CSC413 Neural Networks and Deep Learning

Lecture 10: Generative Models

Mar 19/21, 2024

Table of Contents

1. Generative Models
2. Transposed Convolution
3. Autoencoder
4. An autoencoder for MNIST
5. Variational Autoencoders

Announcement from Accessibility Services

Accessibility Services is seeking volunteer note takers for students in this class who are registered in Accessibility Services. By volunteering to take notes for students with disabilities, you are making a positive contribution to their academic success. By volunteering as a note-taker, you will benefit as well - It is an excellent way to improve your own note-taking skills and to maintain consistent class attendance. At the end of term, we would be happy to provide a Certificate of Appreciation for your hard work. To request a Certificate of Appreciation please fill out the form at this link: Certificate of Appreciation or email us at at as.notetaking@utoronto.ca. You may also qualify for a Co-Curricular Record by registering your volunteer work on Folio before the end of June. We also have a draw for qualifying volunteers throughout the academic year.

Announcement from Accessibility Services

Steps to Register as a Volunteer :

1. Register Online as a Volunteer Note-Taker at: https://clockwork.studentlife.utoronto.ca/custom/misc/home.aspx
2. For a step-to-step guide please follow this link to the Volunteer Notetaking Portal Guide
3. Click on Volunteer Notetakers, and sign in using your UTORid
4. Select the course(s) you wish to take notes for. Please note: you do NOT need to upload sample notes or be selected as a volunteer to begin uploading your notes.
5. Start uploading notes.

Announcement from Accessibility Services

Email us at as.notetaking@utoronto.ca if you have questions or require any assistance with uploading notes. If you are no longer able to upload notes for a course, please also let us know immediately .

For more information about the Accessibility Services Peer Notetaking program, please visit Student Life Volunteer Note Taking.

Thank you for your support and for making notes more accessible for our students.

AS Note-taking Team

Section 1

Generative Models

Generating Images

How to generate new data of certain types

- generate text that looks like our training data
- generate **images** that look like our training data

Models:

- Autoencoder (AE)
- Variational Autoencoder (VAE)
- Generative Adversarial Networks (GANs)
- Generative RNNs
- Diffusion Models
  - lilianweng.github.io/posts/2021-07-11-diffusion-models/

We'll talk about autoencoders and variational autoencoders today.

Autoencoders

There are two ways of thinking of an image autoencoder:

- a model that finds a **low-dimensional representation** of images
- a model that will eventually help us generate new images

Both are considered **unsupervised learning** tasks, since no labels are involved. However, we do have a dataset of unlabelled images.

We talk about *images* as the data type, but autoencoders can be applied to other data types too.

Image Autoencoder

Idea: In order to learn to generate images, we'll learn to **reconstruct** images from a low-dimensional representation.

An image autoencoder has two components:

1. An **encoder** neural network that takes the image as input, and produces a low-dimensional embedding.
2. A **decoder** neural network that takes the low-dimensional embedding as input, and reconstructs the image.

A good, low-dimensional representation should allow us to reconstruct everything about the image.

The components of an autoencoder

Encoder:

- Input = image
- Output = low-dimensional embedding

Decoder:

- Input = low-dimensional embedding
- Output = image

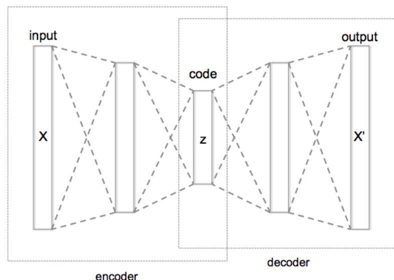


Image Encoder Architecture

What would the architecture of the encoder look like?

- We can use a FC NN (MLP)
  - But MLPs are not the best architecture to deal with images
- We can also use a convolutional neural network

We can use downsampling to reduce the dimensionality of the data

- Q: How can we downsample the dimensionality of an image?

Image Decoder Architecture

What would the architecture of the decoder look like?

We need to be able to **increase** the image resolution.

We haven't learned how to do this yet. Let us introduce a new NN layer: **Transposed Convolution**.

Section 2

Transposed Convolution

Transposed Convolution

Let us have a detour and talk about another type of layer: Transposed Convolution or Deconvolution.

It is used to increase the resolution of a feature map.

