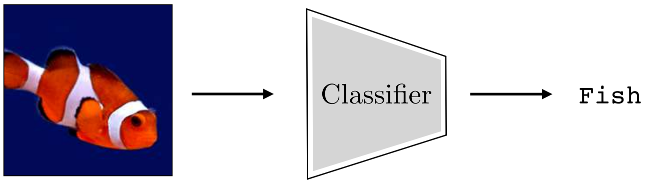
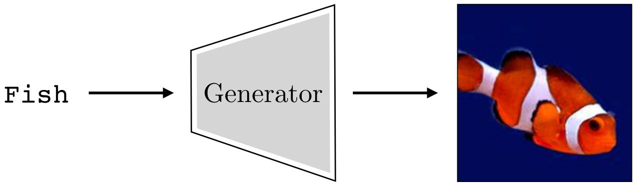
Chapter 1

Generative models

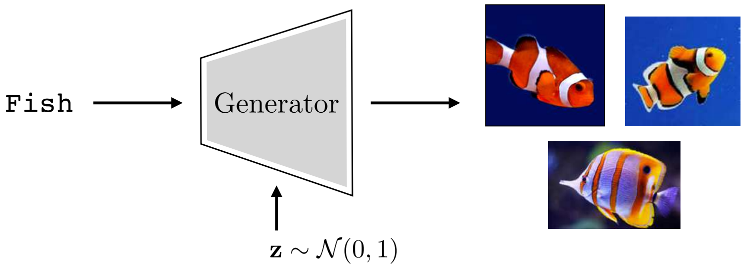
Generative models perform (or describe) the synthesis of data. Recall the image classifier from Chapter **??**:



A generative model does the opposite:

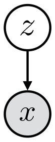


Whereas an image classifier is a function $f : \mathcal{X} \to \mathcal{Y}$, a generative model is a function in the opposite direction $g : \mathcal{Y} \to \mathcal{X}$. Things are a bit different in this direction. The function f is **many-to-one**: there are many images that all should be given the 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 $\mathbf{z} \sim p(\mathbf{z})$, $\mathbf{z} \in \mathcal{Z}$. 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 the input label y. This yields a procedure for making data sampled as:

$$\mathbf{z} \sim p(\mathbf{z}) \tag{1.1}$$

$$\hat{\mathbf{x}} = g(\mathbf{z}) \tag{1.2}$$

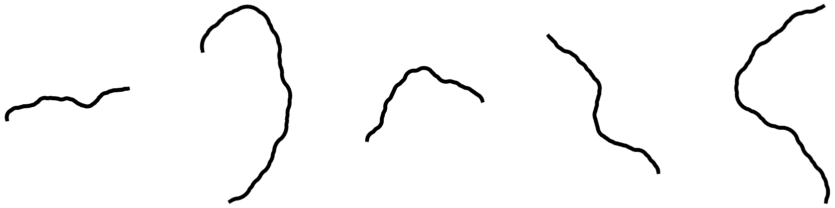
We call this an **unconditional generative model** because it is model of the unconditional distribution $p(\mathbf{x})$. Generally we will refer to unconditional generative models simply as "generative models" and use the term **conditional generative model** for a model of any conditional distribution $p(\mathbf{x}|\mathbf{y})$. Conditional generative models will be the focus of Chapter **??**; in the present chapter we will restrict our attention to unconditional models.

Why bother with (unconditional) generative models, which make up random synthetic data? At first this may seem a silly goal. Why should we care to make up images from scratch? One reason is *content creation*; we will see other reasons later, but content creation is a good starting point. 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 a landscape. We need to decide what path the river should take. A simple program for generating the path could be "walk an increment forward, flip a coin to decide whether to turn left or right, repeat." Here is that program in pseudocode:

Algorithm 1: Generative model of images of rivers

```
Input: Random vector of coin flips $\mathbf{z} = [z_0, \dots, z_N]$ with each $z_i \in \{0, 1\}$
Output: Picture of a river
start drawing at origin with heading = $90^\circ$
for $i = 1, \dots, N$ do
    extend line 1 unit in current heading direction
    if $z_i == 1$ then
        rotate heading $10^\circ$ to the right
    else
        rotate heading $10^\circ$ to the left
```

Here are a few rivers this program draws:



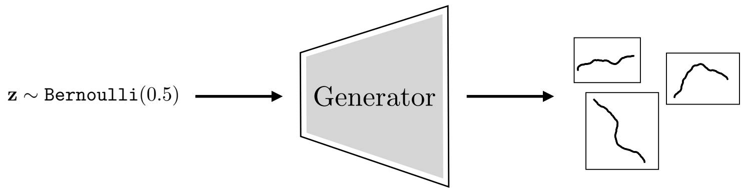
This program relies on a sequence of random coin flips to generate the path of the river. In other words, the program took a sequence of flips as input, and converted this "noise" to a image of the path of the river. It's exactly the same idea as we described above for generating images of fish, just this time the generator is a program that makes rivers:

This generator was written by hand. Next we will see generative models that *learn* the program that synthesizes data.

1.1 Learning generative models

1.1.1 Data generators

What if our model just memorizes all the training examples and generates random draws from this memory? This section focuses on the



How can we learn to synthesize images that look realistic? The machine learning way to do this is to start with a training set of *examples* of real images, $\{\mathbf{x}^{(i)}\}_{i=1}^N$. Recall that in supervised learning, an "example" was defined as an {`input`, `output`} pair; here things are actually no different, only the `input` happens to be the null set. We feed these examples to a learner, which spits out a **generator** function. Later, we may query the generator with random noise source $\mathbf{z}$ to produce novel outputs, a process called sampling from the model:

