Probabilistic (Graphical) Models

and inference

Oliver Obst · Autumn 2024

WESTERN SYDNEY
UNIVERSITY

Probabilistic (Graphical) Models and Inference

(PGM: Probabilistic Graphical Models: Principles and Techniques by Daphne Koller and Nir Friedman. MIT Press)
(PMLI: Probabilistic Machine Learning: An introduction by Kevin Murphy. MIT Press)

Week	Lecture	Required reading	Assessment
1 Monday, 4 March 2024	Introduction, Probability Theory	PGM Chapter 2, PMLI Chapter 6.1	
2 Monday, 11 March 2024	Directed and undirected networks introduction	PGM Chapter 3 & 4	Quiz 1
3 Monday, 18 March 2024	Variable elimination	PGM Chapter 9	
4 Monday, 25 March 2024	Belief propagation	PGM Chapter 10/11	Quiz 2
5 Monday, 1 April 2024	public holiday		5 April 2024: census date
6 Monday, 8 April 2024	Message passing / Graph neural networks	https://distill.pub/2021/gnn-intro/	
7 Monday, 15 April 2024	Sampling	PGM Chapter 12	Quiz 3
8 Monday, 22 April 2024	Mid-term break		
9 Monday, 29 April 2024	Variational inference	https://leimao.github.io/article/Introduction-to-Variational-Inference/	Intra-session exam
10 Monday, 6 May 2024	Autoregressive models	https://sites.google.com/view/berkeley-cs294-158-sp20/home	Quiz 4
11 Monday, 13 May 2024	Variational Auto-Encoders	https://lilianweng.github.io/posts/2018-08-12-vae/	
12 Monday, 20 May 2024	GANs	https://arxiv.org/abs/1701.00160	Quiz 5
13 Monday, 27 May 2024	Energy-based models		
14 Monday, 3 June 2024	Evaluating generative models		Quiz 6
Monday, 17 June 2024			Project due

Variational Auto-Encoder (VAE)

Notation

(Slides based on **“From Autoencoder to Beta-VAE”, Lilian Weng,** https://lilianweng.github.io/posts/2018-08-12-vae/)

Symbol	Meaning
$\mathcal{D}$	The dataset, $\mathcal{D} = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(n)}\}$, contains n data samples; $\lvert\mathcal{D}\rvert = n$.
$\mathbf{x}^{(i)}$	Each data point is a vector of d dimensions, $\mathbf{x}^{(i)} = [x_1^{(i)}, x_2^{(i)}, \dots, x_d^{(i)}]$.
$\mathbf{x}$	One data sample from the dataset, $\mathbf{x} \in \mathcal{D}$.
$\mathbf{x}'$	The reconstructed version of $\mathbf{x}$.
$\tilde{\mathbf{x}}$	The corrupted version of $\mathbf{x}$.
$\mathbf{z}$	The compressed code learned in the bottleneck layer.
$a_j^{(l)}$	The activation function for the j-th neuron in the l-th hidden layer.
$g_\phi(.)$	The **encoding** function parameterized by ϕ.
$f_\theta(.)$	The **decoding** function parameterized by θ.
$q_\phi(\mathbf{z}\vert\mathbf{x})$	Estimated posterior probability function, also known as **probabilistic encoder**.
$p_\theta(\mathbf{x}\vert\mathbf{z})$	Likelihood of generating true data sample given the latent code, also known as **probabilistic decoder**.

Autoencoder

An **autoencoder** is a neural network designed to learn an identity function: $\mathbf{x} = F(\mathbf{x})$

- in an unsupervised way to reconstruct the original input
- while compressing the data in the process

so as to discover a more efficient and compressed representation.

Autoencoder

An **autoencoder** is a neural network designed to learn an identity function: $\mathbf{x} = F(\mathbf{x})$

- in an unsupervised way to reconstruct the original input
- while compressing the data in the process

so as to discover a more efficient and compressed representation.

An autoencoder learns two functions:

- An encoder $g(\cdot)$ with parameters ϕ
- A decoder $f(\cdot)$ with parameters θ

Autoencoder

An **autoencoder** is a neural network designed to learn an identity function: $\mathbf{x} = F(\mathbf{x})$

- in an unsupervised way to reconstruct the original input
- while compressing the data in the process

so as to discover a more efficient and compressed representation.

An autoencoder learns two functions:

- An encoder $g(\cdot)$ with parameters ϕ
- A decoder $f(\cdot)$ with parameters θ

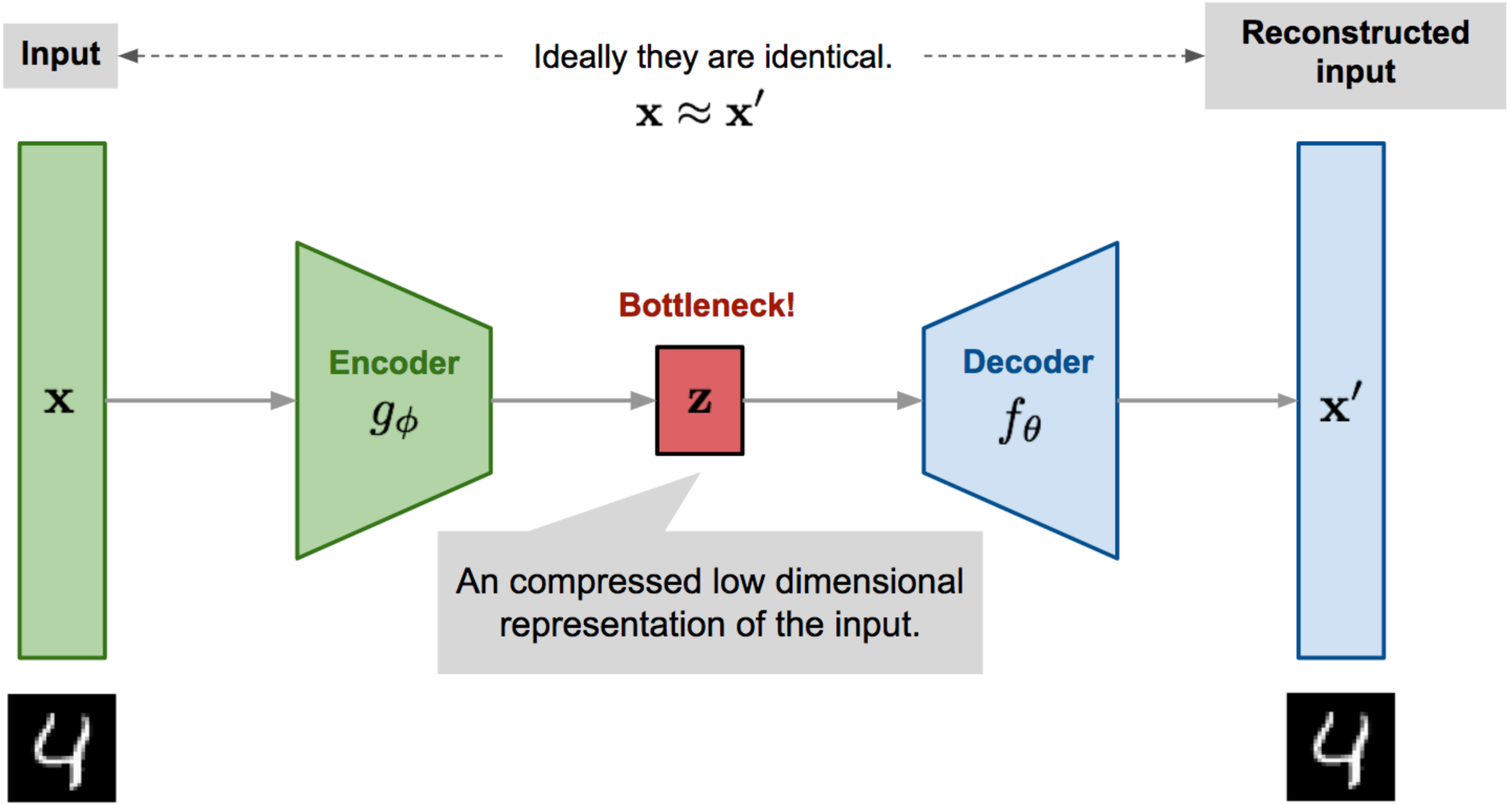
The autoencoder uses these functions to reconstruct the input: $\hat{\mathbf{x}} = f_\theta(g_\phi(\mathbf{x}))$.

- The *encoder* network: It translates the original high-dimension input into the latent low-dimensional code. The input size is larger than the output size.

- The *decoder* network: The decoder network recovers the data from the code, likely with larger and larger output layers.

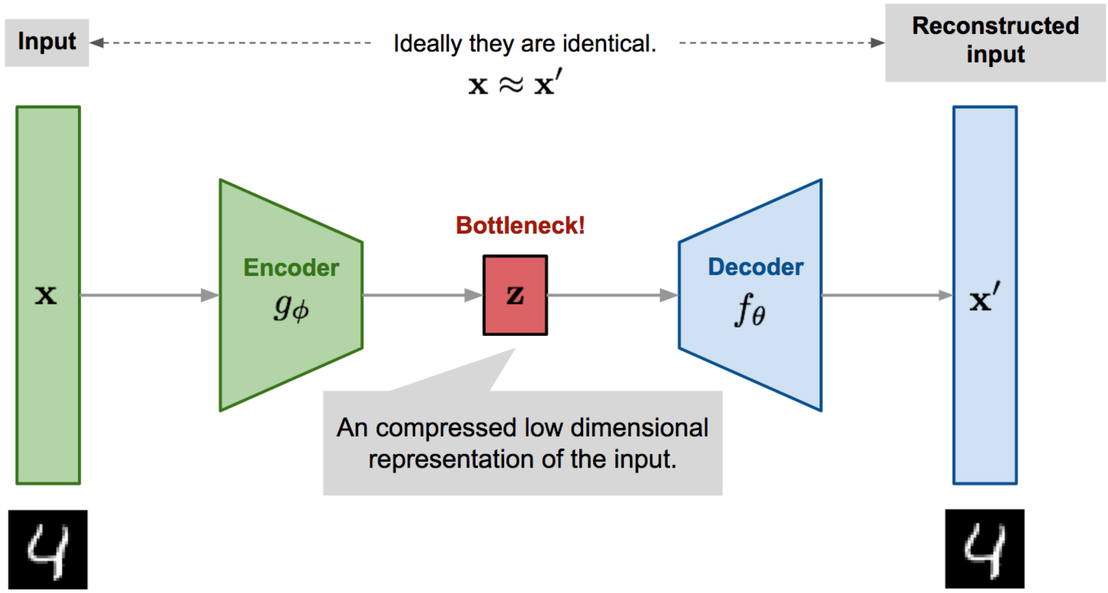
- The *encoder* network: It translates the original high-dimension input into the latent low-dimensional code. The input size is larger than the output size.

- The *decoder* network: The decoder network recovers the data from the code, likely with larger and larger output layers.