This is useful for:

- image generation problems (as we use in the decoder part of the autoencoders)
- pixel-wise prediction problems

Pixel-wise prediction

A prediction problem where we label the content of each pixel is known as a **pixel-wise prediction problem**

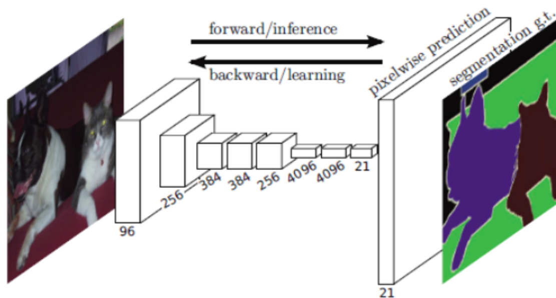


Figure 1: http://deeplearning.net/tutorial/fcn_2D_segm.html

Q: How do we generate pixel-wise predictions?

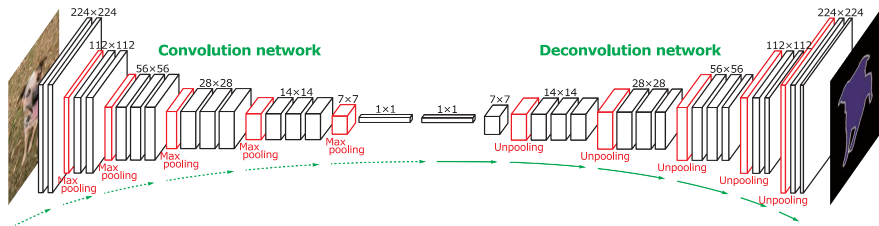
What we need:

We need to be able to **up-sample** features, i.e. to obtain high-resolution features from low-resolution features

- Opposite of max-pooling
- Opposite of a strided convolution

We need an **inverse** convolution, i.e., **transposed convolution** (or commonly used, but incorrect term: **deconvolution**).

Architectures with Transposed Convolution



Transposed Convolution Layer

With stride $= 1$

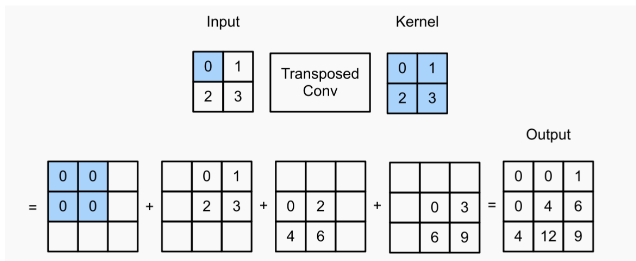


Figure 2: Image credit:
https://d2l.ai/chapter_computer-vision/transposed-conv.html

Transposed Convolution Layer

With stride = 2

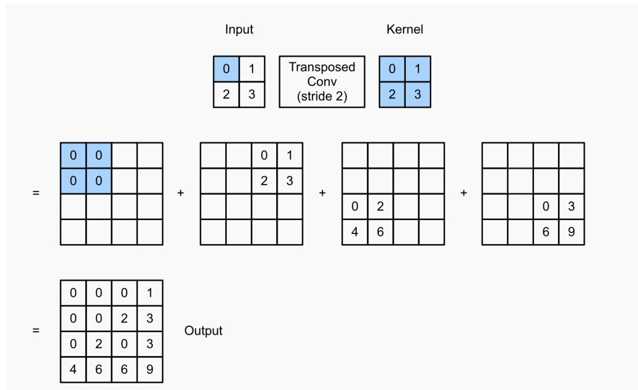


Figure 3: Image credit:
https://d2l.ai/chapter_computer-vision/transposed-conv.html

More at `https://github.com/vdumoulin/conv_arithmetic`

Section 3

Autoencoder

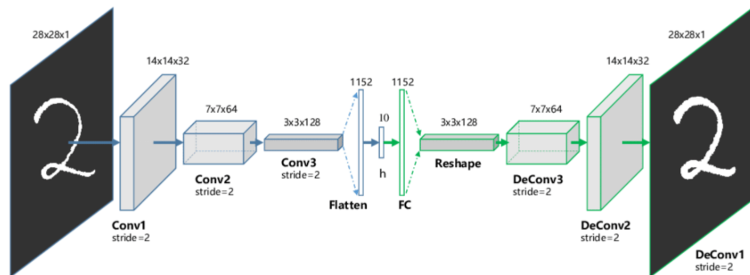
Let's get back to the autoencoder

Recall that we want a model that **generates images** that looks like our training data

Idea:

- In order to learn to generate images, we'll learn to **reconstruct** images from a low-dimensional representation.
- A good, low-dimensional representation should allow us to reconstruct "important" aspects of the image.