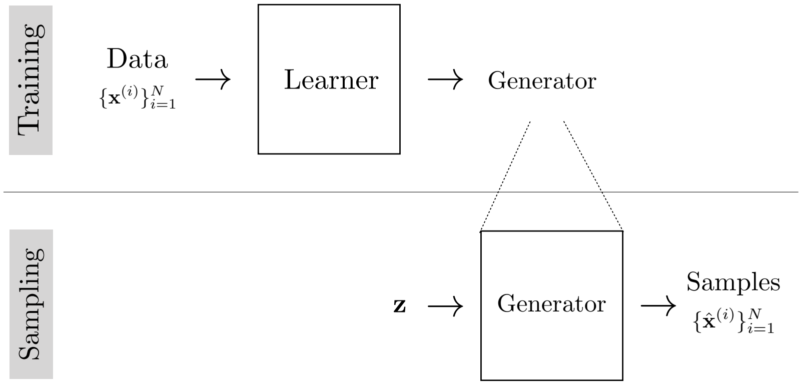

Training
Data
$\{\mathbf{x}^{(i)}\}_{i=1}^N$
Learner
Generator
Sampling
$\mathbf{z}$
Generator
Samples
$\{\hat{\mathbf{x}}^{(i)}\}_{i=1}^N$

Figure 1.1: Learning and using a generator.

The objective of the learner is to create a generator that produces synthetic **fake** data, $\{\hat{\mathbf{x}}^{(i)}\}_{i=1}^N$, that "looks like" the **real** data, $\{\mathbf{x}^{(i)}\}_{i=1}^N$. There are a lot of ways of define "looks like" and they each result in a different kind of generative model. Two examples are:

1. Fake data looks like real data if matches the real data in terms of certain marginal statistics, e.g., it has the same mean color as real photos, or the same color variance, or the same edge statistics, etc.

2. Fake data looks like real data if it has high probability under a density model fit to the real data.

The first approach is the one we saw in Chapter **??** on statistical image models, where fake textures were made that had the same filter response statistics as a real training example. This approach works well when we only want to match certain properties of the training data. The second approach, which is the main focus of this chapter, is better when we want to produce fake data that matches *all* statistics of the training examples.

To be precise, the goal of the deep generative models we consider in this chapter is to produce fake data that is *identically distributed* as the training data, i.e. we want $\hat{\mathbf{x}} \sim p_{\texttt{data}}$ where $p_{\texttt{data}}$ is the true process that produced the training data.

In practice, we may not be able to match all statistics, due to limits of the model's capacity.

Learning data generators There are two general approaches to learn data generators:

1. Direct approach: learn the function $G : \mathcal{Z} \rightarrow \mathcal{X}$.

2. Indirect approach: learn a function $E : X \rightarrow \mathbb{R}$ and generate samples by finding values for $\mathbf{x}$ that score highly under this function.

An example of the direct approach is the **Generative Adversarial Network** that we will see later in this chapter. First, however, we will describe the more classical, indirect approach in the following sections. Indirect approaches come in two general flavors, density models and energy models, which we describe next.

The vocab for these different approaches can be confusing. In the generative modeling literature, method #1 is called an *implicit* model since the probability density is never explicitly represented. However, this usage contrasts with the more usual use of "implicit functions" in math: $y = f(x)$ is explicit while $y = \arg\min_y E(x, y)$ is "implicit" as the output is the implied result of an optimization procedure over E, rather than the direct result of running a function f. Approach #2 to generative modeling is "implicit" in that the output is a sampling procedure over a density or energy function.

1.1.2 Density models

Some generative models not only produce generators but also yield a probability density function p_θ, fit to the training data. This density function may play a role in training the generator (e.g., we want the generator to produce samples from p_θ) or the density function may be the goal itself (e.g., we want to be able to estimate the probability of datapoints in order to detect anomalies).

In fact, some generative models *only* produce a density p_θ and do not learn any explicit generator function. Instead, samples can be down from p_θ using procedures such as **Markov Chain Monte Carlo** (**MCMC**).

Learning density models 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, and then solve the following optimization problem:

$$\arg\min_{p_\theta} D(p_\theta, p_{\texttt{data}}) \tag{1.3}$$

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"[1] 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:

$$\begin{aligned}
p_\theta^* &= \arg\min_{p_\theta} \texttt{KL}(p_{\texttt{data}}, p_\theta) && (1.4)\\
&= \arg\min_{p_\theta} \mathbb{E}_{\mathbf{x}\sim p_{\texttt{data}}}[\log \frac{p_\theta(\mathbf{x})}{p_{\texttt{data}}(\mathbf{x})}] && (1.5)\\
&= \arg\min_{p_\theta} \mathbb{E}_{\mathbf{x}\sim p_{\texttt{data}}}[\log p_\theta(\mathbf{x})] - \mathbb{E}_{\mathbf{x}\sim p_{\texttt{data}}}[\log p_{\texttt{data}}(\mathbf{x})] && (1.6)\\
&= \arg\min_{p_\theta} \mathbb{E}_{\mathbf{x}\sim p_{\texttt{data}}}[\log p_\theta(\mathbf{x})] \qquad \triangleleft \text{ dropped 2nd term since no dependence on } p_\theta && (1.7)\\
&\approx \arg\min_{p_\theta} \frac{1}{N}\sum_{i=1}^{N} \log p_\theta(\mathbf{x}^{(i)}) && (1.8)
\end{aligned}$$

where the final line is an empirical estimate of the expectation by sampling over the training dataset $\{\mathbf{x}^{(i)}\}_{i=1}^N$. Equation 1.7 is the *expected log likelihood of the training data under the model's density function.* Maximizing this objective is therefore a form of **max likelihood learning**. Pictorially we can visualize the max likelihood objective as trying to push up the density over each observed datapoint:

Remember that a probability density function (pdf) is a function $p_\theta : \mathcal{X} \rightarrow [0, \infty)$ with $\int_{\mathbf{x}} p_\theta(\mathbf{x})d\mathbf{x} = 1$ (normalized). To learn a pdf, we will typically learn the parameters of a family of pdfs. All members of the family are normalized nonnegative functions; this way we do not need to add an explicit constraint that the learned function have these properties, we are simply searching over a space of functions *all of which* have these properties. This means that whenever we push up density over datapoints, we are forced to sacrifice density over other regions, so, implicitly, we are removing density from places where there is no data, as indicated by the red regions in Figure **??**.