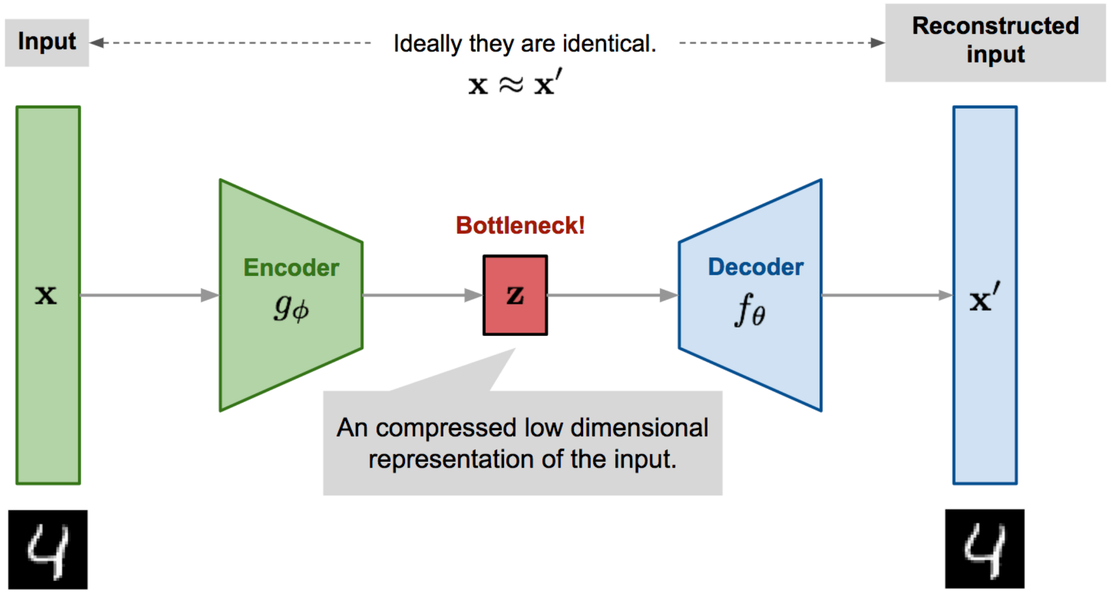
Autoencoder

$$\hat{\mathbf{x}} = f_\theta(g_\phi(\mathbf{x}))$$



Autoencoder

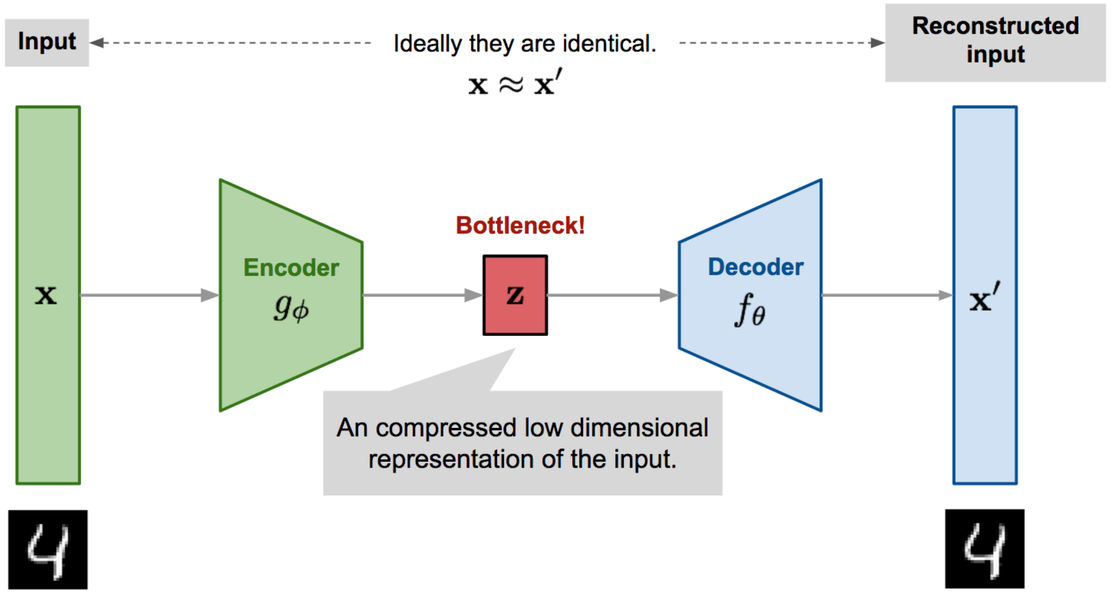
$$\hat{\mathbf{x}} = f_\theta(g_\phi(\mathbf{x}))$$



The parameters θ and ϕ are learned together — using the input as the output.

Autoencoder

$$\hat{\mathbf{x}} = f_\theta(g_\phi(\mathbf{x}))$$



The parameters θ and ϕ are learned together — using the input as the output.

Different metrics (loss functions) are possible, common is the MSE (mean square error):

$$L_{\text{AE}}(\theta, \phi) = \frac{1}{n} \sum_{i=1}^{n} (\mathbf{x}^{(i)} - f_\theta(g_\phi(\mathbf{x}^{(i)})))^2$$

Denoising Autoencoder

Since the autoencoder learns the identity function, we are facing the risk of "overfitting" when there are more network parameters than the number of data points.

To avoid overfitting and improve the robustness, the Denoising Autoencoder (Vincent et al. 2008) proposed a modification to the basic autoencoder.

The input is partially corrupted by adding noises to or masking some values of the input vector in a stochastic manner: $\tilde{\mathbf{x}} \sim \mathcal{M}_{\mathcal{D}}(\tilde{\mathbf{x}} \mid \mathbf{x})$. Then, the model is trained to recover the original input (the uncorrupted image).

Denoising Autoencoder

Since the autoencoder learns the identity function, we are facing the risk of "overfitting" when there are more network parameters than the number of data points.

To avoid overfitting and improve the robustness, the Denoising Autoencoder (Vincent et al. 2008) proposed a modification to the basic autoencoder.

The input is partially corrupted by adding noises to or masking some values of the input vector in a stochastic manner: $\tilde{\mathbf{x}} \sim \mathscr{M}_{\mathscr{D}}(\tilde{\mathbf{x}} \mid \mathbf{x})$. Then, the model is trained to recover the original input (the uncorrupted image).

$$\tilde{\mathbf{x}}^{(i)} \sim \mathscr{M}_{\mathscr{D}}(\tilde{\mathbf{x}}^{(i)} \mid \mathbf{x}^{(i)})$$

$$L_{\text{DAE}}(\theta, \phi) = \frac{1}{n} \sum_{i=1}^{n} (\mathbf{x}^{(i)} - f_\theta(g_\phi(\tilde{\mathbf{x}}^{(i)})))^2$$

Denoising Autoencoder

Since the autoencoder learns the identity function, we are facing the risk of "overfitting" when there are more network parameters than the number of data points.

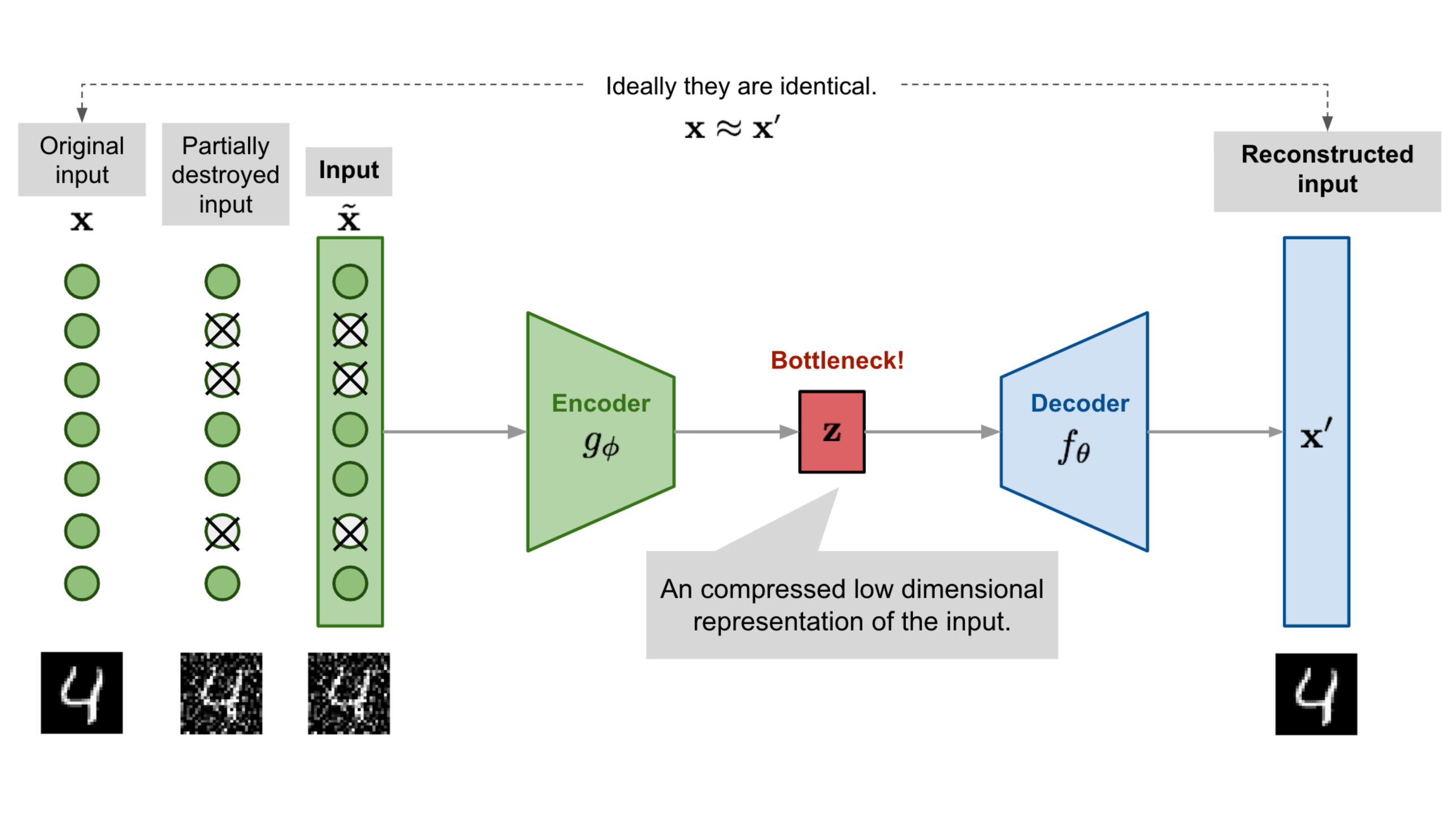
To avoid overfitting and improve the robustness, the Denoising Autoencoder (Vincent et al. 2008) proposed a modification to the basic autoencoder.

The input is partially corrupted by adding noises to or masking some values of the input vector in a stochastic manner: $\tilde{\mathbf{x}} \sim \mathcal{M}_{\mathcal{D}}(\tilde{\mathbf{x}} \mid \mathbf{x})$. Then, the model is trained to recover the original input (the uncorrupted image).

$$\tilde{\mathbf{x}}^{(i)} \sim \mathcal{M}_{\mathcal{D}}(\tilde{\mathbf{x}}^{(i)} \mid \mathbf{x}^{(i)})$$

$$L_{\text{DAE}}(\theta, \phi) = \frac{1}{n} \sum_{i=1}^{n} (\mathbf{x}^{(i)} - f_\theta(g_\phi(\tilde{\mathbf{x}}^{(i)})))^2$$

$\mathcal{M}_{\mathcal{D}}$ defines the mapping from the true data samples to the noisy or corrupted ones.


Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Original input
$\mathbf{x}$
Partially destroyed input
Input
$\tilde{\mathbf{x}}$
Reconstructed input
Encoder
g_ϕ
Bottleneck!
$\mathbf{z}$
Decoder
f_θ
$\mathbf{x}'$
An compressed low dimensional representation of the input.

Denoising Autoencoder

Motivation: humans can easily recognise an object or a scene even the view is partially occluded or corrupted.

To “repair” the partially destroyed input, the denoising autoencoder has to discover and capture relationship between dimensions of input in order to infer missing pieces.

For high dimensional input with high redundancy the model is likely to depend on evidence gathered from a combination of many input dimensions to recover the denoised version rather than to overfit one dimension.

This builds up a good foundation for learning robust latent representation.

Denoising and dropout

In DAE, noise is controlled by a stochastic mapping $\tilde{\mathbf{x}} \sim \mathscr{M}_{\mathscr{D}}(\tilde{\mathbf{x}} \mid \mathbf{x})$.

