CHAPTER 6
Normalizing Flow Models

Chapter Goals

In this chapter you will:

- Learn how normalizing flow models utilize the change of variables equation.
- See how the Jacobian determinant plays a vital role in our ability to compute an explicit density function.
- Understand how we can restrict the form of the Jacobian using coupling layers.
- See how the neural network is designed to be invertible.
- Build a RealNVP model—a particular example of a normalizing flow to generate points in 2D.
- Use the RealNVP model to generate new points that appear to have been drawn from the data distribution.
- Learn about two key extensions of the RealNVP model, GLOW and FFJORD.

So far, we have discussed three families of generative models: variational autoencoders, generative adversarial networks, and autoregressive models. Each presents a different way to address the challenge of modeling the distribution $p(x)$, either by introducing a latent variable that can be easily sampled (and transformed using the decoder in VAEs or generator in GANs), or by tractably modeling the distribution as a function of the values of preceding elements (autoregressive models).

In this chapter, we will cover a new family of generative models—normalizing flow models. As we shall see, normalizing flows share similarities with both autoregressive models and variational autoencoders. Like autoregressive models, normalizing flows are able to explicitly and tractably model the data-generating distribution $p(x)$. Like

VAEs, normalizing flows attempt to map the data into a simpler distribution, such as a Gaussian distribution. The key difference is that normalizing flows place a constraint on the form of the mapping function, so that it is invertible and can therefore be used to generate new data points.

We will dig into this definition in detail in the first section of this chapter before implementing a normalizing flow model called RealNVP using Keras. We will also see how normalizing flows can be extended to create more powerful models, such as GLOW and FFJORD.

Introduction

We will begin with a short story to illustrate the key concepts behind normalizing flows.

Jacob and the F.L.O.W. Machine

Upon visiting a small village, you notice a mysterious-looking shop with a sign above the door that says *JACOB'S*. Intrigued, you cautiously enter and ask the old man standing behind the counter what he sells (Figure 6-1).

Figure 6-1. Inside a steampunk shop, with a large metallic bell (created with Midjourney)

He replies that he offers a service for digitizing paintings, with a difference. After a brief moment rummaging around the back of the shop, he brings out a silver box, embossed with the letters F.L.O.W. He tells you that this stands for Finding Likenesses Of Watercolors, which approximately describes what the machine does. You decide to give the machine a try.

You come back the next day and hand the shopkeeper a set of your favorite paintings, and he passes them through the machine. The F.L.O.W. machine begins to hum and whistle and after a while outputs a set of numbers that appear randomly generated. The shopkeeper hands you the list and begins to walk to the till to calculate how much you owe him for the digitization process and the F.L.O.W. box. Distinctly unimpressed, you ask the shopkeeper what you should do with this long list of numbers, and how you can get your favorite paintings back.

The shopkeeper rolls his eyes, as if the answer should be obvious. He walks back to the machine and passes in the long list of numbers, this time from the opposite side. You hear the machine whir again and wait, puzzled, until finally your original paintings drop out from where they entered.

Relieved to finally have your paintings back, you decide that it might be best to just store them in the attic instead. However, before you have a chance to leave, the shopkeeper ushers you across to a different corner of the shop, where a giant bell hangs from the rafters. He hits the bell curve with a huge stick, sending vibrations around the store.

Instantly, the F.L.O.W. machine under your arm begins to hiss and whirr in reverse, as if a new set of numbers had just been passed in. After a few moments, more beautiful watercolor paintings begin to fall out of the F.L.O.W. machine, but they are not the same as the ones you originally digitized. They resemble the style and form of your original set of paintings, but each one is completely unique!

You ask the shopkeeper how this incredible device works. He explains that the magic lies in the fact that he has developed a special process that ensures the transformation is extremely fast and simple to calculate while still being sophisticated enough to convert the vibrations produced by the bell into the complex patterns and shapes present in the paintings.

Realizing the potential of this contraption, you hurriedly pay for the device and exit the store, happy that you now have a way to generate new paintings in your favorite style, simply by visiting the shop, chiming the bell, and waiting for your F.L.O.W. machine to work its magic!

The story of Jacob and the F.L.O.W. machine is a depiction of a normalizing flow model. Let's now explore the theory of normalizing flows in more detail, before we implement a practical example using Keras.

Normalizing Flows

The motivation of normalizing flow models is similar to that of variational autoencoders, which we explored in Chapter 3. To recap, in a variational autoencoder, we learn an *encoder* mapping function between a complex distribution and a much simpler distribution that we can sample from. We then also learn a *decoder* mapping

function from the simpler distribution to the complex distribution, so that we can generate a new data point by sampling a point z from the simpler distribution and applying the learned transformation. Probabilistically speaking, the decoder models $p(x|z)$ but the encoder is only an approximation $q(z|x)$ of the true $p(z|x)$—the encoder and decoder are two completely distinct neural networks.

In a normalizing flow model, the decoding function is designed to be the exact inverse of the encoding function and quick to calculate, giving normalizing flows the property of tractability. However, neural networks are not by default invertible functions. This raises the question of how we can create an invertible process that converts between a complex distribution (such as the data generation distribution of a set of watercolor paintings) and a much simpler distribution (such as a bell-shaped Gaussian distribution) while still making use of the flexibility and power of deep learning.

