Generative models

Deep Learning (2IMM10)

Vlado Menkovski, Simon Koop

Spring 2020

6 Deep generative models

6.1 Motivation

Point estimate models So far we covered different models developed from data. Specifically for different tasks we developed a formulation where the model maps an input x to the an output y ($f_\theta(x) = y$) or more generally, models that describe a distribution over the values of y given a value for x ($f_\theta(x) = P(y|x)$). In this setting, we reduced the task as finding the parameters θ that minimize some error $L(y, \hat{y}) = L(y, f_\theta(x))$ given a training dataset $D : \{x^{(i)}, y^{(i)}\}_{i=0}^{N}$. We then solve this task $\theta = \arg\min_\theta L(y, f_\theta(x))$ with the SGD algorithm. We refer to the $f_\theta(x) = P(y|x)$ models as discriminative models. An example of such a model is given in fig. 1.

For a binary classification task such as this we define the model as in eq. (1).

$$P(y = c|X = (x_0, x_1)) = \frac{1}{1 + e^{(w_0 x_0 + w_1 x_1 + b)}} \tag{1}$$

In contrast to such a model we can also develop a model of the distribution of the data itself fig. 2.

$$P(X, Y)$$

- describes the relationship between X and Y. This means that we would like our model to express the likelihood of observing the different values of x and y This can be defined as $P(x, y)$ (probability of x and y rather than probability of x given y), the joint probability of x and y.

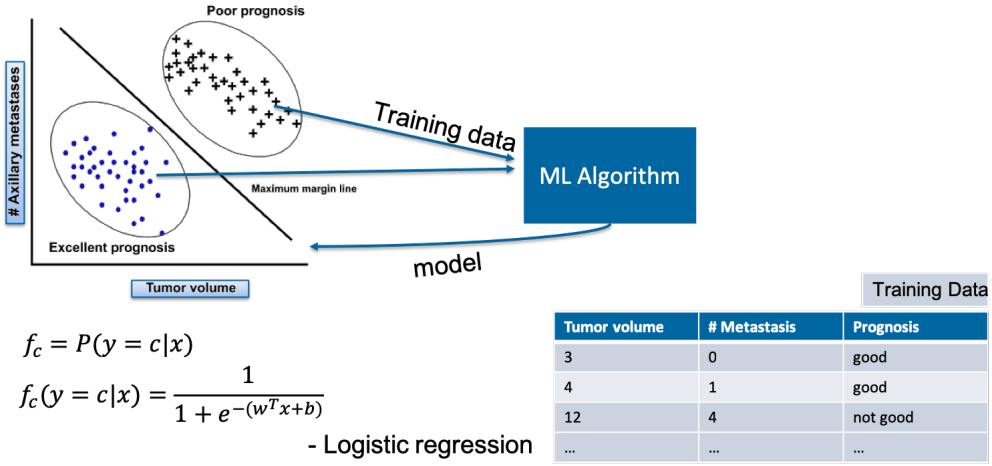


Figure 1: Discriminative model

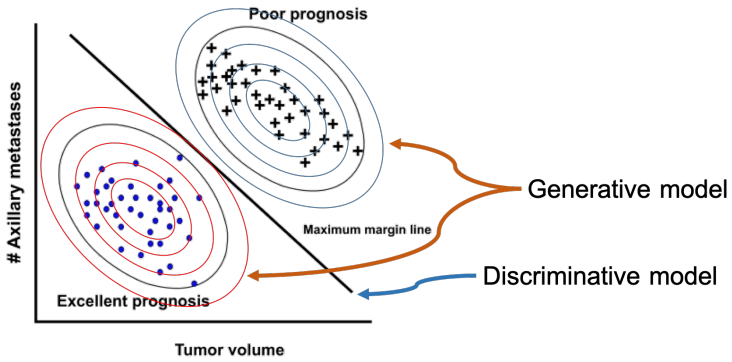


Figure 2: Generative models intuition

Let us for example take the following model:

$$P(X|Y=0) \sim \mathcal{N}(\mu_0, \Sigma_0)$$

$$P(X|Y=1) \sim \mathcal{N}(\mu_1, \Sigma_1)$$

Given such a model, we can compute the most likely class assignment for a datapoint with the following expression:

$$\hat{y} = \arg\max_c P(y=c) P_\theta(X|y=c)$$

Where θ are the parameters for the distribution.

Important: In the rest of the lecture notes we will use terminology and notation from probability theory and Bayesian inference that you need to be familiar with. Please refer to section .1 to familiarize/refresh yourself with the needed background.

6.2 Training a probabilistic model

An important question we have to ask ourselves when making a generative model, is how are we going to measure its performance, and how are we going to train it. We will discuss two commonly used frameworks for doing this:

- Maximizing the (log-)likelihood of the model given the data.
- Minimizing a divergence (or other notion of distance) between our model distribution and the (hypothetical) true or empirical distribution.

These two frameworks are not mutually exclusive. In fact, the cross-entropy loss we have seen popping-up in many instances, fits perfectly in both frameworks.

Maximizing the log-likelihood

The likelihood of the data given the model is an often used to select the best parameters. The parameters maximizing this likelihood are called the maximum likelihood estimate (MSE). The way this works is that we have a model for our data, $X = \{x^{(1)}, \ldots, x^{(n)}\}$, in the form of a parameterized

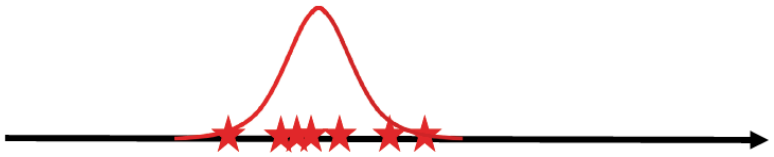


Figure 3: Normal distribution

probability distribution P_θ with density p_θ, and we assume all data points are independent and identically distributed. The joint density is then simply the product of the density for a single random variable, so that the likelihood of the total data set is given by

$$p_\theta(X) = \prod_{i=1}^{n} p_\theta(x^{(i)}).$$

In order to maximize this, we take the logarithm to obtain

$$\log p_\theta(x) = \sum_{i=1}^{n} \log(p_\theta(x^{(i)}))$$

and try to maximize this *log-likelihood*. We illustrate how this works for a simple distribution in the following subsection.

Illustrative Example

To see how this likelihood maximization traditionally works, let's look at a classical statistics exercise. We are given a set of numbers,

$$X = \{x^{(1)}, ..., x^{(m)}\} \subset \mathbb{R}$$

and assume that these numbers are realizations of random variables that are independent and identically distributed (i.i.d.). We define a parametric family of densities $(p_\theta)_{\theta \in \mathbb{R}^d}$ and want to find the parameters of the distribution that maximize the likelihood of our data:

$$\hat{\theta} \leftarrow \arg\max_{\theta \in \mathbb{R}^d} p_\theta(X) \tag{2}$$

For our family of distributions we choose the normal distribution, $\mathcal{N}(\mu, \sigma^2)$, so that

$$p_\theta(x) = \mathcal{N}(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

where $\theta = \{\mu, \sigma\}$. With this choice for our family of distributions, our objective, eq. (2) boils down

to

$$\begin{aligned}\hat{\theta} &= \arg\max_{\theta \in \mathbb{R}^2} \prod_{i=1}^{m} p_\theta(x^{(i)}) \\ &= \arg\max_{\theta \in \mathbb{R}^2} \frac{1}{m} \prod_{i=1}^{m} p_\theta(x^{(i)}) \\ &= \arg\max_{\theta} \frac{1}{m} \sum_{i=1}^{m} \log p_\theta(x^{(i)}) \\ &= \arg\max_{\mu,\sigma} \frac{1}{m} \sum_{i=1}^{m} \frac{-(x^{(i)} - \mu)^2}{2\sigma^2} - \frac{1}{2} \log(2\pi\sigma^2). \end{aligned} \tag{3}$$

To find this maximum, we first take the derivative with respect to μ:

$$\frac{\partial}{\partial \mu}(3) = \frac{-1}{m\,\sigma^2} \sum_{i=1}^{m} (x^{(i)} - \mu).$$

This value is 0 if and only if

$$\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)},$$

so to maximize the log-likelihood of our data set, we must take

$$\hat{\mu} = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}.$$

Similar procedure (differentiating to σ^2) gives us

$$\hat{\sigma}^2 = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)} - \hat{\mu})^2.$$

Maximizing the log-likelihood — continuation

The generative models we deal with in this course typically have a much more complex relation between their parameters and the value of the log-likelihood. Instead of solving things by setting derivatives to 0, we use gradient descent and backpropagation to maximize the log-likelihood (or minimize the negative log-likelihood). Moreover, instead of computing the log-likelihood for the full dataset, we approximate it using smaller batches of datapoints.

Minimizing the distance between distributions

When we have a probabilistic generative model and a dataset, we can see this as having two distributions: a model distribution, and the true distribution we are trying to model. The goal is to get these two distributions to match. To do this, we could try to minimize some kind of distance between the two distributions.

There are many notions of distance between probability distributions available to us, all with their own strengths and weaknesses. Some notions of distance that are popular in machine learning are

Kullback-Leibler divergence, Jensen-Shannon divergence, the Weierstrass metrics, and Maximum Mean Discrepancies (MMD). Here we shall focus on the Kullback-Leibler divergence and the Jensen-Shannon divergence.

