CS-E4890: Deep Learning
Deep autoencoders

Alexander Ilin

Motivation

- Supervised learning problems: datasets consist of input-output pairs

$$(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), \ldots, (\mathbf{x}^{(n)}, \mathbf{y}^{(n)})$$

- Deep learning: supervised learning solved.
- Unsupervised learning: Make computers learn from unlabeled data

$$\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}$$

- Unsupervised learning seems important for building intelligent systems that can learn quickly. We humans learn a lot from unlabeled data.
- Unsupervised learning can be useful for:
  - representation learning (learning features useful for supervised learning problems)
  - detect samples that look different from training population (novelty/anomaly detection)
  - visualize data, discover patterns (information visualization)
  - generate new samples which look similar to the training data (generative models)

Representation learning

- We can use unlabeled data to do representation learning.
- Representation learning: extract features that may be useful for future (downstream) tasks

$$\mathbf{x} \xrightarrow{f} \mathbf{z}$$

- Extracted features might work better than raw data in supervised learning tasks (especially with little labeled data):

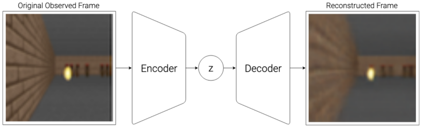
$$\mathbf{x} \xrightarrow{f} \mathbf{z} \rightarrow \mathbf{y}$$

- Problem: we do not know for which downstream tasks we need to prepare.
- Solution: we come up with auxiliary learning problems that would encourage learning useful representations:
    - data compression
    - prediction of the next observation
    - contrastive learning

Unsupervised representation learning with autoencoders

Dimensionality reduction (data compression)

- In many applications, the input data can be highly multi-dimensional (e.g., high-resolution images). Data often contain a lot of redundant information and it is often a good idea to reduce the data dimensionality.
  - Working with reduced dimensionalities can save computations.
  - Working with low-dimensional data might help improve the accuracy of the model, for example, we might reduce the risk of overfitting).
- Consider, for example, a reinforcement learning of playing Doom (Ha and Schmidhuber, 2018).
  - Learning from raw images (pixels) is likely to require a huge number of training episodes.
  - We can compress the data and then train a policy using compressed representations **z**.

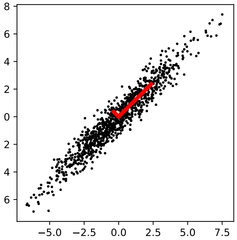


Principal component analysis (PCA)

- PCA is a classical technique of dimensionality reduction.
- It is traditionally formulated as finding data projection $y_1 = \mathbf{w}_1^\top \mathbf{x}$ with the maximum variance.
  - For centered data $\{\mathbf{x}^{(i)}\}$ with covariance matrix $\mathbf{C} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{x}^{(i)}\mathbf{x}^{(i)\top}$:

$$\mathbf{w}_1^* = \arg\max_{\mathbf{w}_1} \mathbf{w}_1^\top \mathbf{C}_\mathbf{x} \mathbf{w}_1, \quad \text{subject to } \|\mathbf{w}_1\| = 1$$

  - The solution is given by the first dominant eigenvector of the covariance matrix $\mathbf{C}_\mathbf{x}$.
  - The second principal component is found by maximizing the variance in the subspace orthogonal to the first eigenvector of $\mathbf{C}_\mathbf{x}$ (and so on).



Finding a principal subspace with a linear autoencoder

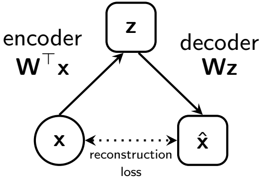
- PCA can be used to find *principal subpaces* of data.
- A principal subspace of size m is found as a linear projection of n-dimensional data

$$\underset{m\times 1}{\mathbf{z}} = \underset{m\times n}{\mathbf{W}}^{\top} \underset{n\times 1}{\mathbf{x}}$$

to minimize the mean-square error

$$\mathbf{W}_* = \arg\min_{\mathbf{W}} \frac{1}{N} \sum_{i=1}^{N} \left\| \mathbf{x}^{(i)} - \mathbf{W}\mathbf{z}^{(i)} \right\|^2, \qquad \text{s.t. } \mathbf{W}^\top \mathbf{W} = \mathbf{I}$$

between original data and its reconstruction from $\mathbf{z}$: $\hat{\mathbf{x}} = \mathbf{W}\mathbf{z}$.



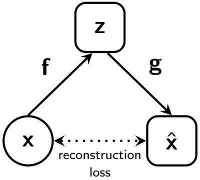
- Such a model is called *autoencoder*: data $\mathbf{x}$ are both model inputs and targets for model outputs.
- A principal subspace can be found with a *linear* autoencoder: both the encoder and decoder are linear functions.

PCA as a bootleneck autoencoder

encoder: $\mathbf{f}(\mathbf{x}) = \mathbf{W}_f \mathbf{x} + \mathbf{b}_f$

decoder: $\hat{\mathbf{x}} = \mathbf{g}(\mathbf{z}) = \mathbf{W}_g \mathbf{z} + \mathbf{b}_g$

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \left\| \mathbf{x}^{(i)} - \mathbf{f}(\mathbf{g}(\mathbf{z}^{(i)})) \right\|^2$$



- If we do not restrict $\mathbf{f}$ and $\mathbf{g}$, we can learn a trivial identity mapping:

$$\hat{\mathbf{x}} = \mathbf{g}(\mathbf{f}(\mathbf{x})) = (\mathbf{W}_g \mathbf{W}_f)\,\mathbf{x} + (\mathbf{W}_g \mathbf{b}_f + \mathbf{b}_g) = \mathbf{x}, \qquad \text{if } \mathbf{W}_g = \mathbf{W}_f^{-1} \text{ and } \mathbf{b}_g = -\mathbf{W}_g \mathbf{b}_f$$

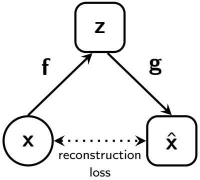
- If the dimensionality of $\mathbf{z}$ is smaller than the dimensionality of $\mathbf{x}$, autoencoding is useful: we compress the data.
    - $\mathbf{z}$ is often called a bottleneck.
    - Thus PCA can be implemented with a bottleneck autoencoder.
- How can we improve compression so that we get a smaller reconstruction error with a bottleneck layer of the same size?