The components of an autoencoder



Encoder:

- Input = image
- Output = low-dimensional embedding

Decoder:

- Input = low-dimensional embedding
- Output = image

Why autoencoders?

- Dimension reduction:
  - find a low dimensional representation of the image
- Image Generation:
  - generate new images not in the training set

Autoencoders are not used for **supervised learning**. The task is *not* to predict something about the image!

Autoencoders are considered a **generative model**.

How to train autoencoders?

- Loss function: How close were the reconstructed image from the original? Here are some ideas...
  - **Mean Square Error (MSE)**: look at the mean square error across all pixels.
  - **Mean Square-Gradient Error (MSGE)**: take the average of the differences of squared gradients (computed with something like the Sobel filter) across all pixels.
  - **Corner Detection**: use computer vision to identify corners. Then across the image (or patches in a partition of the image), compare corner counts, corresponding positions, and/or nearest distances.
- Optimizer:
  - Just like before: use SGD or other optimizers.

Section 4

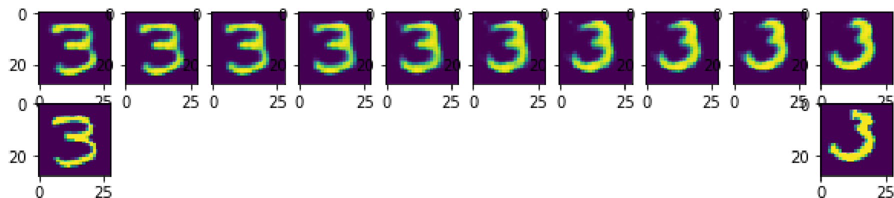
An autoencoder for MNIST

Structure in the Embedding Space

The dimensionality reduction means that there will be structure in the embedding space.

If the dimensionality of the embedding space is not too large, similar images should map to similar locations.

Interpolating in the Embedding Space



Generating New Images

Q: Can we pick a random point in the embedding space, and decode it to get an image of a digit?

A: Unfortunately not necessarily. Can we figure out why not?

Autoencoder Overfitting

Overfitting can occur if the size of the embedding space is too large.

If the dimensionality of the embedding space is small, then the neural network needs to map similar images to similar locations.

If the dimensionality of the embedding space is **too large**, then the neural network can simply memorize the images!

Blurry reconstructions

Q: Why do autoencoders produce blurry images?

Hint: it has to do with the use of the MSELoss.

Read more: `ieeexplore.ieee.org/document/8461664`

Section 5

Variational Autoencoders

Generative Model

In Intro to ML course (CSC311), we learned about **generative models** that describe the distribution that the data comes from.

- Describe the distribution $\mathbf{x} \sim p(\mathbf{x})$, where $\mathbf{x}$ is a single data point

For example, in the Naive Bayes model for data $\mathbf{x}$ (e.g. bag-of-word encoding of an email, which could be spam or not spam) with $\mathbf{x} \sim p(\mathbf{x})$, we assumed that $p(\mathbf{x}) = \sum_c p(\mathbf{x}|c)p(c)$, where c is either spam or not spam. We made further assumptions about $p(\mathbf{x}|c)$, e.g. that each x_i is an independent Bernoulli.

Mathematical Notation and Assumptions

Data $x_i \in \mathbb{R}^d$ are:

- independent, identically distributed (i.i.d.)
- generated from the following joint distribution (with the true parameter θ^* unknown)

$$p_{\theta^*}(\mathbf{z}, \mathbf{x}) = p_{\theta^*}(\mathbf{z}) p_{\theta^*}(\mathbf{x}|\mathbf{z})$$

Where $\mathbf{z}$ is a low-dimensional vector (latent embedding)

- Example $\mathbf{x}$ could be an MNIST digit
- Think of $\mathbf{z}$ as encoding digit features like digit shape, tilt, line thickness, font style, etc...
- To generate an image, we first sample from the prior distribution $p_{\theta^*}(\mathbf{z})$ to decide on these digit features, and use $p_{\theta^*}(\mathbf{x}|\mathbf{z})$ to generate an image given those features

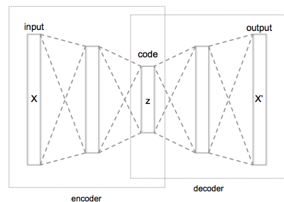
Intractability

Our data set is large, and so the computation of the following is *intractable*:

- evidence $p_{\theta^*}(\mathbf{x})$
- posterior distributions $p_{\theta^*}(\mathbf{z}|\mathbf{x})$

In other words, exactly computing the distribution of $p(\mathbf{x})$ and $p(\mathbf{z}|\mathbf{x})$ using our dataset has high runtime complexity.