[1] The divergence need not be a proper distance *metric*, and often is not; it can be non-symmetric, where $D(p, q) \neq D(q, p)$, and need not satisfy the triangle-inequality. In fact, a divergence is defined by just two properties: non-negativity and $D(p, q) = 0 \iff p = q$.

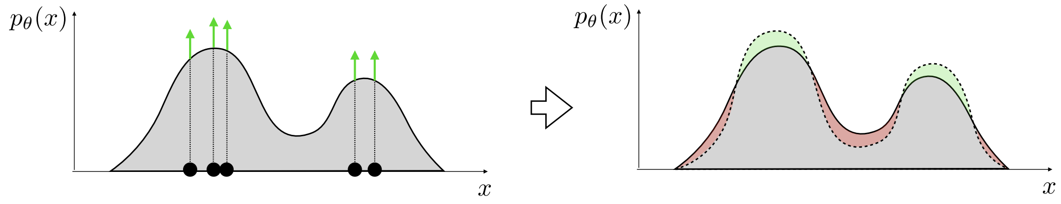


Figure 1.2: Fitting a max likelihood density model to data. The gray region holds a constant amount of mass – think of it as piles of dirt. To increase the amount of dirt at the locations of the green arrow you must remove dirt from other regions, indicated in red.

In the next section we will see an alternative approach where the parametric family we search over need not be normalized.

1.1.3 Energy-based models

Density models constrain the learned function to be normalized, i.e. $\int_{\mathbf{x}} p_\theta(\mathbf{x})d\mathbf{x} = 1$. This constraint is often hard to realize. One approach is to learn an unnormalized function E_θ, then convert it to the normalized density $p_\theta = \frac{e^{-E_\theta}}{Z(\theta)}$, where $Z(\theta) = \int_{\mathbf{x}} e^{-E_\theta(\mathbf{x})}d\mathbf{x}$ is the normalizing constant. $Z(\theta)$ can be very expensive to compute and often can only be approximated.

The parametric form $\frac{e^{-E_\theta}}{Z(\theta)}$ is sometimes referred to as a **Boltzmann** or **Gibbs distribution**.

Energy-based models (**EBM**s) address this by simply skipping the step where we normalize the density, and letting the output of the learner just be E_θ. Even though it is not a true probability density, E_θ can still be used for many of the applications we would want a density for. This is because we can compare *relative probabilities* with E_θ:

$$\frac{p_\theta(\mathbf{x}_1)}{p_\theta(\mathbf{x}_2)} = \frac{e^{-E_\theta(\mathbf{x}_1)}/Z(\theta)}{e^{-E_\theta(\mathbf{x}_2)}/Z(\theta)} = \frac{e^{-E_\theta(\mathbf{x}_1)}}{e^{-E_\theta(\mathbf{x}_2)}} \tag{1.9}$$

Notice that Z is a function of model parameters θ but not of data $\mathbf{x}$, since we integrate over all possible data values.

Knowing relative probabilities is all that is required for sampling (via MCMC), for outlier detection (the relatively lowest probability datapoint in a set of datapoints is the outlier), and even for optimizing over a space of of data to find the datapoint that is max probability (because $\arg\max_{\mathbf{x}\in\mathcal{X}} p_\theta(\mathbf{x}) = \arg\max_{\mathbf{x}\in\mathcal{X}} -E_\theta(\mathbf{x})$). To solve such a maximization problem, we might want to find the gradient of the log probability density w.r.t. $\mathbf{x}$; it turns out this gradient is identical to the gradient of $-E_\theta$ w.r.t. $\mathbf{x}$!

$$\begin{aligned}\nabla_{\mathbf{x}} \log p_\theta(\mathbf{x}) &= \nabla_{\mathbf{x}} \log \frac{e^{-E_\theta(\mathbf{x})}}{Z(\theta)} && (1.10)\\ &= -\nabla_{\mathbf{x}} E_\theta(\mathbf{x}) - \nabla_{\mathbf{x}} \log Z(\theta) && (1.11)\\ &= -\nabla_{\mathbf{x}} E_\theta(\mathbf{x}) && (1.12)\end{aligned}$$

In sum, energy functions can do most of what probability densities can do, except that they do not give normalized probabilities. Therefore, they are insufficient for applications where communicating probabilities is important for either interpretability or for interfacing with downstream systems that require knowing true probabilities.

A case where a density model might be preferable over an energy model is a medical imaging system that needs to communicate to doctors the likelihood that a patient has cancer.

Learning energy-based models Learning the parameters of an energy-based model is a bit different than learning the parameters of a density model. In a density model, we simply increase the probability mass over observed datapoints, and, because the density is a normalized function, this implicitly pushes down the density assigned to regions where there is no observed data. Since energy functions are not normalized, we need to add an explicit "negative" term to push up energy where there are no datapoints, in addition to the positive term of pushing down energy where the data is observed. One way to do so is called **contrastive divergence** [Hinton 2002]. On each iteration of optimization, this method modifies the energy function to decrease the energy assigned to samples from the data (positive term) and to increase the energy assigned to samples from the model (i.e. samples from the energy function itself; negative term):

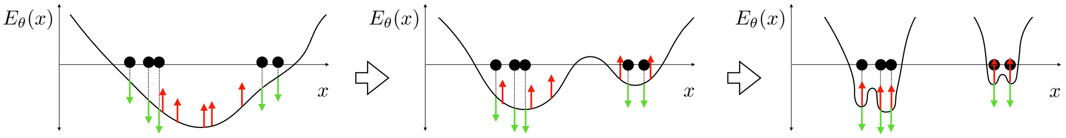


Figure 1.3: Fitting a max likelihood energy function to data, using contrastive divergence [Hinton 2002].