PCA as a bootleneck autoencoder

encoder: $\mathbf{f}(\mathbf{x}) = \mathbf{W}_f\mathbf{x} + \mathbf{b}_f$

decoder: $\hat{\mathbf{x}} = \mathbf{g}(\mathbf{z}) = \mathbf{W}_g\mathbf{z} + \mathbf{b}_g$

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}\left\|\mathbf{x}^{(i)} - \mathbf{f}(\mathbf{g}(\mathbf{z}^{(i)}))\right\|^2$$



- If we do not restrict $\mathbf{f}$ and $\mathbf{g}$, we can learn a trivial identity mapping:

$$\hat{\mathbf{x}} = \mathbf{g}(\mathbf{f}(\mathbf{x})) = (\mathbf{W}_g\mathbf{W}_f)\,\mathbf{x} + (\mathbf{W}_g\mathbf{b}_f + \mathbf{b}_g) = \mathbf{x}, \qquad \text{if } \mathbf{W}_g = \mathbf{W}_f^{-1} \text{ and } \mathbf{b}_g = -\mathbf{W}_g\mathbf{b}_f$$

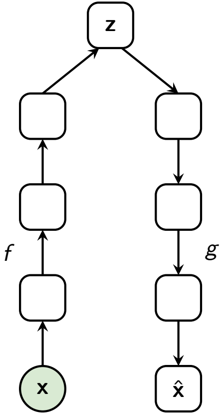
- If the dimensionality of $\mathbf{z}$ is smaller than the dimensionality of $\mathbf{x}$, autoencoding is useful: we compress the data.
  - $\mathbf{z}$ is often called a bottleneck.
  - Thus PCA can be implemented with a bottleneck autoencoder.
- How can we improve compression so that we get a smaller reconstruction error with a bottleneck layer of the same size? We can use nonlinear encoder $\mathbf{f}$ and decoder $\mathbf{g}$.

- Deep autoencoders (Bourlard and Kamp, 1988; Oja, 1991) is an extension of this idea to using nonlinear encoders and decoders. Both are implemented as deep neural networks:

$$\mathbf{z}^{(i)} = \mathbf{f}\left(\mathbf{x}^{(i)}, \boldsymbol{\theta}_f\right)$$

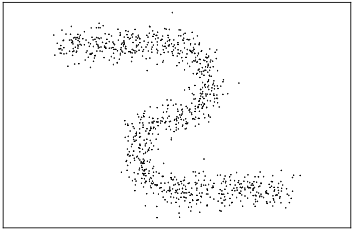
$$\boldsymbol{\theta}_f, \boldsymbol{\theta}_g = \arg\min_{\boldsymbol{\theta}_f, \boldsymbol{\theta}_g} \frac{1}{N} \sum_{i=1}^{N} \left\| \mathbf{x}^{(i)} - \mathbf{g}(\mathbf{z}^{(i)}, \boldsymbol{\theta}_g) \right\|^2$$

- To prevent learning a trivial (identity) function, **z** has fewer dimensions than **x** (a bottleneck layer). Such autoencoders are often called *bottleneck autoencoders*.



Deep autoencoders can learn complex data manifolds

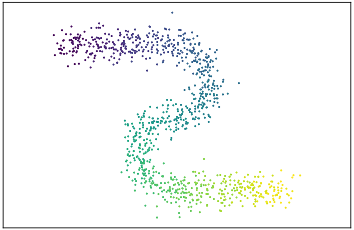
- In this hypothetical example, the data lie on one-dimensional manifold.
- Principal component analysis is not be able to learn the one-dimensional manifold because it is a linear model.



A one-dimensional data manifold in the two-dimensional space.

Deep autoencoders can learn complex data manifolds

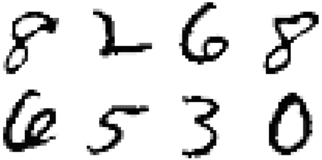
- In this hypothetical example, the data lie on one-dimensional manifold.
- Principal component analysis is not be able to learn the one-dimensional manifold because it is a linear model.
- With a nonlinear autoencoder, we can learn a curved data manifold.
- In our example, colors represents the values of the latent code z that may be found by an autoencoder.

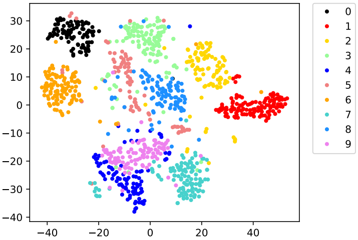


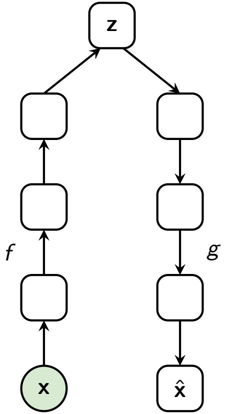
A one-dimensional data manifold in the two-dimensional space.

Deep bottleneck autoencoder: MNIST example

- In the home assignment, you will train a bottleneck autoencoder for the MNIST dataset.







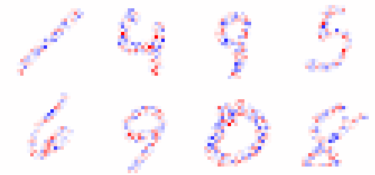
- Visualization of the **z**-space using t-SNE:

Denoising autoencoders

Vanilla autoencoders fail to extract more complex features

- Vanilla autoencoders cannot extract complex features, for example, features related to higher-order statistics (e.g., variance).

- Example: a variant of the MNIST dataset in which pixel intensities have high variance in the locations of the strokes.
- A vanilla autoencoder fails to extract features that allow classification of the images.



- The problem of the vanilla autoencoder is the mean-squared error loss which significantly constraints which the types of features that can be extracted.

Denoising autoencoder (Vincent et al., 2008)

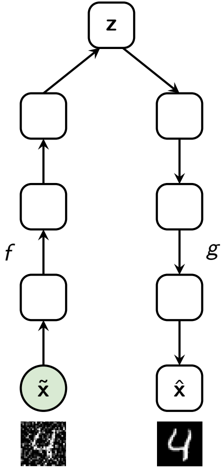
- Denoising autoencoders are conceptually similar to vanilla autoencoders. The difference is that the inputs of the autoencoder are always corrupted with noise (for example, Gaussian):

$$\tilde{\mathbf{x}}^{(i)} = \mathbf{x}^{(i)} + \boldsymbol{\epsilon}^{(i)} \qquad \boldsymbol{\epsilon}^{(i)} \sim \mathcal{N}(0, \sigma^2 \mathbf{I})$$

$$\mathbf{z}^{(i)} = \mathbf{f}\left(\tilde{\mathbf{x}}^{(i)}, \boldsymbol{\theta}_f\right)$$

$$\boldsymbol{\theta}_f, \boldsymbol{\theta}_g = \underset{\boldsymbol{\theta}_f, \boldsymbol{\theta}_g}{\arg\min} \frac{1}{N} \sum_{i=1}^{N} \left\| \mathbf{x}^{(i)} - \mathbf{g}(\mathbf{z}^{(i)}, \boldsymbol{\theta}_g) \right\|^2$$

- One can view adding noise to inputs as a way to regularize the autoencoder (regularization by noise injection) but there is more theory behind denoising autoencoders.



What does denoising autoencoder learn?

- For Gaussian corruption $\epsilon^{(i)} \sim \mathcal{N}(0, \sigma^2 \mathbf{I})$, the optimal denoising can be shown to be

$$\mathbf{d}(\tilde{\mathbf{x}}) = \tilde{\mathbf{x}} + \sigma^2 \nabla_{\tilde{\mathbf{x}}} \log p(\tilde{\mathbf{x}})$$

(see Alain and Bengio, 2014, Raphan and Simoncelli, 2011)

- $\mathbf{d}(\cdot)$ learns to point towards higher probability density.
- Thus, by learning the optimal denoising function $d(\mathbf{x})$, we implicitly model the data distribution $p(\mathbf{x})$.