If we have two discrete probability distributions Q and P on the sample space S, the *Kullback-Leibler divergence* from Q to P is defined as

$$D_{KL}(Q \mid\mid P) = \sum_{s \in S} Q(s) \log \left(\frac{Q(s)}{P(s)} \right),$$

For distributions corresponding to continuous random variables with range $S \subset \mathbb{R}$, the Kullback-Leibler divergence is given by

$$D_{KL}(Q \mid\mid P) = \int_S q(s) \log \left(\frac{q(s)}{p(s)} \right) \mathrm{d}s,$$

where p and q are the densities of P and Q. The definition for joint distributions is analogous.

If Q is the true distribution, and P is an approximation of this distribution, the Kullback-Leibler divergence from P to Q can be interpreted as the amount of information lost by using the approximation over the true distribution. It has the following properties:

- The divergence is non-negative — i.e. $D_{KL}(Q \mid\mid P) \geq 0$ — with equality if and only if $P = Q$;
- the divergence is not symmetric, i.e. $D_{KL}(P \mid\mid Q) \neq D_{KL}(Q \mid\mid P)$ in general;
- it sums over dimensions, i.e. if $P(x, y) = P_1(x)P_2(y)$ and $Q(x, y) = Q_1(x)Q_2(Y)$, then

  $$D_{KL}(Q \mid\mid P) = D_{KL}(Q_1 \mid\mid P_1) + D_{KL}(Q_2 \mid\mid P_2)$$

  and similar for continuous distributions.

As we will see in Section 6.2, maximizing the log-likelihood is equivalent to minimizing the Kullback-Leibler divergence between the model distribution and the empirical distribution of the data. The Kullback-Leibler divergence also plays an important role in the Variational Auto-Encoder discussed in Section 6.6.

Another divergence is the *Jensen-Shannon divergence*. This divergence can be seen as a symmetrized version of the Kullback-Leibler divergence, and is given by

$$D_{JS}(P \mid\mid Q) = \frac{1}{2} D_{KL}(P \mid\mid M) + \frac{1}{2} D_{KL}(Q \mid\mid M)$$
$$M = \frac{P + Q}{2}.$$

This divergence has the following properties:

- it is symmetric, i.e. $D_{JS}(P \mid\mid Q) = D_{JS}(Q \mid\mid P)$;
- it is bounded by $0 \leq D_{JS}(P \mid\mid Q) \leq \log(2)$;

- it is 0 if and only if the distributions mach, i.e. $D_{JS}(P \mid\mid Q) \iff P = Q$.

The training objective of the original GAN can be seen as minimizing the Jensen-Shannon divergence.

When we train a generative model by minimizing some notion of distance, we usually see the dataset, $X = \{x^{(1)}, \ldots, x^{(n)}\}$ as drawn from independent random variables that all have the same true probability distribution[1], $q(x)$, often called the empirical distribution. Our generative model implements an approximation $p_\theta(x)$ to that distribution and we want to pick our model-parameters θ such that some distance, e.g. the Kullback-Leibler divergence, or the Jensen-Shannon divergence, is minimal.

Cross-entropy

So far we have discussed two common frameworks for training a generative model:

1. maximizing the log-likelihood of the data according to our model distribution;
2. minimizing some distance between our model distribution, p_θ and the empirical distribution q.

Let's have a quick look at how these two relate. We assume we have a dataset $X = \{x^{(1)}, \ldots, x^{(N)}\}$ which we view as realizations of i.i.d. random variables distributed according to $q(x)$, the empirical distribution. We try to approximate this using a model distribution $p_\theta(x)$ for which we want to find the right parameters.

We can do this by trying to maximizing the log-likelihood of the data, or equivalently, to minimize the negative log-likelihood:

$$\hat{\theta} = \arg\min_\theta \sum_{i=1}^{n} -\log(p_\theta(x^{(i)})).$$

When the model has a complex distribution, we often want to use gradient descent to do this optimization. In that case, instead of looking at the log-likelihood of the full dataset, we take batches of datapoints and use those for computing the gradient:

$$I \subset \{1, \ldots, n\}$$
$$\sum_{i=1}^{n} -\log(p_\theta(x^{(i)})) \approx \frac{n}{|I|} \sum_{i \in I} -\log(p_\theta(x^{(i)})).$$

Changing the factor from $\frac{n}{|I|}$ to $\frac{1}{|I|}$ doesn't change the direction of the gradient, only its size. Since we can freely choose the learning rate of gradient descent this need not be of any consequence, and we can use

$$\frac{1}{|I|} \sum_{i \in I} -\log(p_\theta(x^{(i)}))$$

[1]Notation: we will use $q(x)$ to denote both the distribution and its density. Similarly $p_\theta(x)$ denotes both the distribution and the density.

instead, which comes down to replacing a sum by an average. But due to the law of large numbers, this expression is a good approximation of

$$\frac{1}{|I|}\sum_{i\in I} -\log(p_\theta(x^{(i)})) \approx \mathbb{E}_{q(x)}[-\log(p_\theta(x))]$$

(at least when $|I|$ is large). Put more concisely, *minimizing the negative log-likelihood of the dataset is equivalent*[2] *to minimizing the expected negative log-likelihood:*

$$\arg\min_\theta \sum_{i=1}^{n} -\log(p_\theta(x^{(i)})) \approx \arg\min_\theta \mathbb{E}_{q(x)}[-\log(p_\theta(x))].$$

This expected negative log-likelihood is called the **cross entropy** of the model distribution $p_\theta(x)$ to the true distribution $q(x)$, and is denoted by

$$H(q(x), p_\theta(x)) = \mathbb{E}_{q(x)}[-\log(p_\theta(x))].$$

We can rewrite this cross entropy as follows:

$$\begin{aligned} H(q(x), p_\theta(x)) &= H(q(x)) + D_{KL}(q(x) \,||\, p_\theta(x)) \\ H(q(x)) &= \mathbb{E}_{q(x)}[-\log(q(x))] \end{aligned}$$

where the entropy of $q(X)$, $H(q(x))$, is a constant with respect to the model parameters θ. Since $H(q(x))$ is constant, minimizing the cross-entropy is equivalent to minimizing the Kullback-Leibler divergence:

$$\begin{aligned} \arg\min_\theta \sum_{i=1}^{n} -\log(p_\theta(x^{(i)})) &\approx \arg\min_\theta H(q(x), p_\theta(x)) \\ &= \arg\min_\theta D_{KL}(q(x) \,||\, p_\theta(x)). \end{aligned}$$

In conclusion we can see maximizing the log-likelihood of the data-set as a special case of minimizing a distance between the model distribution and the empirical distribution, where we use the Kullback-Leibler divergence as our notion of distance. Algorithmically we try to solve them in the same way.

6.3 Some challenges with probabilistic models

So far we have discussed why we might want to use generative models to attempt to approximate the distribution of our dataset, and what kind of loss functionals we can use to train them. However, there are some challenges we face when trying to model data this way. These challenges come from the problem of dimensionality: it is hard and costly to represent very high dimensional probability distributions, especially when the distributions are complex. Moreover, the integrals that show up in the various loss functions we want to use become impossible to calculate exactly, and even become difficult to approximate in a deterministic way. Many calculations that can be done cheaply for low dimensional data become very expensive when the number of dimensions becomes large.

[2] Or at least they are equivalent in so far as that we would solve both minimization problems algorithmically in the same way and in that the quantities that we try to minimize are, after scaling, good approximations of each other.

Representations of distributions

The data we try to model is often very high dimensional. Images of only 100×100 pixels already are 10 000 dimensional. Suppose we have very small black-and-white pictures of 10×10 pixels which have value 1 if they are white, and 0 if they are black. If we want to specify a probability distribution over this set as an arbitrary joint distribution, that would mean we have to specify $2^{100} - 1$ parameters.

For continuous distributions we have various families of parameterized probability distributions such as the family multivariate normal distributions. These make it possible to easily specify very high dimensional distributions, but are generally unfit for modelling the complex data we try to represent.

Clearly we need better ways to represent probability distributions. We will discuss three ways of doing this here, and later in the chapter we will see these in action. For those who are not familiar with conditional probability we have added a short summary of basic probability theory in Section .1.

The first way to factorize the joint distribution using the chain rule of probability:

$$\begin{aligned} p_\theta(\mathbf{x}) &= \prod_{i=1}^{n} p_\theta(x_i \mid x_1, \ldots, x_{i-1}) \\ &= \prod_{i=1}^{n} p_\theta(x_i \mid x_{<i}). \end{aligned}$$

— think of $x_1, \ldots, x_k$ as for example the pixels of an image. As is this chain rule doesn't help much, but if we implement $p_\theta(x_i \mid x_{<i})$ using some RNN, we can learn very complex and high dimensional probability distributions using relatively few parameters. Models using this technique are called auto-regressive models. In Section 6.4 we will look at how this works in detail and see how this is used by PixelRNN to generate images.