The decoder and encoder



With this assumption, we can think of the autoencoder as doing the following:

Decoder: A point approximation of the true distribution $p_{\theta^*}(\mathbf{x}|\mathbf{z})$

Encoder: Making a **point prediction** for the value of the latent vector z that generated the image x

Alternative:

- what if, instead, we try to infer the **distribution** $p_{\theta^*}(\mathbf{z}|\mathbf{x})$?

VAE Setup so far

Decoder: An approximation of the true distribution $p_{\theta^*}(\mathbf{x}|\mathbf{z})$

Encoder: An approximation of the true distribution $p_{\theta^*}(\mathbf{z}|\mathbf{x})$

Computing the encoding distribution $p_{\theta^*}(\mathbf{z}|\mathbf{x})$

Unfortunately, the true distribution $p_{\theta^*}(\mathbf{z}|\mathbf{x})$ is complex (e.g. can be multi-modal).

But can we approximate this distribution with a **simpler distribution**?

Let's restrict our estimate $q_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$ to be a multivariate Gaussian distribution with $\phi = (\boldsymbol{\mu}, \boldsymbol{\Sigma})$

- It suffices to estimate the mean $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$ of $q_\phi(\mathbf{z}|\mathbf{x})$
- Let's make it simpler and assume that the covariance matrix is diagonal, $\boldsymbol{\Sigma} = \sigma^2 \mathbf{I}_{d \times d}$

(Note: we don't have to make this assumption, but it will make computation easier later on)

VAE Setup so far

Decoder: An approximation of the true distribution $p_{\theta^*}(\mathbf{x}|\mathbf{z})$

Encoder: Predicts the mean and standard deviations of a distribution $q_\phi(\mathbf{z}|\mathbf{x})$, so that the distribution is close to the true distribution $p_{\theta^*}(\mathbf{z}|\mathbf{x})$

We want our estimate distribution to be close to the true distribution. How do we measure the difference between distributions?

(Discrete) Entropy

$$H[X] = \sum_x p(X = x) \log \frac{1}{p(X = x)} = \mathbb{E}[\log \frac{1}{p(X)}]$$

Many ways to think about this quantity:

- The expected number of yes/no questions you would need to ask to correctly predict the next symbol sampled from distribution $p(X)$
- The expected "surprise" or "information" in the possible outcomes of random variable X
- The minimum number of bits required to compress a symbol x sampled from distribution $p(X)$

(Discrete) Entropy of a Coin Flip

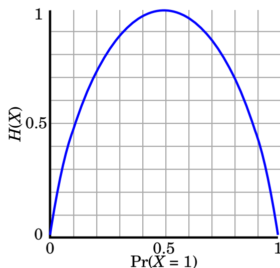


Figure 4: Binary Entropy

- Entropy of a fair coin flip is $0.5 log(2) + 0.5 log(2) = log(2) = 1$ bits
- Entropy of a fair dice is $log(6) = 2.58$ bits
- Entropy of characters in English words is about 2.62 bits
- Entropy of characters from the English alphabet selected uniformly at random is $log(26) = 4.7$ bits

Kullback-Leibler Divergence

Also called: KL Divergence, Relative Entropy

For discrete probability distributions:

$$KL[q(z) \mid\mid p(z)] = \sum_z q(z) \log \frac{q(z)}{p(z)}$$

For continuous probability distributions:

$$KL[q(z) \mid\mid p(z)] = \int q(z) \log \frac{q(z)}{p(z)} dz$$

KL Divergence Example Computation

Approximating an unfair coin with a fair coin.

- $p(z = 1) = 0.7$ and $p(z = 0) = 0.3$
- $q(z = 1) = q(z = 0) = 0.5$

$$
\begin{aligned}
KL[q(z) \mid\mid p(z)] &= \sum_z q(z) \log \frac{q(z)}{p(z)} \\
&= q(0) \log \frac{q(0)}{p(0)} + q(1) \log \frac{q(1)}{p(1)} \\
&= 0.5 \log \frac{0.5}{0.3} + 0.5 \log \frac{0.5}{0.7} \\
&= 0.872
\end{aligned}
$$

KL Divergence is not symmetric!