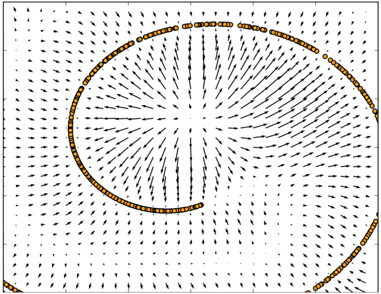
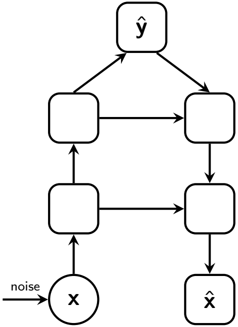


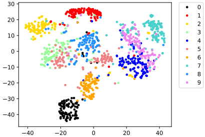
Image from (Alain and Bengio, 2014)

Denoising autoencoder: variance MNIST example

- Since the inputs of the autoencoder are *noisy* versions of the targets, the model cannot learn an identity mapping. Therefore:
    - A bottleneck layer is not needed in principle, but having a bottleneck layer often helps.
    - There can be skip connections between the encoder and the decoder (like in the U-net).

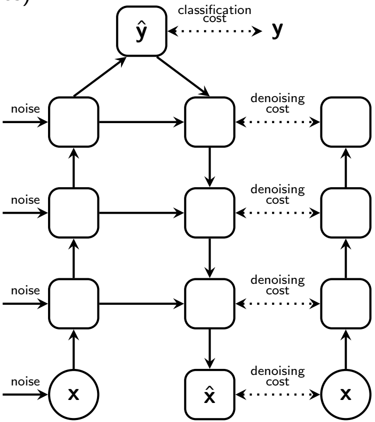


- For the variance-MNIST data, a denoising autoencoder can learn features that capture the shapes of the digits (see the visualization of the **z**-space using t-SNE).

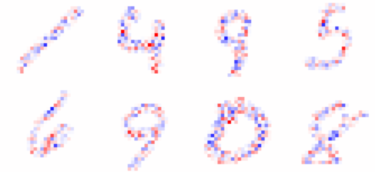


Ladder networks (Rasmus et al., 2015)

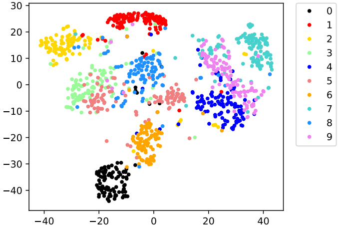
- Ladder networks used the principle of denoising to learn useful features in the semi-supervised settings (learning from both labeled and unlabeled examples).
- The architecture resembles a ladder (or a U-net): it is a denoising autoencoder with skip connections.
- The primary task is classification (bottleneck layer).
- The auxiliary task is denoising (output of the DAE).
- Intuition: In order to reconstruct the clean image from a noisy one, one has to learn features which are commonly present in images, which can help with the primary classification task.
- Ladder networks inspired modern models for deep semi-supervised learning.



Denoising autoencoder: variance MNIST example



- In the home assignment, we create a synthetic dataset (which we call variance MNIST).



- A denoising autoencoder can extract meaningful features. Visualization of the **z**-space using t-SNE:

Converting autoencoders into generative models with latent variables

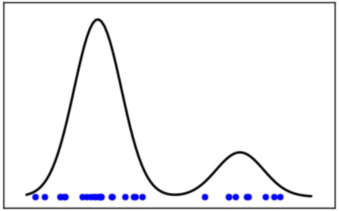
Generative models

- Generative models:
  - learn to represent the data distribution $p(\mathbf{x})$
  - can be used to generate new examples from $p(\mathbf{x})$.

- An example: a mixture-of-Gaussians model

$$p(x \mid \boldsymbol{\theta}) = w_1 \mathcal{N}(x \mid \mu_1, \sigma_1^2) + w_2 \mathcal{N}(x \mid \mu_2, \sigma_2^2)$$

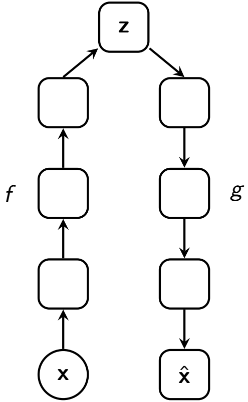
  Parameters $\boldsymbol{\theta} = \{w_1, \mu_1, \sigma_1, w_2, \mu_2, \sigma_2\}$ can be estimated by maximum likelihood.



- This model is an example of an explicit density model: $p(\mathbf{x} \mid \boldsymbol{\theta})$ has an explicit parametric form.

Converting autoencoders into generative models

- Vanilla autoencoders are not generative models.
  - We cannot generate new samples from $p(\mathbf{x})$.
  - We cannot compute the probability that a new sample $\mathbf{x}$ comes from the same distribution (e.g., for novelty detection).



Converting autoencoders into generative models

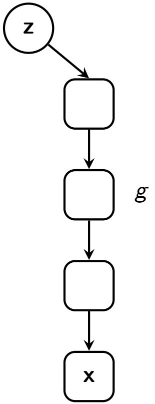
- Vanilla autoencoders are not generative models.
  - We cannot generate new samples from $p(\mathbf{x})$.
  - We cannot compute the probability that a new sample $\mathbf{x}$ comes from the same distribution (e.g., for novelty detection).
- We can build a generative model, for example, in this way:
  - Assume that variables $\mathbf{z}$ are normally distributed:

$$\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$$

  - Data samples $\mathbf{x}$ are nonlinear transformations of latent variables $\mathbf{z}$:

$$\mathbf{x} = g(\mathbf{z}, \boldsymbol{\theta}) + \boldsymbol{\varepsilon}$$

    with possibly noise added: $\boldsymbol{\varepsilon} \sim \mathcal{N}(0, \sigma^2 \mathbf{I})$
- Function $g(\mathbf{z}, \boldsymbol{\theta})$ can be modeled as a neural network.
- Now we can draw samples from the model.

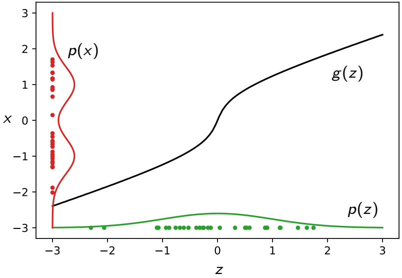


Latent variable model

- Our model contains latent (unobserved) variables $\mathbf{z}$:

$$\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$$
$$\mathbf{x} = g(\mathbf{z}, \boldsymbol{\theta}) + \boldsymbol{\varepsilon}$$
$$\boldsymbol{\varepsilon} \sim \mathcal{N}(0, \sigma^2 \mathbf{I})$$

- A simple example to illustrate the idea: We model one-dimensional data x as a Gaussian variable z transformed with nonlinearity g with some noise added.
- We need to learn the latent variable model from training data $\{\mathbf{x}_i\}$. We should tune parameters $\boldsymbol{\theta}, \sigma^2$ so that the training examples are likely to be produced by the model.