To answer this question, we first need to understand a technique known as *change of variables*. For this section, we will work with a simple example in just two dimensions, so that you can see exactly how normalizing flows work in fine detail. More complex examples are just extensions of the basic techniques presented here.

Change of Variables

Suppose we have a probability distribution $p_X(x)$ defined over a rectangle X in two dimensions ($x = (x_1, x_2)$), as shown in Figure 6-2.

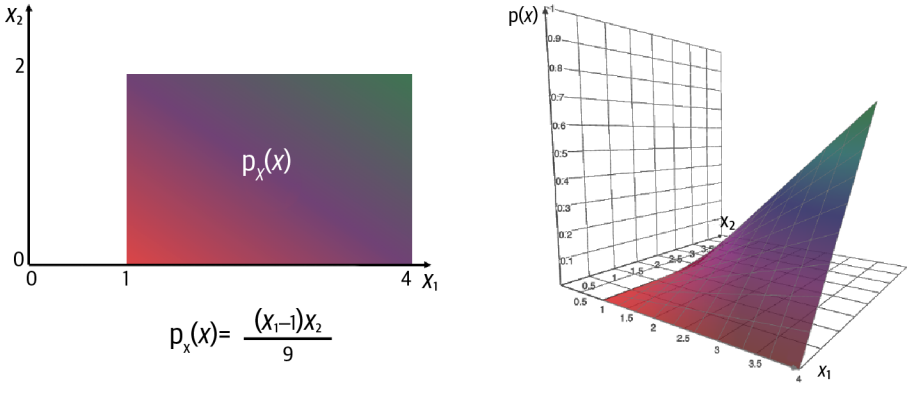

Figure 6-2. A probability distribution $p_X(x)$ defined over two dimensions, shown in 2D (left) and 3D (right)

This function integrates to 1 over the domain of the distribution (i.e., x_1 in the range [1, 4] and x_2 in the range [0, 2]), so it represents a well-defined probability distribution. We can write this as follows:

$$\int_0^2 \int_1^4 p_X(x) dx_1 dx_2 = 1$$

Let's say that we want to shift and scale this distribution so that it is instead defined over a unit square Z. We can achieve this by defining a new variable $z = (z_1, z_2)$ and a function f that maps each point in X to exactly one point in Z as follows:

$$\begin{aligned} z &= f(x) \\ z_1 &= \frac{x_1 - 1}{3} \\ z_2 &= \frac{x_2}{2} \end{aligned}$$

Note that this function is *invertible*. That is, there is a function g that maps every z back to its corresponding x. This is essential for a change of variables, as otherwise we cannot consistently map backward and forward between the two spaces. We can find g simply by rearranging the equations that define f, as shown in Figure 6-3.

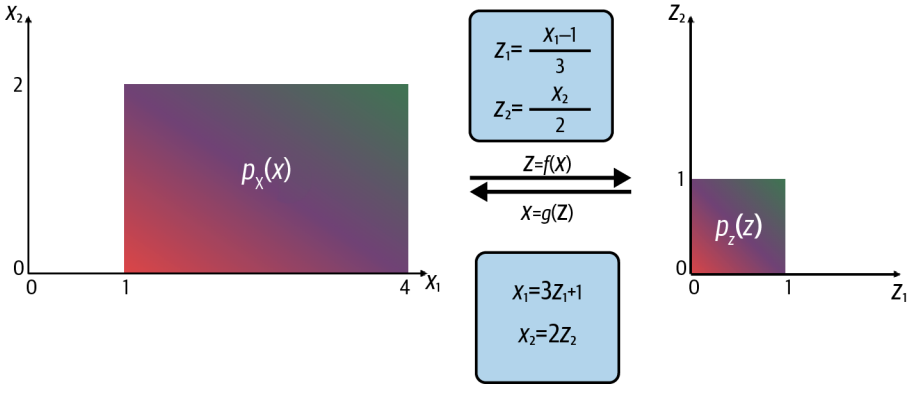

Figure 6-3. Changing variables between X and Z

We now need to see how the change of variables from X to Z affects the probability distribution $p_X(x)$. We can do this by plugging the equations that define g into $p_X(x)$ to transform it into a function $p_Z(z)$ that is defined in terms of z:

$$\begin{aligned} p_Z(z) &= \frac{\big((3z_1 + 1) - 1\big)(2z_2)}{9} \\ &= \frac{2z_1 z_2}{3} \end{aligned}$$

However, if we now integrate $p_Z(z)$ over the unit square, we can see that we have a problem!

$$\int_0^1 \int_0^1 \frac{2z_1 z_2}{3} dz_1 dz_2 = \frac{1}{6}$$

The transformed function $p_Z(z)$ is now no longer a valid probability distribution, because it only integrates to 1/6. If we want to transform our complex probability distribution over the data into a simpler distribution that we can sample from, we must ensure that it integrates to 1.

The missing factor of 6 is due to the fact that the domain of our transformed probability distribution is six times smaller than the original domain—the original rectangle X had area 6, and this has been compressed into a unit square Z that only has area 1. Therefore, we need to multiply the new probability distribution by a normalization factor that is equal to the relative change in area (or volume in higher dimensions).

