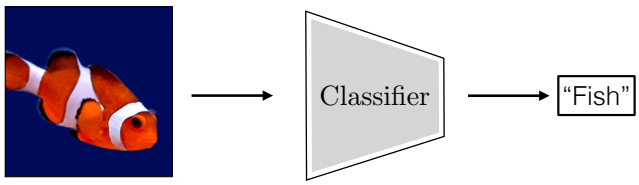
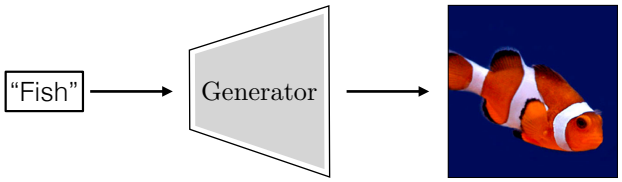
Chapter 22

Generative models

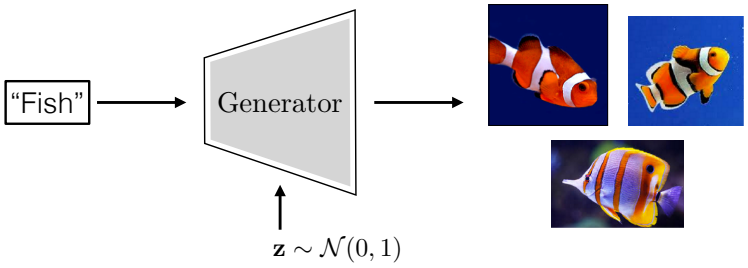
Generative models perform (or describe) the synthesis of data. Recall the image classifier from Chapter 11:



A generative model does the opposite:

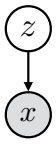


Whereas an image classifier is a function $f : \mathcal{X} \to \mathcal{Y}$, a generative model is a function in the opposite direction $g : \mathcal{Y} \to \mathcal{X}$. Things are a bit different in this direction. The function f is **many-to-one**: there are many images that all should be given the same label "fish". The function g, on the other hand, is **one-to-many**: there are many possible outputs for any given input. Generative models handle this ambiguity by making g a **stochastic function**. One way to make a function stochastic is to make it a deterministic function of a stochastic input: $g : \mathcal{Z} \times \mathcal{Y} \to \mathcal{X}$, with $z \sim p(\mathbf{z})$, $\mathbf{z} \in \mathcal{Z}$. Often, we use Gaussian noise as the stochastic input: $p(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{1})$:



Sometimes we wish to simply make up data from scratch – in fact this is the canonical setting in which generative models are often studied. To do so, we can simply drop the

Corresponds to this graphical model:



dependency on y. This yields a procedure for making data sampled as follows:

$$\mathbf{z} \sim p(\mathbf{z}) \tag{22.1}$$

$$\mathbf{x} = g(\mathbf{z}) \tag{22.2}$$

In this form of generative modeling, our goal is to learn g such that the distribution of random variable $\mathbf{x}$ matches some target distribution.

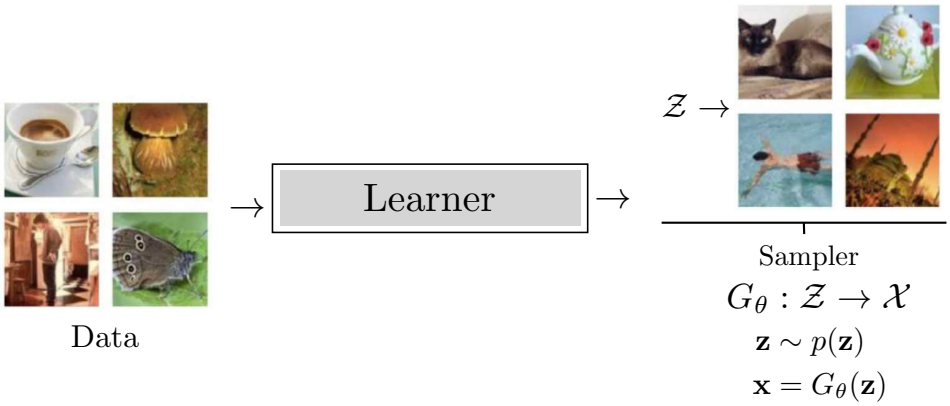
At first it may seem a silly goal. Why should we care to make up images from scratch? One reason is *content creation*. Suppose we are making a video game and we want to automatically generate a bunch of exciting levels for the player the explore. We would like a procedure for making up new levels from scratch. Such **procedural graphics** have been successfully used to generate random landscapes for game levels. Suppose we want to add a river to our landscape. We need to decide what path the river should take. A simple program for generating the path would be "walk an increment forward, flip a coin to decide whether to turn left or right, repeat."

This program relies on a sequence of random coin flips to generate the path of the river. In other words, the program took a sequence of flips as input, and converted this "noise" to a image of the path of the river.

The conversion process was written by hand. Generative models learn the program, by setting its parameters such that the output matches some target distribution, for example, we may wish the output to match the distribution of actual river paths in the wild.