- We can tune parameters $\boldsymbol{\theta}, \sigma^2$ by maximizing the probability of the training data (maximum likelihood estimate):

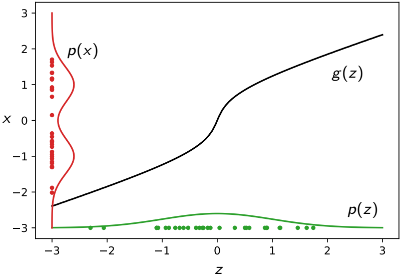
$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} \log p(\mathbf{x}_1, \ldots, \mathbf{x}_N \mid \boldsymbol{\theta})$$

$$\log p(\mathbf{x}_1, \ldots, \mathbf{x}_N \mid \boldsymbol{\theta}) = \sum_{i=1}^{N} \log p(\mathbf{x}_i \mid \boldsymbol{\theta}) = \sum_{i=1}^{N} \log \int p(\mathbf{x}_i \mid \mathbf{z}_i, \boldsymbol{\theta}) p(\mathbf{z}_i) d\mathbf{z}$$

- The probability density functions are defined by our model:

$$p(\mathbf{x}_i \mid \mathbf{z}_i, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{x}_i \mid g(\mathbf{z}_i, \boldsymbol{\theta}), \sigma^2 \mathbf{I})$$
$$p(\mathbf{z}_i) = \mathcal{N}(\mathbf{z}_i \mid 0, \mathbf{I})$$

- Direct optimization of $\log p(\mathbf{x}_1, \ldots, \mathbf{x}_N \mid \boldsymbol{\theta})$ is difficult because the above integrals are intractable.

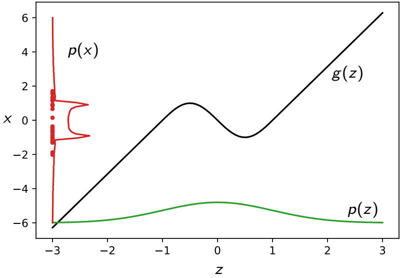


ML estimation with the EM algorithm

- The classical way to estimate parameters $\boldsymbol{\theta}$ of a latent variable model

$$p(\mathbf{x}_1, \dots, \mathbf{x}_N, \mathbf{z}_1, \dots, \mathbf{z}_N \mid \boldsymbol{\theta}) = \prod_{i=1}^{N} p(\mathbf{x}_i \mid \mathbf{z}_i, \boldsymbol{\theta}) p(\mathbf{z}_i)$$

  is the expectation-maximization (EM) algorithm.
- The EM-algorithm iterates between two steps: E-step and M-step.
  - E-step: Compute posterior probabilities $p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta})$ given current values of $\boldsymbol{\theta}$.
  - M-step: Update the values of $\boldsymbol{\theta}$ using computed $p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta})$.



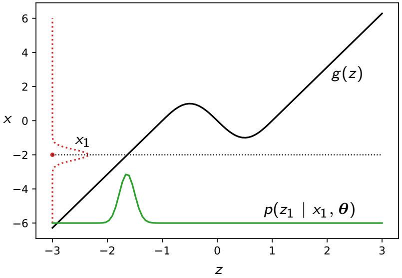
Consider our simple example. We initialize $\boldsymbol{\theta}$ with values that give us $\mathbf{g}$ of the form shown in the figure.

EM algorithm: E-step

$$\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$$
$$\mathbf{x} = g(\mathbf{z}, \boldsymbol{\theta}) + \varepsilon$$
$$\varepsilon \sim \mathcal{N}(0, \sigma^2 \mathbf{I})$$

- The E-step: Compute the posterior probabilities of the unobserved latent variables $\mathbf{z}_i$ given the data and the current estimates of the model parameters $\boldsymbol{\theta}$:

$$q(\mathbf{z}_1, ..., \mathbf{z}_N) = q(\mathbf{z}_1)...q(\mathbf{z}_N)$$
$$q(\mathbf{z}_i) = p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta})$$



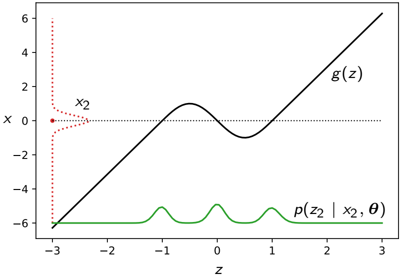
E-step: For each training data point, find the distribution over the latent variables that could have produced that data point according to the model.

EM algorithm: E-step

$$\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$$
$$\mathbf{x} = g(\mathbf{z}, \boldsymbol{\theta}) + \varepsilon$$
$$\varepsilon \sim \mathcal{N}(0, \sigma^2 \mathbf{I})$$

- The E-step: Compute the posterior probabilities of the unobserved latent variables $\mathbf{z}_i$ given the data and the current estimates of the model parameters $\boldsymbol{\theta}$:

$$q(\mathbf{z}_1, ..., \mathbf{z}_N) = q(\mathbf{z}_1)...q(\mathbf{z}_N)$$
$$q(\mathbf{z}_i) = p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta})$$

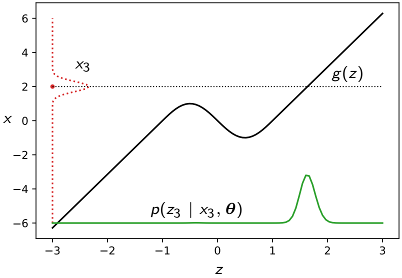


E-step: For each training data point, find the distribution over the latent variables that could have produced that data point according to the model.

$$\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$$
$$\mathbf{x} = g(\mathbf{z}, \boldsymbol{\theta}) + \varepsilon$$
$$\varepsilon \sim \mathcal{N}(0, \sigma^2 \mathbf{I})$$

- The E-step: Compute the posterior probabilities of the unobserved latent variables $\mathbf{z}_i$ given the data and the current estimates of the model parameters $\boldsymbol{\theta}$:

$$q(\mathbf{z}_1, ..., \mathbf{z}_N) = q(\mathbf{z}_1)...q(\mathbf{z}_N)$$
$$q(\mathbf{z}_i) = p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta})$$



E-step: For each training data point, find the distribution over the latent variables that could have produced that data point according to the model.

EM algorithm: M-step

- In the M-step, we use the computed distributions $q(\mathbf{z}_i)$ to form the following objective function:

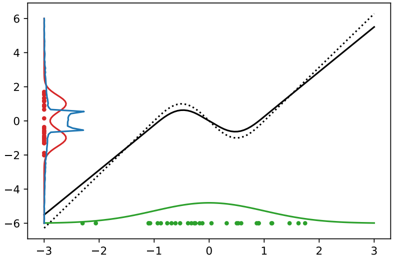
$$\begin{aligned}\mathcal{F}(\boldsymbol{\theta}) &= \langle \log p(\mathbf{x}_1, \ldots, \mathbf{x}_N, \mathbf{z}_1, \ldots, \mathbf{z}_N \mid \boldsymbol{\theta}) \rangle_{q(\mathbf{z}_1, \ldots, \mathbf{z}_N)} \\ &= \sum_{i=1}^{N} \langle \log p(\mathbf{x}_i, \mathbf{z}_i \mid \boldsymbol{\theta}) \rangle_{q(\mathbf{z}_i)} \\ &= \sum_{i=1}^{N} \int q(\mathbf{z}_i) \log p(\mathbf{x}_i, \mathbf{z}_i \mid \boldsymbol{\theta}) d\mathbf{z}_i\end{aligned}$$

  and maximize it wrt model parameters $\boldsymbol{\theta}$.