Once the energy function perfectly adheres to the data, samples from the model are identical to samples from the data and the positive and negative terms cancel out. This should make intuitive sense because we don't want the energy function to change once we have perfectly fit the data. It turns out that mathematically this procedure is an approximation to the gradient of the log likelihood function. Defining $p_\theta = \frac{e^{-E_\theta}}{Z(\theta)}$, start by decomposing the gradient of the data log likelihood into two terms, which, as we will see, will end up playing the role of positive and negative terms:

$$\nabla_\theta \mathbb{E}_{\mathbf{x}\sim p_{\mathtt{data}}}[\log p_\theta(\mathbf{x})] = \nabla_\theta \mathbb{E}_{\mathbf{x}\sim p_{\mathtt{data}}}[\log \frac{e^{-E_\theta(\mathbf{x})}}{Z(\theta)}] \tag{1.13}$$

$$= -\mathbb{E}_{\mathbf{x}\sim p_{\mathtt{data}}}[\nabla_\theta E_\theta(\mathbf{x})] - \nabla_\theta \log Z(\theta) \tag{1.14}$$

The first term is the positive term gradient, which tries to modify parameters to decrease the energy placed on data samples. The second term is the negative term gradient – here it appears as an intractable integral, so our strategy is to rewrite it as an expectation, which can be approximated via sampling:

Here we use a very useful identity from the chain rule of calculus, which appears often in machine learning: $\nabla_x \log f(x) = \frac{1}{f(x)} \nabla_x f(x)$

$$-\nabla_\theta \log Z(\theta) = \frac{1}{Z(\theta)} \nabla_\theta Z(\theta) \tag{1.15}$$

$$= \frac{1}{Z(\theta)} \nabla_\theta \int_x e^{-E_\theta(\mathbf{x})} d\mathbf{x} \qquad \triangleleft \quad \text{definition of } Z \tag{1.16}$$

$$= \frac{1}{Z(\theta)} \int_x \nabla_\theta e^{-E_\theta(\mathbf{x})} d\mathbf{x} \qquad \triangleleft \quad \text{exchange sum and grad} \tag{1.17}$$

$$= \frac{1}{Z(\theta)} - \int_x e^{-E_\theta(\mathbf{x})} \nabla_\theta E_\theta(\mathbf{x}) d\mathbf{x} \tag{1.18}$$

$$= -\int_x \frac{e^{-E_\theta(\mathbf{x})}}{Z(\theta)} \nabla_\theta E_\theta(\mathbf{x}) d\mathbf{x} \tag{1.19}$$

$$= -\int_x p_\theta(\mathbf{x}) \nabla_\theta E_\theta(\mathbf{x}) d\mathbf{x} \qquad \triangleleft \quad \text{definition of } p_\theta \tag{1.20}$$

$$= -\mathbb{E}_{\mathbf{x}\sim p_\theta}[\nabla_\theta E_\theta(\mathbf{x})] \qquad \triangleleft \quad \text{definition of expectation} \tag{1.21}$$

Plugging Eqn 1.21 back into Eqn 1.14, we arrive at:

$$\nabla_\theta \mathbb{E}_{\mathbf{x}\sim p_{\mathtt{data}}}[\log p_\theta(\mathbf{x})] = -\mathbb{E}_{\mathbf{x}\sim p_{\mathtt{data}}}[\nabla_\theta E_{\theta(\mathbf{x})}] + \mathbb{E}_{\mathbf{x}\sim p_\theta}[\nabla_\theta E_\theta(\mathbf{x})] \tag{1.22}$$

Both expectations can be approximated via sampling: defining $\mathbf{x}^{(i)} \sim p_{\mathtt{data}}$ and $\hat{\mathbf{x}}^{(i)} \sim p_\theta$, and taking N such samples, we have:

$$-\mathbb{E}_{\mathbf{x}\sim p_{\texttt{data}}}[\nabla_\theta E_{\theta(\mathbf{x})}] + \mathbb{E}_{\mathbf{x}\sim p_\theta}[\nabla_\theta E_\theta(\mathbf{x})] \approx -\frac{1}{N}\sum_{i=1}^{N} \nabla_\theta E_\theta(\mathbf{x}^{(i)}) + \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta E_\theta(\hat{\mathbf{x}}^{(i)}) \tag{1.23}$$

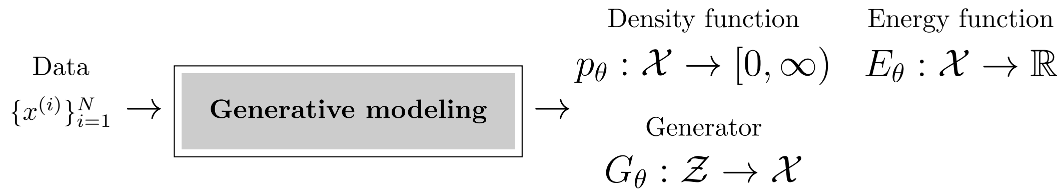
$$= \frac{1}{N}\nabla_\theta \sum_{i=1}^{N}(-E_\theta(\mathbf{x}^{(i)}) + E_\theta(\hat{\mathbf{x}}^{(i)})) \tag{1.24}$$

The last line should make clear the intuition: we establish a “contrast” between data samples and current model samples, then update the model to decrease this contrast, bringing the model closer to the data. Once $\mathbf{x}^{(i)}$ and $\hat{\mathbf{x}}^{(i)}$ are identically distributed, this gradient will be zero (in expectation) – we have perfectly fit the data and no further updates should be taken.

Under our formalization of a learning problem as an objective, hypothesis space, and optimizer, contrastive divergence is an *optimizer*; it’s an optimizer specifically built for max likelihood objectives over energy functions. Contrastive divergence tells us how to approximate the gradient of the likelihood function, which can then be plugged into any gradient descent method.

1.2 Types of generative models

Some generative models give density or energy functions, others give generator functions, and still others give both. We can summarize all these kinds of models with the following learning diagram:


Data
$\{x^{(i)}\}_{i=1}^N$
Generative modeling
Density function
$p_\theta : \mathcal{X} \to [0,\infty)$
Energy function
$E_\theta : \mathcal{X} \to \mathbb{R}$
Generator
$G_\theta : \mathcal{Z} \to \mathcal{X}$

It’s worth noting a special property of generative models that learn a generator G_θ. The “noise” $\mathbf{z}$ input to the generator acts as **latent variables** that control the properties of the generated image. Changing these variables can change the generated image in meaningful ways – we will explore this idea in greater detail in the next two chapters. In contrast, density and energy models do not have latent variables that directly control generated samples[2].