Luckily, there is a way to calculate this volume change for a given transformation—it is the absolute value of the Jacobian determinant of the transformation. Let's unpack that!

The Jacobian Determinant

The *Jacobian* of a function $z = f(x)$ is the matrix of its first-order partial derivatives, as shown here:

$$\mathrm{J} = \frac{\partial z}{\partial x} = \begin{bmatrix} \frac{\partial z_1}{\partial x_1} & \cdots & \frac{\partial z_1}{\partial x_n} \\ \ddots & \vdots & \\ \frac{\partial z_m}{\partial x_1} & \cdots & \frac{\partial z_m}{\partial x_n} \end{bmatrix}$$

The best way to explain this is with our example. If we take the partial derivative of z_1 with respect to x_1, we obtain $\frac{1}{3}$. If we take the partial derivative of z_1 with respect to x_2, we obtain 0. Similarly, if we take the partial derivative of z_2 with respect to x_1, we obtain 0. Lastly, if we take the partial derivative of z_2 with respect to x_2, we obtain $\frac{1}{2}$.

Therefore, the Jacobian matrix for our function $f(x)$ is as follows:

$$J = \begin{pmatrix} \frac{1}{3} & 0 \\ 0 & \frac{1}{2} \end{pmatrix}$$

The *determinant* is only defined for square matrices and is equal to the signed volume of the parallelepiped created by applying the transformation represented by the matrix to the unit (hyper)cube. In two dimensions, this is therefore just the signed area of the parallelogram created by applying the transformation represented by the matrix to the unit square.

There is a general formula for calculating the determinant of a matrix with n dimensions, which runs in $\mathcal{O}\left(n^3\right)$ time. For our example, we only need the formula for two dimensions, which is simply as follows:

$$\det\begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc$$

Therefore, for our example, the determinant of the Jacobian is $\frac{1}{3} \times \frac{1}{2} - 0 \times 0 = \frac{1}{6}$. This is the scaling factor of 1/6 that we need to ensure that the probability distribution after transformation still integrates to 1!

By definition, the determinant is signed—that is, it can be negative. Therefore we need to take the absolute value of the Jacobian determinant in order to obtain the relative change of volume.

The Change of Variables Equation

We can now write down a single equation that describes the process for changing variables between X and Z. This is known as the *change of variables equation* (Equation 6-1).

Equation 6-1. The change of variables equation

$$p_X(x) = p_Z(z)\left|\det\left(\frac{\partial z}{\partial x}\right)\right|$$

How does this help us build a generative model? The key is understanding that if $p_Z(z)$ is a simple distribution from which we can easily sample (e.g., a Gaussian), then in theory, all we need to do is find an appropriate invertible function $f(x)$ that can map from the data X into Z and the corresponding inverse function $g(z)$ that can be

used to map a sampled z back to a point x in the original domain. We can use the preceding equation involving the Jacobian determinant to find an exact, tractable formula for the data distribution $p(x)$.

However, there are two major issues when applying this in practice that we first need to address!

Firstly, calculating the determinant of a high-dimensional matrix is computationally extremely expensive—specifically, it is $\mathcal{O}(n^3)$. This is completely impractical to implement in practice, as even small 32×32–pixel grayscale images have 1,024 dimensions.

Secondly, it is not immediately obvious how we should go about calculating the invertible function $f(x)$. We could use a neural network to find some function $f(x)$ but we cannot necessarily invert this network—neural networks only work in one direction!

To solve these two problems, we need to use a special neural network architecture that ensures that the change of variables function f is invertible and has a determinant that is easy to calculate.

We shall see how to do this in the following section using a technique called *real-valued non-volume preserving (RealNVP) transformations*.

RealNVP

RealNVP was first introduced by Dinh et al. in 2017.[1] In this paper the authors show how to construct a neural network that can transform a complex data distribution into a simple Gaussian, while also possessing the desired properties of being invertible and having a Jacobian that can be easily calculated.

Running the Code for This Example

The code for this example can be found in the Jupyter notebook located at *notebooks/06_normflow/01_realnvp/realnvp.ipynb* in the book repository.

The code has been adapted from the excellent RealNVP tutorial created by Mandolini Giorgio Maria et al. available on the Keras website.

The Two Moons Dataset

The dataset we will use for this example is created by the `make_moons` function from the Python library `sklearn`. This creates a noisy dataset of points in 2D that resemble two crescents, as shown in Figure 6-4.

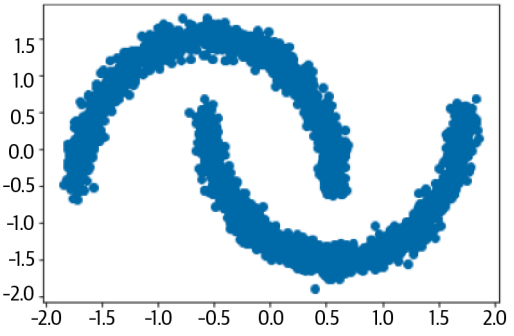

Figure 6-4. The two moons dataset in two dimensions

The code for creating this dataset is given in Example 6-1.