- We are guaranteed to improve the likelihood

$$\log p(\mathbf{x}_1, \ldots, \mathbf{x}_N \mid \boldsymbol{\theta})$$

  for each iteration of the EM-algorithm.



Iteration 1

EM algorithm: M-step

- In the M-step, we use the computed distributions $q(\mathbf{z}_i)$ to form the following objective function:

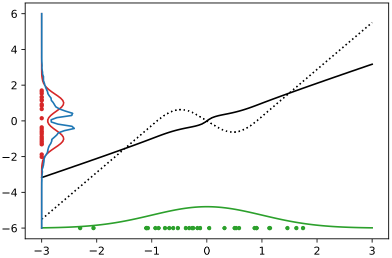
$$\begin{aligned}\mathcal{F}(\boldsymbol{\theta}) &= \langle \log p(\mathbf{x}_1, \ldots, \mathbf{x}_N, \mathbf{z}_1, \ldots, \mathbf{z}_N \mid \boldsymbol{\theta}) \rangle_{q(\mathbf{z}_1, \ldots, \mathbf{z}_N)} \\ &= \sum_{i=1}^{N} \langle \log p(\mathbf{x}_i, \mathbf{z}_i \mid \boldsymbol{\theta}) \rangle_{q(\mathbf{z}_i)} \\ &= \sum_{i=1}^{N} \int q(\mathbf{z}_i) \log p(\mathbf{x}_i, \mathbf{z}_i \mid \boldsymbol{\theta}) d\mathbf{z}_i\end{aligned}$$

  and maximize it wrt model parameters $\boldsymbol{\theta}$.

- We are guaranteed to improve the likelihood

$$\log p(\mathbf{x}_1, \ldots, \mathbf{x}_N \mid \boldsymbol{\theta})$$

  for each iteration of the EM-algorithm.



Iteration 2

EM algorithm: M-step

- In the M-step, we use the computed distributions $q(\mathbf{z}_i)$ to form the following objective function:

$$\mathcal{F}(\boldsymbol{\theta}) = \langle \log p(\mathbf{x}_1, \dots, \mathbf{x}_N, \mathbf{z}_1, \dots, \mathbf{z}_N \mid \boldsymbol{\theta}) \rangle_{q(\mathbf{z}_1, \dots, \mathbf{z}_N)}$$

$$= \sum_{i=1}^{N} \langle \log p(\mathbf{x}_i, \mathbf{z}_i \mid \boldsymbol{\theta}) \rangle_{q(\mathbf{z}_i)}$$

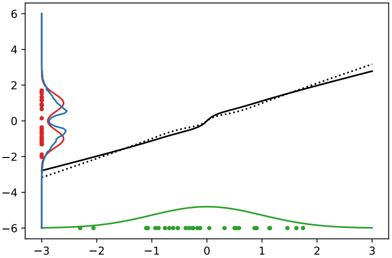
$$= \sum_{i=1}^{N} \int q(\mathbf{z}_i) \log p(\mathbf{x}_i, \mathbf{z}_i \mid \boldsymbol{\theta}) d\mathbf{z}_i$$

  and maximize it wrt model parameters $\boldsymbol{\theta}$.

- We are guaranteed to improve the likelihood

$$\log p(\mathbf{x}_1, \dots, \mathbf{x}_N \mid \boldsymbol{\theta})$$

  for each iteration of the EM-algorithm.



Iteration 3

EM algorithm: M-step

- In the M-step, we use the computed distributions $q(\mathbf{z}_i)$ to form the following objective function:

$$\mathcal{F}(\boldsymbol{\theta}) = \langle \log p(\mathbf{x}_1, \ldots, \mathbf{x}_N, \mathbf{z}_1, \ldots, \mathbf{z}_N \mid \boldsymbol{\theta}) \rangle_{q(\mathbf{z}_1, \ldots, \mathbf{z}_N)}$$

$$= \sum_{i=1}^{N} \langle \log p(\mathbf{x}_i, \mathbf{z}_i \mid \boldsymbol{\theta}) \rangle_{q(\mathbf{z}_i)}$$

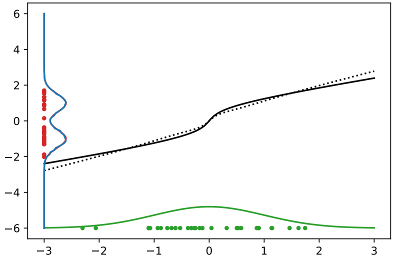
$$= \sum_{i=1}^{N} \int q(\mathbf{z}_i) \log p(\mathbf{x}_i, \mathbf{z}_i \mid \boldsymbol{\theta}) d\mathbf{z}_i$$

and maximize it wrt model parameters $\boldsymbol{\theta}$.

- We are guaranteed to improve the likelihood

$$\log p(\mathbf{x}_1, \ldots, \mathbf{x}_N \mid \boldsymbol{\theta})$$

for each iteration of the EM-algorithm.

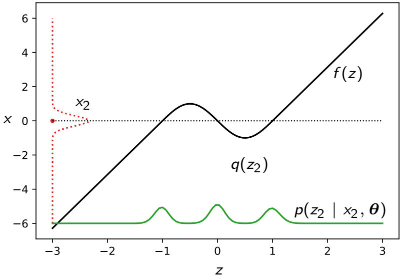


Iteration 4

Learning latent variable models with variational approximations

Intractability of the true conditional distributions

- There are a few problems with the direct application of the EM-algorithm in nonlinear latent variable models.
- One problem is the intractability of the true conditional distributions $q(\mathbf{z}_i) = p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta})$ that we need to compute on the E-step.
- The true distributions can be very complex (for example, a multi-modal distribution in our simple example).



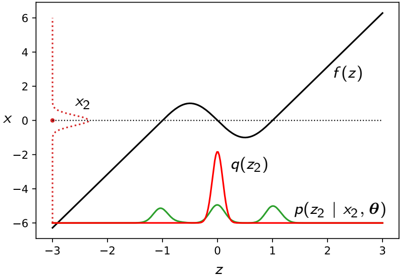
Example of multi-modal $p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta})$

E-step: Variational approximations

- Solution: Instead of using true conditional distributions, use their approximations $q(\mathbf{z}_i) \approx p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta})$.
- $q(\mathbf{z}_i)$ is selected to have a simple form, most often a Gaussian:

$$q(\mathbf{z}_i) = \mathcal{N}(\mu_{\mathbf{z}_i}, \sigma^2_{\mathbf{z}_i})$$

  Note: we have two parameters $\mu_{\mathbf{z}_i}$ and $\sigma^2_{\mathbf{z}_i}$ describing $q(\mathbf{z}_i)$ for *each training sample*.
- Parameters describing the posterior distributions of the latent variables $\boldsymbol{\theta}_q = \{\mu_{\mathbf{z}_i}, \sigma^2_{\mathbf{z}_i}\}_{i=1}^N$ are called variational parameters.



- A popular way to find the approximation is by minimizing the Kullback-Leibler divergence between $q(\mathbf{z}_i)$ and $p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta})$.