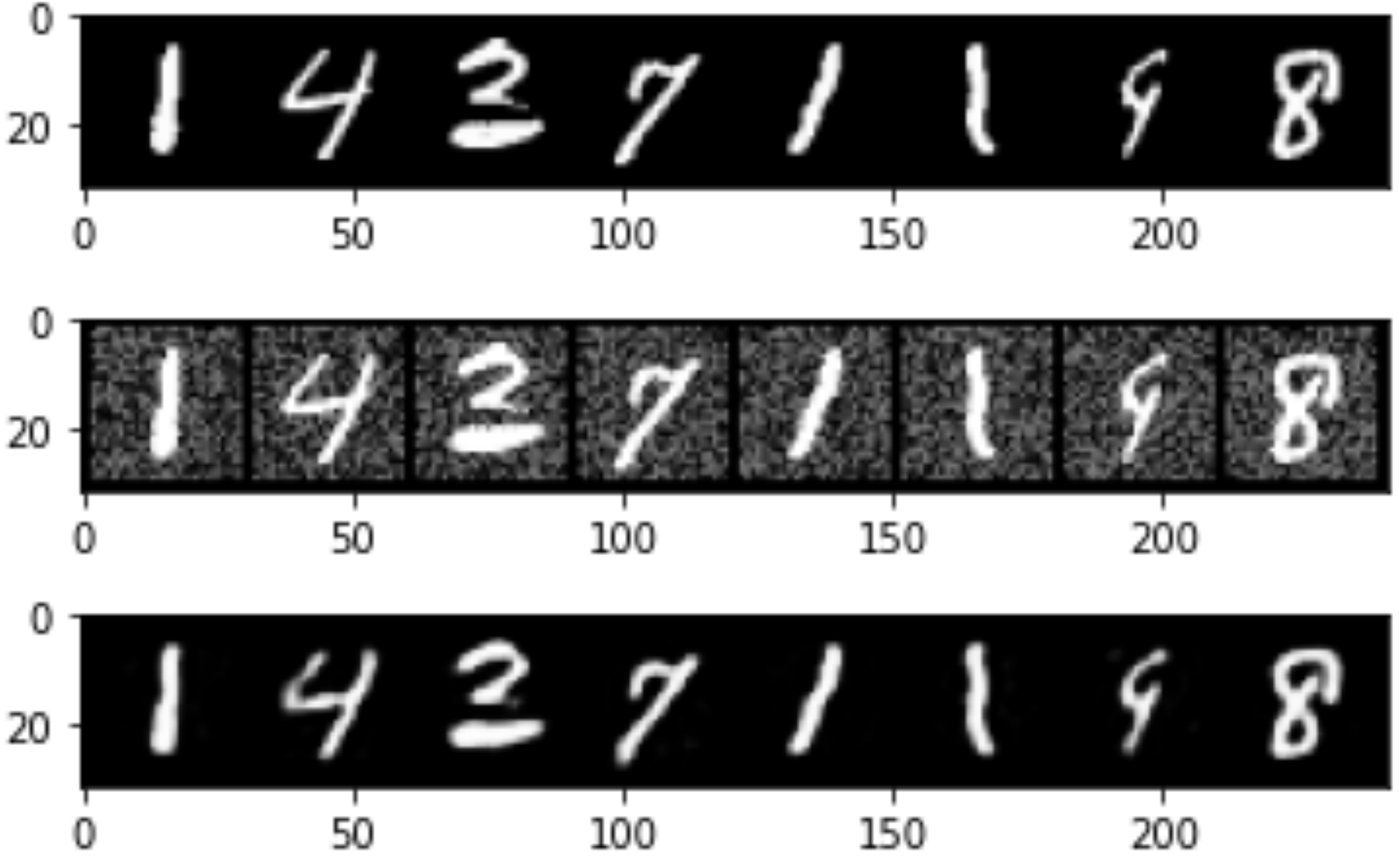
This mapping is not specific to particular types of noise
(that is, it could be masking noise, Gaussian noise, salt-and-pepper noise, etc).

In the original DAE paper, the noise is applied in this way: a fixed proportion of input dimensions are selected at random and their values are forced to 0.

This sounds like dropout (approach for regularisation applied in deep learning), but DAE came actually before dropout (Hinton, et al. 2012).

```
model = Autoencoder(input_size, hidden_size).to(device)

# Loss and optimiser
criterion = nn.MSELoss()
optimiser = torch.optim.Adam(model.parameters(), lr=learning_rate)

# Train the model
for epoch in range(num_epochs):
    for i, (images, _) in enumerate(train_loader):
        images = images.reshape(-1, 28*28).to(device)

        # Add noise to the images
        noise = torch.rand(images.shape) / 2.  # Modify the denominator to add more or less noise
        noisy_images = images + noise
        noisy_images = torch.clamp(noisy_images, 0., 1.)  # Clamp the values between 0 and 1

        # Forward pass
        outputs = model(noisy_images)
        loss = criterion(outputs, images)

        # Backward and optimise
        optimiser.zero_grad()
        loss.backward()
        optimiser.step()

        if (i + 1) % 100 == 0:
            print ('Epoch [{}/{}], Loss: {:.4f}'
                   .format(epoch+1, num_epochs, loss.item()))
```

```
# Test the model
model.eval()
total_loss = 0
with torch.no_grad():
    for i, (images, _) in enumerate(train_loader):
        images = images.reshape(-1, 28*28).to(device)
        # Add noise to the images
        noise = torch.rand(images.shape) / 2.
        noisy_images = images + noise
        noisy_images = torch.clamp(noisy_images, 0., 1.)  # Clamp the values between 0 and 1

        outputs = model(noisy_images)
        loss = criterion(outputs, images)
        total_loss += loss.item()

        if i == 0:
            print('Original images')
            out_imgs = torchvision.utils.make_grid(images.reshape(-1, 1, 28, 28)[:8])
            plt.imshow(out_imgs.permute(1, 2, 0).cpu().numpy())
            plt.show()
            print('Noisy inputs')
            out_imgs = torchvision.utils.make_grid(noisy_images.reshape(-1, 1, 28, 28)[:8])
            plt.imshow(out_imgs.permute(1, 2, 0).cpu().numpy())
            plt.show()
            print('Reconstructed images')
            out_imgs = torchvision.utils.make_grid(outputs.reshape(-1, 1, 28, 28)[:8])
            plt.imshow(out_imgs.permute(1, 2, 0).cpu().numpy())
            plt.show()

print('Average test loss: {:.4f}'.format(total_loss / len(test_loader)))
```


0
20
0
50
100
150
200
0
20
0
50
100
150
200
0
20
0
50
100
150
200

Sparse Autoencoder

Sparse Autoencoder apply a “sparse” constraint on the hidden unit activation.

This is another way to avoid overfitting and improve robustness.

It forces the model to only have a small number of hidden units being “activated” at the same time, or in other words, a hidden neuron should be inactive most of time.

Sparse Autoencoder

Sparse Autoencoder apply a “sparse” constraint on the hidden unit activation.

This is another way to avoid overfitting and improve robustness.

It forces the model to only have a small number of hidden units being “activated” at the same time, or in other words, a hidden neuron should be inactive most of time.

What is “active”?

Recall that common activation functions include sigmoid, tanh, relu, leaky relu, etc.
A neuron is activated when the value is close to 1 (or larger), inactive with a value close to 0.

Sparse Autoencoder

Sparse Autoencoder apply a “sparse” constraint on the hidden unit activation.

This is another way to avoid overfitting and improve robustness.

It forces the model to only have a small number of hidden units being “activated” at the same time, or in other words, a hidden neuron should be inactive most of time.

What is “active”?

Recall that common activation functions include sigmoid, tanh, relu, leaky relu, etc.
A neuron is activated when the value is close to 1 (or larger), inactive with a value close to 0.

(What is “inactive” for tanh?)

Sparse Autoencoder

If there are s_l neurons in the l-th hidden layer, and the activation function for the j-th neuron in that layer is labeled as $a_j^{(l)}(\cdot), \quad j = 1,\ldots,s_l$.

The fraction of activation of this neuron, $\hat{\rho}_j$, is expected to be a small number ρ.

Sparse Autoencoder

If there are s_l neurons in the l-th hidden layer, and the activation function for the j-th neuron in that layer is labeled as $a_j^{(l)}(\cdot), \quad j = 1,\ldots,s_l$.

The fraction of activation of this neuron, $\hat{\rho}_j$, is expected to be a small number ρ.

ρ is also called sparsity parameter. A typical values, for example, is $\rho = 0.05$.

$$\hat{\rho}_j^{(l)} = \frac{1}{n} \sum_{i=1}^{n} [a_j^{(l)}(\mathbf{x}^{(i)})] \approx \rho$$

Sparse Autoencoder

The sparsity constraint is achieved by adding a penalty term into the loss function.

We can view ρ and $\hat{\rho}_j^{(l)}$ as the means of two Bernoulli distributions.

Sparse Autoencoder

The sparsity constraint is achieved by adding a penalty term into the loss function.

We can view ρ and $\hat{\rho}_j^{(l)}$ as the means of two Bernoulli distributions.

We can measure the difference between distributions using the KL-divergence D_{KL}.

Sparse Autoencoder

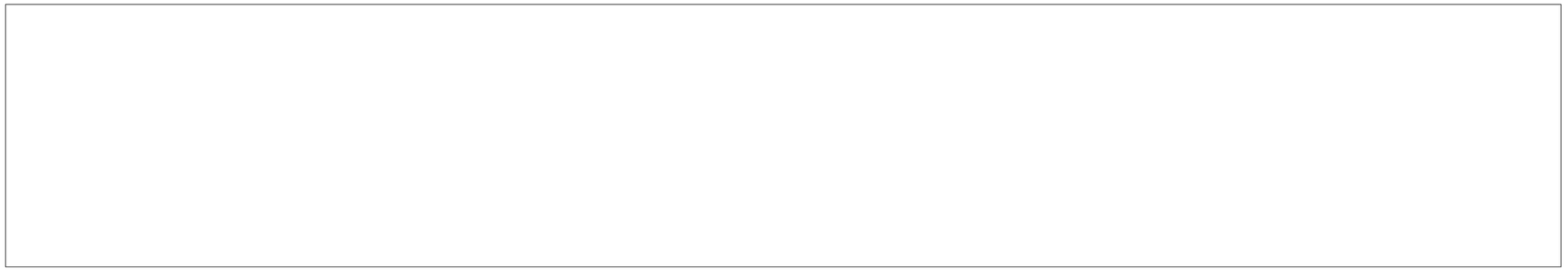
The sparsity constraint is achieved by adding a penalty term into the loss function.

We can view ρ and $\hat{\rho}_j^{(l)}$ as the means of two Bernoulli distributions.

We can measure the difference between distributions using the KL-divergence D_{KL}.

When we add D_{KL} as a penalty to the loss, we can control how strong the penalty should be by setting the hyperparameter β.

$$L_{\mathrm{SAE}}(\theta) = L(\theta) + \beta \sum_{l=1}^{L} \sum_{j=1}^{s_l} D_{\mathrm{KL}}(\rho \| \hat{\rho}_j^{(l)})$$