Example 6-1. Creating a moons dataset

```
data = datasets.make_moons(3000, noise=0.05)[0].astype("float32") ❶
norm = layers.Normalization()
norm.adapt(data)
normalized_data = norm(data) ❷
```

❶ Make a noisy, unnormalized moons dataset of 3,000 points.

❷ Normalize the dataset to have mean 0 and standard deviation 1.

We will build a RealNVP model that can generate points in 2D that follow a similar distribution to the two moons dataset. Whilst this is a very simple example, it will help us understand how a normalizing flow model works in practice, in fine detail.

First, however, we need to introduce a new type of layer, called a coupling layer.

Coupling Layers

A *coupling layer* produces a scale and translation factor for each element of its input. In other words, it produces two tensors that are exactly the same size as the input, one for the scale factor and one for the translation factor, as shown in Figure 6-5.

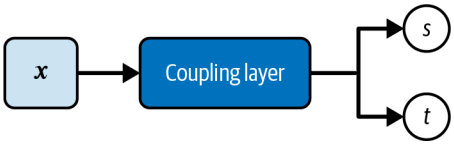

Figure 6-5. A coupling layer outputs two tensors that are the same shape as the input: a scaling factor (s) and a translation factor (t)

To build a custom `Coupling` layer for our simple example, we can stack `Dense` layers to create the scale output and a different set of `Dense` layers to create the translation output, as shown in Example 6-2.

For images, `Coupling` layer blocks use `Conv2D` layers instead of `Dense` layers.

Example 6-2. A `Coupling` layer in Keras

```
def Coupling():
    input_layer = layers.Input(shape=2) ❶

    s_layer_1 = layers.Dense(
        256, activation="relu", kernel_regularizer=regularizers.l2(0.01)
    )(input_layer) ❷
    s_layer_2 = layers.Dense(
        256, activation="relu", kernel_regularizer=regularizers.l2(0.01)
    )(s_layer_1)
    s_layer_3 = layers.Dense(
        256, activation="relu", kernel_regularizer=regularizers.l2(0.01)
    )(s_layer_2)
    s_layer_4 = layers.Dense(
        256, activation="relu", kernel_regularizer=regularizers.l2(0.01)
    )(s_layer_3)
    s_layer_5 = layers.Dense(
        2, activation="tanh", kernel_regularizer=regularizers.l2(0.01)
    )(s_layer_4) ❸

    t_layer_1 = layers.Dense(
        256, activation="relu", kernel_regularizer=regularizers.l2(0.01)
    )(input_layer) ❹
    t_layer_2 = layers.Dense(
        256, activation="relu", kernel_regularizer=regularizers.l2(0.01)
    )(t_layer_1)
    t_layer_3 = layers.Dense(
        256, activation="relu", kernel_regularizer=regularizers.l2(0.01)
    )(t_layer_2)
    t_layer_4 = layers.Dense(
        256, activation="relu", kernel_regularizer=regularizers.l2(0.01)
    )(t_layer_3)
    t_layer_5 = layers.Dense(
        2, activation="linear", kernel_regularizer=regularizers.l2(0.01)
    )(t_layer_4) ❺

    return models.Model(inputs=input_layer, outputs=[s_layer_5, t_layer_5]) ❻
```

❶ The input to the `Coupling` layer block in our example has two dimensions.

❷ The *scaling* stream is a stack of `Dense` layers of size 256.

❸ The final scaling layer is of size 2 and has `tanh` activation.

❹ The *translation* stream is a stack of `Dense` layers of size 256.

❺ The final translation layer is of size 2 and has `linear` activation.

❻ The `Coupling` layer is constructed as a Keras `Model` with two outputs (the scaling and translation factors).

Notice how the number of channels is temporarily increased to allow for a more complex representation to be learned, before being collapsed back down to the same number of channels as the input. In the original paper, the authors also use regularizers on each layer to penalize large weights.

Passing data through a coupling layer

The architecture of a coupling layer is not particularly interesting—what makes it unique is the way the input data is masked and transformed as it is fed through the layer, as shown in Figure 6-6.

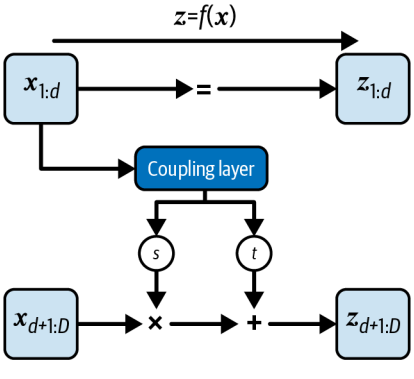

Figure 6-6. The process of transforming the input x through a coupling layer

Notice how only the first d dimensions of the data are fed through to the first coupling layer—the remaining $D - d$ dimensions are completely masked (i.e., set to zero). In our simple example with $D = 2$, choosing $d = 1$ means that instead of the coupling layer seeing two values, (x_1, x_2), the layer sees $(x_1, 0)$.