E-step: Variational approximations

- We can minimize the KL divergence between $q(\mathbf{z}_i)$ and $p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta})$ using the following trick:
  - Add to the objective function used in the M-step the entropies of the approximate distributions:

$$\mathcal{F}(\boldsymbol{\theta}, \boldsymbol{\theta}_q) = \sum_{i=1}^{N} \underbrace{\int q(\mathbf{z}_i) \log p(\mathbf{x}_i, \mathbf{z}_i \mid \boldsymbol{\theta}) d\mathbf{z}_i}_{\text{what we had in the M-step}} \underbrace{- \int q(\mathbf{z}_i) \log q(\mathbf{z}_i) d\mathbf{z}_i}_{\text{entropy}}$$

$$= \sum_{i=1}^{N} \int q(\mathbf{z}_i) \log \frac{p(\mathbf{x}_i, \mathbf{z}_i \mid \boldsymbol{\theta})}{q(\mathbf{z}_i)} dz_i = \sum_{i=1}^{N} \int q(\mathbf{z}_i) \log \frac{p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta}) p(\mathbf{x}_i \mid \boldsymbol{\theta})}{q(\mathbf{z}_i)} dz_i$$

$$= \sum_{i=1}^{N} -D_{\mathrm{KL}}(q(\mathbf{z}_i) \parallel p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta})) + \log p(\mathbf{x}_i \mid \boldsymbol{\theta})$$

- One can see that maximizing $\mathcal{F}(\boldsymbol{\theta}, \boldsymbol{\theta}_q)$ wrt variational parameters $\boldsymbol{\theta}_q$ is equivalent to minimizing the KL divergence between $q(\mathbf{z}_i)$ and $p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta})$.

EM algorithm with variational approximations

- We can now maximize a single function $\mathcal{F}$ wrt $\boldsymbol{\theta}$ and $\boldsymbol{\theta}_q$ jointly without the need to alternate between the E- and M-steps:

$$\begin{aligned}\mathcal{F}(\boldsymbol{\theta}, \boldsymbol{\theta}_q) &= \sum_{i=1}^{N} \int q(\mathbf{z}_i) \log p(\mathbf{x}_i, \mathbf{z}_i \mid \boldsymbol{\theta}) d\mathbf{z}_i - \int q(\mathbf{z}_i) \log q(\mathbf{z}_i) d\mathbf{z}_i \\ &= \sum_{i=1}^{N} -D_{\mathrm{KL}}(q(\mathbf{z}_i) \parallel p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta})) + \log p(\mathbf{x}_i \mid \boldsymbol{\theta})\end{aligned}$$

  - Maximizing $\mathcal{F}(\boldsymbol{\theta}, \boldsymbol{\theta}_q)$ wrt $\boldsymbol{\theta}$ is equivalent to the M-step.
  - Maximizing $\mathcal{F}(\boldsymbol{\theta}, \boldsymbol{\theta}_q)$ wrt $\boldsymbol{\theta}_q$ is done in the E-step with variational approxiations.

- We can solve this optimization problem using any optimizer of our choice.

Evidence lower bound (ELBO)

- The objective function

$$\mathcal{F}(\boldsymbol{\theta}, \boldsymbol{\theta}_q) = \sum_{i=1}^{N} -D_{\mathrm{KL}}(q(\mathbf{z}_i) \parallel p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta})) + \log p(\mathbf{x}_i \mid \boldsymbol{\theta})$$

  is the *lower bound* of the true likelihood that we want to optimize. Since $D_{\mathrm{KL}}(q \parallel p) \geq 0$:

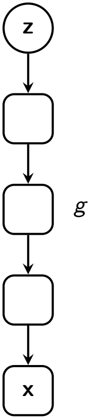
$$\mathcal{F}\left(\boldsymbol{\theta}, \boldsymbol{\theta}_q\right) \leq \sum_{i=1}^{N} \log p(\mathbf{x}_i \mid \boldsymbol{\theta}) = \log p(\mathbf{x}_1, \ldots, \mathbf{x}_N \mid \boldsymbol{\theta})$$

- This function is often called *evidence lower bound* or ELBO.
- The closer our approximation $q(\mathbf{z}_i)$ to the true posterior $p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta})$, the tighter the bound.

ELBO for our deep generative model

- ELBO can be re-written in the following form:

$$\mathcal{F}(\boldsymbol{\theta}, \boldsymbol{\theta}_q) = \sum_{i=1}^{N} \int q(\mathbf{z}_i) \log p(\mathbf{x}_i \mid \mathbf{z}_i, \boldsymbol{\theta}) dz_i - \int q(\mathbf{z}_i) \log \frac{q(\mathbf{z}_i)}{p(\mathbf{z}_i)} dz_i \qquad (1)$$



- Recall our deep generative model: $p(\mathbf{x}_i \mid \mathbf{z}_i, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{x}_i \mid g(\mathbf{z}_i, \boldsymbol{\theta}), \sigma^2 \mathbf{I})$,
- The first term in equation (1) can be written as

$$\left\langle -\frac{D}{2} \log 2\pi\sigma^2 - \frac{1}{2\sigma^2} \sum_{d=1}^{D} (\mathbf{x}_i(d) - g_d(\mathbf{z}_i, \boldsymbol{\theta}))^2 \right\rangle_{q(\mathbf{z}_i)}$$

  where D is the number of dimensions in $\mathbf{x}$, $\mathbf{x}_i(d)$ is the d-th element of $\mathbf{x}_i$ and g_d is the d-th element of the output of function g.
- The first term contains the mean-squared error between data sample $\mathbf{x}_i$ and its reconstruction $g_d(\mathbf{z}_i, \boldsymbol{\theta})$ from the latent code $\mathbf{z}_i$.

ELBO for our deep generative model

$$\mathcal{F}(\boldsymbol{\theta}, \boldsymbol{\theta}_q) = \sum_{i=1}^{N} \underbrace{\int q(\mathbf{z}_i) \log p(\mathbf{x}_i \mid \mathbf{z}_i, \boldsymbol{\theta}) d\mathbf{z}_i}_{\text{minus mean-square reconstruction error}} \underbrace{- \int q(\mathbf{z}_i) \log \frac{q(\mathbf{z}_i)}{p(\mathbf{z}_i)} d\mathbf{z}_i}_{\text{regularization term}}$$

- The second term is minus KL-divergence between $q(\mathbf{z}_i)$ and the prior $p(\mathbf{z}_i) = \mathcal{N}(0, \mathbf{I})$:

$$- \int q(\mathbf{z}_i) \log \frac{q(\mathbf{z}_i)}{p(\mathbf{z}_i)} dz_i = -D_{\mathrm{KL}}(q(\mathbf{z}_i) \parallel p(\mathbf{z}_i))$$

- It is a kind of a regularization term: We want the conditional distributions $q(\mathbf{z}_i)$ to be close to the prior $p(\mathbf{z}_i) = \mathcal{N}(0, \mathbf{I})$.

Variational autoencoders

First algorithms for learning this type of models

- The first algorithm for learning latent variable model

$$\mathbf{z} \sim \mathcal{N}(0, \mathbf{I}) \qquad \mathbf{x} = g(\mathbf{z}, \boldsymbol{\theta}) + \boldsymbol{\varepsilon} \qquad \boldsymbol{\varepsilon} \sim \mathcal{N}(0, \sigma^2 \mathbf{I})$$

  using variational approximations was proposed in this university (Lappalainen and Honkela, 2001).