Sparse Autoencoder

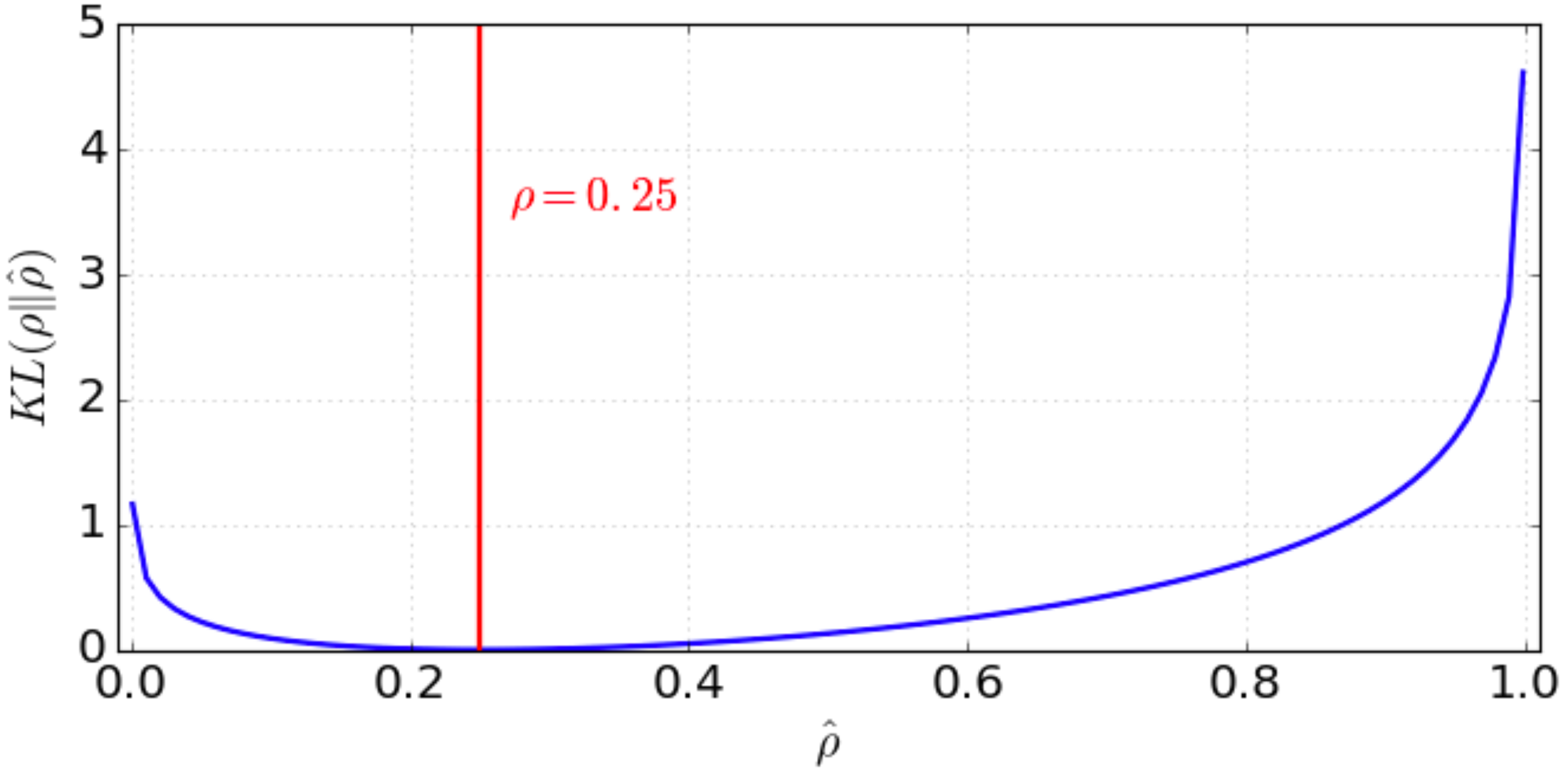
The sparsity constraint is achieved by adding a penalty term into the loss function.

We can view ρ and $\hat{\rho}_j^{(l)}$ as the means of two Bernoulli distributions.

We can measure the difference between distributions using the KL-divergence D_{KL}.

When we add D_{KL} as a penalty to the loss, we can control how strong the penalty should be by setting the hyperparameter β.

$$
\begin{aligned}
L_{\mathrm{SAE}}(\theta) &= L(\theta) + \beta \sum_{l=1}^{L} \sum_{j=1}^{s_l} D_{\mathrm{KL}}(\rho \| \hat{\rho}_j^{(l)}) \\
&= L(\theta) + \beta \sum_{l=1}^{L} \sum_{j=1}^{s_l} \rho \log \frac{\rho}{\hat{\rho}_j^{(l)}} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j^{(l)}}
\end{aligned}
$$

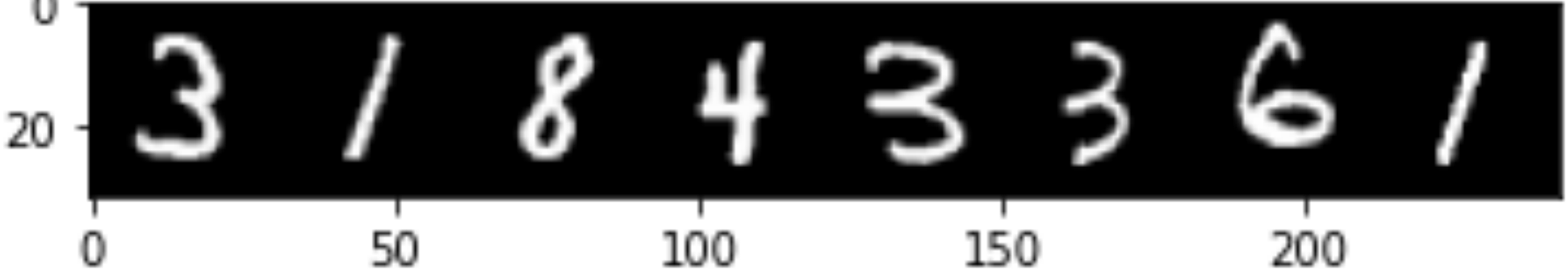


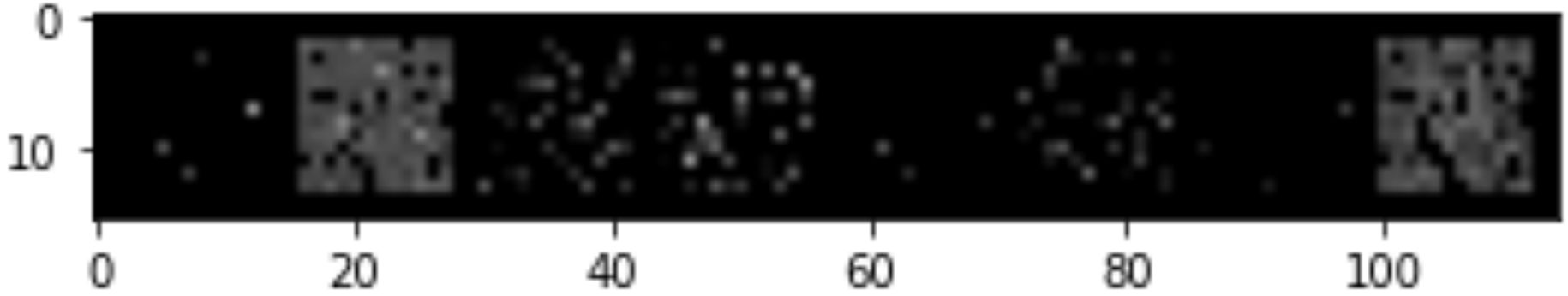
The KL divergence between a Bernoulli distribution with mean $\rho = 0.25$, and another Bernoulli distribution, $0 \leq \hat{\rho} \leq 1$.

```python
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

# Parameters
input_size = 784  # 28x28
hidden_size = 128
num_epochs = 5
batch_size = 100
learning_rate = 0.001
rho = 0.05  # Sparsity parameter
beta = 3  # Weight of sparsity penalty term
```

```
# Autoencoder model
class SparseAutoencoder(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(SparseAutoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            nn.ReLU())
        self.decoder = nn.Sequential(
            nn.Linear(hidden_size, input_size),
            nn.Sigmoid())

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x

    def encode(self, x): return self.encoder(x)

# Model, Loss and optimiser
model = SparseAutoencoder(input_size, hidden_size).to(device)
criterion = nn.MSELoss()
optimiser = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

```
# Train the model
for epoch in range(num_epochs):
    for i, (images, _) in enumerate(train_loader):
        images = images.reshape(-1, 28*28).to(device)

        # Forward pass
        outputs = model(images)
        mse_loss = criterion(outputs, images)

        # Sparsity penalty
        rho_hat = torch.mean(model.encoder(images), 0)  # Average hidden layer activation
        sparsity_penalty = torch.sum(rho * torch.log(rho / rho_hat) + (1 - rho) *
torch.log((1 - rho) / (1 - rho_hat)))

        # Total loss
        loss = mse_loss + beta * sparsity_penalty

        # Backward and optimise
        optimiser.zero_grad()
        loss.backward()
        optimiser.step()

        if (i+1) % 100 == 0:
            print ('Epoch [{}/{}], Loss: {:.4f}'
                   .format(epoch+1, num_epochs, loss.item()))
```





144 hidden units
Sparse activation

Visualised as
12x12 images

k-sparse auto encoder

Only keep the top k activated units

Sparsity is enforced by only keeping the top k highest activations in the bottleneck layer.

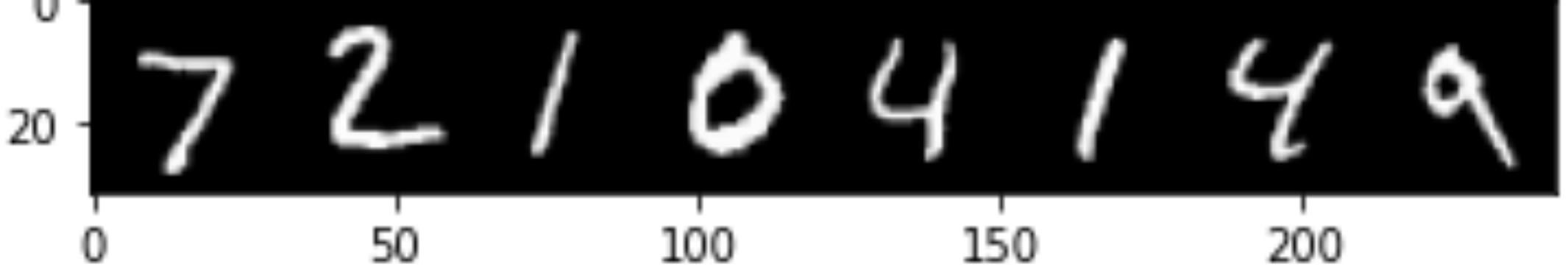
We run feedforward through the encoder network to get the compressed code: $\mathbf{z} = g(\mathbf{x})$.

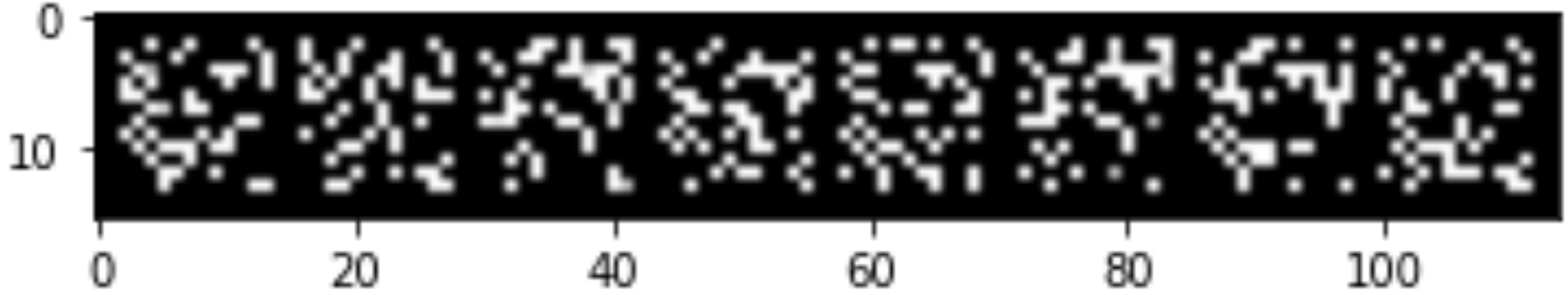
Then, sorted by values in $\mathbf{z}$, we leave the top k ones, and set all others in $\mathbf{z}$ to 0.

$\mathbf{z}' = \text{k\_sparse}(\mathbf{z})$

The loss is computed from this sparsified code: $L = |\mathbf{x} - f(\mathbf{z}')|_2^2$

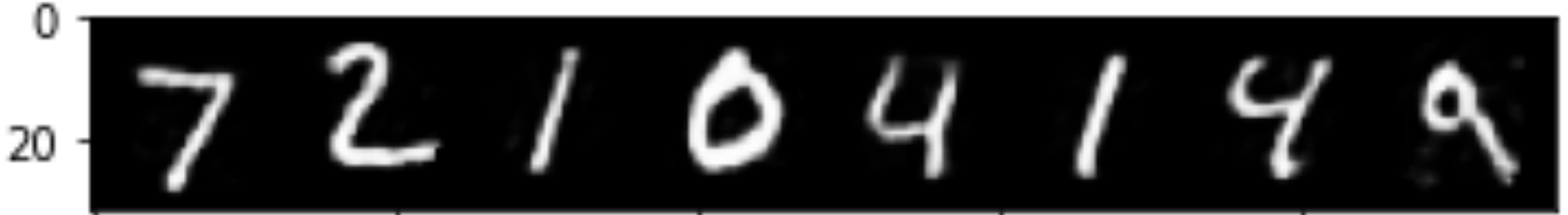
```
class KSparseAutoencoder(nn.Module):
    def __init__(self, input_size, hidden_size, k):
        super(KSparseAutoencoder, self).__init__()
        self.k = k
        self.encoder = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            nn.ReLU())
        self.decoder = nn.Sequential(
            nn.Linear(hidden_size, input_size),
            nn.Sigmoid())