22.1 Learning generative models

How can we learn to do this? If we are given data $\hat{\mathbf{x}} \sim p_{\texttt{data}}$, our goal is to learn G_θ from this data such that G_θ produces additional identically distributed samples, i.e. $\mathbf{x} \sim p_{\texttt{data}}$.[1]



Generative models that just produce a **sampler** $G_\theta : \mathcal{Z} \rightarrow \mathcal{X}$ are sometimes called **implicit generative models**, because they do not explicitly give us the likelihood (a.k.a. probability) of the samples they generate. Other generative models instead fit an explicit probability **density** to the input data, $p_\theta : \mathcal{X} \rightarrow [0, 1]$:

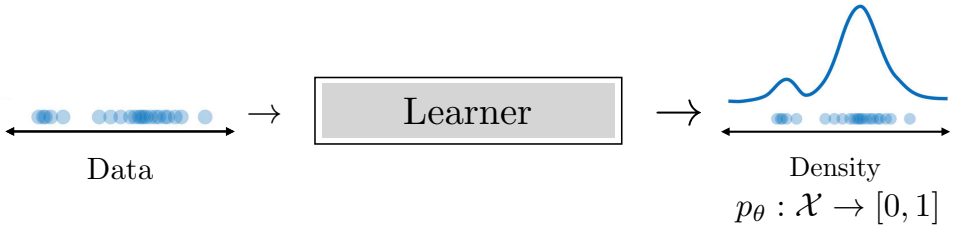
The objective of the learner for a density model is to output a density function p_θ that is as close as possible to $p_{\texttt{data}}$. How should we measure closeness? We need to define a **divergence** between two distributions, D, which yields the following optimization problem:

$$\arg\min_{p_\theta} D(p_\theta, p_{\texttt{data}}) \tag{22.3}$$

A problem is that we do not actually have the function $p_{\texttt{data}}$, we only have samples from this function, $\mathbf{x} \sim p_{\texttt{data}}$. Therefore, we need a divergence D that measures the "distance"[2]

[1] the figures on this page and the next are derived from: `http://introtodeeplearning.com/materials/2019_6S191_L4.pdf`

[2] The divergence need not be a proper distance *metric*, and often is not, e.g., it can be non-symmetric ($D(p, q) \neq D(q, p)$).



between p_θ and $p_{\texttt{data}}$ while only accessing samples from $p_{\texttt{data}}$. A common choice is to use the "forward" **KL-divergence**, which is defined as follows:

$$p^*_\theta = \arg\min_{p_\theta} \texttt{KL}(p_{\texttt{data}}, p_\theta) \tag{22.4}$$

$$= \arg\min_{p_\theta} \mathbb{E}_{\mathbf{x}\sim p_{\texttt{data}}}[\log \frac{p_\theta(\mathbf{x})}{p_{\texttt{data}}(\mathbf{x})}] \tag{22.5}$$

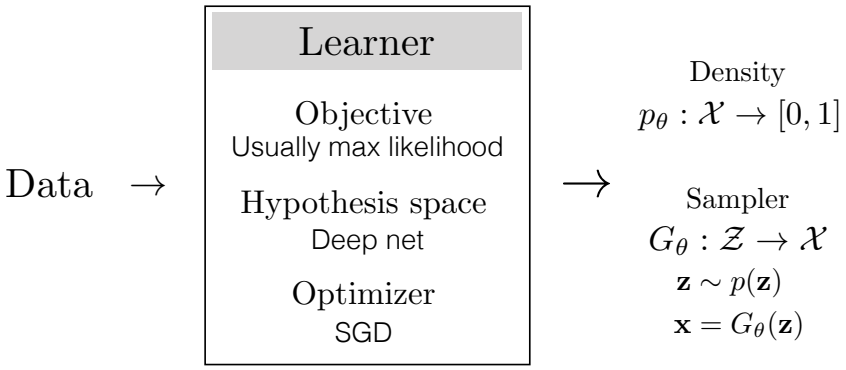
$$= \arg\min_{p_\theta} \mathbb{E}_{\mathbf{x}\sim p_{\texttt{data}}}[\log p_\theta(\mathbf{x})] - \mathbb{E}_{\mathbf{x}\sim p_{\texttt{data}}}[\log p_{\texttt{data}}(\mathbf{x})] \tag{22.6}$$

$$= \arg\min_{p_\theta} \mathbb{E}_{\mathbf{x}\sim p_{\texttt{data}}}[\log p_\theta(\mathbf{x})] \quad \triangleleft \text{dropped 2nd term since no dependence on } p_\theta \tag{22.7}$$

$$\approx \arg\min_{p_\theta} \frac{1}{N}\sum_{i=1}^{N} \log p_\theta(\mathbf{x}_i) \tag{22.8}$$

where the final line is an empirical estimate of the expectation by sampling over the training dataset $\{\mathbf{x}_i\}_{i=1}^N$. Equation 22.7 is the *expected log likelihood of the training data under the model's distribution*. Maximizing this objective is a form of **max likelihood learning**.