The second way we want to discuss is to use a latent variable. We describe the rationale behind this in Section 6.5. This is the approach taken in the Variational Auto-Encoder. With these models we assume that our data $\mathbf{x}$ can best be explained using an unobserved *latent* variable $\mathbf{z}$. We model the joint distribution of $\mathbf{x}$ and $\mathbf{z}$ as

$$p_\theta(\mathbf{x}, \mathbf{z}) = p_\theta(\mathbf{z}) p_\theta(\mathbf{x} \mid \mathbf{z})$$

which makes that our data distribution is given by

$$p_\theta(\mathbf{x}) = \int_Z p_\theta(\mathbf{x} \mid \mathbf{z}) p_\theta(\mathbf{z}) \mathrm{d}\mathbf{z}.$$

In order to sample from $p_\theta(\mathbf{x})$ we can simply sample z from $p_\theta(\mathbf{z})$ and then sample x from $p_\theta(\mathbf{x} \mid \mathbf{z})$. Here $p_\theta(\mathbf{z})$ is can often be chosen to be a relatively simple distribution such as a multivariate standard normal distribution. We can then choose $p_\theta(\mathbf{x} \mid \mathbf{z})$ to be some parameterized family of distributions, where the parameter is determined based on $\mathbf{z}$ using a neural network. The classical example from a VAE is

$$\begin{aligned} p_\theta(\mathbf{z}) &= \mathcal{N}(0, I), \\ p_\theta(\mathbf{x} \mid \mathbf{z}) &= \mathcal{N}(f_\theta(\mathbf{z}), \lambda I), \end{aligned}$$

	Autoregressive	VAE	GAN
Advantages	- simple and stable training - good log likelihood estimation	- efficient inference with latent variables	- sharp sampling of datapoints
Drawbacks	- inefficient sampling	- blurry reconstruction	- unstable training

Figure 4: Generative models comparison

where f_θ is a neural network with weights and biases θ, and λ is a parameter mainly important for the training of the model.

This idea of drawing from a simple distribution and using a neural network to add complexity is the basis of the third approach too: using a push-forward measure as our model distribution. We take a neural network f_θ, and a simple distribution such as a multivariate normal distribution $\mathcal{N}(0, I)$.We then very easily sample from our model distribution by sampling some

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

and computing our x as

$$x = f_\theta(\epsilon).$$

This is how the model distribution is represented in the original GAN. A downside to this is that it is very hard to do density estimation using this representation of a probability distribution.

We will illustrate these techniques by discussing three algorithms that use them: PixelRNN, VAE, and GAN. Many more recent algorithms use combinations of these techniques or variations on them to model the probability distribution of a dataset. In Figure 4 we give a short comparison of Autoregressive models, VAE, and GAN.

6.4 Autoregressive models

There are a number of solutions to the challenges discussed in Section 6.3 in developing generative models for high dimensional data. In continuation we take an in-depth look of three approaches for developing generative models. The first type of generative models we will look into are the autoregressive models of which PixelRNN is a great example. Autoregressive models in deep learning are models where we specify the distribution through factorization:

$$\begin{aligned} p_\theta(\mathbf{x}) &= \prod_{i=1}^{n} p_\theta(x_i \mid x_1, \ldots, x_{i-1}) \\ &= \prod_{i=1}^{n} p_\theta(x_i \mid x_{<i}). \end{aligned}$$

Figure 5: Autoregressive model - graphical representation

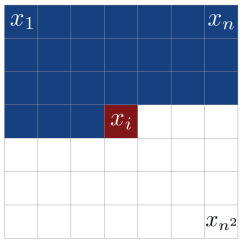


Figure 6: Sample generation with the PixelRNN method
[3]

The factorization allows us to reformulate the joint probability modeling problem into a sequence problem fig. 5. You can reflect back to the seq2seq models discussed in Chapter 5, where the output of the decoder at time t is made available for decoding at time $t + 1$.

PixelRNN models the joint distribution of the pixels in an image and allows us to sample from that distribution. In other words, the model allows us to generate images form the same distribution as the training data. PixelRNN views an image as a sequence of pixels and applies the factorization of the distribution to predict the next pixel from all previous pixels. To accomplish this it uses a variation of the LSTM inspired RNN cell.

In principle the PixelRNN model produces one pixel at the time. Starting from the first pixel the model produces a probability distribution over the values of the color intensities for that pixel. This values are discretized such that the model can model the output with a discrete distribution over the colors fig. 6. The decision for the value of the pixel at the location is then presented as input the the RNN cell such that the model can condition the probability of the next pixel value based on the decision that was made for the previous pixel value fig. 7.

Training

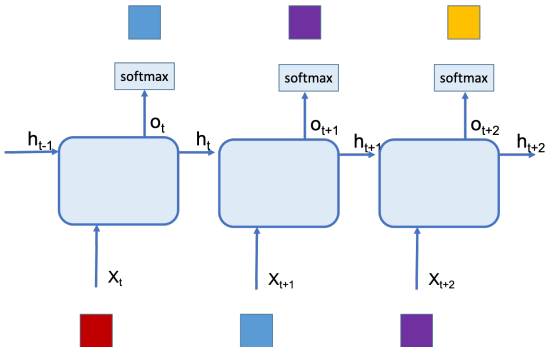


Figure 7: Sample generation PixelRNN with an RNN cell

Figure 8: PixelRNN example

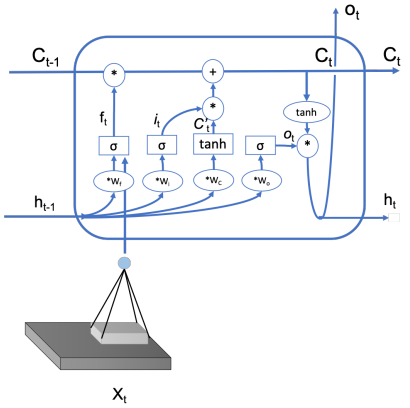


Figure 9: LSTM with convolutional filters attached to the input

To train the pixelRNN model we start by selecting an image from the training dataset. Starting from the first pixel we now use the model to generate each pixel of the image. As we did with the autoregressive seq2seq models, during training we present the ground truth to the model as a the previous output. Specifically, at time t we present to the RNN input the value x_{t-1} from training data rather than from the model's output. The loss is then compute from the output at each pixel location by computing the crossentropy between the model's output and the one-hot encoding of pixel values at each location in the training image. We apply this training process to cycling through all the images in the dataset.

A trained PixelRNN model can then be used to generate new images or part of images (fig. 8). This capability can be useful in tasks such as reconstruction of corrupted image or inference of occluded areas. The method also allows for estimating the likelihood that an image is a sample from the distribution of images given our training data. This can be useful to solving tasks such as anomaly detection.

PixelRNN - architecture

The major limitation of the PixelRNN model in its most simple form presented so far is that the conditional dependencies of the pixel values have to be propagated by the hidden state of the RNN cell. Even though different variation of the RNN cell have made improvements to the capability of this model to store longer term dependencies in general this is still a limitation of the RNN architecture. In the context of this particular task the different PixelRNN architectures that improve the performance were introduced.

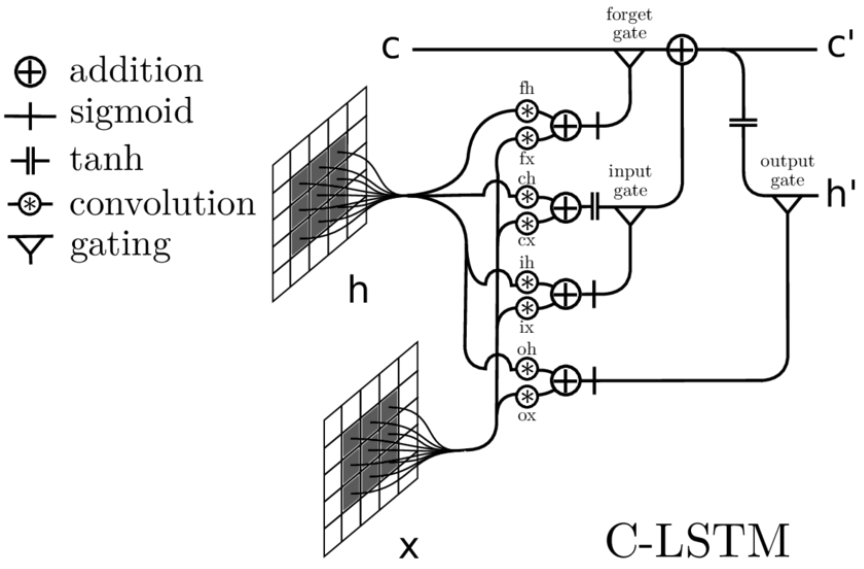


Figure 10: PixelRNN architectures. Shows convolutions over the input image and the previous hidden states.

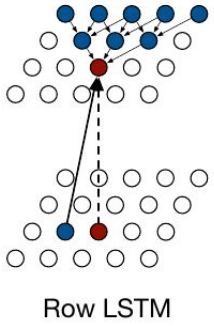


Figure 11: RowLSTM (looking at a row at the time)

One immediate improvement is to use the characteristics of the data to improve the expressiveness of the model. Specifically for image data, this means to use the localized correlations that are present. We can do that by using convolutional layers to process the image rather than processing one pixel at the time (fig. 9).

Further improvements are achieved by processing of the first layer of 2D convolutions with different strategies within the RNN cell it self. The PixelRNN approach actually extend the RNN cell to include convolutions both of the pixel values and the hidden states computed in previous steps (fig. 10. Two different strategies of processing the previous states are introduced. The RowLSTM model develops a probability distribution of a pixel at particular location based on the estimated value of neighboring pixels in the previous row (fig. 11 and the DiagonalBiLSTM model develops a conditional distribution of the pixels on an even wider area of neighbours (fig. 12). These improvements allow for more effective training of these models as the patterns present in the data are more explicit during training. The full technical details are available in the publication.