    def forward(self, x):
        x = self.encoder(x)
        x = self.k_sparse(x)
        x = self.decoder(x)
        return x

    def k_sparse(self, x):
        topk, indices = torch.topk(x, self.k)
        x_sparse = torch.zeros_like(x).to(x.device)
        x_sparse.scatter_(1, indices, topk)
        return x_sparse

    def encode(self, x):  return self.k_sparse(self.encoder(x))
```

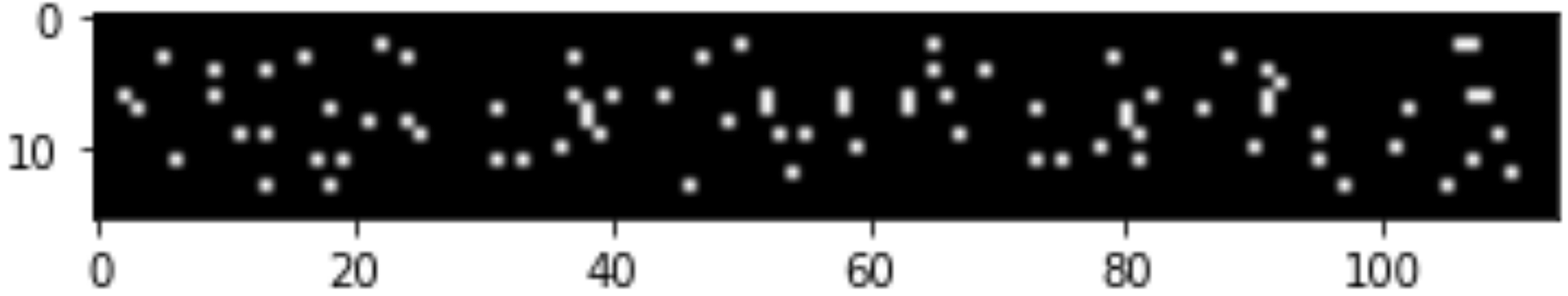




144 hidden units
44-Sparse activation

Visualised as
12x12 images





144 hidden units
10-Sparse activation

Visualised as
12x12 images

Contractive autoencoder

Robust to small changes

"Contractive": a function that brings inputs closer together in the output space. Similar goal to sparse autoencoder.

How to achieve "contractive"? By adding a term in the loss function to penalise representations being *too sensitive* to the input.

"Sensitive": the hidden representation shouldn't change much when the input changes a little.

We can measure change by looking at the partial derivatives (the Jacobian matrix J). The "length" of such a matrix is measured by the Frobenius norm (think Euclidean distance).

$$\|J_f(\mathbf{x})\|_F^2 = \sum_{ij} \left(\frac{\partial h_j(\mathbf{x})}{\partial x_i} \right)^2$$

Variational Autoencoder

VAE

Mapping input to a distribution

We map the input into a distribution, instead of mapping it to a single data point.

Calling this distribution p_θ, we can then describe the relation between input $\mathbf{x}$ and $\mathbf{z}$:

- Prior $p_\theta(\mathbf{z})$

- Likelihood $p_\theta(\mathbf{x} \mid \mathbf{z})$

- Posterior $p_\theta(\mathbf{z} \mid \mathbf{x})$

We have to find the optimal parameter θ^*.

In order to generate a sample that looks like a real data point $\mathbf{x}^{(i)}$:

- We sample a $\mathbf{z}^{(i)}$ from a prior distribution $p_{\theta^*}(\mathbf{z})$
- Generate a value $\mathbf{x}^{(i)}$ from the conditional distribution $p_{\theta^*}(\mathbf{x} \mid \mathbf{z} = \mathbf{z}^{(i)})$

In order to generate a sample that looks like a real data point $\mathbf{x}^{(i)}$:

- We sample a $\mathbf{z}^{(i)}$ from a prior distribution $p_{\theta*}(\mathbf{z})$
- Generate a value $\mathbf{x}^{(i)}$ from the conditional distribution $p_{\theta*}(\mathbf{x} \mid \mathbf{z} = \mathbf{z}^{(i)})$

The optimal parameter $\theta*$ is the one that maximises the probability of generating "real" data samples:

$$\theta* = \arg\max_{\theta} \prod_{i=1}^{n} p_{\theta}(\mathbf{x}^{(i)})$$

Computing the product: $\arg\max_{\theta} \prod_{i=1}^{n} p_\theta(\mathbf{x}^{(i)})$

to maximise θ^* is equivalent to computing $\theta^* = \arg\max_{\theta} \sum_{i=1}^{n} \log p_\theta(\mathbf{x}^{(i)})$.

Computing the product: $\arg\max_\theta \prod_{i=1}^{n} p_\theta(\mathbf{x}^{(i)})$

to maximise θ^* is equivalent to computing $\theta^* = \arg\max_\theta \sum_{i=1}^{n} \log p_\theta(\mathbf{x}^{(i)})$.

As we discussed in the class on variational inference, this is what we need to compute:

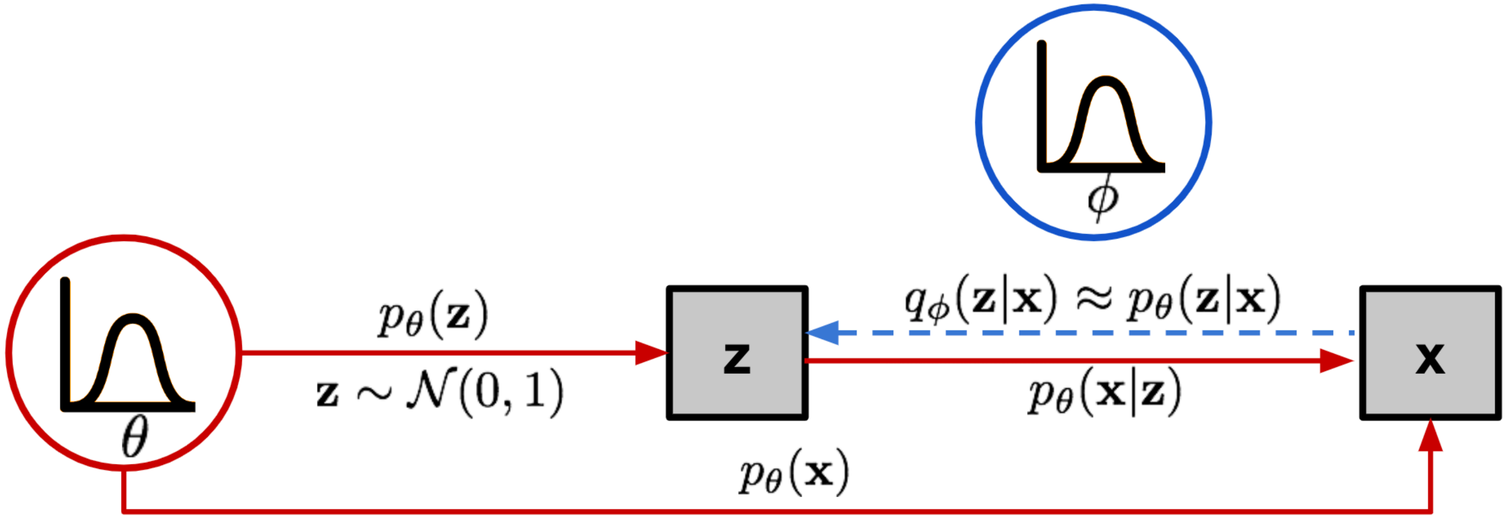
$$p_\theta(\mathbf{x}^{(i)}) = \int p_\theta(\mathbf{x}^{(i)} \mid \mathbf{z})\, p_\theta(\mathbf{z})\, d\mathbf{z}$$

but in many cases this is not easy to compute, because it involves summing up for all possible values of $\mathbf{z}$.

In variational inference, the way to solve this intractable problem is to approximate the output:

The function q_ϕ should tell us what is the code $\mathbf{z}$ given some input $\mathbf{x}$.

$q_\phi(\mathbf{z} \mid \mathbf{x})$ is an approximation of $p_\theta(\mathbf{z} \mid \mathbf{x})$, parameterised by ϕ.



Now the structure looks like an auto encoder

- The conditional probability $p_\theta(\mathbf{x} \mid \mathbf{z})$ defines a generative model, similar to the decoder $f_\theta(\mathbf{x} \mid \mathbf{z})$ introduced above. $p_\theta(\mathbf{x} \mid \mathbf{z})$ is also known as *probabilistic decoder*.

- The approximation function $q_\phi(\mathbf{z} \mid \mathbf{x})$ is the *probabilistic encoder*, playing a similar role to $g_\phi(\mathbf{z} \mid \mathbf{x})$ above.

Now the structure looks like an auto encoder

- The conditional probability $p_\theta(\mathbf{x} \mid \mathbf{z})$ defines a generative model, similar to the decoder $f_\theta(\mathbf{x} \mid \mathbf{z})$ introduced above. $p_\theta(\mathbf{x} \mid \mathbf{z})$ is also known as *probabilistic decoder*.

- The approximation function $q_\phi(\mathbf{z} \mid \mathbf{x})$ is the *probabilistic encoder*, playing a similar role to $g_\phi(\mathbf{z} \mid \mathbf{x})$ above.

In the VI lecture (week 9), we defined the Evidence Lower Bound (ELBO) to come up with a cost function. There, our aim was to approximate the true posterior distribution $p(\mathbf{z} \mid \mathbf{x})$ with a simpler, tractable distribution $q(\mathbf{z})$.

ELBO

The evidence lower bound

Defined as:

$$\text{ELBO}(q) = \mathbb{E}_q\left[\log p(\mathbf{x}, \mathbf{z})\right] - \mathbb{E}_q\left[\log q(\mathbf{z})\right]$$

ELBO

The evidence lower bound

Defined as:

$$\mathrm{ELBO}(q) = \mathbb{E}_q\left[\log p(\mathbf{x}, \mathbf{z})\right] - \mathbb{E}_q\left[\log q(\mathbf{z})\right]$$

It is called the evidence lower bound, because it is the lower bound of logarithmic evidence:

$$\begin{aligned}\log p(\mathbf{x}) &= \mathrm{ELBO}(q) + D_{\mathrm{KL}}(q(\mathbf{z}) \| p(\mathbf{z} \mid \mathbf{x})) \\ &\geq \mathrm{ELBO}(q)\end{aligned}$$

ELBO

The evidence lower bound

$$D_{\mathrm{KL}}(q(\mathbf{z})\|p(\mathbf{z}\,|\,\mathbf{x})) = \mathbb{E}_q\left[\log q(\mathbf{z})\right] - \mathbb{E}_q\left[\log p(\mathbf{x},\mathbf{z})\right] + \log p(\mathbf{x})$$

Defined as:

$$\mathrm{ELBO}(q) = \mathbb{E}_q\left[\log p(\mathbf{x},\mathbf{z})\right] - \mathbb{E}_q\left[\log q(\mathbf{z})\right]$$

It is called the evidence lower bound, because it is the lower bound of logarithmic evidence:

$$\begin{aligned}\log p(\mathbf{x}) &= \mathrm{ELBO}(q) + D_{\mathrm{KL}}(q(\mathbf{z})\|p(\mathbf{z}\,|\,\mathbf{x})) \\ &\geq \mathrm{ELBO}(q)\end{aligned}$$

Maximising the ELBO for VAEs

Goal: maximise the ELBO with respect to the variational distribution $q(\mathbf{z})$.

This is equivalent to minimising the KL divergence between $q(\mathbf{z})$ and the true posterior $p(\mathbf{z} \mid \mathbf{x})$, which is the ultimate goal.

Maximising the ELBO for VAEs

Goal: maximise the ELBO with respect to the variational distribution $q(\mathbf{z})$.

This is equivalent to minimising the KL divergence between $q(\mathbf{z})$ and the true posterior $p(\mathbf{z} \mid \mathbf{x})$, which is the ultimate goal.

In VAE, we work with a conditional distribution $q(\mathbf{z} \mid \mathbf{x})$ that depends on observed data.

Maximising the ELBO for VAEs

Goal: maximise the ELBO with respect to the variational distribution $q(\mathbf{z})$.

This is equivalent to minimising the KL divergence between $q(\mathbf{z})$ and the true posterior $p(\mathbf{z} \mid \mathbf{x})$, which is the ultimate goal.

In VAE, we work with a conditional distribution $q(\mathbf{z} \mid \mathbf{x})$ that depends on observed data.

The ELBO then becomes:

Maximising the ELBO for VAEs

Goal: maximise the ELBO with respect to the variational distribution $q(\mathbf{z})$.

This is equivalent to minimising the KL divergence between $q(\mathbf{z})$ and the true posterior $p(\mathbf{z} \mid \mathbf{x})$, which is the ultimate goal.

In VAE, we work with a conditional distribution $q(\mathbf{z} \mid \mathbf{x})$ that depends on observed data.

The ELBO then becomes:

$$\text{ELBO}(\mathbf{q}) = \mathbb{E}_q[\log p(\mathbf{x} \mid \mathbf{z})] - D_{\text{KL}}(q(\mathbf{z} \mid \mathbf{x}) \,\|\, p(\mathbf{z}))$$

Maximising the ELBO for VAEs

Goal: maximise the ELBO with respect to the variational distribution $q(\mathbf{z})$.

This is equivalent to minimising the KL divergence between $q(\mathbf{z})$ and the true posterior $p(\mathbf{z} \mid \mathbf{x})$, which is the ultimate goal.

In VAE, we work with a conditional distribution $q(\mathbf{z} \mid \mathbf{x})$ that depends on observed data.

The ELBO then becomes:

$$\text{ELBO}(\mathbf{q}) = \mathbb{E}_q[\log p(\mathbf{x} \mid \mathbf{z})] - D_{\text{KL}}(q(\mathbf{z} \mid \mathbf{x}) \,\|\, p(\mathbf{z}))$$

Expected log likelihood of the data
(Reconstruction term)

Maximising the ELBO for VAEs

Goal: maximise the ELBO with respect to the variational distribution $q(\mathbf{z})$.

This is equivalent to minimising the KL divergence between $q(\mathbf{z})$ and the true posterior $p(\mathbf{z} \mid \mathbf{x})$, which is the ultimate goal.

In VAE, we work with a conditional distribution $q(\mathbf{z} \mid \mathbf{x})$ that depends on observed data.

The ELBO then becomes:

$$\text{ELBO}(\mathbf{q}) = \mathbb{E}_q[\log p(\mathbf{x} \mid \mathbf{z})] - D_{\text{KL}}(q(\mathbf{z} \mid \mathbf{x}) \,\|\, p(\mathbf{z}))$$

Expected log likelihood of the data (Reconstruction term)

KL divergence between variational dist and the prior (regularisation term)

Maximising the ELBO for VAEs

Goal: maximise the ELBO with respect to the variational distribution $q(\mathbf{z})$.

This is equivalent to minimising the KL divergence between $q(\mathbf{z})$ and the true posterior $p(\mathbf{z} \mid \mathbf{x})$, which is the ultimate goal.

In VAE, we work with a conditional distribution $q(\mathbf{z} \mid \mathbf{x})$ that depends on observed data.

The ELBO then becomes:

$$\text{ELBO}(\mathbf{q}) = \mathbb{E}_q[\log p(\mathbf{x} \mid \mathbf{z})] - D_{\text{KL}}(q(\mathbf{z} \mid \mathbf{x}) \,\|\, p(\mathbf{z}))$$

Expected log likelihood of the data (Reconstruction term)

KL divergence between variational dist and the prior (regularisation term)

$$= \log p_\theta(\mathbf{x}) - D_{\text{KL}}(q_\phi(\mathbf{z} \mid \mathbf{x}) \,\|\, p_\theta(\mathbf{z} \mid \mathbf{x}))$$

Maximising the ELBO for VAEs

Goal: maximise the ELBO with respect to the variational distribution $q(\mathbf{z})$.

This is equivalent to minimising the KL divergence between $q(\mathbf{z})$ and the true posterior $p(\mathbf{z} \mid \mathbf{x})$, which is the ultimate goal.

In VAE, we work with a conditional distribution $q(\mathbf{z} \mid \mathbf{x})$ that depends on observed data.

The ELBO then becomes:

$$\text{ELBO}(\mathbf{q}) = \mathbb{E}_q[\log p(\mathbf{x} \mid \mathbf{z})] - D_{\text{KL}}(q(\mathbf{z} \mid \mathbf{x}) \,\|\, p(\mathbf{z}))$$

Expected log likelihood of the data (Reconstruction term)

