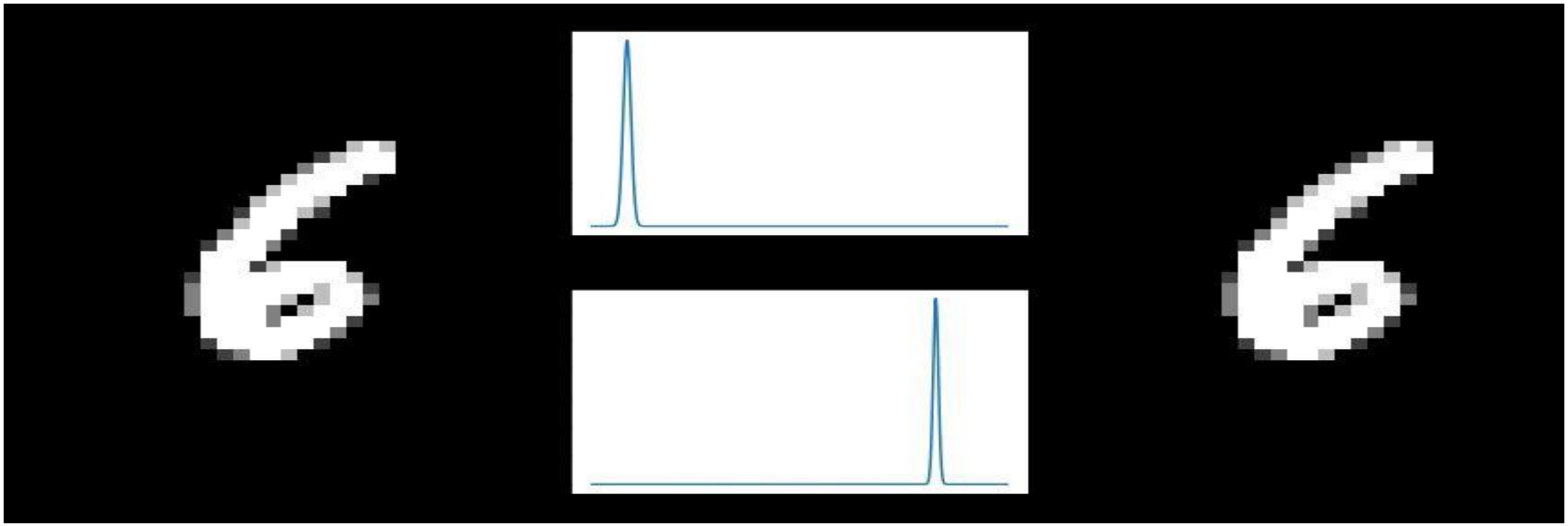
- we can use q to infer the possible hidden variables (ie. latent state) which was used to generate an observation.
- We can further construct this model into a neural network architecture where the encoder model learns a mapping from x to z and the decoder model learns a mapping from z back to x.

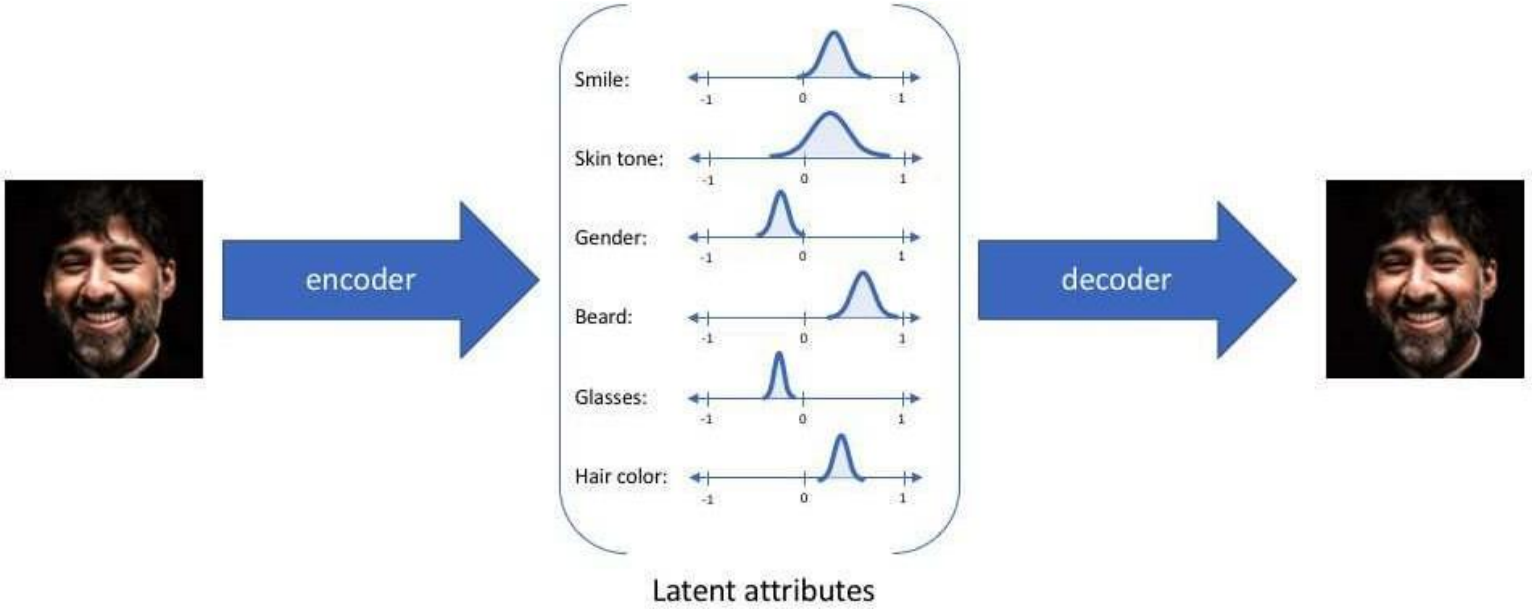


We'd like to use our observations to understand the hidden variable.



- The main benefit of a variational autoencoder is that we're capable of learning *smooth* latent state representations of the input data.
- When constructing a variational autoencoder, inspect the latent dimensions for a few samples from the data to see the characteristics of the distribution.




encoder
Smile:
Skin tone:
Gender:
Beard:
Glasses:
Hair color:
-1
0
1
decoder
Latent attributes