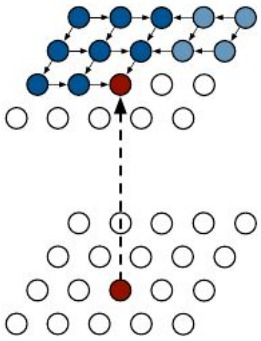


Figure 12: DiagonalBiLSTM

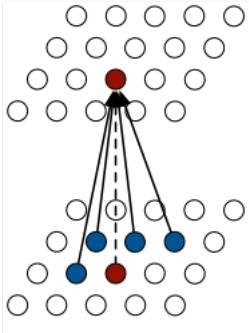


Figure 13: PixelCNN model

In this work the the we are also introduced to the PixelCNN model. This model is a fully convolutional neural networks that without a RNN cell and effectively limits the conditioning of each pixel values only to the immediate neighborhood (fig. 13). This limitation may be significant for images where a larger context may be required to form a good representation of the data. But in cases where global patterns are not present or they have little influence on value of the pixels the PixelCNN model presents a significantly more efficient solutions.

6.5 Latent Variable Models

A second strategy to develop generative models for high dimensional data is to introduce latent variables with simple distribution on which we can condition the observed variables these models are referred to as latent variable models. Imagine expressing the distribution of possible pixel values in a dataset of images. We assume that there is a simple description of the data in a lower dimensional space. In this space we can express the data with a simple distribution.

For example imaging a dataset of image of one chair (fig. 14). Each of the images in this dataset is a picture of the same chair, but at a different angle. Even though this dataset is high dimensional as the images contain many pixels, we can easily see that there is only one factors that describes the image and that is the orientation of the chair. This is a latent factor and we cannot observe it directly. The goal of the latent variable models is to develop a such descriptions of the data. Typically this is not an easy task as with most datasets we need more factors to develop an efficient representation of the data (fig. 15). Furthremore, we typically do not know the number of factors that we need and the most suited distribution to assign to them. Nevertheless the latent variable

Figure 14: A dataset of images of the same chair at a different angle

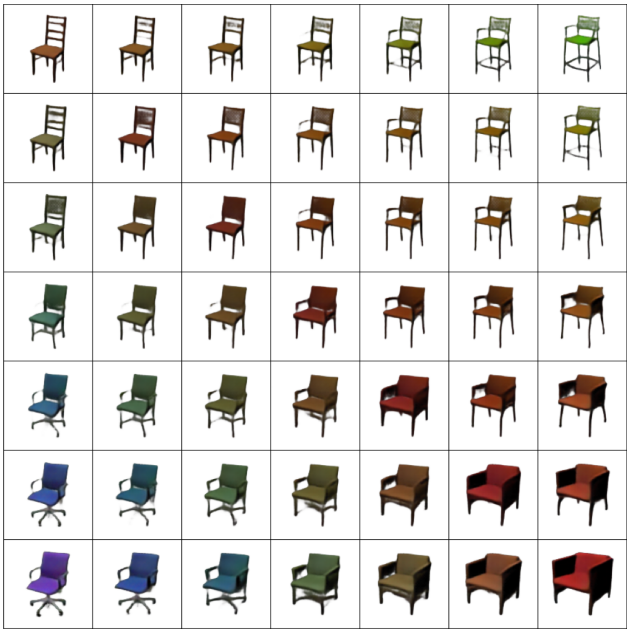


Figure 15: A dataset of images of different chairs

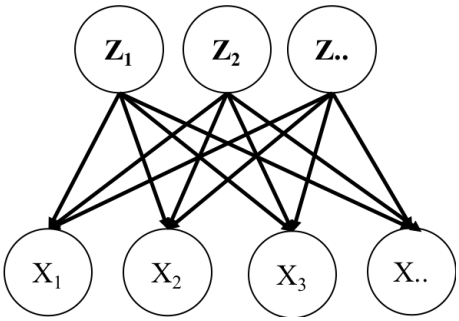


Figure 16: Graphical model representation of a Latent Variable Model

model give us the flexibility that eventually allows us develop generative models.

As a graphical representation the latent variable model can be depicted as in (fig. 16).

The joint distribution of the observed variables is

$$P(X) = \sum_{Z} P(X|Z)P(Z)$$

The Variational Auto-Encoder (VAE) is an of latent variable model that is a generative model for high dimensional data.

6.6 Variational Auto-Encoder

The VAE uses a latent variable z to describe the data x. That means it describes a distribution

$$p_\theta(x, z) = p_\theta(x \mid z)p_\theta(z),$$

which in turn gives a distribution over the data of the form

$$\begin{aligned} p_\theta(x) &= \int p_\theta(x, z)\mathrm{d}z \\ &= \int p_\theta(x \mid x)p_\theta(z)\mathrm{d}z. \end{aligned}$$

This integral however is itself intractable. We want the distribution we learn to match the true[4] distribution, which we will denote by $q(x)$, over the data as closely as possible. We can go at this in two ways, which as we will see, result in the same loss function. In both cases we introduce an extra conditional probability distribution $q_\phi(z \mid x)$ that we want to approximate $p_\theta(z \mid x)$ because the latter is intractable:

$$q_\phi(z \mid x) \approx p_\theta(z \mid x) = \frac{p_\theta(x \mid z)p_\theta(z)}{p_\theta(x)}.$$

We will write $q_\phi(x, z)$ for $q_\phi(z \mid x)q(x)$ which is, as a whole, a parameterized joint distribution on x and z.

[4]Also called the empirical distribution.

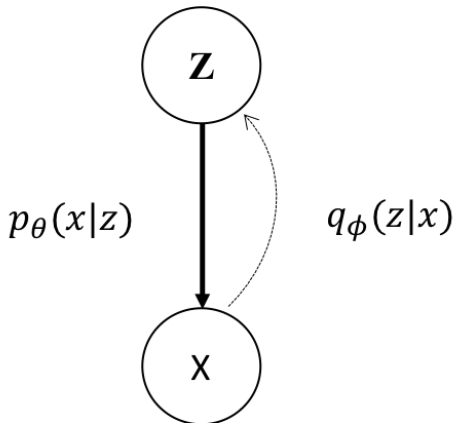


Figure 17: The graphical model for the Variational Auto-Encoder. The solid arrow represents the decoding distribution $p_\theta(x \mid z)$ and the dashed arrow represents the encoding distribution $q_\phi(z \mid x)$.

The first perspective is that we want to maximize the (log-)likelihood of the observed data according to our model distribution, i.e.

$$\arg\max_\theta \mathbb{E}_{q(x)} \left[\log p_\theta(x)\right].$$

In order to do this, we rewrite the expected log-likelihood as

$$\begin{aligned}
\mathbb{E}_{q(x)} \log p_\theta(x) &= \int_x 1 \cdot \log(p_\theta(x))\, q(x) \mathrm{d}x \\
&= \int_X \int_Z q_\phi(z \mid x) \mathrm{d}z \log(p_\theta(x))\, q(x) \mathrm{d}x \\
&= \int_{X \times Z} \log(p_\theta(x))\, q_\phi(x, z) \mathrm{d}x\, \mathrm{d}z \\
&= \int_{X \times Z} \log\left(\frac{p_\theta(x \mid z) p_\theta(z) q_\phi(z \mid x)}{p_\theta(z \mid x) q_\phi(z \mid x)}\right) q_\phi(x, z) \mathrm{d}x\, \mathrm{d}z \\
&= \int_{X \times Z} \log\left(\frac{q_\phi(z \mid x)}{p_\theta(z \mid x)}\right) q_\phi(z \mid x) \mathrm{d}z\, q(x) \mathrm{d}x - \int_{X \times Z} \log\left(\frac{q_\phi(z \mid x)}{p_\theta(z)}\right) q_\phi(z \mid x) \mathrm{d}z\, q(x) \mathrm{d}x \\
&\quad + \int_{X \times Z} \log(p_\theta(x \mid z))\, q_\phi(z \mid x) \mathrm{d}z\, q(x) \mathrm{d}x \\
&= \mathbb{E}_{q(x)} D_{KL}\left(q_\phi(z \mid x) \,||\, p_\theta(z \mid x)\right) - \mathbb{E}_{q(x)} D_{KL}\left(q_\phi(z \mid x) \,||\, p_\theta(z)\right) \\
&\quad + \mathbb{E}_{q(x)} \mathbb{E}_{q_\phi(z|x)} \log p_\theta(x \mid z).
\end{aligned}$$

Seeing as the Kullback-Leibler divergence in the intractable first term is always non-negative, we can bound the average log-likelihood from below by

$$\begin{aligned}
\mathbb{E}_{q(x)} \log p_\theta(x) \geq & - \mathbb{E}_{q(x)} D_{KL}\left(q_\phi(z \mid x) \,||\, p_\theta(z)\right) \\
& + \mathbb{E}_{q(x)} \mathbb{E}_{q_\phi(z|x)} \log p_\theta(x \mid z).
\end{aligned}$$

We can then try to maximize this "variational lower bound", also called the ELBO, in hopes of maximizing the log-likelihood. This is shown in Figure 18.

Another perspective is that q_ϕ and p_θ are both models for the same data with different properties: p_θ allows us to easily generate data but $p_\theta(x)$ is just an approximation to the real distribution and

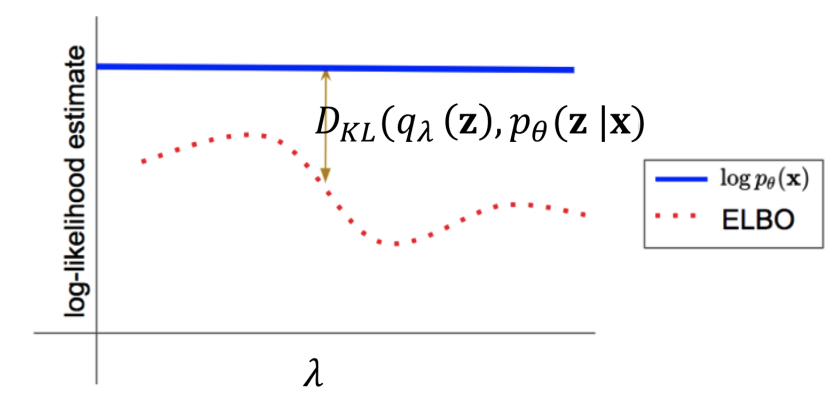


Figure 18: We try to maximize the log-likelihood by maximizing the ELBO. Source: Grover and Ermon (2018) DGM tutorial.

we have no easy way of knowing what latent variable corresponds to what data point; on the other hand, $q(x)$ is the correct distribution over the data, and $q_\phi(z \mid x)$ tells us what latent variable corresponds to a data point, but we have no easy way of generating data using this description of the distribution.