The outputs from the layer are the scale and translation factors. These are again masked, but this time with the *inverse* mask to previously, so that only the second halves are let through—i.e., in our example, we obtain $(0, s_2)$ and $(0, t_2)$. These are then applied element-wise to the second half of the input x_2 and the first half of the input x_1 is simply passed straight through, without being updated at all. In summary, for a vector with dimension D where $d < D$, the update equations are as follows:

$$\begin{aligned} z_{1:d} &= x_{1:d} \\ z_{d+1:D} &= x_{d+1:D} \odot \exp\left(s\left(x_{1:d}\right)\right) + t\left(x_{1:d}\right) \end{aligned}$$

You may be wondering why we go to the trouble of building a layer that masks so much information. The answer is clear if we investigate the structure of the Jacobian matrix of this function:

$$\frac{\partial z}{\partial x} = \begin{bmatrix} \mathbf{I} & 0 \\ \dfrac{\partial z_{d+1:D}}{\partial x_{1:d}} & \operatorname{diag}\left(\exp\left[s\left(x_{1:d}\right)\right]\right) \end{bmatrix}$$

The top-left $d \times d$ submatrix is simply the identity matrix, because $z_{1:d} = x_{1:d}$. These elements are passed straight through without being updated. The top-right submatrix is therefore 0, because $z_{1:d}$ is not dependent on $x_{d+1:D}$.

The bottom-left submatrix is complex, and we do not seek to simplify this. The bottom-right submatrix is simply a diagonal matrix, filled with the elements of $\exp\left(s\left(x_{1:d}\right)\right)$, because $z_{d+1:D}$ is linearly dependent on $x_{d+1:D}$ and the gradient is dependent only on the scaling factor (not on the translation factor). Figure 6-7 shows a diagram of this matrix form, where only the nonzero elements are filled in with color.

Notice how there are no nonzero elements above the diagonal—for this reason, this matrix form is called *lower triangular*. Now we see the benefit of structuring the matrix in this way—the determinant of a lower-triangular matrix is just equal to the product of the diagonal elements. In other words, the determinant is not dependent on any of the complex derivatives in the bottom-left submatrix!

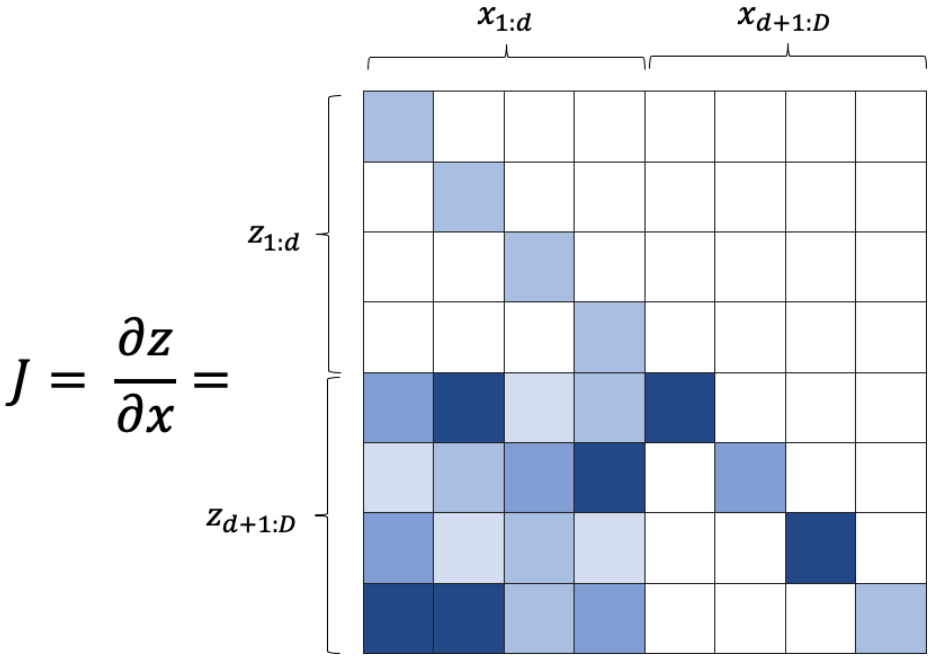

Figure 6-7. The Jacobian matrix of the transformation—a lower triangular matrix, with determinant equal to the product of the elements along the diagonal

Therefore, we can write the determinant of this matrix as follows:

$$\det(\mathrm{J}) = \exp\left[\sum_j s(x_{1:d})_j\right]$$

This is easily computable, which was one of the two original goals of building a normalizing flow model.

The other goal was that the function must be easily invertible. We can see that this is true as we can write down the invertible function just by rearranging the forward equations, as follows:

$$\begin{aligned} x_{1:d} &= z_{1:d} \\ x_{d+1:D} &= \left(z_{d+1:D} - t(x_{1:d})\right) \odot \exp\left(-s(x_{1:d})\right) \end{aligned}$$

The equivalent diagram is shown in Figure 6-8.

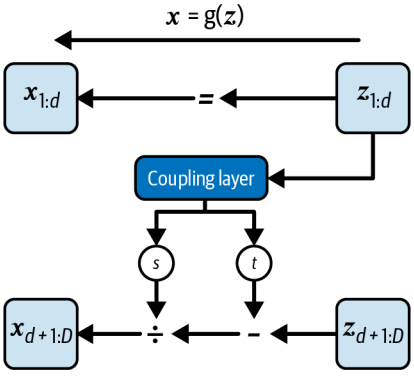

Figure 6-8. The inverse function x = g(z)

We now have almost everything we need to build our RealNVP model. However, there is one issue that still remains—how should we update the first d elements of the input? Currently they are left completely unchanged by the model!

