3.4. Generative neural networks

How can we use neural networks for generating random values? Neural networks have the job of taking an input and processing it to produce an output. This is an obvious fit for supervised learning, where we aim to predict the output label given the input predictor variables. But how can we frame generative modelling — in which we are given an unlabelled dataset $x_1, \ldots, x_n$ and we want to find a probability model that might have generated these values — as an input/output problem?

There's an almost embarrassingly simple trick: we simply decide to model the data as $X = f(Z)$ where Z is some standard random variable for example a Uniform$[0, 1]$ and f is the neural network. Here Z is called the *latent* random variable, meaning *hidden*, since we're not told the z_i that led to each datapoint x_i.

In principle it doesn't matter what distribution we use for Z, whether it's Uniform or Normal or a collection of independent Normals. The neural network will (we hope) learn to transform Z into something interesting like a photo or a piece of text, and in comparison the job of transforming from a Uniform to a Normal distribution is trivial. It's even possible, in principle, for the network to turn a single random variable into a collection of independent random variables.[24] In practice, there's no point making training more demanding than it needs to be, and neural networks 'like to be fed randomness', so the pragmatic choice is to let Z have as many dimensions as there are free dimensions in the dataset. This hand-wavey advice will be made quantifiable when we look at autoencoders in section 5.4.

the inversion method, section 4.6, shows how to transform between $U[0, 1]$ and any other real-valued random variable

Training a generative neural network. Training a generative model is just the same as any other probabilistic modelling task: it's just maximum likelihood estimation. In other words, we'll choose the parameters of the network to maximize the log likelihood of the dataset, $\sum_i \log \Pr_X(x_i)$.

The challenge of generative neural networks, and the reason they are understudied compared to neural networks for supervised learning, is that it takes mathematical subtlety to set things up so that we can even compute $\Pr_X(x_i)$. There are several approaches. We'll see a very simple approach here, only suitable for toy examples. In later chapters, after we've covered the necessary probability tools, we'll look at two other approaches: autoencoders in section 5.4 and recurrent neural networks in section 10.6.

Evaluating a generative neural network. It's good machine learning practice to set aside a holdout dataset, and to evaluate a neural network's performance by its prediction accuracy on the holdout data. It never saw the holdout data during training, so it can't cheat by memorizing the answers. But how should we evaluate a generative model? We can't measure its prediction accuracy on a holdout set, since it's not designed to make predictions.

We could conceivably ask humans to evaluate the outputs it generates, to judge whether they are realistic. Or, better, take some machine-generated outputs and some holdout datapoints, and see if a human can tell them apart. But this doesn't scale—it's hardly *machine* learning if it needs a human in the loop. Instead, we could build a 'discriminator' neural network and measure how well it can tell the difference. This is a powerful idea—the basis for Generative Adversarial Networks—but it's hard to get right. Did our generator do well because it's a good generator, or because the discriminator we trained was inadequate?

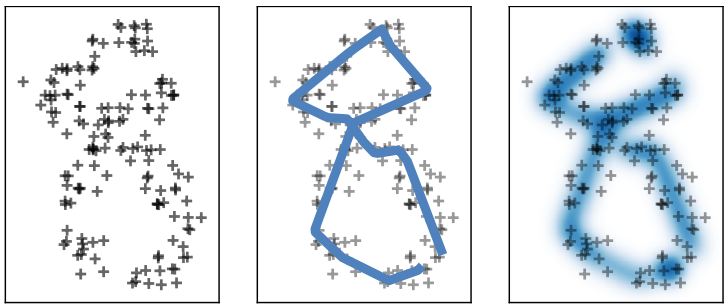
Probabilistic modelling rides in to the rescue. A generative neural network is nothing more nothing less than a probability model, and a probability model can be evaluated by the log likelihood of observed data. We simply train the network on the training dataset, then measure the log likelihood of holdout data, and that is our evaluation metric. There's no cleverness needed, no special treatment: it's exactly the same evaluation for supervised learning as for generative modelling, and it's exactly the same metric we use for training as for evaluation.

In Natural Language Processing this evaluation metric is widely used (or rather, an oddly-transformed version of it), and it's given the name *perplexity*. It's woefully underused in other areas of machine learning. We'll discuss evaluation much more deeply in section 9.

[24] Here's how to transform a single Uniform random variable into two independent Uniform random variables. Let the binary expansion of $U \sim$ Uniform$[0, 1]$ be $0.U_1U_2U_3\ldots$, and then simply let $X_1 = 0.U_1U_3U_5\ldots$ and $X_2 = 0.U_2U_4U_6\ldots$. This is all well and good for infinite precision mathematics, not helpful for floating point computation.

Example 3.4.1.

Train a generative model for a collection of points $x_1, \dots, x_n$ in $\mathbb{R}^2$. (The points shown here are pixels from a handwritten MNIST digit.) The model should have the form "pick a random point on a curved path, then offset it randomly".