Now suppose that these models agree, i.e. $p_\theta(x, z) = q_\phi(x, z)$. In that case we have a correct description of the data distribution that allows us to generate new data points and that allows us to relate the latent variable to the data in both directions.

To achieve this we measure the difference between the two distributions using the Kullback-Leibler divergence and jointly train θ and ϕ using gradient descent and backpropagation to minimize this divergence:

$$\arg\min_{\theta,\phi} D_{KL}(q_\phi(x, z) \,||\, p_\theta(x, z)).$$

We rewrite this divergence as

$$\begin{aligned}
D_{KL}(q_\phi(x, z) \,||\, p_\theta(x, z)) &= \int_{X\times Z} \log\left(\frac{q_\phi(x, z)}{p_\theta(x, z)}\right) q_\phi(x, z)\mathrm{d}x\,\mathrm{d}z \\
&= \int_{X\times Z} \log\left(\frac{q(x)q_\phi(z \mid x)}{p_\theta(z)p_\theta(x \mid z)}\right) q(x)q_\phi(z \mid x)\mathrm{d}x\,\mathrm{d}z \\
&= \int_X \log(q(x))q(x)\mathrm{d}x \qquad (4) \\
&\quad + \int_X \int_Z \log\left(\frac{q_\phi(z \mid x)}{p_\theta(z)}\right) q_\phi(z \mid x)\mathrm{d}z\, q(x)\mathrm{d}x \qquad (5) \\
&\quad - \int_X \int_Z \log(p_\theta(x \mid z))q_\phi(z \mid x)\mathrm{d}z\, q(x)\mathrm{d}x. \qquad (6)
\end{aligned}$$

Here the first term, (4), is the negative (differential) entropy of the true distribution. Since this term contains no parameters, its gradients in training are 0 and we can ignore it for our minimization

problem. This leaves us with (5) and (6) making our loss function

$$L_{VAE} = \mathbb{E}_{q(x)} \left[D_{KL}(q_\phi(z \mid x) \mid\mid p_\theta(z)) \right] \tag{7}$$

$$- \mathbb{E}_{q(x)} \left[\mathbb{E}_{q_\phi(z|x)} \big(\log p_\theta(x \mid z) \big) \right], \tag{8}$$

which is the earlier variational lower bound with its sign swapped — i.e. both perspectives lead to the same optimization objective.

As in Section 6.2, we can approximate the expectations with respect to $q(x)$ by using a random batch of data points, say $x^{(1)}, \ldots, x^{(m)}$. We approximate as

$$L_{VAE} = \frac{1}{m} \sum_{i=1}^{m} D_{KL}(q_\phi(z \mid x = x^{(i)}) \mid\mid p_\theta(z)) \tag{9}$$

$$- \frac{1}{m} \sum_{i=1}^{m} \mathbb{E}_{q_\phi(z|x=x^{(i)})} \big(\log p_\theta(x = x^{(i)} \mid z) \big), \tag{10}$$

We then still have two integrals to either compute or approximate: the expectation in (10) and the Kullback-Leibler divergence in (9). In some cases, most notably when both $q_\phi(z \mid x)$ and $p_\theta(z)$ are multivariate normal distributions, we can use an exact formula for the Kullback-Leibler divergence. Otherwise we can approximate it in the same way as we approximate the expectation in (10): by sampling from $q_\phi(z \mid x = x^{(i)})$.

In general, if we have some probability distribution $\rho(y)$ and an integrable function f, we can approximate

$$\mathbb{E}_{\rho(y)}[f(y)] \tag{11}$$

by sampling a number of independent drawings $y_1, \ldots, y_l \sim \rho(y)$, and computing

$$\mathbb{E}_{\rho(y)}[f(y)] \approx \frac{1}{l} \sum_{j=1}^{l} f(y_j).$$

Due to the law of large numbers, if l is large enough this will likely be a good approximation. This way of approximating integrals is a basic example of a *Monte Carlo method*.

Both the Kullback-Leibler divergence from (9) and the expected value in (10) are integrals like (11) where $\rho = q_\phi(z \mid x = x^{(i)})$. So if for very $x^{(i)}$, we can draw $z_1^{(i)}, \ldots, z_l^{(i)} \sim q_\phi(z \mid x = x^{(i)})$ independently, then we can approximate (9) and (10) as

$$(9) = \frac{1}{m \cdot l} \sum_{i=1}^{m} \sum_{j=1}^{l} \log(q_\phi(z = z_j^{(i)} \mid x = x^{(i)})) - \log(p_\theta(z = z_j^{(i)}))$$

$$(10) = \frac{1}{m \cdot l} \sum_{i=1}^{m} \sum_{j=1}^{l} \log(p_\theta(x = x^{(i)} \mid z = z_j^{(i)})).$$

In practice, l is usually chosen as $l = 1$.

One difficulty we have to address is that to be able to train p_θ and q_ϕ jointly using backpropagation and gradient-descent, we need both of our approximations[5]to be differentiable with respect to ϕ.

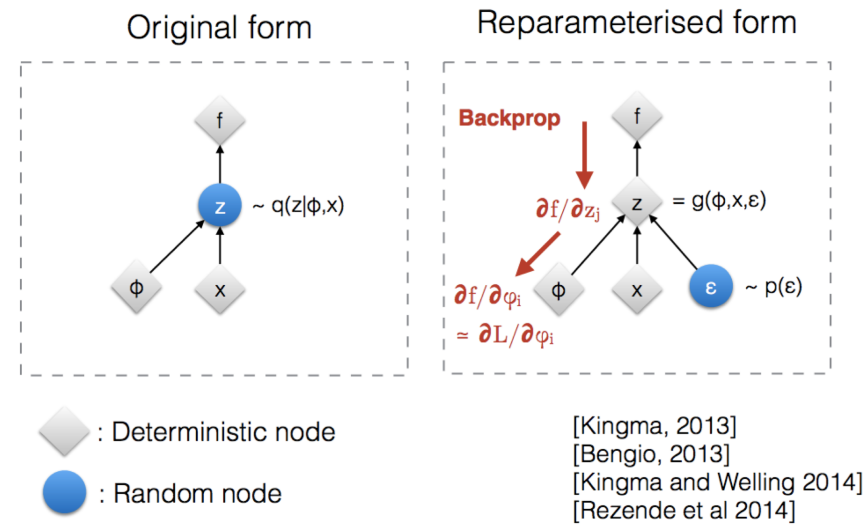


Figure 19: The reparameterization trick

More specifically, we want our samples $z_j^{(i)}$ to be differentiable with respect to ϕ. In order to do this, we can use the reparameterization trick.

The *reparameterization trick* consists of drawing a stochastic variable ϵ from some fixed distribution, and applying a differentiable[6] function g_ϕ such that

$$g_\phi(\epsilon, x^{(i)}) \sim q_\phi(z \mid x = x^{(i)}).$$

For example, if $q_\phi(z \mid x = x^{(i)}) = \mathcal{N}(\mu, \sigma^2)$, then if $\epsilon \sim (0, 1)$, we have $\mu + \sigma\epsilon \sim q_\phi(z \mid x = x^{(i)})$. Using this trick we can draw $z_j^{(i)}$ as

$$z_j^{(i)} = g_\phi(\epsilon_j^{(i)}, x^{(i)})$$

where $\epsilon_j^{(i)}$ are independent samples from the same fixed distribution. The original paper on the VAE mentions three basic approaches to finding a transformation g_ϕ:

1. For any location-scale family of distributions, such as Gaussian distributions, we can sample ϵ from the distribution with location=0 and scale = 1, and set $g_\phi(\epsilon, x) = \text{location}_\phi(x) + \text{scale}_\phi(x)\epsilon$.

2. If the distribution has a tractable inverse cumulative distribution function, we can let $\epsilon \sim U(0, \mathbf{1})$ (where $\mathbf{1} = (1, \ldots, 1)$), and let g_ϕ be the inverse CDF.

3. Many distributions can be drawn from by drawing from a different distribution and applying a standard transformation, e.g. a log-normal distribution can be drawn from by drawing from a normal distribution and exponentiating the result.

[5]Really it's not just that we want the approximations to be differentiable with respect to ϕ, but also that we want the resulting derivative is a good approximation of the derivative of the terms we are approximating.