KL divergence between variational dist and the prior (regularisation term)

$$= \log p_\theta(\mathbf{x}) - D_{\text{KL}}(q_\phi(\mathbf{z} \mid \mathbf{x}) \,\|\, p_\theta(\mathbf{z} \mid \mathbf{x}))$$

That is, we want to maximise the (log-)likelihood of generating real data , while minimising the difference between the real and the estimated posterior.

D_{KL} is not symmetric

Use $D_{\mathrm{KL}}(q_\phi \| p_\theta)$ or $D_{\mathrm{KL}}(p_\theta \| q_\phi)$?

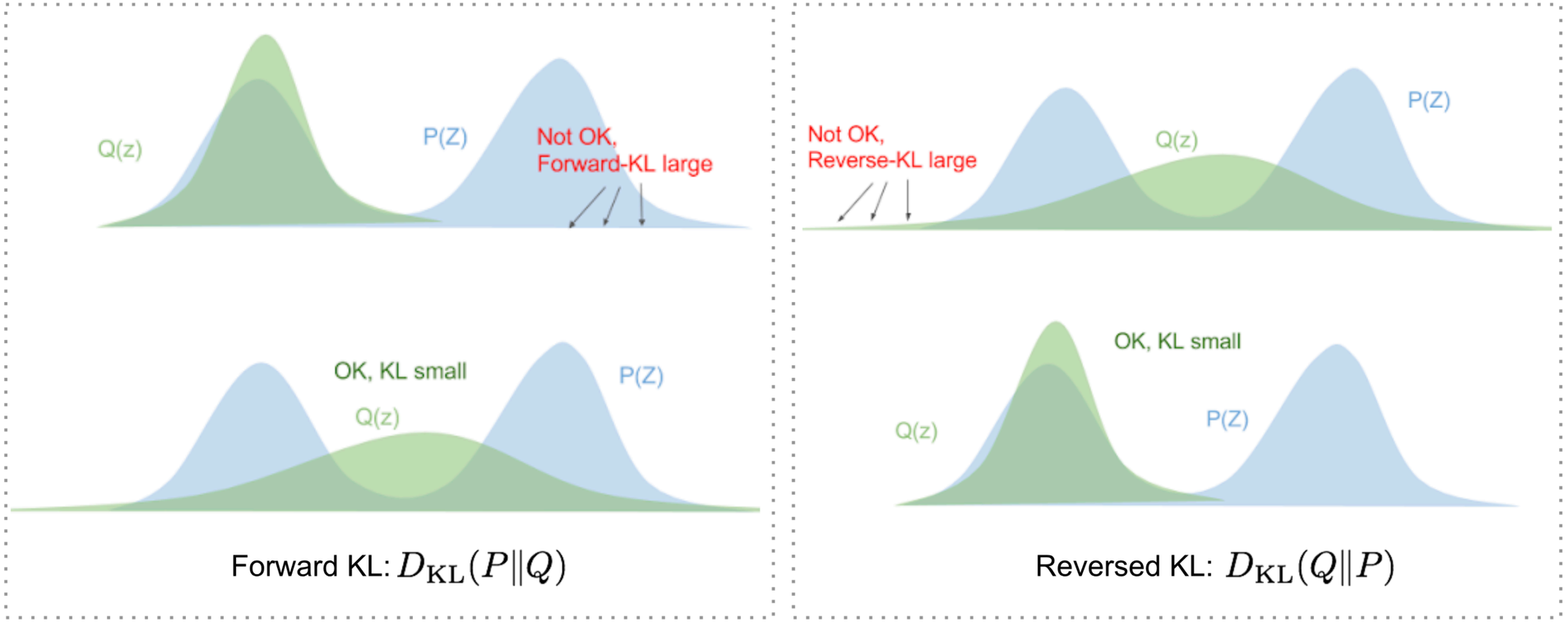


Figure by Eric Jang

For a discussion, see also "A Beginner's Guide to Variational Methods" https://blog.evjang.com/2016/08/variational-bayes.html by Eric Jang.

Reparametrisation trick

When we use neural networks to represent the distribution, we have to compute gradients to optimise (for gradient descent).

The expectation term in the loss function invokes generating samples from $\mathbf{z} \sim q_\phi(\mathbf{z} \mid \mathbf{x})$.
But sampling is a stochastic process and therefore we cannot backpropagate the gradient.

Reparametrisation trick

When we use neural networks to represent the distribution, we have to compute gradients to optimise (for gradient descent).

The expectation term in the loss function invokes generating samples from $\mathbf{z} \sim q_\phi(\mathbf{z} \mid \mathbf{x})$. But sampling is a stochastic process and therefore we cannot backpropagate the gradient.

A ‘reparameterisation trick’ solves this problem:

Reparametrisation trick

When we use neural networks to represent the distribution, we have to compute gradients to optimise (for gradient descent).

The expectation term in the loss function invokes generating samples from $\mathbf{z} \sim q_\phi(\mathbf{z} \mid \mathbf{x})$. But sampling is a stochastic process and therefore we cannot backpropagate the gradient.

A 'reparameterisation trick' solves this problem:

- express the random variable $\mathbf{z}$ as a deterministic variable $\mathbf{z} = T_\phi(\mathbf{x}, \epsilon)$. Here, ϵ is an auxiliary random variable, and the transformation function T_ϕ converts ϵ to $\mathbf{z}$.

Reparametrisation trick

When we use neural networks to represent the distribution, we have to compute gradients to optimise (for gradient descent).

The expectation term in the loss function invokes generating samples from $\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})$. But sampling is a stochastic process and therefore we cannot backpropagate the gradient.

A 'reparameterisation trick' solves this problem:

- express the random variable $\mathbf{z}$ as a deterministic variable $\mathbf{z} = T_\phi(\mathbf{x}, \epsilon)$. Here, ϵ is an auxiliary random variable, and the transformation function T_ϕ converts ϵ to $\mathbf{z}$.

- Example:

$$\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x}^{(i)}) = \mathcal{N}(\mathbf{z}; \mu^{(i)}, \sigma^{2(i)}\mathbf{I})$$

$$\mathbf{z} = \mu + \sigma \odot \epsilon, \quad \text{where } \epsilon \sim \mathcal{N}(0,\mathbf{I})$$

$\odot$ is the element-wise multiplication

Reparameterisation trick

Summary

- we have a random variable $\mathbf{z}$ that is generated by transforming a parameter θ through a probability distribution: $\mathbf{z} = p(\mathbf{z} \mid \theta)$
- We want to create a sample from this, AND
  compute the gradient of a function $f(\mathbf{z})$ with respect to θ.
- We express $\mathbf{z}$ as a deterministic function $g(\epsilon, \theta)$
- While ϵ is still a source of randomness, the function $g(\epsilon, \theta)$ is diff'able wrt θ.
  This means we can compute the gradient.

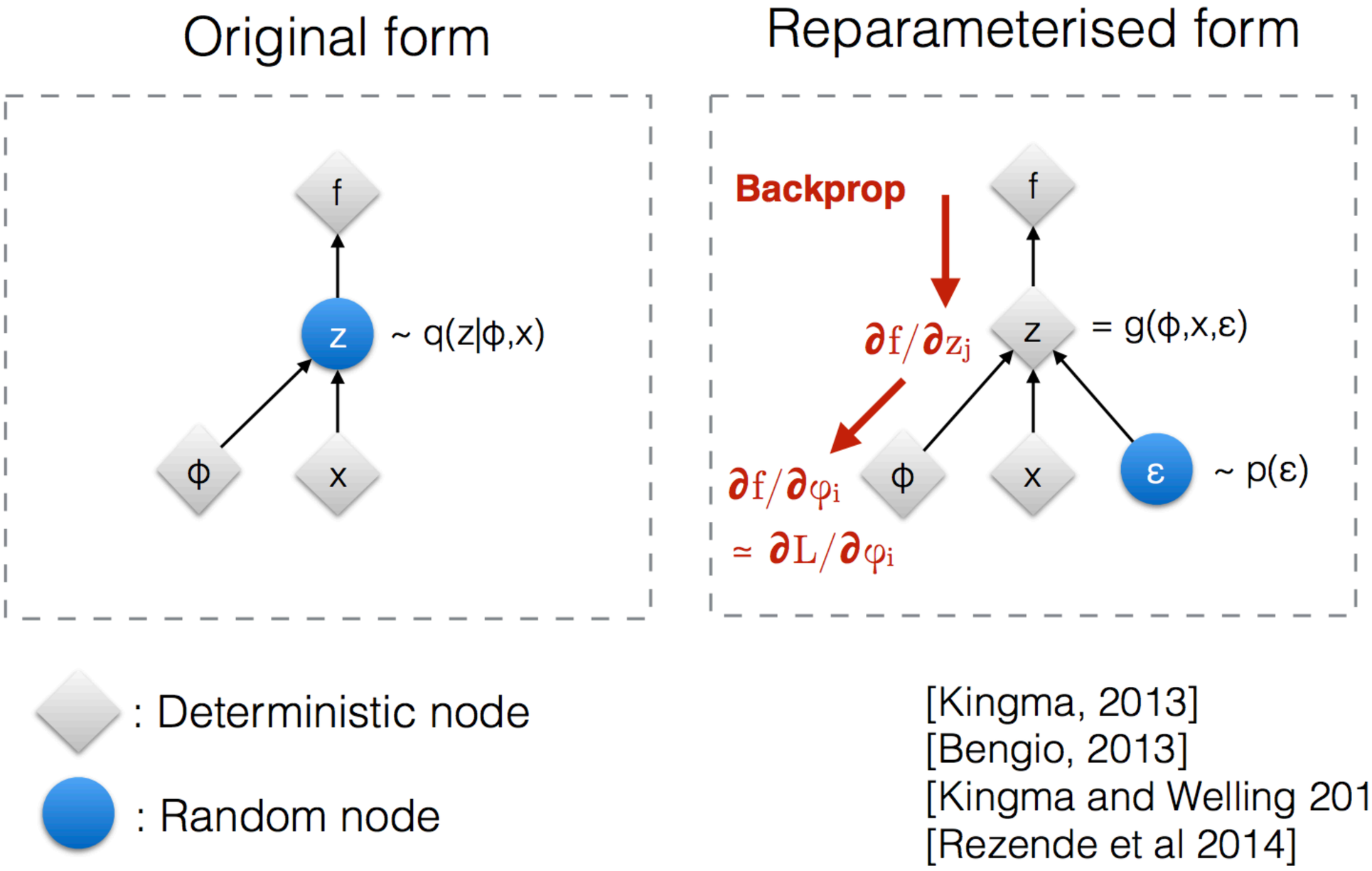
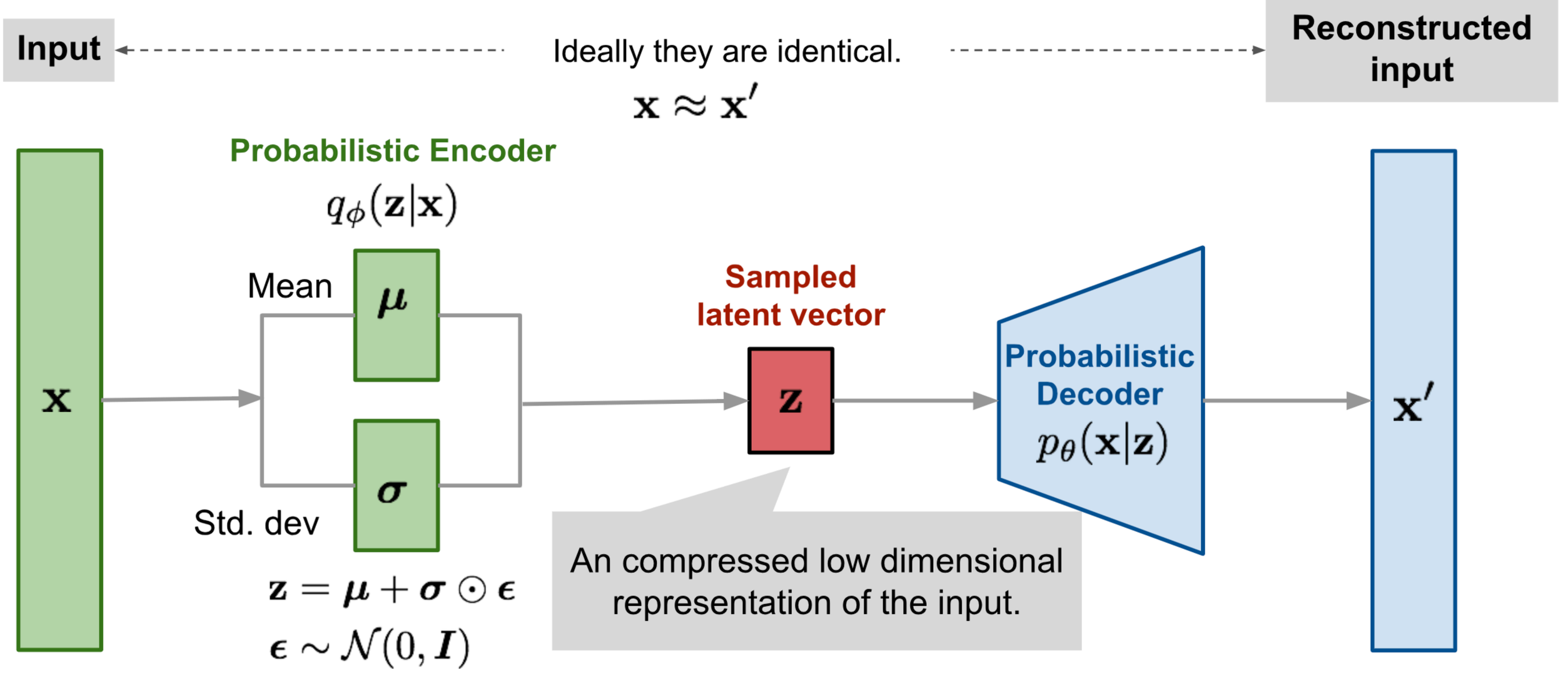
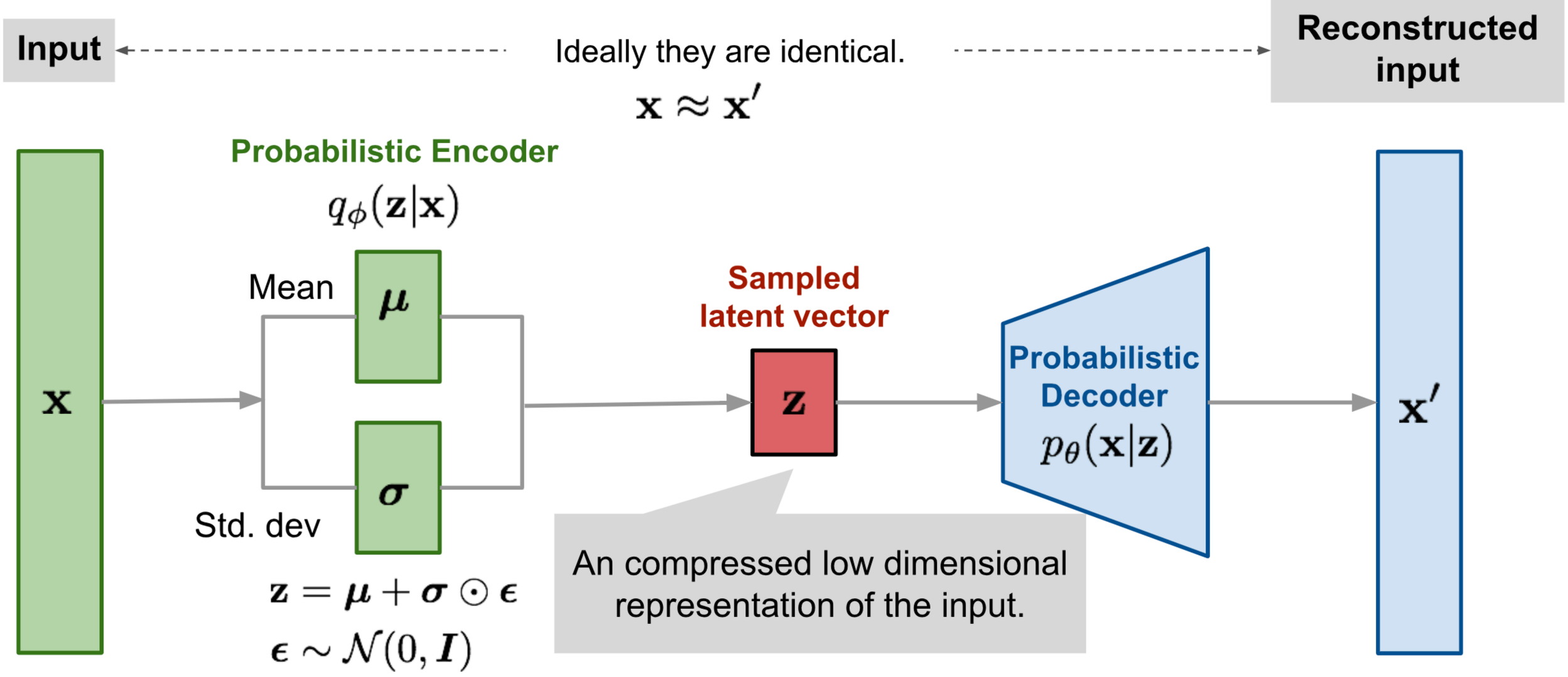


Illustration of how the reparameterisation trick makes the **z** sampling process trainable.


Input
Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Reconstructed input
Probabilistic Encoder
$q_\phi(\mathbf{z}|\mathbf{x})$
Mean
$\boldsymbol{\mu}$
$\boldsymbol{\sigma}$
Std. dev
$\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$
$\boldsymbol{\epsilon} \sim \mathcal{N}(0, \boldsymbol{I})$
$\mathbf{x}$
Sampled latent vector
$\mathbf{z}$
An compressed low dimensional representation of the input.
Probabilistic Decoder
$p_\theta(\mathbf{x}|\mathbf{z})$
$\mathbf{x}'$

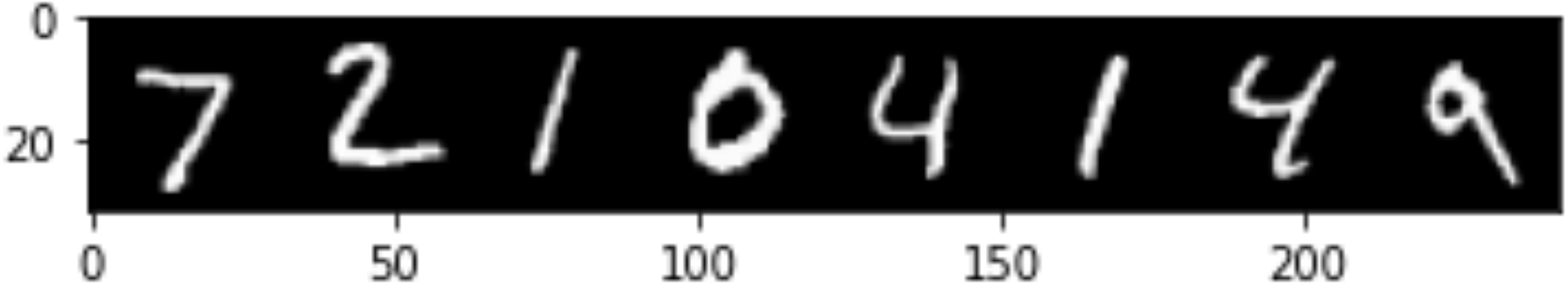


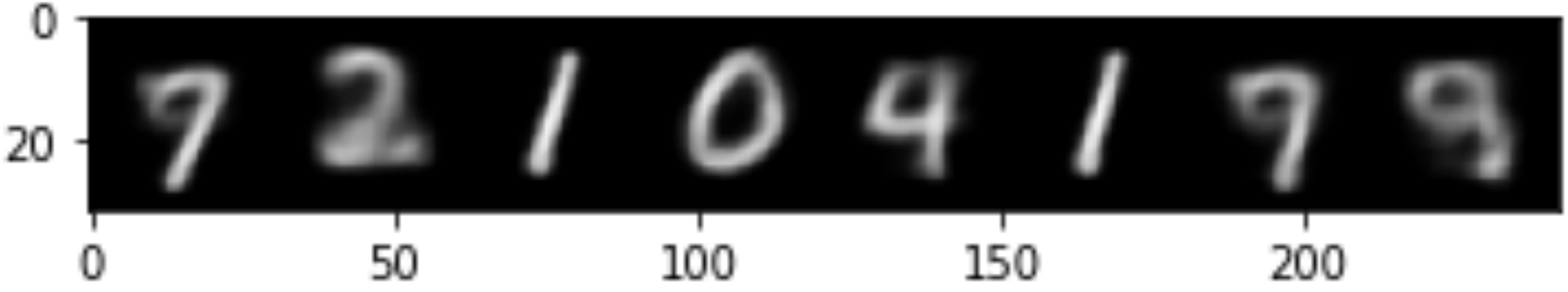
The reparameterisation trick works for other types of distributions too, not only Gaussian. In the multivariate Gaussian case, we make the model trainable by learning the mean and variance of the distribution explicitly using the reparameterisation trick, while the stochasticity remains in the random variable $\epsilon \sim \mathcal{N}(0,\mathbf{I})$

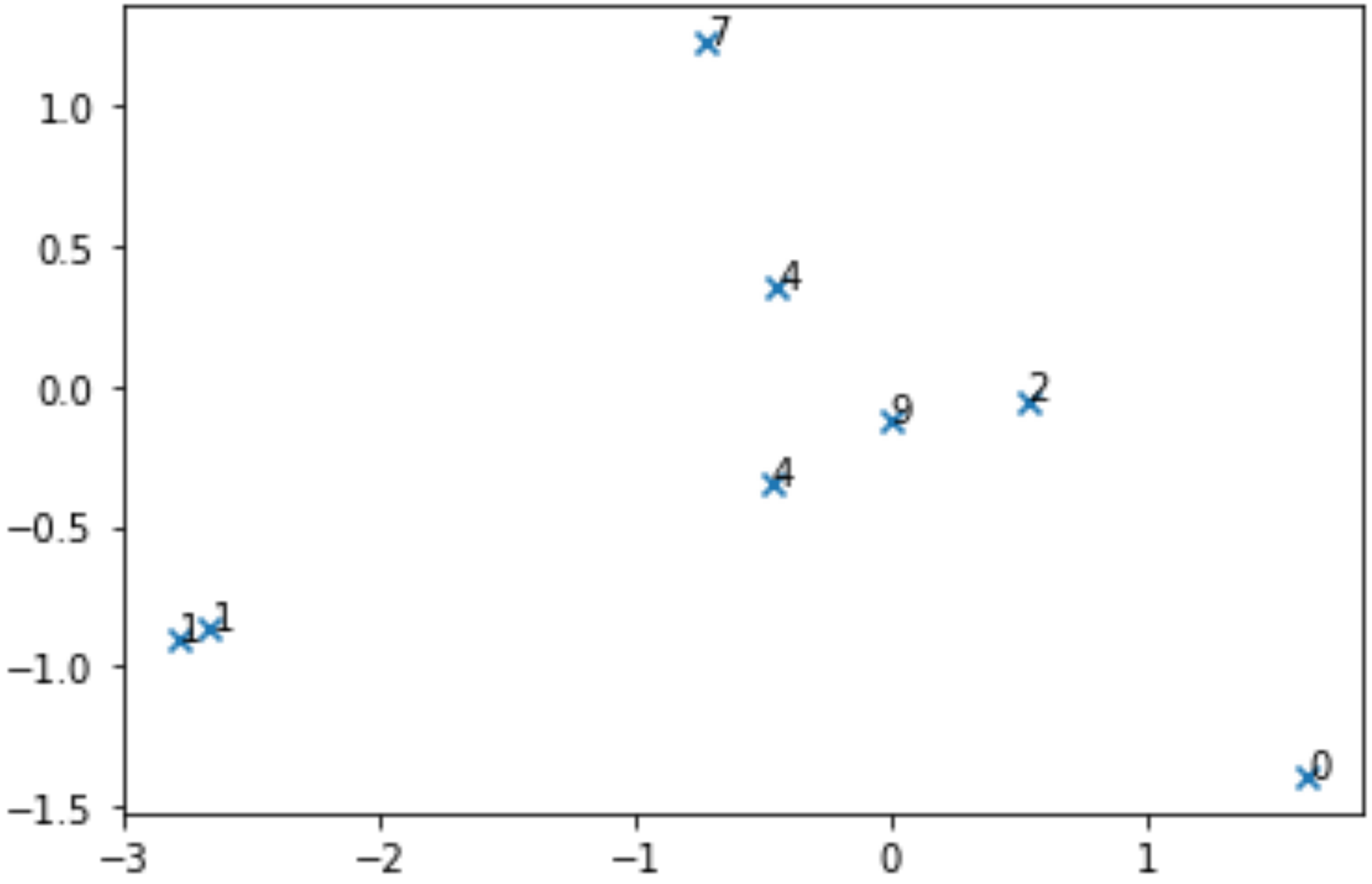
```python
class VAE(nn.Module):
    def __init__(self, input_dim, hidden_dim, latent_dim):
        super(VAE, self).__init__()
        # Encoder
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc_mu = nn.Linear(hidden_dim, latent_dim)
        self.fc_logvar = nn.Linear(hidden_dim, latent_dim)
        # Decoder
        self.fc3 = nn.Linear(latent_dim, hidden_dim)
        self.fc4 = nn.Linear(hidden_dim, input_dim)