Some generative models give density functions, others give sampler functions, and still others give both. The kinds of **deep generative models** that are popular these days, have the following form:



22.2 Types of deep generative models [Section under construction]

Many generative models have the form described above. They map a base distribution over $\mathbf{z}$ to a transformed distribution over $\mathbf{x}$. The difference is mainly in the learning objective, which encourages this mapping to fit the data distribution.

Usually the objective is a form of max likelihood, so we need to calculate the likelihood function, $L(\mathbf{x}, \theta)$. In explicit density models, we directly learn the likelihood function: $L(\mathbf{x}, \theta) = p_\theta(\mathbf{x})$. Then we adjust the parameters to maximize this quantity.

In implicit models, things are a bit harder since we don't have an explicit likelihood function. Instead we learn a sampler G_θ. If the sampler is bijective, so that G^{-1} exists and can be computed, things are actually still easy. We can use the **change of variables** formula to compute the likelihood of data implied by our sampler:

$$L(\mathbf{x}, \theta) = p_z(G_\theta^{-1}(\mathbf{x}))|\det(\frac{\partial G_\theta^{-1}(\mathbf{x})}{\partial \mathbf{x}})|$$

where p_z is our prior (usually a Normal distribution or uniform).

Max likelihood learning corresponds to maximizing the log likelihood of the training data, $\{\mathbf{x}_i\}_{i=1}^N$, so here it looks like:

$$\arg\max_{G_\theta} \frac{1}{N}\sum_{i=1}^{N} \log p_z(G_\theta^{-1}(\mathbf{x}_i)) + \log|\det(\frac{\partial G_\theta^{-1}(\mathbf{x}_i)}{\partial \mathbf{x}_i})| \tag{22.9}$$

Normalizing flows are a family of generative models learned in this fashion.

If the mapping is not bijective, the change of variables formula does not apply. Instead, one option is to build up a density model, from our sampler, by putting a little blip of Gaussian probability around each image, $G_\theta(\mathbf{z}_i)$, generated by our sampler. Commonly, we augment G to also output the variance, σ, of this Gaussian, so we have G_θ^μ and G_θ^σ. This gives us an infinite mixture of Gaussians as our likelihood model: $L(\mathbf{x}, \theta) = \int_{\mathbf{z}} p_\theta(\mathbf{x}|\mathbf{z})p(\mathbf{z})d\mathbf{z}$, where $p_\theta(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mathbf{x}; G_\theta^\mu, G_\theta^\sigma)$.

This integral is intractable, so we turn to the calculus of variations of approximate it. The calculus of variations deals with finding functions that maximize some functional. A functional is just a function of functions. In variational inference, the functional we use is typically an integral. The function is a probability density. Saying we are using "variational inference" usually just means we are using a tractable density q to approximate an intractable target density p.

Here is how it works in **Variational Autoencoders** (**VAEs**). We want to approximate $p(\mathbf{x}) = \int_{\mathbf{z}} p_\theta(\mathbf{x}|\mathbf{z})p(\mathbf{z})d\mathbf{z}$ in a tractable manner. We use $\int_{\mathbf{z}} p_\theta(\mathbf{x}|\mathbf{z})p(\mathbf{z})d\mathbf{z} \approx \int_{\mathbf{z}} p_\theta(\mathbf{x}|\mathbf{z})q_\phi(\mathbf{z}|\mathbf{x})d\mathbf{z}$, for some well chosen q_ϕ. Which q_ϕ should we use? We'd like a q_ϕ such that $q_\phi(\mathbf{z}|\mathbf{x})$ places high density on exactly those $\mathbf{z}$'s that have high $p_\theta(\mathbf{x}|\mathbf{z})$. Then a few samples from q will suffice to be a good approximation to the full integral. We can jointly learn q_ϕ and p_θ to achieve this. A fuller treatment of how VAEs achieve this goal is given in [Doersch2016].

Other popular deep generative models include **Generative adversarial networks**, **Autoregressive models**, and **Energy-based models**.

22.3 Generative versus discrimatinative [Advanced topic]

Classically, a distinction was made between "discriminative models" and generative models, in the context of data classification problems. The former referred to models of $p(y|\mathbf{x})$, where y represented a *label* and $\mathbf{x}$ represented *data*. The latter were models of $p(y, \mathbf{x})$, which can be factored as $p(\mathbf{x}|y)p(y)$ (the idea is that $p(\mathbf{x}|y)$ is a model of how the data is generated). This distinction has become less useful in modern AI, since often it's hard to tell what is a *label* and what is *data* – generally, both the inputs and outputs to our models are high-dimensional structured objects.

These days, "generative models" usually simply refer to models of $p(\mathbf{x}|\mathbf{y})$ with two key properties: 1) $\mathbf{x}$ is high-dimensional (usually we think of it as "data"), 2) the dimensions of $\mathbf{x}$ are non-independent.

Bibliography

[Doersch2016] Doersch, Carl. 2016. Tutorial on Variational Autoencoders.