Stacking coupling layers

To resolve this problem, we can use a really simple trick. If we stack coupling layers on top of each other but alternate the masking pattern, the layers that are left unchanged by one layer will be updated in the next. This architecture has the added benefit of being able to learn more complex representations of the data, as it is a deeper neural network.

The Jacobian of this composition of coupling layers will still be simple to compute, because linear algebra tells us that the determinant of a matrix product is the product of the determinants. Similarly, the inverse of the composition of two functions is just the composition of the inverses, as shown in the following equations:

$$\det(\mathrm{A} \cdot \mathrm{B}) = \det(\mathrm{A})\det(\mathrm{B})$$
$$(f_b \circ f_a)^{-1} = f_a^{-1} \circ f_b^{-1}$$

Therefore, if we stack coupling layers, flipping the masking each time, we can build a neural network that is able to transform the whole input tensor, while retaining the essential properties of having a simple Jacobian determinant and being invertible. Figure 6-9 shows the overall structure.

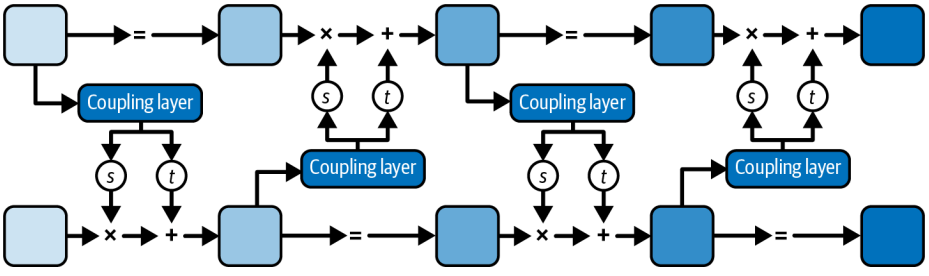

Figure 6-9. Stacking coupling layers, alternating the masking with each layer

Training the RealNVP Model

Now that we have built the RealNVP model, we can train it to learn the complex distribution of the two moons dataset. Remember, we want to minimize the negative log-likelihood of the data under the model $-\log p_X(x)$. Using Equation 6-1, we can write this as follows:

$$-\log p_X(x) = -\log p_Z(z) - \log \left| \det \left(\frac{\partial z}{\partial x} \right) \right|$$

We choose the target output distribution $p_Z(z)$ of the forward process f to be a standard Gaussian, because we can easily sample from this distribution. We can then transform a point sampled from the Gaussian back into the original image domain by applying the inverse process g, as shown in Figure 6-10.

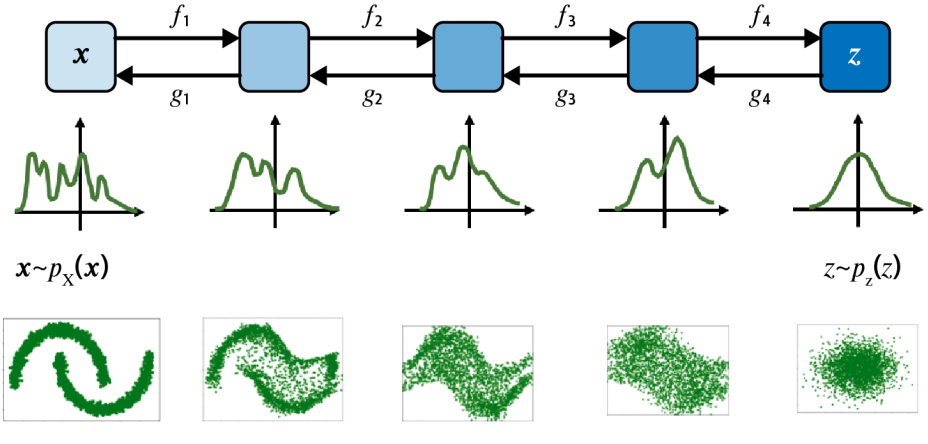

Figure 6-10. Transforming between the complex distribution $p_X(x)$ and a simple Gaussian $p_Z(z)$ in 1D (middle row) and 2D (bottom row)

Example 6-3 shows how to build a RealNVP network, as a custom Keras `Model`.