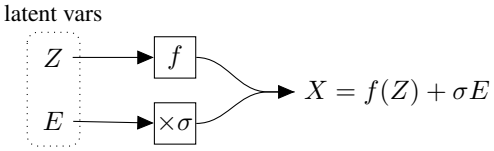
points x_i path $f(z)$ likelihood $\Pr_X(x)$

To be precise, model the dataset as independent samples from

$$X \sim f(Z) + N(0, \sigma^2 I)$$

where $f : [0,1] \to \mathbb{R}^2$ is a neural network to be trained, σ is a noise parameter to be fitted, and Z is a discretized $U[0,1]$ random variable taking evenly spaced values[25] $\{z_1 = 0, \dots, z_m = 1\}$ where m is given. The notation $N(0, \sigma^2 I)$ means "generate two independent $N(0, \sigma^2)$ random variables, one for each of the $\mathbb{R}^2$ coordinates", and it's a special case of the multivariate Normal distribution.

The notation here obscures what the true latent variable is. It's actually a pair (Z, E) where Z is the discretized uniform random variable specified in the question, and $E \sim N(0, I)$ is made up of two independent Normal$(0,1)$ random variables. The overall function we're learning is $g(Z, E) = f(Z) + \sigma E$. Latent variables are meant to capture all of the randomness of X; and their distributions are not meant to depend on whatever it is we aim to learn. This isn't terribly important—it's a matter of terminology, not a matter of modelling—but it's good mental hygiene to be explicit about all the sources of randomness in our model, and about what is learnt versus what is given.



The first step is to derive a formula for the log likelihood of the dataset. As usual, we're modelling the datapoints as independent samples, so the total log likelihood is the sum of the log likelihoods of individual datapoints. For an individual datapoint $x \in \mathbb{R}^2$,

$$\begin{aligned}
\log \Pr_X(x) &= \log\left(\sum_{j=1}^{m} \Pr_X(x \mid Z = z_j) \Pr_Z(z_j)\right) && \textit{by law of total prob.} \\
&= \log\left(\frac{1}{m}\sum_{j=1}^{m} \frac{1}{2\pi\sigma^2} e^{-\|x - f(z_j)\|^2/2\sigma^2}\right) && \textit{as } X \textit{ is normal with mean } f(z_j) \\
&= -\log(2\pi\sigma^2) + \log\left(\frac{1}{m}\sum_{j=1}^{m} e^{-\|x - f(z_j)\|^2/2\sigma^2}\right).
\end{aligned}$$

(Remember that each datapoint x has two dimensions, and each dimension of $x - f(z)$ is an independent $N(0, \sigma^2)$ random variable, which is why there is $1/2\pi\sigma^2$ rather than $1/\sqrt{2\pi\sigma^2}$.)

Now we can fit the model. This code defines a PyTorch module corresponding to the random variable, and its evaluation function computes the log likelihood:

$$\mathsf{forward}([x_1, \dots, x_n]) = [\log \Pr_X(x_1), \dots, \log \Pr_X(x_n)].$$

[25] It would be more natural to let $Z \sim U[0,1]$, but that needs more advanced probability for computing $\Pr_X(x_i)$, and is left to section 5.4.

```
class RCurve(nn.Module):
    def __init__(self, σ0=0.1, m=100):
        super().__init__()
        self.f = nn.Sequential( # input.shape [m]
            nn.Linear(1,4),      # → [m × 4]
            nn.LeakyReLU(),
            nn.Linear(4,20),
            nn.LeakyReLU(),
            nn.Linear(20,20),
            nn.LeakyReLU(),
            nn.Linear(20,2)      # → [m × 2]
            )
        self.σ = nn.Parameter(torch.tensor(σ0))
        self.z = torch.linspace(0,1,m)
    def forward(self, x):  # x.shape [B × 2]
        μ = self.f(self.z.reshape(-1,1)).reshape(1,-1,2)
        x = x.reshape(-1,1,2)
        d = torch.linalg.norm(x - μ, dim=2)
        lik = torch.exp(- 0.5 * torch.pow(d/self.σ, 2))
        lik = torch.log(torch.mean(lik, dim=1)) - torch.log(2*np.pi*torch.pow(self.σ,2))
        return lik  # shape [B]

X = ... # the datapoints, as a n × 2 torch tensor
m = RCurve(σ0=0.03)
optimizer = optim.Adam(m.parameters())

with Interruptable() as check_interrupted:
    while True:
        check_interrupted()
        optimizer.zero_grad()
        loglik = m(X)
        e = - torch.mean(loglik)
        e.backward()
        optimizer.step()
```

You should ignore the contents of RCurve.f. *It's simply an arbitrary function* $[0,1] \to \mathbb{R}^2$ *plucked out of thin air. I experimented before settling on this particular number of layers and nodes—too simple and I only get trivial paths, too complex and I get messy wiggles rather than smooth lines.*

∎