**Unsupervised Learning with Normalising Flows 🌊**

[[Hylke C. Donker](https://medium.com/@hylke.donker?source=post_page-----59ebc463c367--------------------------------)](https://medium.com/@hylke.donker?source=post_page-----59ebc463c367--------------------------------)

[Hylke C. Donker](https://medium.com/@hylke.donker?source=post_page-----59ebc463c367--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Fea1bfe4db7a8&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40hylke.donker%2Funsupervised-learning-with-normalising-flows-59ebc463c367&user=Hylke+C.+Donker&userId=ea1bfe4db7a8&source=post_page-ea1bfe4db7a8----59ebc463c367---------------------post_header-----------)

9 min read

·

Jul 15

29

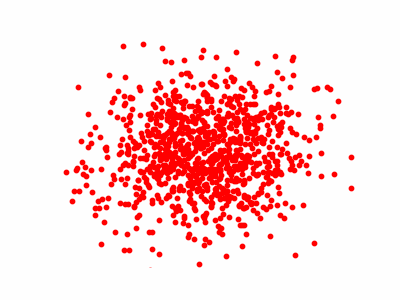


Fig. 1: Normalising flow gradually solving the moons dataset. Image by Author.

Without a doubt, 2023 is the year of generative AI. These unsupervised models derive their name from what they can do: generate new data. Probably you already know two: GPT (generative pre-trained transformer) famous from its’ chat app, and Stable Diffusion, a text-to-image generator that can turn short text descriptions into stunning images.

In this post, I’ll cover a specific class of generative models called normalising flows. 🌊

**Historic Background**

The AI community has had a long tradition of generative AI. Well before the invention of generative adverserial networks (GAN) [1] and variational auto encoders (VAE) [2], AI pioneers Geoffrey Hinton and Yann LeCun popularised energy-based models back in the early 00’s [3,4]. These probabilistic predecessors paved the way for *deep* energy-based networks [5], planting the seeds for [AlexNet](https://en.wikipedia.org/wiki/AlexNet" \t "_blank) — famously beating the [ImageNet](https://www.image-net.org/) visual recognition competition by a large margin. These developments eventually led to the Cambrian AI explosion we’re seeing now. With unsupervised learning back at the centre stage, Google, Facebook, and Microsoft are now racing to build the largest generative model.

**Objectives**

* Applications of normalising flows.
* What are normalising flows?
* Implementing a normalising flow — RealNVP [6] — from scratch.

**Applications of Normalising Flows**

Because normalising flows are generative models, they inherit the capacity to generate new examples, complete existing examples, do inference, and make predictions. The distinguishing characteristic of normalising flows is that the likelihood *p*(***x***) of an example ***x*** is easy to compute. This makes normalising flows particularly powerful for:

**Variational inference**: How do you choose your surrogate (or, the variational distribution) *q*? Normalising flows are an excellent choice for *q* because they can be made very flexible. Increasing the capacity of your normalising flow — i.e., expanding the set of distributions the model can accommodate — reduces the approximation error. (See [Variational Inference: The Basics](https://towardsdatascience.com/variational-inference-the-basics-f70ac511bcea), to learn more.)

**Representation learning**: By construction, normalising flows *deterministically* map examples, ***x***, to latent — usually Gaussian — space ***z***. Examples in Gaussian space are typically (i) easy to visualise, (ii) simple to quantify with traditional statistics, and (iii) distances are intuitive to interpret. All these aspects make for a great latent representation.

**Anomaly detection**: Because normalising flows are unsupervised, there is no need to collect rare *abnormal* examples. Instead, to detect outliers you flag unlikely samples — with low *p*(***x***) — as anomalies. Moreover, because of the one-to-one correspondence between ***x*** and ***z***, you can inspect anomalous examples to see how ***z*** semantically differed from expected.

**What are Normalising Flows?**

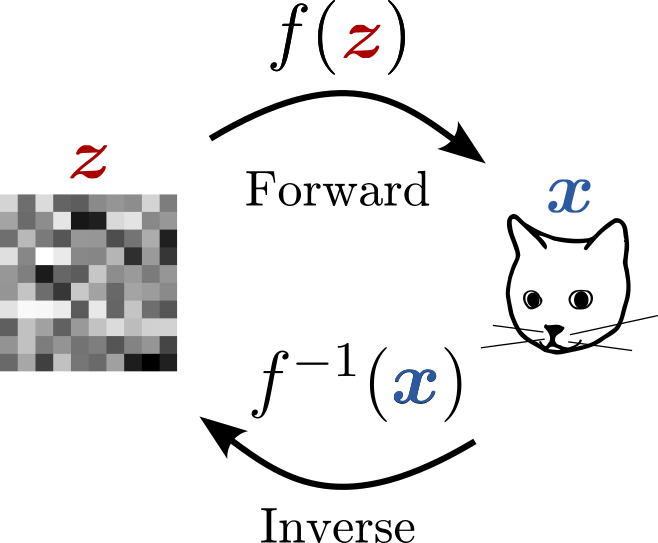


Fig. 2: A normalising flow transforming Gaussian noise, **z**, into an image of a cat, **x**. Image by Author.