[6]With respect to ϕ

Figure 20: The MNIST dataset

Let us look at a more concrete example. Suppose we want to use a VAE with d-dimensional latent space to develop a generative model for the MNIST dataset in Figure 20. We can choose the following distributions:

$$\begin{aligned} q_\phi(z \mid x) &= \mathcal{N}\left(\mu_\phi(x), \mathrm{diag}(\sigma_\phi^2(x))\right), \\ p_\theta(z) &= \mathcal{N}(0, I_{\mathbb{R}^d}), \\ p_\theta(x \mid z) &= \mathcal{N}\left(m_\theta(z), s\, I\right), \end{aligned}$$

where μ_ϕ and σ_ϕ are neural networks[7]with parameters ϕ, and m_θ is a neural network with parameters θ, and where s is some fixed number. To train this we can take a batch of data points $x^{(1)}, \ldots, x^{(N)}$ and for each data point compute $\mu_\phi(x^{(i)})$ and $\sigma_\phi(x^{(i)})$. Next we can, for each data point, draw a random vector $\epsilon^{(i)} \sim \mathcal{N}(0, I)$ to use in the parameterization trick, as shown in Figure 21. The resulting value for the latent variables then becomes

$$z^{(i)} = \mu_\phi(x^{(i)}) + \sigma_\phi(x^{(i)}) \cdot \epsilon^{(i)},$$

where the multiplication is element-wise. We can compute the first loss term, (7), using

$$\begin{aligned} D_{KL}(q_\phi(z \mid x = x^{(i)}) \mid\mid p_\theta(z)) &= D_{KL}\left(\mathcal{N}(\mu_\phi(x^{(i)}), \mathrm{diag}(\sigma_\phi^2(x^{(i)}))) \mid\mid \mathcal{N}(0, I)\right) \\ &= \frac{1}{2}\sum_{j=1}^{d}\left(\sigma_\phi^2(x^{(i)})_j + \mu_\phi(x^{(i)})_j - \log(\sigma_\phi^2(x^{(i)})_j) - 1\right), \end{aligned}$$

and approximate the second, (8), using a one sample Monte-Carlo approximation and using

$$-\log p_\theta(x = x^{(i)} \mid z^{(i)}) = C + \frac{1}{2s}||x^{(i)} - \mu_\theta(z^{(i)})||^2$$

[7]For σ_ϕ we typically use an exponential function as the final activation to ensure we get a positive number.

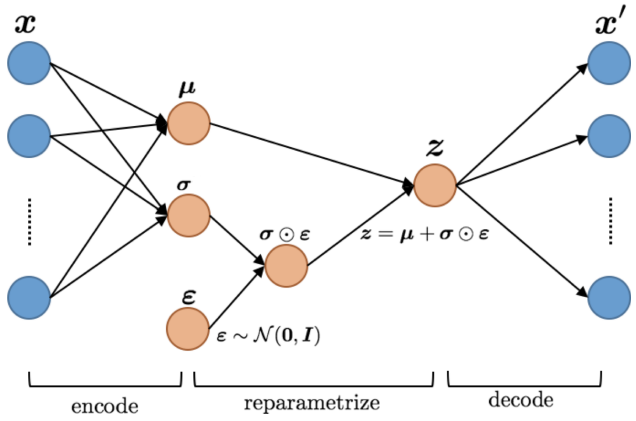


Figure 21: Reparameterization trick with Gaussian distributions with diagonal covariance.

where C is a constant (and thus of no influence on the training). This loss term is usually referred to as the *Reconstruction loss*. We could also have used a different probability distribution. For example, if we had modelled $p(x \mid z)$ as a bunch of independent Bernoulli random variables with success probabilities based on z, we would have gotten binary cross-entropy as the reconstruction loss.

Note that if we ignore the first loss term, (7), and make $q_\phi(z \mid x)$ deterministic (e.g. setting $\sigma_\phi = 0$), this model is simply an auto-encoder as discussed in Chapter 3. Let us look at the full model from this perspective too. If we have a regular auto-encoder, it is generally very hard to, without encoding real data, come up with codes that correspond to realistic data. The way in which data and code relate to each other can often be very irregular, and only small portions of the space of possible codes correspond to realistic data. By making the encoder stochastic with a distribution that covers a portion of the latent space with relatively high probability, we ensure that all points in the latent space that are close to $\mu_\phi(x)$ for some x will correspond to realistic data.

This however is not yet enough to ensure that we can use the model to generate new, realistic looking data. The encoder might simply learn to make $\sigma_\phi(x)$ very small, and to send $\mu_\phi(x)$ for different x very far apart (compared to $\sigma_\phi(x)$). The Kullback-Leibler divergence loss on the latent space prevents this, forcing the encoder to cover the latent space as similar to a normal distribution as it can without seriously sacrificing the quality of reconstruction. This makes that we can use the VAE to generate new data points, and even interpolate between two data points, as shown in Figure 22.

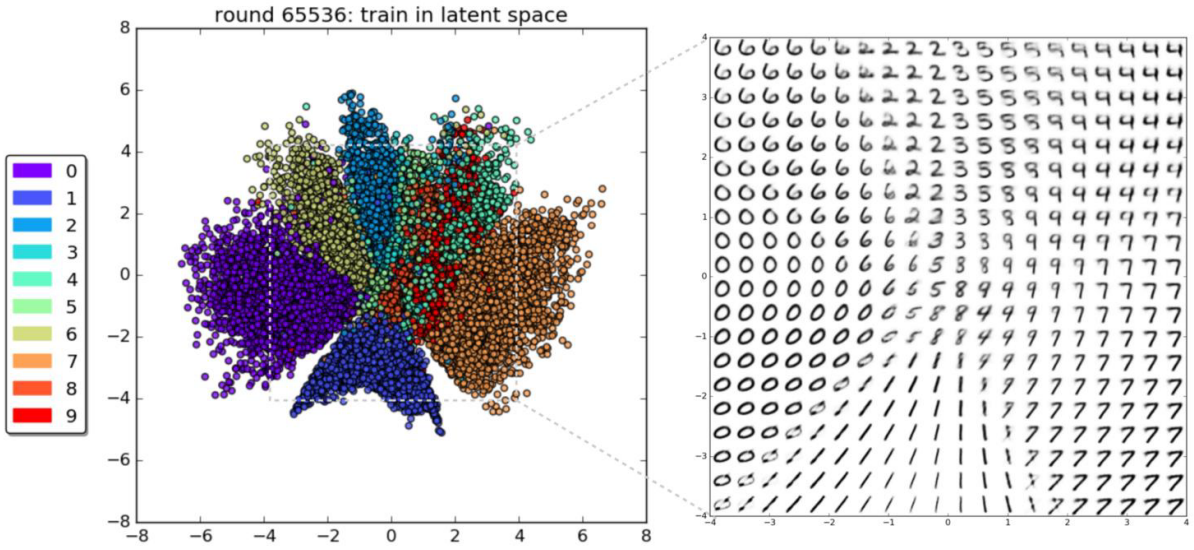


Figure 22: The 2-dimensional latent space of a VAE trained on MNIST

6.7 Generative Adversarial Networks

The main idea behind the Generative Adversarial Networks (GAN) is that the performance metric against which we train the generative model should be learned itself so as to prevent over-fitting to a specific loss function, and because no simple loss function really accurately captures how humans sees e.g. whether two pictures look similar. This is done by having a generative model G and a discriminative model D which we train jointly to try to achieve[8]

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim P(x)}[\log(D(x))] + \mathbb{E}_{z\sim P_z(z)}[\log(1-D(G(z)))]. \tag{12}$$

In words, we try to train a discriminator D to be able to decide whether its input comes from the data set (maximizing the first term) or is generated (maximizing the second term). In the meanwhile we try to train a generator G to fool the discriminator (minimizing the second term by picking the best G). In terms of loss functions this comes down to

$$\begin{aligned}\text{Loss}_D &= -\mathbb{E}_{x\sim P(x)}[\log(D(x))] - \mathbb{E}_{z\sim P_z(z)}[\log(1-D(G(z)))],\\ \text{Loss}_G &= -\text{Loss}_D\end{aligned}$$

or equivalently for gradient descent

$$\text{Loss}_{G,\,equivalent} = \mathbb{E}_{z\sim P_z(z)}[\log(1-D(G(z)))].$$

The generative model G learns the distribution of the data in an implicit way. We sample from it by sampling a noise variable z from some noise distribution $p(z)$ which can be e.g. a uniform distribution, or a Gaussian distribution, and by applying a function G to z:

$$\begin{aligned}z &\sim p(z)\\ x &= G(z).\end{aligned}$$

Here G is a neural network, and we optimize (12) by training the parameters of the network. The discriminator D is a neural network too, and again, optimizing is done by training its parameters.

We can train the model by doing the following every training step:

1. Take a batch of data points $x^{(i)}$ and sample a batch of noise variables $z^{(i)}$ and use them to compute the discriminator loss and gradients
2. Update the discriminator using those gradients
3. (Optionally) repeat step 1 and 2 a number of times
4. Take a batch of noise variables and use them to compute the generator loss
5. Update the generator using those gradients

[8] This is the learning objective for the original GAN. Since then a number of variations on this have been developed, some of which we will discuss in short.

This adversarial training is often compared to real world examples such as criminals innovating to avoid getting caught by police, and police innovating to become better at catching criminals. Think of the generator as some art forger trying to create pictures such that they pass for creations by some great artist, and think of the discriminator as some expert trying to distinguish the forgeries from the masterpieces.

The objective in eq. (12) is very useful for theoretical analysis of the GAN, but has a major drawback: whenever the discriminator manages to distinguish well between real data and generated examples, the gradient of the loss function for the generator is rather flat, making it hard for the generator to learn. To overcome this a heuristic loss is often used for the generator instead. This heuristic loss is given by

$$\text{Loss}_{G,\,heuristic} = -\mathbb{E}_z \log D(G(z)). \tag{13}$$

Instead of trying to minimize the probability of the discriminator being right, this loss encourages the generator to maximize the probability of the discriminator being wrong.