- The objective function was ELBO:

$$\mathcal{F}(\boldsymbol{\theta}, \boldsymbol{\theta}_q) = \sum_{i=1}^{N} \underbrace{\int q(\mathbf{z}_i) \log p(\mathbf{x}_i \mid \mathbf{z}_i, \boldsymbol{\theta}) d\mathbf{z}_i}_{\text{needs approximations}} - \underbrace{\int q(\mathbf{z}_i) \log \frac{q(\mathbf{z}_i)}{p(\mathbf{z}_i)} d\mathbf{z}_i}_{\text{can be computed analytically}}$$

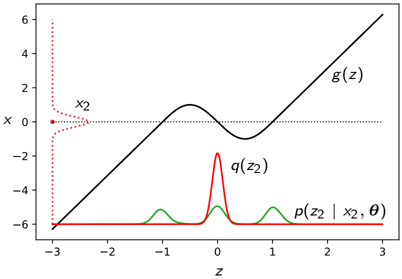
- The posterior approximations were Gaussian $q(\mathbf{z}_i) = \mathcal{N}(\mu_{\mathbf{z}_i}, \sigma^2_{\mathbf{z}_i})$. The number of variational parameters $\boldsymbol{\theta}_q = \{\mu_{\mathbf{z}_i}, \sigma^2_{\mathbf{z}_i}\}_{i=1}^{N}$ was proportional to the number of training samples.

Adding encoder

- We want to get rid of the large number of variational parameters $\boldsymbol{\theta}_q = \{\mu_{\mathbf{z}_i}, \sigma^2_{\mathbf{z}_i}\}_{i=1}^N$.
- For fixed model parameters $\boldsymbol{\theta}$, the optimal $q(\mathbf{z})$ only depends on $\mathbf{x}$. The inference procedure does the following mapping:

$$\mathbf{x} \rightarrow q(\mathbf{z})$$

  For Gaussian approximation: $\mathbf{x} \rightarrow \mu_{\mathbf{z}}, \sigma^2_{\mathbf{z}}$.



- In variational autoencoders (VAE) (Kingma and Welling, 2014), mapping $\mathbf{x} \rightarrow q(\mathbf{z})$ is done using a neural network (encoder).
- The encoder performs so called *amortized inference*: When doing inference for a particular sample $\mathbf{x}_i$, we leverage the knowledge of the inference results for other samples. If two samples $\mathbf{x}_i$ and $\mathbf{x}_j$ are close to each other, the corresponding $q(\mathbf{z}_i)$, $q(\mathbf{z}_j)$ should be close as well.

Variational autoencoder (VAE): Encoder and decoder

- Our generative model is defined by the decoder.

$$\mathbf{z} \sim \mathcal{N}(0, \mathbf{I}) \qquad \mathbf{x} = g(\mathbf{z}, \boldsymbol{\theta}) + \varepsilon \qquad \varepsilon \sim \mathcal{N}(0, \sigma^2 \mathbf{I})$$

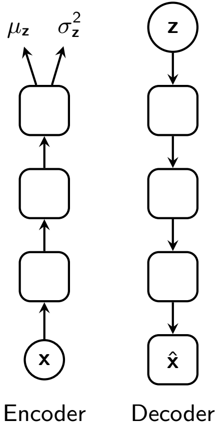
- Encoder is a neural network that is trained to perform variational inference:

$$\mathbf{x} \to q(\mathbf{z})$$

- For Gaussian approximation $q(\mathbf{z})$, the neural network needs to produce:

$$\mathbf{x} \to \mu_{\mathbf{z}}, \sigma^2_{\mathbf{z}}$$

- In practice, this is done using one neural network with two heads.
- The encoder is similar to the encoder in a bottleneck autoencoder but produces the mean and variance of the code $\mathbf{z}$.
- The encoder and decoder are two components of the variational autoencoder.



Monte Carlo estimates of the objective function

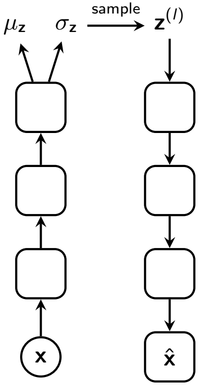
- The first term of the objective function cannot computed analytically

$$\mathcal{F}(\boldsymbol{\theta}, \boldsymbol{\theta}_q) = \sum_{i=1}^{N} \underbrace{\int q(\mathbf{z}_i) \log p(\mathbf{x}_i \mid \mathbf{z}_i, \boldsymbol{\theta}) d\mathbf{z}_i}_{\text{needs approximations}} - \underbrace{\int q(\mathbf{z}_i) \log \frac{q(\mathbf{z}_i)}{p(\mathbf{z}_i)} d\mathbf{z}_i}_{\text{can be computed analytically}}$$

- Kingma and Welling (2014) proposed to use Monte Carlo estimates:

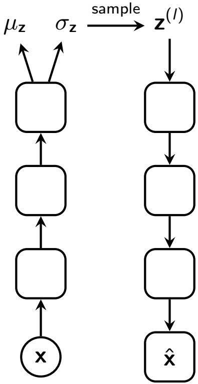
$$\int q(\mathbf{z}_i) \log \mathcal{N}(\mathbf{x}_i \mid g(\mathbf{z}_i, \boldsymbol{\theta}), \sigma^2 \mathbf{I}) d\mathbf{z}_i \approx \frac{1}{L} \sum_{l=1}^{L} \log \mathcal{N}(\mathbf{x}_i \mid g(\mathbf{z}_i^{(l)}, \boldsymbol{\theta}), \sigma^2 \mathbf{I})$$

where $\mathbf{z}_i^{(l)}$ are drawn from $q(\mathbf{z}_i)$. Using $L = 1$ works well in practice.



Computation of the objective function

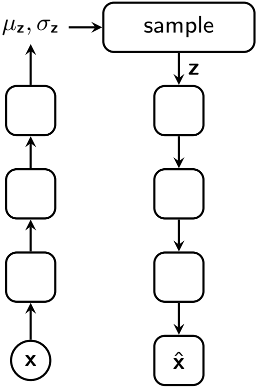
$$\mathcal{F}(\boldsymbol{\theta}, \boldsymbol{\theta}_q) = \sum_{i=1}^{N} \underbrace{\log \mathcal{N}(\mathbf{x}_i \mid g(\mathbf{z}_i^{(l)}, \boldsymbol{\theta}), \sigma^2 \mathbf{I})}_{\text{Monte Carlo estimate}} - \underbrace{\int q(\mathbf{z}_i) \log \frac{q(\mathbf{z}_i)}{p(\mathbf{z}_i)} d\mathbf{z}_i}_{\text{can be computed analytically}}$$



- For each training example $\mathbf{x}_i$:
  - compute means $\mu_{\mathbf{z}_i}$ and $\sigma_{\mathbf{z}_i}$ using the encoder
  - compute the second term analytically
  - draw $L = 1$ samples $\mathbf{z}_i^{(l)}$ from $q(\mathbf{z}_i) = \mathcal{N}(\mu_{\mathbf{z}_i}, \sigma^2_{\mathbf{z}_i})$
  - propagate $\mathbf{z}_i^{(l)}$ through the decoder and compute the first term
- Problem: We can use backpropagation to compute the derivatives wrt the parameters of the decoder but we need an extra trick to propagate derivatives through the encoder.