Approximating a fair coin with an unfair coin.

- $p(z=1) = 0.7$ and $p(z=0) = 0.3$
- $q(z=1) = q(z=0) = 0.5$

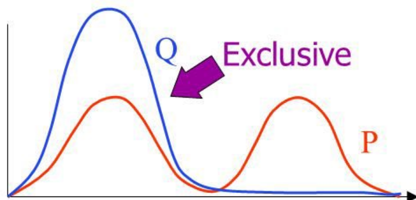
$$
\begin{aligned}
KL[p(z) \mid\mid q(z)] &= \sum_z p(z) \log \frac{p(z)}{q(z)} \\
&= p(0) \log \frac{p(0)}{q(0)} + p(1) \log \frac{p(1)}{q(1)} \\
&= 0.3 \log \frac{0.3}{0.5} + 0.7 \log \frac{0.7}{0.5} \\
&= 0.823 \\
&\neq KL[q(z) \mid\mid p(z)]
\end{aligned}
$$

Minimizing KL Divergence

Minimising

$\mathrm{KL}(Q||P)$

$$= \sum_H Q(H) \ln \frac{Q(H)}{P(H|V)}$$



Minimising

$\mathrm{KL}(P||Q)$

$$= \sum_H P(H|V) \ln \frac{P(H|V)}{Q(H)}$$

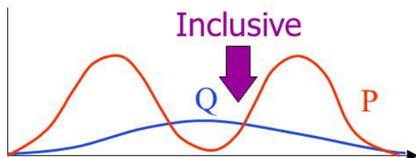


Figure 5: Two directions of the KL Divergence

KL divergence Properties

The KL divergence is a measure of the difference between probability distributions.

KL divergence is an asymmetric, nonnegative measure, not a norm. It doesn't obey the triangle inequality.

KL divergence is always non-negative. Hint: you can show this using the inequality $ln(x) \leq x - 1$ for $x > 0$

KL Divergence: continuous example

Suppose we have two Gaussian distributions $p(x) \sim N(\mu_1, \sigma_1^2)$ and $q(x) \sim N(\mu_2, \sigma_2^2)$.

What is the KL divergence $KL[p(z) \mid\mid q(z)]$?

Recall:

$$p(z; \mu_1, \sigma_1^2) = \frac{1}{\sqrt{2\pi\sigma_1^2}} e^{-\frac{(z-\mu_1)^2}{2\sigma_1^2}}$$

$$\log p(z; \mu_1, \sigma_1^2) = -\log \sqrt{2\pi\sigma_1^2} - \frac{(z-\mu_1)^2}{2\sigma_1^2}$$

KL Divergence: Entropy and Cross-Entropy

We can split the KL divergence into two terms, which we can compute separately:

$$
\begin{aligned}
KL[p(z) \mid\mid q(z)] &= \int p(z) \log \frac{p(z)}{q(z)} dz \\
&= \int p(z)(\log p(z) - \log q(z)) dz \\
&= \int p(z) \log p(z) dz - \int p(z) \log q(z) dz \\
&= -\text{entropy} - \text{cross-entropy}
\end{aligned}
$$

KL Divergence: continuous example, entropy computation

$$\begin{aligned}
\int p(z) \log p(z) dz &= \int p(z)(-\log \sqrt{2\pi\sigma_1^2} - \frac{(z-\mu_1)^2}{2\sigma_1^2}) dz \\
&= -\int p(z) \frac{1}{2} \log(2\pi\sigma_1^2) dz - \int p(z) \frac{(z-\mu_1)^2}{2\sigma_1^2} dz \\
&= -\frac{1}{2} \log(2\pi\sigma_1^2) \int p(z) dz - \frac{1}{2\sigma_1^2} \int p(z)(z-\mu_1)^2 dz \\
&= -\frac{1}{2} \log(2\pi\sigma_1^2) - \frac{1}{2} \\
&= -\frac{1}{2} \log(\sigma_1^2) - \frac{1}{2} \log(2\pi) - \frac{1}{2}
\end{aligned}$$