In essence, normalising flows 🌊 are functions, *f*(***z***) = ***x***, with an inverse. The idea is to let ***x*** be the example to learn (e.g., image of a cat) and ***z*** the corresponding latent representation (Fig. 2). Usually, we choose a distribution *p*(***z***) that is easy to sample from (e.g., Gaussian). We then look for *f* that can transform ***z*** into ***x***.

One subtlety though: the flow from latent ***z*** to data ***x***curves the underlying space. To relate the data likelihood, *p*(***x***), back to *p*(***z***) requires a volume correction to account for the space warping of *f*(***z***). This correction, the inverse determinant of the Jacobian,

*p*(***x***) = *p*(***z***) |det ∂*f*⁻¹/∂***x***|,

ensures that the distributions remain normalised. Hence, the name *normalising* flows.

While the choice of *f* comes in many flavours — Real NVP [6], masked autoregressive flows [7] and inverse autoregressive flows [8]—they share two key requirements:

* (i) Inverting *f* is easy.
* (ii) The volume correction — the determinant of the Jacobian — is cheap to compute.

Let’s showcase one particularly simple *f* that satisfies both requirements: real-valued non-volume preserving transformation (Real NVP) [6].

**Normalising flow from scratch: Real NVP**

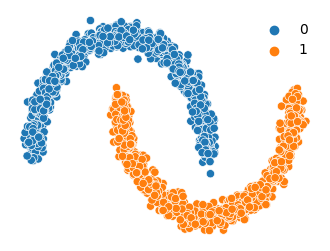


Fig. 3: The two moons dataset from sci-kit learn, coloured by label value. Image by Author.

Real NVP is a normalising flow that is conceptually simple, easy to implement, and with great baseline performance.

In this tutorial (inspired by [Eric Jang’s notebook](https://github.com/ericjang/nf-jax/blob/master/nf-tutorial-jax.ipynb)), you'll train a model that generates samples from sci-kit learn’s [moons dataset](https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_moons.html#sklearn.datasets.make_moons) (Fig. 3). You can follow along with the [Colab notebook](https://colab.research.google.com/drive/1vXLAfsli2i09skbNsqQUVIng4AoTvfHq?usp=sharing" \t "_blank).

**Model definition**

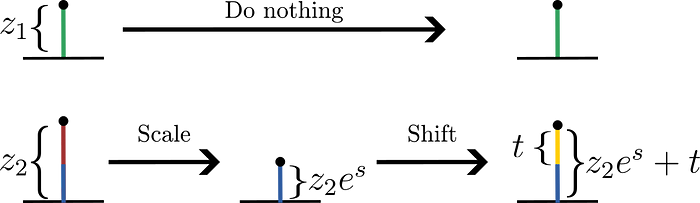


Fig. 4: Shift and scale transformation. Image by Author.

In essence, the forward function *f* of RealNVP is a sequence of block-wise transformations. In each step, we partition ***z*** in ***z***₁ and ***z***₂ and then transform (Fig. 4):

***x***₁ = ***z***₁,

***x***₂ = ***z***₂ exp[***s***(***z***₁)] + ***t***(***z***₁)*,*

where both the scaling factor ***s***(***z***₁) and the shift ***t***(***z***₁) are functions. The block-wise form makes the transformation easy to invert and the volume correction cheap to compute. RealNVP stacks multiple transformations in sequence, *f*(***z***)= …∘*f*₂∘*f*₁ , to make increasingly complicated transformations.

In Python, a single transformation with a 50:50 split reads as follows:

import jax.numpy as jnp  
  
# Make a 50:50 split.  
z\_1, z\_2 = jnp.split(z, 2)  
  
# Apply shift and scale transformation to lower block.  
x\_1 = z\_1  
x\_2 = z\_2 \* jnp.exp(scale) + shift  
x = jnp.concatenate([x\_1, x\_2])

Here, we used the [JAX](https://jax.readthedocs.io/en/latest/) [9] implementation of NumPy so that later on we can take gradients and compute the inverse. (To learn the bare essentials, read [JAX: Fast as PyTorch, Simple as NumPy](https://medium.com/@hylke.donker/jax-fast-as-pytorch-simple-as-numpy-a0c14893a738).)

In practice, we make both the scaling factor ***s***(***z***₁) and the translation ***t***(***z***₁) neural networks. 🤖 This is easily done with the JAX neural network library [Haiku](https://dm-haiku.readthedocs.io/en/latest/). For a one minute primer on Haiku, I suggest my post [Object Orient Programming in JAX with Haiku](https://medium.com/@hylke.donker/object-orient-programming-in-jax-with-haiku-f59dc08c712c).

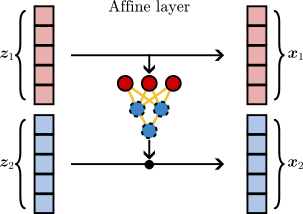


Fig. 5: A single affine layer that uses neural networks for computing the scale and shift. Image by Author.