Some of the generative models we will describe in this chapter and the next are categorized, along these dimensions, in the table 1.1.

Generative models can also be distinguished according to their objectives, hypothesis spaces, and optimization algorithms. Indeed, some classes of model, such as autoregressive models refer to just a particular kind of hypothesis space, whereas other classes of model, such as variational autoencoders, are much more specific in referring to the conjunction of an objective, a general family of hypothesis spaces, and a particular optimization algorithm.

1.2.1 Gaussian density models

One of the simplest and most useful density models is the Gaussian distribution, which, in 1D is:

$$p_\theta(x) = \frac{1}{Z}e^{-(x-\theta_1)^2/(2\theta_2)} \tag{1.25}$$

[2] The *sampling algorithm* that draws samples from a density/energy function necessarily *does* depend on “noise” factors that control properties of the generated samples, but this dependency is typically much less explicit than the relationship between noise inputs and image outputs in a generator function.

Method	Latents?	Density?	Generator?
Autoregressive models	✗	✓	slow
Diffusion models	high-dim	✗	slow
GANs	✓	✗	✓
VAEs	✓	lower-bound	✓
Energy-based models	✗	unnormalized	✗

Table 1.1: Three desirable properties in a generative model. No method achieves all three (without caveats). VAEs and GANs are good at representation learning (i.e. at finding a low-dimensional latent space). Autoregressive models are great if you want to estimate the likelihood of your data points (e.g., for anomaly detection). Energy-based models can be an especially efficient way to model an unnormalized density.

This density has two parameters θ_1 and θ_2, which are the mean and variance of the distribution. The normalization constant Z ensures that the function is normalized. This is the typical strategy in defining density models: create a parameterized family of functions such that any function in the family is a normalized. Given such a family, we search over the parameters to optimize a generative modeling objective.

For 1D Gaussians $Z = \frac{1}{\sqrt{2\pi\theta_2}}$. We prefer to write it as Z to emphasize its structural role as a normalization constant. Usually we will not be explicitly evaluating normalization constants, and will not require knowing their analytical form.

For density models, the most common objective is max likelihood:

$$\mathbb{E}_{x\sim p_{\texttt{data}}}[\log p_\theta(x)] \approx \frac{1}{N}\sum_{i=1}^{N}\log p_\theta(x^{(i)}) \tag{1.26}$$

For a 1D Gaussian, this has a simple form:

$$\frac{1}{N}\sum_{i=1}^{N}\log\frac{1}{Z}e^{-(x^{(i)}-\theta_1)^2/(2\theta_2)} = -\log(Z) + \frac{1}{N}\sum_{i=1}^{N}(x^{(i)}-\theta_1)^2/(2\theta_2) \tag{1.27}$$

Optimizing w.r.t. θ_1 and θ_2 could be done via gradient descent or random search, but in this case there is also an analytical solution we can find by setting the gradient to be zero. For θ_1^* we have:

$$\frac{\partial - \log Z + \frac{1}{N}\sum_{i=1}^{N}(x^{(i)}-\theta_1^*)^2/(2\theta_2^*)}{\partial\theta_1^*} = 0 \tag{1.28}$$

$$\frac{1}{N}\sum_{i=1}^{N}2(x^{(i)}-\theta_1^*)/(2\theta_2^*) = 0 \tag{1.29}$$

$$\sum_{i=1}^{N}x^{(i)} - \sum_{i=1}^{N}\theta_1^* = 0 \tag{1.30}$$

$$\theta_1^* = \frac{1}{N}\sum_{i=1}^{N}x^{(i)} \tag{1.31}$$

For θ_2^* we need to note that Z depends on θ_2^* and in particular notice that $\frac{\partial - \log Z}{\partial \theta_2^*} = \frac{1}{2\theta_2^*}$:

$$\frac{\partial - \log Z + \frac{1}{N}\sum_{i=1}^{N}(x^{(i)} - \theta_1^*)^2/(2\theta_2^*)}{\partial \theta_2^*} = 0 \tag{1.32}$$

$$\frac{1}{2\theta_2^*} - \frac{1}{N}\sum_{i=1}^{N} 2(x^{(i)} - \theta_1^*)^2/(2\theta_2^*)^2 = 0 \tag{1.33}$$

$$2\theta_2^* - \frac{1}{N}\sum_{i=1}^{N} 2(x^{(i)} - \theta_1^*)^2 = 0 \tag{1.34}$$

$$\theta_2^* = \frac{1}{N}\sum_{i=1}^{N}(x^{(i)} - \theta_1^*)^2 \tag{1.35}$$

You might recognize the solutions for θ_1^* and θ_2^* as, repsectively, the empirical mean and variance of the data. This makes sense: we have just shown that to maximize the probability of the data under a Gaussian, we should set the mean and variance of the Gaussian to be the empirical mean and variance of the data.

This fully describes the learning problem, and solution, for a 1D Gaussian density model. We can put it all together in the learning diagram below:

Data

$\{x^{(i)}\}_{i=1}^{N} \rightarrow$

1D Gaussian density model

Objective

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

Hypothesis space

$p_\theta(x) = \frac{1}{Z}e^{-(x-\theta_1)^2/(2\theta_2)}$

Optimizer

$\theta_1^* = \frac{1}{N}\sum_{i=1}^{N} x^{(i)}$

$\theta_2^* = \frac{1}{N}\sum_{i=1}^{N}(x^{(i)} - \theta_1^*)^2$

$\rightarrow p_\theta$

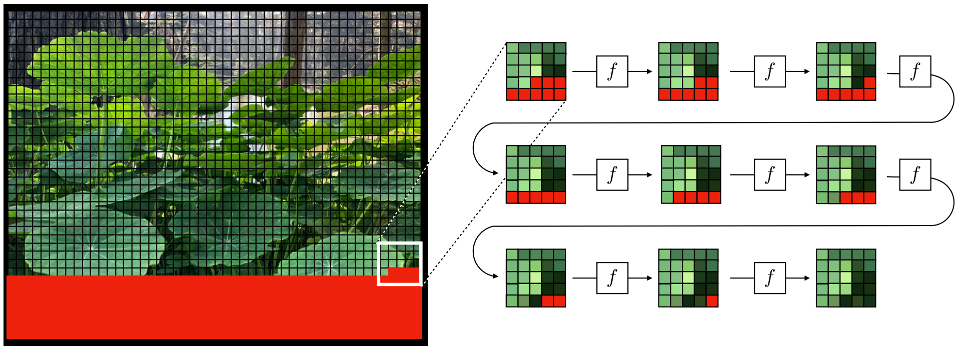
Gaussian density models are just about the simplest density models one can come up with. You may be wondering, do we actually use them for anything in computer vision, or are they just a toy example? The answer is that *yes* we do use them, and all the time. For example, in least-squares regression, we are simply fitting a Gaussian density to the conditional probability $p(y|x)$. If we want a more complicated density, we may use a mixture of Gaussians, which we will encounter in the next chapter. It's useful to get comfortable with Gaussian fits because 1) they are a subcomponent of many more sophisticated models, and 2) they showcase all the key components of density modeling, with a clear objective, a parameterized hypothesis space, and an optimizer that finds the parameters that maximize the objective.

1.2.2 Autoregressive density models

A single Gaussian is a very limited model, and the real utility of Gaussians only shows up when they are part of a larger modeling framework. Next we will consider a recipe for building highly expressive models out of simple ones. There are many such recipes and the one we focus on here is called an **autoregressive model**.

The idea of an autoregressive model is to synthesize an image pixel by pixel. Each new pixel is decided on based on the sequence already generated. You can think of this as a simple sequence prediction problem: given a sequence of observed pixels, predict the color of the next one. We use the same learned function f to make each subsequent prediction:

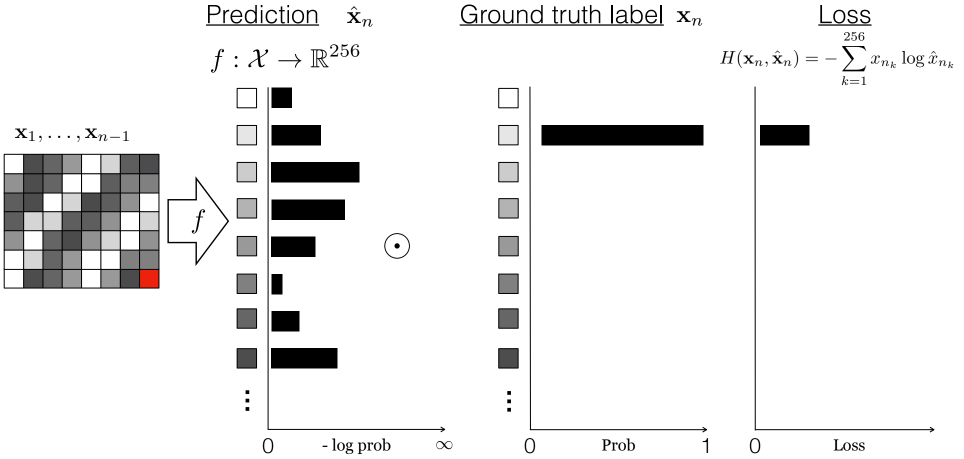
Here we have a partially synthesized image. The red pixels are the remaining pixels to



These models can be easily understood by first considering the problem of synthesizing one pixel, then two, and so on. The first observation to make is that it's pretty easy to synthesize a single grayscale pixel. Such a pixel can take on 256 possible values (for a standard 8-bit grayscale image). So it suffices to use a categorical distribution to represent the probability that the pixel takes on each of these possible values. The categorical distribution is fully expressive: any possible pmf over the 256 values can be represented with the categorical distribution. Fitting this categorical distribution to training data just amounts to counting how often we observe each of the 256 values in the training pixels, and normalizing by the total number of training pixels. So we know how to model one grayscale pixel. We can sample from this distribution to synthesize a random 1-pixel image.

How do we model the distribution over a second grayscale pixel given the first? In fact, we already know how to model this setting; mathematically, we are just trying to model $p(\mathbf{x_2}|\mathbf{x_1})$ where $\mathbf{x_1}$ is the first pixel and $\mathbf{x_2}$ is the second. Treating x_2 as a categorical variable (just like $\mathbf{x_1}$), we can simply use a softmax regression, which we saw in Chapter **??**. In that chapter we were modeling a K-way categorical distribution over K object classes, conditioned on an input image. Now we can use exactly the same tools to model a 256-way distribution over a the second pixel in a sequence conditioned on the first.

What about the third pixel, conditioned on the first two? Well, this is again a problem of the same form: a 256-way softmax regression conditioned on some observations. Now you can see the induction: modeling each next pixel in the sequence that forms the image is a softmax regression problem that models $p(\mathbf{x_n}|\mathbf{x_1}, \dots, \mathbf{x}_{n-1})$. In fact, it looks almost identical to Figure **??** in Chapter **??**:


Prediction $\hat{\mathbf{x}}_n$
$f : \mathcal{X} \rightarrow \mathbb{R}^{256}$
Ground truth label $\mathbf{x}_n$
Loss
$H(\mathbf{x}_n, \hat{\mathbf{x}}_n) = -\sum_{k=1}^{256} x_{n_k} \log \hat{x}_{n_k}$
$\mathbf{x}_1, \dots, \mathbf{x}_{n-1}$
f
$\odot$
0
- log prob
∞
0
Prob
1
0
Loss

You might be wondering: how do we turn an image into a sequence of pixels? Good question! There are innumerable ways, but the simplest is often good enough: just vectorize the 2D grid of pixels by first listing the first row of pixels, then the second row, and so forth. In general, any fixed ordering of the pixels into a sequence is actually valid, but this simple method is perhaps the most common.