Since $\int p(z) dz = 1$ and $\int p(z)(z-\mu_1)^2 dz = \sigma_1^2$

KL Divergence: continuous example, cross-entropy computation

$$
\begin{aligned}
\int p(z) \log q(z) dz &= \int p(z)(-\log \sqrt{2\pi\sigma_2^2} - \frac{(z-\mu_2)^2}{2\sigma_2^2}) dz \\
&= -\int p(z) \frac{1}{2} \log(2\pi\sigma_2^2) dz - \int p(z) \frac{(z-\mu_2)^2}{2\sigma_2^2} dz \\
&= -\frac{1}{2} \log(2\pi\sigma_2^2) - \frac{1}{2\sigma_2^2} \int p(z)(z-\mu_2)^2 dz \\
&= \ldots \\
&= -\frac{1}{2} \log(2\pi\sigma_2^2) - \frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{2\sigma_2^2}
\end{aligned}
$$

Back to autoencoders: summary so far

Autoencoder:

- Decoder: point estimate of $p_{\theta^*}(\mathbf{x}|\mathbf{z})$
- Encoder: point estimate of the value of $\mathbf{z}$ that generated the image $\mathbf{x}$

VAE:

- Decoder: probabilistic estimate of $p_{\theta^*}(\mathbf{x}|\mathbf{z})$
- Encoder: probabilistic estimate of a Gaussian distribution $q_\phi(\mathbf{z}|\mathbf{x})$ that approximates the distribution $p_{\theta^*}(\mathbf{z}|\mathbf{x})$
  - In particular, our encoder will be a neural network that predicts the mean and standard deviation of $q_\phi(\mathbf{z}|\mathbf{x})$
  - We can then sample $\mathbf{z}$ from this distribution!

VAE Objective

But how do we train a VAE?

We want to maximize the likelihood of our data:

$\log p(x) = \log \int p(x,z)dz = \log \int p(x|z)p(z)dz$

And we want to make sure that the distributions $q(z|x)$ and $p(z|x)$ are close:

- We want to minimize $KL[q(\mathbf{z}|\mathbf{x}) \,||\, p(\mathbf{z}|\mathbf{x})]$
- This is a measure of encoder quality

In other words, we want to maximize

$$-KL[q(\mathbf{z}|\mathbf{x}) \,||\, p(\mathbf{z}|\mathbf{x})] + \log p(x)$$

How can we optimize this quantity in a tractable way?

VAE: Evidence Lower-Bound

$$
\begin{aligned}
KL[q(\mathbf{z}|\mathbf{x}) \mid\mid p(\mathbf{z}|\mathbf{x})] &= \int q(\mathbf{z}|\mathbf{x}) \log \frac{q(\mathbf{z}|\mathbf{x})}{p(\mathbf{z}|\mathbf{x})} dz \\
&= \mathbb{E}_q[\log \frac{q(\mathbf{z}|\mathbf{x})}{p(\mathbf{z}|\mathbf{x})}] \\
&= \mathbb{E}_q[\log q(\mathbf{z}|\mathbf{x})] - \mathbb{E}_q[\log p(\mathbf{z}|\mathbf{x})] \\
&= \mathbb{E}_q[\log q(\mathbf{z}|\mathbf{x})] - \mathbb{E}_q[\log p(\mathbf{z},\mathbf{x})] + \mathbb{E}_q[\log p(\mathbf{x})] \\
&= \mathbb{E}_q[\log q(\mathbf{z}|\mathbf{x})] - \mathbb{E}_q[\log p(\mathbf{z},\mathbf{x})] + \log p(\mathbf{x})
\end{aligned}
$$

We'll define the **evidence lower-bound**:

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

So we have

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

Optimizing the ELBO

The ELBO gives us a way to estimate the gradients of $\log p(\mathbf{x}) - KL[q(\mathbf{z}|\mathbf{x}) \mid\mid p(\mathbf{z}|\mathbf{x})]$

How?

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

- The right hand side of this expression is an expectation over $z \sim q(z|x)$
- To estimate the ELBO, we can **sample** from the distribution $z \sim q(z|x)$, and compute the terms inside.
- We can estimate gradients in the same way—this is called a **Monte-Carlo gradient estimator**

Monte Carlo Estimation

(This notation is unrelated to other slides: $p(z)$ is just a univariate Gaussian distribution, and $f_\phi(z)$ is a function parameterized by ϕ)

Suppose we want to optimize an objective $\mathcal{L}(\phi) = \mathbb{E}_{z \sim p(z)}[f_\phi(z)]$ where $p(z)$ is a normal distribution.