    def encode(self, x):
        h = F.relu(self.fc1(x))
        mu = self.fc_mu(h)
        logvar = self.fc_logvar(h)
        return mu, logvar

    def reparameterise(self, mu, logvar):
        std = torch.exp(0.5*logvar)
        eps = torch.randn_like(std)
        return mu + eps*std

    def decode(self, z):
        h = F.relu(self.fc3(z))
        return torch.sigmoid(self.fc4(h))

    def forward(self, x):
        mu, logvar = self.encode(x.view(-1, 784))
        z = self.reparameterise(mu, logvar)
        return self.decode(z), mu, logvar
```

```python
# Reconstruction + KL divergence losses summed over all elements and batch
def loss_function(recon_x, x, mu, logvar):
    BCE = F.binary_cross_entropy(recon_x, x.view(-1, 784), reduction='sum')

    # see Appendix B from VAE paper:
    # Kingma and Welling. Auto-Encoding Variational Bayes. ICLR, 2014
    # https://arxiv.org/abs/1312.6114
    # 0.5 * sum(1 + log(sigma^2) - mu^2 - sigma^2)
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())

    return BCE + KLD
```

```
import torch
from torch import nn
from torch.nn import functional as F
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

# Parameters
input_size = 784  # 28x28
latent_dim = 2
hidden_sqrt = 12   # 12x12 = 144 hidden units
hidden_size = hidden_sqrt * hidden_sqrt
num_epochs = 40
batch_size = 100
learning_rate = 0.001
```


0
20
0
50
100
150
200

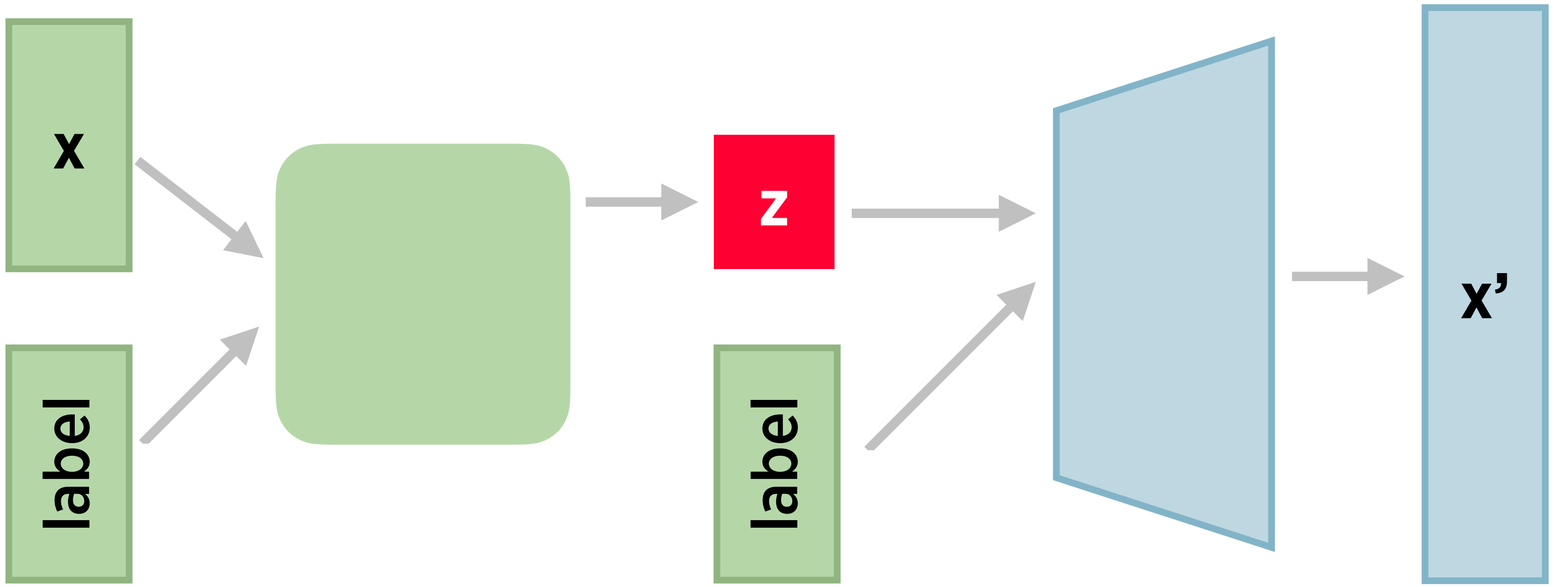

0
20
0
50
100
150
200


7
4
2
9
4
1
1
0
1.0
0.5
0.0
−0.5
−1.0
−1.5
−3
−2
−1
0
1

Conditional Variational Autoencoder (CVAE)

See also: https://medium.com/@sofeikov/implementing-conditional-variational-auto-encoders-cvae-from-scratch-29fcbb8cb08f

- Generating new images of a specific class seems to be tricky
- A Conditional Variational Autoencoder fixes this:
- It takes as input both the data and the label (both at the encoder and the decoder)



Summary

Autoencoders

- “Vanilla” autoencoder
- Denoising autoencoder
- Sparse autoencoder
- K-sparse autoencoder
- Variational autoencoder

Creating representations from our data sets (for adding, e.g., a classifier),

Data compression (transmitting only the hidden layer representation),

Generating new data