Example 6-3. Building the RealNVP model in Keras

```
class RealNVP(models.Model):
    def __init__(self, input_dim, coupling_layers, coupling_dim, regularization):
        super(RealNVP, self).__init__()
        self.coupling_layers = coupling_layers
        self.distribution = tfp.distributions.MultivariateNormalDiag(
            loc=[0.0, 0.0], scale_diag=[1.0, 1.0]
        ) ❶
        self.masks = np.array(
            [[0, 1], [1, 0]] * (coupling_layers // 2), dtype="float32"
        ) ❷
        self.loss_tracker = metrics.Mean(name="loss")
        self.layers_list = [
            Coupling(input_dim, coupling_dim, regularization)
            for i in range(coupling_layers)
        ] ❸

    @property
    def metrics(self):
        return [self.loss_tracker]

    def call(self, x, training=True):
        log_det_inv = 0
        direction = 1
        if training:
            direction = -1
        for i in range(self.coupling_layers)[::direction]: ❹
            x_masked = x * self.masks[i]
            reversed_mask = 1 - self.masks[i]
            s, t = self.layers_list[i](x_masked)
            s *= reversed_mask
            t *= reversed_mask
            gate = (direction - 1) / 2
            x = (
                reversed_mask
                * (x * tf.exp(direction * s) + direction * t * tf.exp(gate * s))
                + x_masked
            ) ❺
            log_det_inv += gate * tf.reduce_sum(s, axis = 1) ❻
        return x, log_det_inv

    def log_loss(self, x):
        y, logdet = self(x)
        log_likelihood = self.distribution.log_prob(y) + logdet ❼
        return -tf.reduce_mean(log_likelihood)

    def train_step(self, data):
        with tf.GradientTape() as tape:
```

```
            loss = self.log_loss(data)
        g = tape.gradient(loss, self.trainable_variables)
        self.optimizer.apply_gradients(zip(g, self.trainable_variables))
        self.loss_tracker.update_state(loss)
        return {"loss": self.loss_tracker.result()}

    def test_step(self, data):
        loss = self.log_loss(data)
        self.loss_tracker.update_state(loss)
        return {"loss": self.loss_tracker.result()}

model = RealNVP(
    input_dim = 2
    , coupling_layers= 6
    , coupling_dim = 256
    , regularization = 0.01
)

model.compile(optimizer=optimizers.Adam(learning_rate=0.0001))

model.fit(
    normalized_data
    , batch_size=256
    , epochs=300
)
```

❶ The target distribution is a standard 2D Gaussian.

❷ Here, we create the alternating mask pattern.

❸ A list of `Coupling` layers that define the RealNVP network.

❹ In the main `call` function of the network, we loop over the `Coupling` layers. If `training=True`, then we move forward through the layers (i.e., from data to latent space). If `training=False`, then we move backward through the layers (i.e., from latent space to data).

❺ This line describes both the forward and backward equations dependent on the `direction` (try plugging in `direction = -1` and `direction = 1` to prove this to yourself!).

❻ The log determinant of the Jacobian, which we need to calculate the loss function, is simply the sum of the scaling factors.

❼ The loss function is the negative sum of the log probability of the transformed data, under our target Gaussian distribution and the log determinant of the Jacobian.

Analysis of the RealNVP Model

Once the model is trained, we can use it to transform the training set into the latent space (using the forward direction, f) and, more importantly, to transform a sampled point in the latent space into a point that looks like it could have been sampled from the original data distribution (using the backward direction, g).

Figure 6-11 shows the output from the network before any learning has taken place—the forward and backward directions just pass information straight through with hardly any transformation.

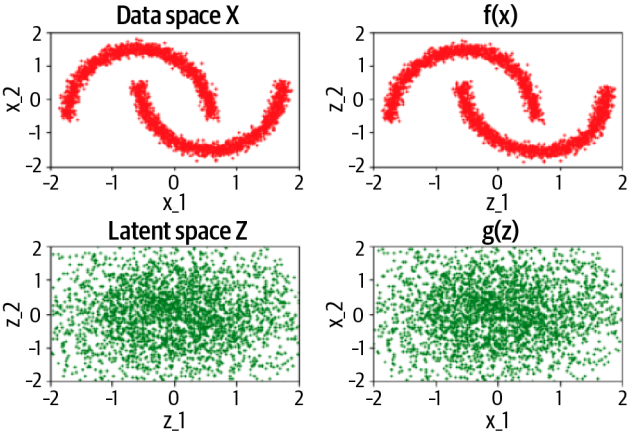

Figure 6-11. The RealNVP model inputs (left) and outputs (right) before training, for the forward process (top) and the reverse process (bottom)

After training (Figure 6-12), the forward process is able to convert the points from the training set into a distribution that resembles a Gaussian. Likewise, the backward process can take points sampled from a Gaussian distribution and map them back to a distribution that resembles the original data.

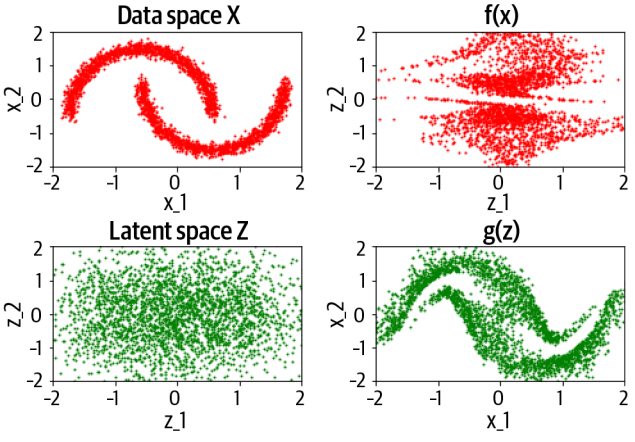

Figure 6-12. The RealNVP model inputs (left) and outputs (right) after training, for the forward process (top) and the reverse process (bottom)

The loss curve for the training process is shown in Figure 6-13.