We can **estimate** $\mathcal{L}(\phi)$ by sampling $z_i \sim p(z)$ and computing

$$\begin{aligned}
\mathcal{L}(\phi) &= \mathbb{E}_{z \sim p(z)}[f_\phi(z)] \\
&= \int_z p(z) f_\phi(z) dz \\
&\approx \frac{1}{N} \sum_{i=1}^{N} f_\phi(z_i)
\end{aligned}$$

Monte Carlo Gradient Estimation

Likewise, if we want to estimate $\nabla_\phi \mathcal{L}$, we can sample $z_i \sim p(z)$ and compute

$$
\begin{aligned}
\nabla_\phi \mathcal{L} &= \nabla_\phi \mathbb{E}_{z \sim p(z)}[f_\phi(z)] \\
&= \nabla_\phi \int_z p(z) f_\phi(z) dz \\
&\approx \nabla_\phi \frac{1}{N} \sum_{i=1}^{N} f_\phi(z_i) \\
&= \frac{1}{N} \sum_{i=1}^{N} \nabla_\phi f_\phi(z_i)
\end{aligned}
$$

The reparamaterization trick

$\text{ELBO}_{\theta,\phi}(\mathbf{x}) = \mathbb{E}_{q_\phi}[\log p_\theta(\mathbf{z}, \mathbf{x}) - \log q_\phi(\mathbf{z}|\mathbf{x})]$

Problem: typical Monte-Carlo gradient estimator with samples $\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})$ has very high variance

Reparameterization trick: instead of sampling $\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})$ express $\mathbf{z} = g_\phi(\epsilon, \mathbf{x})$ where g is deterministic and only ϵ is stochastic.

In practise, the reparameterization trick is what makes the VAE encoder deterministic. When running a VAE forward pass:

1. We get the means and standard deviations from the VAE
2. We sample from $\mathcal{N}(\mathbf{0}, \mathbf{I})$
3. We use the samples from step 2 to get a sample from $q(z)$ obtained from step 1

VAE: Summary so far

Decoder: estimate of $p_{\theta^*}(\mathbf{x}|\mathbf{z})$

Encoder: estimate of a Gaussian distribution $q_\phi(\mathbf{z}|\mathbf{x})$ that approximates the distribution $p_{\theta^*}(\mathbf{z}|\mathbf{x})$

- Encoder is a NN that predicts the mean and standard deviation of $q_\phi(\mathbf{z}|\mathbf{x})$
- Use the **reparameterization trick** to sample from this distribution

The VAE objective is equal to the evidence lower-bound:

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

Which we can estimate using Monte Carlo

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

VAE: Summary so far

But given a value $z \sim q(z|x)$, how can we compute

$$\log p(\mathbf{z}, \mathbf{x}) - \log q(\mathbf{z}|\mathbf{x})$$

...or its derivative with respect to the neural network parameters?

We need to do some more math to write this quantity in a form that is easier to estimate.

VAE: a simpler form

$$\begin{aligned}
\mathrm{ELBO}_{\theta,\phi}(\mathbf{x}) &= \mathbb{E}_{q_\phi}[\log p_\theta(\mathbf{z},\mathbf{x}) - \log q_\phi(\mathbf{z}|\mathbf{x})] \\
&= \mathbb{E}_{q_\phi}[\log p_\theta(\mathbf{x}|\mathbf{z}) + \log p_\theta(\mathbf{z}) - \log q_\phi(\mathbf{z}|\mathbf{x})] \\
&= \mathbb{E}_{q_\phi}[\log p_\theta(\mathbf{x}|\mathbf{z})] - \mathbb{E}_{q_\phi}[\log q_\phi(\mathbf{z}|\mathbf{x}) - \log p_\theta(\mathbf{z})] \\
&= \mathbb{E}_{q_\phi}[\log p_\theta(\mathbf{x}|\mathbf{z})] - \mathrm{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \;||\; p_\theta(\mathbf{z})) \\
&= \text{decoding quality} - \text{encoding regularization}
\end{aligned}$$

Both terms can be computed easily if we make some simplifying assumptions

Let's see how. . .