So we can model the probability of each subsequent pixel given the preceding pixels. To generate an image we can sample a value for the first pixel, then the second given the first, then the third given the first and second, and so forth. But is this a valid model of $p(\mathbf{X}) = p(\mathbf{x}_1, \dots, \mathbf{x}_n)$, the probability distribution of the full set of pixels? Does this way of sequential sampling draw a valid sample from $p(\mathbf{X})$? It turns out it does, according tothe **chain rule of probability**. This rule allows us to factorize *any* joint distribution into a product of conditionals as follows:

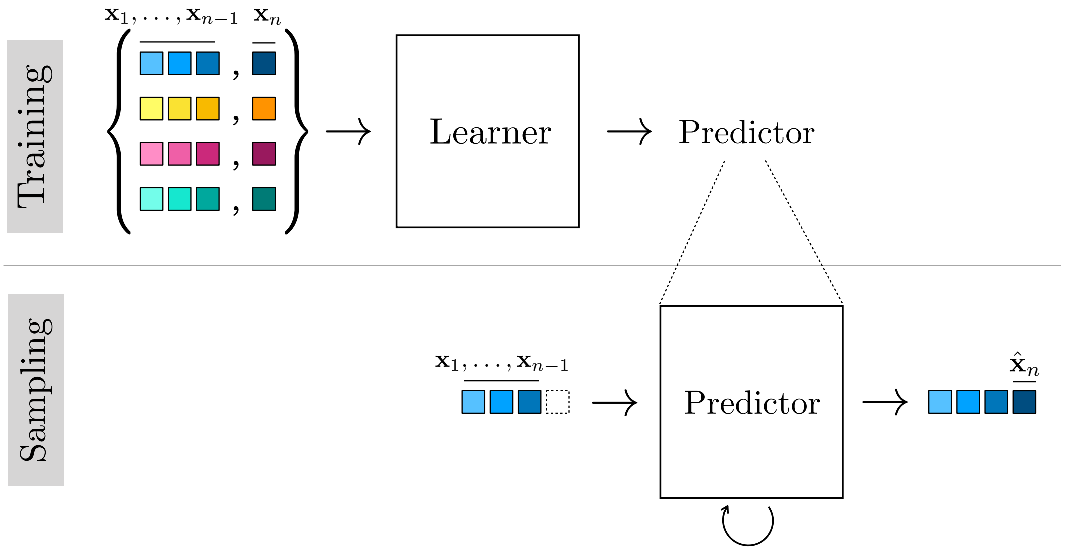
$$p(\mathbf{X}) = p(\mathbf{x_n}|\mathbf{x_1}, \dots, \mathbf{x}_{n-1})p(\mathbf{x}_{n-1}|\mathbf{x}_1, \dots, \mathbf{x}_{n-2}) \quad \dots \quad p(\mathbf{x}_2|\mathbf{x}_1)p(\mathbf{x}_1) \tag{1.36}$$

$$p(\mathbf{X}) = \prod_{i=1}^{n} p(\mathbf{x}_i|\mathbf{x}_1, \dots, \mathbf{x}_{i-1}) \tag{1.37}$$

As a notational convenience, we define here that $p(\mathbf{x}_i|\mathbf{x}_1, \dots, \mathbf{x}_{i-1}) = p(\mathbf{x}_1)$ when $i = 1$.

This factorization demonstrates that sampling from such a model can be indeed done in sequence because all the conditional distributions are independent of each other, and we just take a product over all of them. This sampling method is also called **ancestral sampling** (sample your ancestors first, then each next "generation" given the preceding generation).

To train an autogressive a model, you just need to extract supervised pairs of desired input-output behavior, as usual. For an autoregressive model of pixels, that means extracting sequences of pixels $\mathbf{x}_1, \dots, \mathbf{x}_{n-1}$ and corresponding observed next pixel $\mathbf{x}_n$. These can be extracted by traversing images in raster order. The full training and testing setup looks like this:


Training
$\mathbf{x}_1, \dots, \mathbf{x}_{n-1}$
$\mathbf{x}_n$
Learner
Predictor
Sampling
$\mathbf{x}_1, \dots, \mathbf{x}_{n-1}$
Predictor
$\hat{\mathbf{x}}_n$

Training efficiency trick: When training autoregressive models, the naive way to set up the training batches is to take as input a sequence $\mathbf{x}_1, \dots, \mathbf{x}_4$, then predict $\hat{\mathbf{x}}_5$, compare the prediction to the ground truth $\mathbf{x}_5$, and backprob the loss. It turns out there is a much more efficient way to do it: typically, to predict $\hat{\mathbf{x}}_5$, the autoregressive model had to also predict $\hat{\mathbf{x}}_1, \dots, \hat{\mathbf{x}}_4$ – this is the case, for example, if the autoregressive model is an RNN, where the hidden state has to be updated sequentially and, after you have computed the sequence of hidden states, making predictions for each item in the sequence requires very little compute (maybe a linear layer and a softmax). In these cases, it's much more efficient to compare each prediction $\hat{\mathbf{x}}_1, \dots, \hat{\mathbf{x}}_5$ to the ground truth and backprop all five losses.

Autogressive models give us an explicit density function, Equation 1.37. To sample from this density we can use **ancestral sampling**, which refers to sampling the first pixel from $p(x_1)$, then, conditioned on this pixel, sample the second from $p(x_2|x_1)$ an so forth. Since each of these densities is a categorical distribution, sampling is easy: one option is to partition a unit line segment into regions of length equal to the categorical probabilities and see where a uniform random draw along this line falls. Autogressive models do not have latent variables $\mathbf{z}$, which makes them incompatible with applications that involve extracting or manipulating latent variables.

1.2.3 Generative Adversarial Networks

Autoregressive models sample simple distributions step by step to build up a complex distribution. Could we instead create a system that directly, in one step, outputs samples from the complex distribution. It turns out we can, and one model that does this is the **Generative adversarial network**, or **GAN**, which was introduced by [Goodfellow et al. 2014].