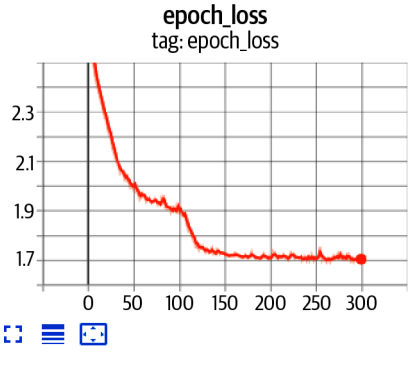

Figure 6-13. The loss curve for the RealNVP training process

This completes our discussion of RealNVP, a specific case of a normalizing flow generative model. In the next section, we'll cover some modern normalizing flow models that extend the ideas introduced in the RealNVP paper.

Other Normalizing Flow Models

Two other successful and important normalizing flow models are *GLOW* and *FFJORD*. The following sections describe the key advancements they made.

GLOW

Presented at NeurIPS 2018, GLOW was one of the first models to demonstrate the ability of normalizing flows to generate high-quality samples and produce a meaningful latent space that can be traversed to manipulate samples. The key step was to replace the reverse masking setup with invertible 1×1 convolutional layers. For example, with RealNVP applied to images, the ordering of the channels is flipped after each step, to ensure that the network gets the chance to transform all of the input. In GLOW a 1×1 convolution is applied instead, which effectively acts as a general method to produce any permutation of the channels that the model desires. The authors show that even with this addition, the distribution as a whole remains tractable, with determinants and inverses that are easy to compute at scale.

Figure 6-14. Random samples from the GLOW model (source: Kingma and Dhariwal, 2018)[2]

FFJORD

RealNVP and GLOW are discrete time normalizing flows—that is, they transform the input through a discrete set of coupling layers. FFJORD (Free-Form Continuous Dynamics for Scalable Reversible Generative Models), presented at ICLR 2019, shows how it is possible to model the transformation as a continuous time process (i.e., by taking the limit as the number of steps in the flow tends to infinity and the step size tends to zero). In this case, the dynamics are modeled using an ordinary differential equation (ODE) whose parameters are produced by a neural network (f_θ). A black-box solver is used to solve the ODE at time t_1—i.e., to find z_1 given some initial point z_0 sampled from a Gaussian at t_0, as described by the following equations:

$$
\begin{aligned}
z_0 &\sim p(z_0) \\
\frac{\partial z(t)}{\partial t} &= f_\theta(x(t), t) \\
x &= z_1
\end{aligned}
$$

A diagram of the transformation process is shown in Figure 6-15.

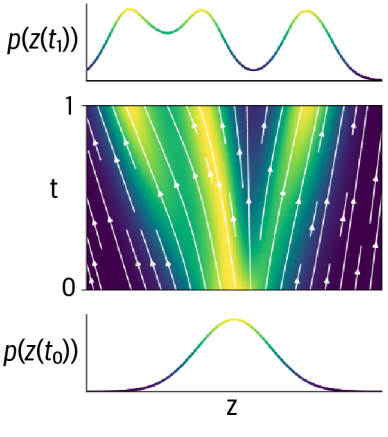

Figure 6-15. FFJORD models the transformation between the data distribution and a standard Gaussian via an ordinary differential equation, parameterized by a neural network (source: Will Grathwohl et al., 2018)[3]

Summary

In this chapter we explored normalizing flow models such as RealNVP, GLOW, and FFJORD.

A normalizing flow model is an invertible function defined by a neural network that allows us to directly model the data density via a change of variables. In the general case, the change of variables equation requires us to calculate a highly complex Jacobian determinant, which is impractical for all but the simplest of examples.

To sidestep this issue, the RealNVP model restricts the form of the neural network, such that it adheres to the two essential criteria: it is invertible and has a Jacobian determinant that is easy to compute.

It does this through stacking coupling layers, which produce scale and translation factors at each step. Importantly, the coupling layer masks the data as it flows through the network, in a way that ensures that the Jacobian is lower triangular and therefore has a simple-to-compute determinant. Full visibility of the input data is achieved through flipping the masks at each layer.

By design, the scale and translation operations can be easily inverted, so that once the model is trained it is possible to run data through the network in reverse. This means that we can target the forward transformation process toward a standard Gaussian, which we can easily sample from. We can then run the sampled points backward through the network to generate new observations.

The RealNVP paper also shows how it is possible to apply this technique to images, by using convolutions inside the coupling layers, rather than densely connected layers. The GLOW paper extended this idea to remove the necessity for any hardcoded permutation of the masks. The FFJORD model introduced the concept of continuous time normalizing flows, by modeling the transformation process as an ODE defined by a neural network.

Overall, we have seen how normalizing flows are a powerful generative modeling family that can produce high-quality samples, while maintaining the ability to tractably describe the data density function.

References

1. Laurent Dinh et al., "Density Estimation Using Real NVP," May 27, 2016, *https://arxiv.org/abs/1605.08803v3*.

2. Diedrick P. Kingma and Prafulla Dhariwal, "Glow: Generative Flow with Invertible 1x1 Convolutions," July 10, 2018, *https://arxiv.org/abs/1807.03039*.

3. Will Grathwohl et al., "FFJORD: Free-Form Continuous Dynamics for Scalable Reversible Generative Models," October 22, 2018, *https://arxiv.org/abs/1810.01367*.