Computing Decoding Quality

In order to estimate this quantity

$$\mathbb{E}_{q_\phi}[\log p_\theta(\mathbf{x}|\mathbf{z})]$$

. . . we need to make some assumptions about the distribution $p_\theta(\mathbf{x}|\mathbf{z})$.

If we make the assumption that $p_\theta(\mathbf{x}|\mathbf{z})$ is a normal distribution centered around some pixel intensity, then optimizing $p_\theta(\mathbf{x}|\mathbf{z})$ is equivalent to optimizing the *square loss*!

That is, $p_\theta(\mathbf{x}|\mathbf{z})$ tells us how intense a pixel could be, but that pixel could be a bit darker/lighter, following a normal distribution).

Bonus: A traditional autoencoder is optimizing this same quantity!

Computing Encoding Quality

This KL divergence computes the difference in distribution between two distributions:

$$\mathrm{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \mid\mid p_\theta(\mathbf{z}))$$

- $q_\phi(\mathbf{z}|\mathbf{x})$ is a normal distribution that approximates $p_\theta(\mathbf{z}|\mathbf{x})$
- $p_\theta(\mathbf{z})$ is the **prior distribution on z**
  - distribution of z when we don't know anything about $\mathbf{x}$ or any other quantity

Since $\mathbf{z}$ is a *latent* variable, not actually observed in the real word, we can choose $p_\theta(\mathbf{z})$

- we choose $p_\theta(z) = \mathcal{N}(\mathbf{0}, \mathbf{I})$

. . . and we know how to compute the KL divergence of two Gaussian distributions!

Interpretation

The VAE objective

$$\mathbb{E}_{q_\phi}[\log p_\theta(\mathbf{x}|\mathbf{z})] - \mathrm{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \mid\mid p_\theta(\mathbf{z}))$$

has an extra regularization term that the traditional autoencoder does not.

This extra regularization term pushes the values of $\mathbf{z}$ to be closer to 0!

MNIST results

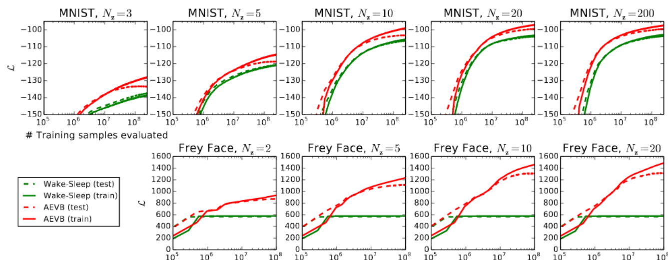
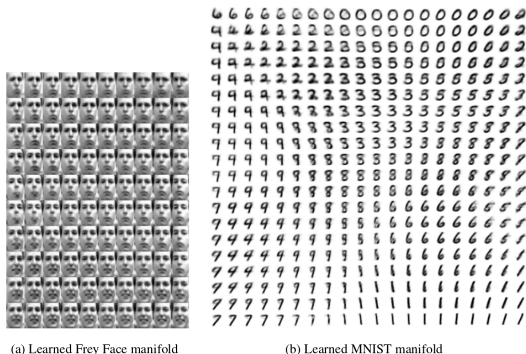


Figure 2: Comparison of our AEVB method to the wake-sleep algorithm, in terms of optimizing the lower bound, for different dimensionality of latent space ($N_\mathbf{z}$). Our method converged considerably faster and reached a better solution in all experiments. Interestingly enough, more latent variables does not result in more overfitting, which is explained by the regularizing effect of the lower bound. Vertical axis: the estimated average variational lower bound per datapoint. The estimator variance was small (< 1) and omitted. Horizontal axis: amount of training points evaluated. Computation took around 20-40 minutes per million training samples with a Intel Xeon CPU running at an effective 40 GFLOPS.

Frey Faces results



(a) Learned Frey Face manifold (b) Learned MNIST manifold

Figure 4: Visualisations of learned data manifold for generative models with two-dimensional latent space, learned with AEVB. Since the prior of the latent space is Gaussian, linearly spaced coordinates on the unit square were transformed through the inverse CDF of the Gaussian to produce values of the latent variables $\mathbf{z}$. For each of these values $\mathbf{z}$, we plotted the corresponding generative $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$ with the learned parameters $\boldsymbol{\theta}$.

Dimension of latent variables

(a) 2-D latent space (b) 5-D latent space (c) 10-D latent space (d) 20-D latent space

Figure 5: Random samples from learned generative models of MNIST for different dimensionalities of latent space.