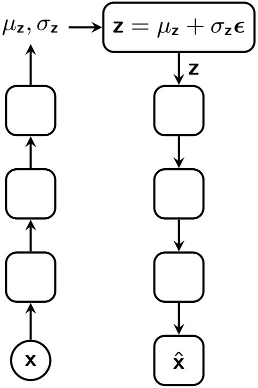
Reparameterization trick

- We need a computational block that would
  - take as inputs $\mu_\mathbf{z}$ and $\sigma_\mathbf{z}$
  - produce a sample from distribution $\mathbf{z} \sim \mathcal{N}(\mu_\mathbf{z}, \sigma_\mathbf{z})$
  - would be differentiable wrt $\mu_\mathbf{z}$ and $\sigma_\mathbf{z}$



Reparameterization trick

- We need a computational block that would
  - take as inputs $\mu_\mathbf{z}$ and $\sigma_\mathbf{z}$
  - produce a sample from distribution $\mathbf{z} \sim \mathcal{N}(\mu_\mathbf{z}, \sigma_\mathbf{z})$
  - would be differentiable wrt $\mu_\mathbf{z}$ and $\sigma_\mathbf{z}$
- We can obtain this with the reparameterization trick:
  - Sample $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$
  - Compute $\mathbf{z} = \mu_\mathbf{z} + \sigma_\mathbf{z}\boldsymbol{\epsilon}$
- Now we can also backpropagate through the sampling block and then further through the encoder.

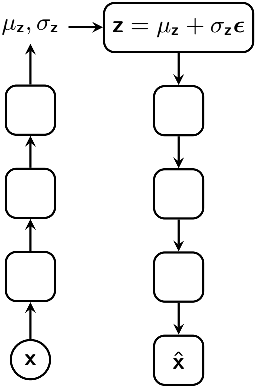


VAE training algorithm

- VAE training algorithm:
  - Take a mini-batch $\{\mathbf{x}_i\}$ of training samples.
  - Use the encoder to compute means $\mu_{\mathbf{z}_i}$ and standard deviations $\sigma_{\mathbf{z}_i}$ for each sample $\mathbf{x}_i$ in the mini-batch.
  - Draw $\epsilon_i \sim \mathcal{N}(0, \mathbf{I})$ and compute samples $\mathbf{z}_i = \mu_{\mathbf{z}_i} + \sigma_{\mathbf{z}_i}\epsilon_i$
  - Propagate samples $\mathbf{z}_i$ through the decoder to compute reconstructions $\hat{\mathbf{x}}_i$.
  - Compute the loss which is the negative of

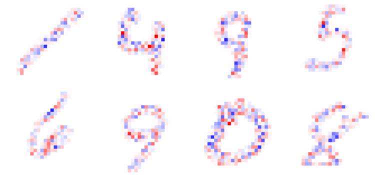
$$\mathcal{F}(\boldsymbol{\theta}, \boldsymbol{\theta}_q) = \frac{1}{n}\sum_{i=1}^{n} \underbrace{\log \mathcal{N}(\mathbf{x}_i \mid g(\mathbf{z}_i^{(l)}, \boldsymbol{\theta}), \sigma^2 \mathbf{I})}_{\text{Monte Carlo estimate}} \underbrace{- \int q(\mathbf{z}_i) \log \frac{q(\mathbf{z}_i)}{p(\mathbf{z}_i)} dz_i}_{\text{can be computed analytically}}$$

  - Perform backpropagation and update the parameters of the encoder and the decoder.



Variational autoencoder: variance MNIST example

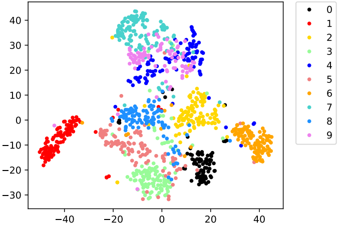
- In the home assignment, we train a variational autoencoder on a synthetic (variance MNIST) dataset.



- In order to extract meaningful features for this dataset, we need to use a generator (decoder) that models the variances of pixel intensities:

$$\mathbf{z} \sim \mathcal{N}(0, \mathbf{I}) \qquad \mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}(\mathbf{z}), \mathrm{diag}(\boldsymbol{\sigma}(\mathbf{z})))$$

$$\boldsymbol{\mu}(\mathbf{z}) = g_\mu(\mathbf{z}, \boldsymbol{\theta}) \qquad \boldsymbol{\sigma}(\mathbf{z}) = \exp(g_\sigma(\mathbf{z}, \boldsymbol{\theta}))$$



Why should I use a VAE?

- VAE is more complex than a simple bottleneck autoencoder. Do we need these complications?
- As we will see in the home assignment, VAEs are more powerful. In some problems when vanilla autoencoders fail, VAEs can develop useful representations.
- The problem of the vanilla autoencoder is the mean-squared error loss, which makes too simplistic assumptions about the data distribution.
- One advantage of VAE is in greater flexibility in defining the generative model.
- Note that denoising autoencoders are more powerful than standard autoencoders even though they also use the mean-squared error loss.

VAEs as generative models

- The main benefit of VAEs is that we can encode data into a lower-dimensional representation.
- But VAEs are generative models and we can draw samples using VAEs.
- Traditionally, the quality of the VAE-generated samples have not been very impressive: samples and reconstructions usually look blurry.

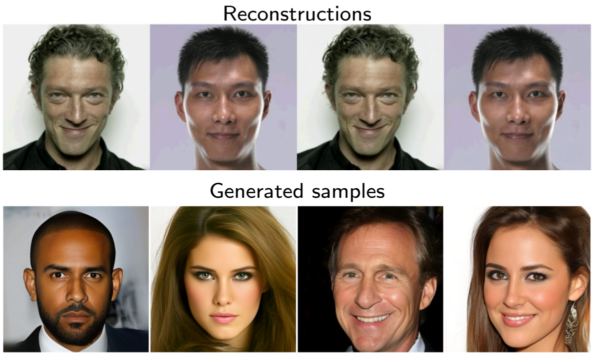
Reconstructions

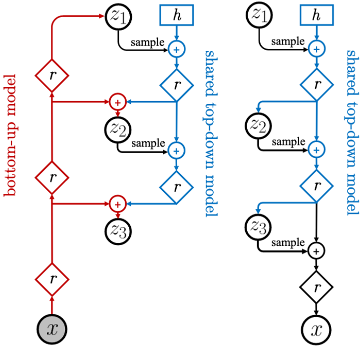
Generated samples

Images from (Tolstikhin et al., 2017)

Nouveau VAE (NVAE; Vahdat and Kautz, 2020)

- Vahdat and Kautz (2020) presented a VAE model that is able to generate high-quality images.
- It is a hierarchical latent variable model, that is there are multiple levels of latent variables.





Home assignment

Assignment 07_ae

- In the home assignment, you will have to implement three types of autoencoders:
    1. Vanilla bottleneck autoencoder
    2. Denoising autoencoder
    3. Variational autoencoder

Recommended reading

- Chapter 14 of the Deep Learning book
- Papers cited in the lecture slides