Training a GAN can be difficult for several reasons. Training can often be unstable where the discriminator and the generator simply keep undoing each others progress by focusing on irrelevant details, or where the generator is unable to learn at all due to the discriminator being too good. The latter problem can be reduced by using the heuristic loss for the generator given in eq. (13). Another trick for stabilizing training is *one-sided label smoothing*. This means that instead of

$$\begin{aligned}\text{Loss}_D &= \sum_{x^{(i)}} -\log(D(x^{(i)})) + \sum_{z^{(i)}} -\log(1 - D(G(z^{(i)}))) \\ &= \sum_{x^{(i)}} H_{bin}(1, D(x^{(i)})) + \sum_{z^{(i)}} H_{bin}(0, D(G(z^{(i)})))\end{aligned}$$

we replace the labels for the true samples by slightly lower values:

$$\text{Loss}_{D,\,smooth} = \sum_{x^{(i)}} H_{bin}(1 - \epsilon, D(x^{(i)})) + \sum_{z^{(i)}} H_{bin}(0, D(G(z^{(i)})))$$

where H_{bin} denotes binary cross-entropy

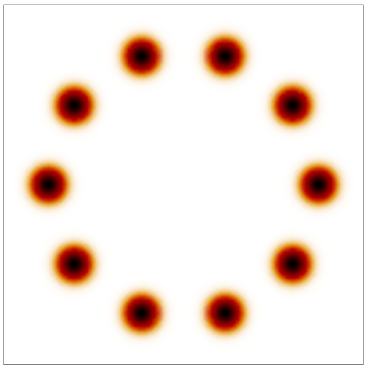
$$H_{bin}(y, x) = y \log(x) + (1 - y) \log(1 - x),$$

and $0 < \epsilon \ll 1$ can either be fixed or random. This is done to prevent overly confident classification through interpolation by the discriminator.

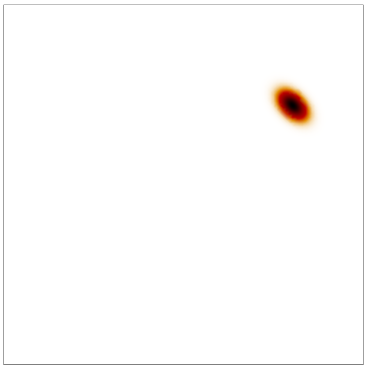
Another recurring problem with the training of GANs is that of mode collapse. This is where the true distribution as a number of "modes" or most likely outcomes and the generator only produces examples corresponding to one of the modes. Think e.g. of images from different classes where the different modes correspond to the different classes, the generator might only create examples from one class, or of a few of the classes, ignoring the others. This is illustrated in Figure 23.

In theory optimizing (12) is equivalent to minimizing the Jensen-Shannon divergence between the learned distribution and the real distribution.[9] By modifying (12) or putting some extra restrictions on the discriminator, the corresponding divergence can be changed to various other notions of

[9] Due to the need to represent the functions by neural networks and having to learn parameters instead of functions directly, this equivalence doesn't really fully hold. It is still useful for theoretical analysis of GANs.



(a) The target distribution that we want to learn has ten modes.



(b) The GAN only learns one of the modes.

Figure 23: Mode collapse in a GAN model

distance. In attempts to address the problems with training GANs various such modifications have been proposed, including WGAN (Wasserstein-1 or Earth-Mover metric) and f-GAN (any f-divergence including Jensen-Shannon and Kullback-Leibler, the latter of which theoretically would allow a GAN to be used for likelihood maximization).

Finally, if labels are present for (part of) the data set, they can be used in various ways to significantly improve the quality of the generated samples and to stabilize the training process and prevent mode collapse.

Figure 24: An example from `thispersondoesnotexist.com`, generated by StyleGAN2. A link to the corresponding article can be found on that website.

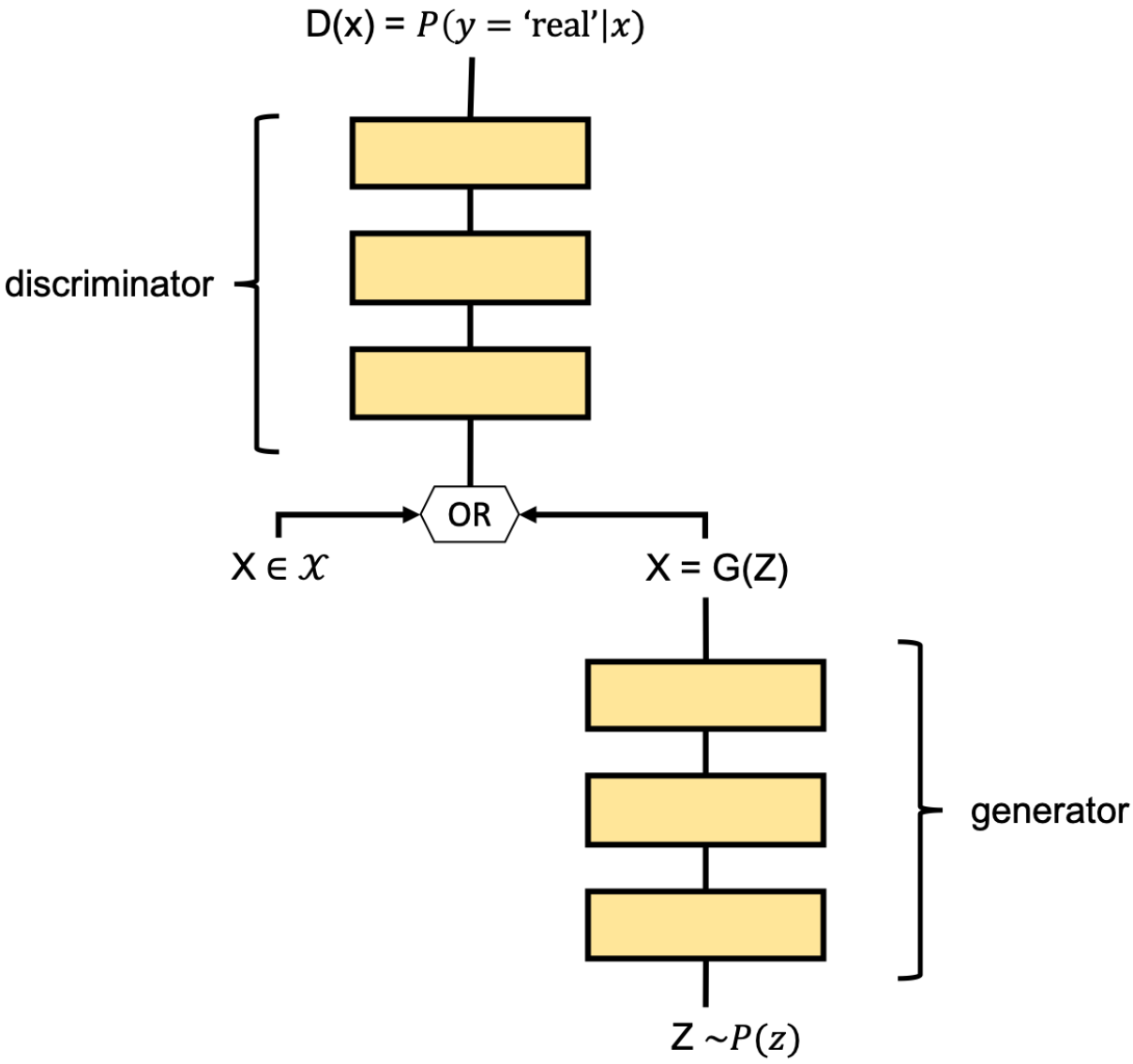


Figure 25: A visual representation of a Generative Adversarial Network

Appendices

.1 Short recap of probability theory

Throughout this chapter we will be working with a number of concepts from probability theory. For those who haven't worked with probability theory for a long time, we give a short description of these concepts. Moreover, we will talk a bit about the notation we use throughout this chapter. You do not need to know all of the technical details behind these concepts to understand the material in this chapter, so do not worry if you don't understand some of these technicalities.

Probability distributions and random variables

The most commonly used framework for probability theory is built upon the concept of a probability space and a probability measure. The details of these are often not needed for practicing probability theory, but a very basic understanding can help you understand how things relate to each other.

In this framework we have a set of possible outcomes called the **sample space**, often denoted by Ω. Think of this set as follows: if we run an experiment where we throw seven fair six-sided dice in a row, this space would consist of all 6-tuples of possible outcomes:

$$\Omega = \{\omega = (\omega_1, \ldots, \omega_7) \mid \omega_1, \ldots, \omega_7 \in \{1, \ldots, 6\}\}$$

On top of this, we have a set of **events**, often denoted by $\mathcal{F} \subset 2^\Omega$, which is a set of subsets of Ω. One event in our experiment with six dice might be that the second die was a three. The corresponding element of $\mathcal{F}$ would be

$$\{\omega \in \Omega \mid \omega_2 = 3\} \in \mathcal{F}.$$

The final ingredient of a probability space is the **probability measure**. This probability measure tells us how likely an event is, and is a function from $\mathcal{F}$ to the interval $[0, 1]$, in this case

$$P : \mathcal{F} \to [0, 1]$$
$$P : A \mapsto \frac{\#A}{6^7},$$

where $\#A$ denotes the number of elements in A. E.g. for the event $A = \{\omega \in \Omega \mid \omega_2 = 3\}$ we have 6^6 different outcomes that belong to A, so the probability of A is $P(A) = 6^6/6^7 = 1/6$, which is precisely what we expect intuitively.