Recall that the goal of generative modeling is to make synthetic data that looks like real data. We stated above that there are many different ways to measure "looks like" and each leads to a different kind of generative model. GANs take a very direct and intuitive approach: synthetic data looks like real data if a classifier cannot distinguish between synthetic images and real images.

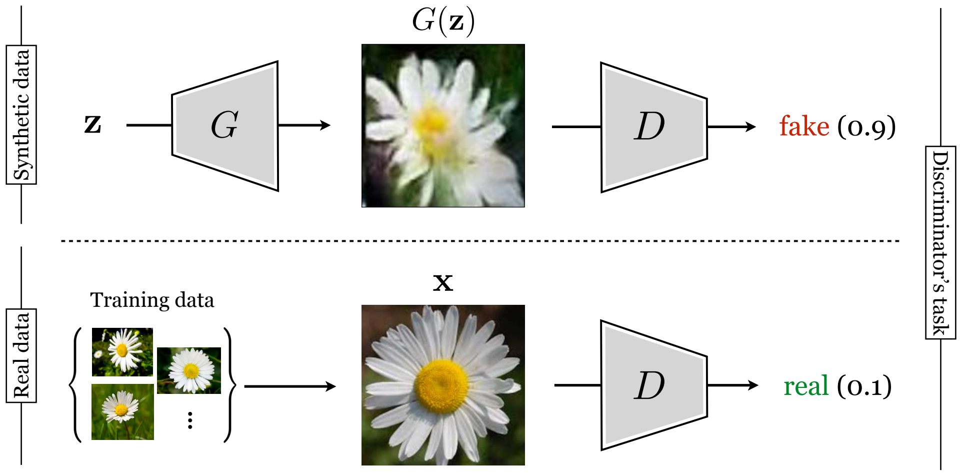
GANs consist of two neural networks, the **generator**, $G : \mathcal{Z} \rightarrow \mathcal{X}$, which synthesizes data from noise, and a **discriminator**, $D : \mathcal{X} \rightarrow \Delta^1$, that tries to classify between synthesized data and real data.

Notice that D has a form similar to an energy function, but, unlike energy functions, D is not, in general, interpretable as an unnormalized probability density. Nonetheless, we can roughly think of a GAN as a type of energy-based generative model where we train another network G to directly sample from the energy function rather than relying on MCMC to sample from the energy function.

G and D play an adversarial game in which G tries to become better and better at generating synthetic images while D tries to become better and better at detecting any errors G is making. The learning problem can be written as a minimax game:

$$\arg\min_G \max_D \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}[\log(1 - D(G(\mathbf{z})))] \tag{1.38}$$

Schematically, the generator synthesizes images that are then fed as input to the discriminator. The discriminator tries to assign a high score to generated images (classifying them as "fake") and a low score to real images from some training set (classifying them as "real"):



Although we call this an adversarial game, the discriminator is in fact helping the generator to perform better and better, by pointing out its current errors (we call it an adversary because it tries to point out errors). You can think of the generator as a student taking a painting class and the discriminator as the teacher. The student is trying to produce new paintings that match the quality and style of the teacher. At first the student paints flat landscapes that lack shading and the illusion of depth; the teacher gives feedback: "this mountain is not well shaded, it looks 2D." So the student improves and corrects the error, adding haze and shadows to the mountain. The teacher is pleased but now points out a different error: "the trees all look identical, there is not enough variety." The teacher and student continue on in this fashion until the student has succeeded at satisfying the teacher.

Eventually, in theory, the student – the generator – produces paintings that are just as good as the teacher's paintings.

This objective may be easier to understand if we think of the objectives for G and D separately. Given a particular generator G, D tries to maximize its ability to discriminate between real and fake images (fake images are anything output by G). D's objective is logistic regression between a set of real data $\{\mathbf{x}^{(i)}\}_{i=1}^{N}$ and fake data $\{\mathbf{x}^{(i)}\}_{i=1}^{N}$, where $\hat{\mathbf{x}}^{(i)} = G(\mathbf{z}^{(i)})$.

Let the optimal discriminator be labeled D^*. We have that:

$$D^* = \arg\max_{D} \mathbb{E}_{\mathbf{x}\sim p_{\texttt{data}}}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}[\log(1 - D(G(\mathbf{z})))] \tag{1.39}$$

Now we turn to G's perspective. Given D^*, G tries to solve the following problem:

$$\arg\min_{G} \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}[\log(1 - D^*(G(\mathbf{z})))] \tag{1.40}$$

Now, because the optimal discriminator D^* depends on the current behavior of G, as soon as we *change* G, updating it to better fool D^*, D^* no longer is the optimal discriminator and we need to again solve problem in Eqn. 1.39. To optimize a GAN, we simply alternate between taking one gradient on Eqn. 1.40 and then K gradient steps on Eqn. 1.39, where the larger the K, the closer we are to approximating the true D^*. In practice, setting $K = 1$ is often sufficient.

GANs are statistical image models GANs are related to the statistical image models we saw in Chapter **??**. In Heeger and Bergen (1995), for example, we synthesize images with the same statistics as a source texture. This can be phrased as an optimization problem in which we optimize image pixels until certain statistics of the images match those same statistics measured on a source (training) set of images. We can write it as $\|\phi(x) - \phi(\hat{x})\|$. This is a kind of discriminator – it outputs a score related to the difference between a generated image and real data. However, unlike a GAN, this discriminator is hand-defined in terms of certain statistics of interest rather than learned. Additionally, GANs amortize the optimization over pixels that satisfy the "discriminator". That is GANs learned a mapping G from latent noise to samples rather than arriving at samples via an optimization process that starts from scratch each time we want to make a new sample.

1.3 Generative versus discriminative

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

I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

D. J. Heeger and J. R. Bergen. Pyramid-based texture analysis/synthesis. In *ACM SIGGRAPH*, pages 229–236, 1995. In *Computer Graphics* Proceedings, Annual Conference Series.

G. E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural computation*, 14(8):1771–1800, 2002.