The sample space, Ω, doesn't have to be finite, or even countable. Say for example its raining and we have a square of 1×1 meter on the ground. If we look at just the next rain drop, we could say that our sample space is just a square

$$\Omega = [0, 1] \times [0, 1]$$

and our events are regions of the square[10]where the rain drop might fall

$$\mathcal{F} \subset 2^\Omega,$$

with the probability of an event being the area of that region

$$P(A) = \text{area}(A).$$

This model of the situation is manageable if we want to model just one rain drop, but if we want to do more — e.g. multiple rain drops and the times in between them — it easily becomes impractical. In practice, the **probability space** $(\Omega, \mathcal{F}, P)$ is kept abstract[11], and all calculations are done using *random variables*.

A **random variable**, X is a measurable function

$$X : \Omega \to \mathbb{R},$$

or more generally

$$\mathbf{X} : \Omega \to \mathbb{R}^n.$$

Such a random variable comes with its own events of the form[12]

$$\{X \in A\} = X^{-1}(A) = \{\omega \in \Omega \mid X(\omega) \in A\}$$

and with its own probability measure on its range:

$$P_X = P_{\#}X \ : \ A \mapsto P(X^{-1}(A)).$$

This probability measure is called the **distribution**, or the **law** of X.

In this course all our random variables are either discrete, where the range of X is some countable discrete set — e.g. $\{1, \ldots, 10\}$, $\mathbb{N}$, or $\mathbb{Z}$ — or continuous, where the range of X is some interval of $\mathbb{R}$ (or some Cartesian product thereof).

In the case of a discrete random variable, we can characterize the whole random variable by the probability of it taking specific values, i.e. we can describe X using the function

$$f_X : R \to [0, 1] : r \mapsto P(X = r).$$

This function, f_X is called the **probability mass function (PMF)** of X.

For a distribution to be continuous, there is an additional requirement besides what the range of X is: the distribution must have a **probability density function**[13] **(PDF)**, $f_X \geq 0$, so that

$$P(X \in A) = \int_A f_X(x) \mathrm{d}x.$$

Throughout this course all PDFs will be assumed to be continuous and strictly positive.

When the range of X is some ordered set such as $\mathbb{R}$, $\mathbb{N}$, $\mathbb{Z}$, or $[0, 1]$, we also have a **cumulative distribution function (CDF)** given by

$$F_X : x \mapsto P(X \leq x).$$

For continuous random variables, the PDF is the derivative of the CDF.

[10] Whether $\mathcal{F}$ can be all of 2^Ω in this case depends on your axioms of set theory. For the most commonly used axioms the answer is *no*. This however is beyond the scope of this course.

[11] Such an abstract probability space needs to satisfy some conditions: $\mathcal{F}$ needs to contain the full sample space, it needs to be closed under taking complements, and under taking countable unions of its elements. The probability measure P must be a function on $\mathcal{F}$ such that $P(\Omega)$=1, and such that for any *countable* collection of *disjoint* elements of $\mathcal{F}$, $(A_i)_{i=1}^\infty$, we have $P(\bigcup_{i=1}^\infty A_i) = \sum_{i=1}^\infty P(A_i)$.

[12] Notation like $X \in A$ or $X \leq x$ is often used to denote the corresponding events $\{\omega \in \Omega \mid X(\omega) \in A\}$ and $\{\omega \in \Omega \mid X(\omega) \leq x\}$. This way direct references to Ω are seldom needed.

[13] With respect to the Lebesgue measure. You can have densities with respect to other measures too, e.g. a PMF is a density with respect to the counting measure supported on the range of the random variable, but that's besides the point.

Conditional probability, joint distributions, and independence.

When we have multiple random variables, $X_1, \ldots, X_n$ we can look at them together. Suppose two people, person 1 and person 2, are living together. Say that on a given day the probability of one of them having the flue is around 2%, i.e. $P(X_i = 1) = .02$, where $X_i = 1$ means person i has the flue, and $X_i = 0$ means the student doesn't have it. Because the students have a lot of contact with each other, if one student has the flue, others are likely to get it as well, so the probability that student 1 has the flue given the fact that student 2 has it, is much larger than the a-priori probability that student 1 has it without extra information. This leads us to conditional probability and to joint distributions for random variables.

If we have two events, A and B, the probability of A **conditioned** on B is given by

$$P(A \mid B) = \frac{P(A \cap B)}{B}.$$

In the example above, the contagiousness of the flue makes that

$$P(X_1 = 1 \mid X_2 = 1) > P(X_1 = 1).$$

In general $P(A \mid B)$ can be larger than, less than, or equal to $P(A)$. The case $P(A \mid B) = P(A)$ is special: if

$$P(A \mid B) = A,$$

or equivalently[14]

$$P(A \cap B) = P(A) \cdot P(B),$$

we say the events A and B are **independent**, denoted by $A \perp B$. Two random variables, X and Y, are called independent, denoted by $X \perp Y$ if all their associated events are independent mutually independent, i.e.

$$X \perp Y \iff \forall_{A,B \text{ measurable}} \ : \ X^{-1}(A) \perp Y^{-1}(B).$$

An important theorem for dealing with conditional probabilities is **Bayes' theorem**, which says that for any two events, A and B, we have

$$P(B \mid A) = \frac{P(A \mid B)P(B)}{P(A)}.$$

Especially when two or more random variables are not independent, it is interesting to look at their joint distribution. For simplicity we will look at continuous random variables taking values in $\mathbb{R}$, but all of this can easily be generalized to other cases. We can view a collection of random variables $X_1, \ldots, X_n$ as function

$$\mathbf{X} \ : \ \Omega \to \mathbb{R}^n \ : \ \omega \mapsto (X_1(\omega), \ldots, X_n(\omega)).$$

[14] Here and in the definition of conditional probability we assume that $P(B) > 0$. Conditioning on events of probability 0 is tricky, and we will only do so when the events come from continuous random variables with continuous and nowhere vanishing densities.

This **random vector** again has associated events of the form

$$\{\mathbf{X} \in A\} = \mathbf{X}^{-1}(A) = \bigcap_{i=1}^{n} X_i^{-1}(A),$$

and an associated probability measure on $\mathbb{R}$ given by

$$P_{\mathbf{X}} = \mathbf{X}_{\#}P \ : \ A \mapsto P(\mathbf{X}^{-1}(A)).$$

This measure is called the joint distribution of the random variables. Throughout this chapter, all our joint distributions will be assumed to be continuous (at least if needed) so that this joint distribution again has a **joint density** $f_{\mathbf{X}} : \mathbb{R}^n \to [0, 1]$ such that

$$P(\mathbf{X} \in A) = \int_A f_{\mathbf{X}}(x_1, \ldots, x_n)\mathrm{d}x_1 \cdots \mathrm{d}x_n.$$

From the joint density we obtain the density of a single random variable through the process of **marginalization**, for example if we have two random variables X and Y with joint density $f_{X,Y}$, then:

$$f_X(x) = \int_{\mathbb{R}} f_{X,Y}(x, y)\mathrm{d}y.$$

From the joint density and marginal densities we can get the **conditional distribution**. If we have two random variables X and Y with joint density $f_{X,Y}$, then the conditional distribution is given by

$$f_{Y|X=x}(y) = f_{Y|X}(y \mid x) = \frac{f_{X,Y}(x, y)}{f_X(x)}.$$

Note that this means we can **factorize** the density of the joint distribution as

$$\begin{aligned} f_{X,Y}(x, y) &= f_X(x) f_{Y|X}(y \mid x) \\ &= f_Y(y) f_{X|Y}(x \mid y). \end{aligned}$$

Expectation

When we are dealing with random phenomena, we often want to make predictions, or comparisons. Although we usually can't make claims about the precise outcomes, we can often say something about the expected outcome, or the average outcome if we have a large number of independent and identically distributed random variables. For example, when throwing fair six sided dice, we don't know in advance what the outcome will be, but we do expect that if we throw a lot of these dice, the average outcome will be around 3.5. This idea that we can expect a certain average outcome is formalized by the **expected value**, and the idea that we will obtain this value if we do a lot of independent experiments is formalized by the **Law of Large Numbers**. The expected value of a discrete random variable X is simply a weighted average of the possible outcomes where the weights are the probabilities of the outcomes — i.e. if the set of possible outcomes is S, and X has probability mass function f_X

$$\mathbb{E}[X] = \sum_{s \in S} s \cdot f_X(s).$$

We generalize this to continuous random variables by replacing the sum by an integral, and the mass function by a density:

$$\mathbb{E}[X] = \int_S s \cdot f_X(s) \mathrm{d}s,$$

where again S is the range of the random variable. Whenever this integral, or in case of discrete random variables the sum, is not well-defined, we say that the random variable does not have an expected value.

Sometimes we want to know what the expected value of some function of a random variable instead of the random variable itself. For this we have two useful results: the first is the **law of the unconscious statistician** which tells us that for a random variable X and a function g, we have

$$\mathbb{E}[g(X)] = \sum_{s \in S} g(s) \cdot f_X(s) \qquad \text{if } X \text{ is discrete,}$$

and

$$\mathbb{E}[g(X)] = \int_S g(s) \cdot f_X(s) \mathrm{d}s \qquad \text{if } X \text{ is continuous,}$$

provided that the expressions are well-defined.

The second useful result is **Jensen's inequality**, which tells us that if a function g is **convex** we have

$$g(\mathbb{E}[X]) \leq \mathbb{E}[g(X)],$$

and consequently if g is **concave** we get

$$g(\mathbb{E}[X]) \geq \mathbb{E}[g(X)].$$

Whenever there might be confusion against what random variable, or what probability distribution, we are taking the expected value, we will indicate the distribution in subscript.