We let the neural network simultaneously predict the shift ***t***(***z***₁) and the scale ***s***(***z***₁), thus producing n\_features predictions in total. It turns out that a vanilla three-layer neural net works well for our moons dataset: haiku.nets.MLP(output\_sizes=[512, 128, n\_features]).

Let’s put these two code snippets together in a Haiku module (Fig. 5) for re-usability.

import haiku as hk  
  
class Affine(hk.Module):  
 """A shift and scale layer."""  
 def \_\_init\_\_(self, flip: bool = False):  
 super().\_\_init\_\_()  
 self.flip = flip # Permute lower and upper blocks.  
  
 def \_\_call\_\_(self, z):  
 """Do affine transformation on lower block."""  
 # Make a 50:50 split.  
 z\_1, z\_2 = jnp.split(z, 2)  
 if self.flip:  
 z\_1, z\_2 = z\_2, z\_1  
  
 # Predict shift and scale parameters using neural network.  
 n\_features = z.shape[-1]  
 neural\_net = hk.nets.MLP(output\_sizes=[512, 128, n\_features])  
 shift\_scale = neural\_net(z\_1)  
 shift, scale = jnp.split(shift\_scale, 2, axis=-1)  
  
 # Apply shift and scale transformation to lower block.  
 x\_1 = z\_1  
 x\_2 = z\_2 \* jnp.exp(scale) + shift  
 return jnp.concatenate([x\_1, x\_2])

Here, we anticipated that we’re going to alternate the block that is being transformed (flip=True) when chaining multiple layers in sequence.

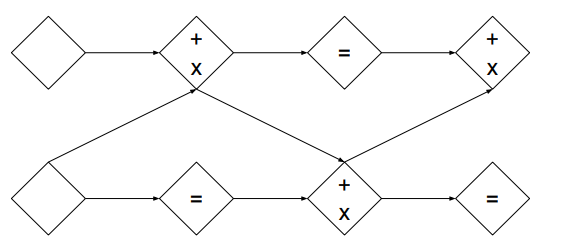


Fig. 6: We alternate the partitions so that all variables participate in shift and scale transformations. Image taken from Ref. [6].

By flipping the blocks we ensure that all variables participate in the transformation as a whole (Fig. 6).

In total, our normalising flow network *f*(***z***) will comprise of four affine transformations. To chain the four layers, we’ll use Haiku’s [hk.Sequential](https://dm-haiku.readthedocs.io/en/latest/api.html" \l "sequential" \t "_blank) module. Putting everything together, our forward function *f*(***z***) looks as follows:

@hk.without\_apply\_rng  
@hk.transform  
def forward(z):  
 flow = hk.Sequential(layers=[  
 Affine(),  
 Affine(flip=True),  
 Affine(),  
 Affine(flip=True),  
 ])  
 return flow(z)

Here, we converted forward to a pure JAX function with [hk.transform](https://dm-haiku.readthedocs.io/en/latest/api.html" \l "haiku.transform" \t "_blank) and removed the random number generator argument from forward.apply with [hk.without\_apply\_rng](https://dm-haiku.readthedocs.io/en/latest/api.html" \l "haiku.without_apply_rng" \t "_blank).

This completes the specification of the model.

**Loss function**

We train the model by maximising the log likelihood:

ln*p*(***x***) = ln *p*(***z***) + ln |det ∂*f*⁻¹/∂***x***|,

which requires two ingredients. We need to (i) compute ***z*** = *f*⁻¹(***x***) and (ii) evaluate the inverse log determinant Jacobian: ln |det ∂*f*⁻¹/∂***x***|. Note that we haven’t explicitly defined *f*⁻¹(***x***). Fortunately, we don’t have to with [Oryx](https://github.com/jax-ml/oryx) (pip install oryx)[10]. Oryx has a function inverse\_and\_idlj that can automatically infer *f*⁻¹(***x***) and ln |det ∂*f*⁻¹/∂***x***| given an invertible JAX function *f*(***z***) [10].

Below, we implement ln*p*(***x***) called log\_prob where params contains the weights and biases of our flow network.

from oryx.core import inverse\_and\_ildj  
  
# Make f(z) invertible by combining params and forward.apply.  
f\_inv\_and\_ildj = inverse\_and\_ildj(lambda p, z: [p, forward.apply(p, z)])  
  
@partial(vmap, in\_axes=[None, 0])  
def log\_prob(params, x):  
 """Log-likelihood of a single example x."""  
 # Compute: z = f⁻¹(x) and ln |det ∂f⁻¹/∂x|.  
 (\_, z), inv\_log\_det\_jac = f\_inv\_and\_ildj(params, x)  
 logp\_z = jsp.stats.norm.logpdf(z)  
 logp\_x = logp\_z.sum() + inv\_log\_det\_jac  
 return logp\_x

Here, we used JAX’s vmap to vectorise log\_prob across a batch **[*x***₁, ***x***₂,..] of examples (second argument x). Finally, the loss is defined as the negative log-likelihood of a batch:

def loss(params, x\_batch):  
 """Negative log likelihood: ℒ = 1/m ∑ ln p(xᵢ)."""  
 return -jnp.mean(log\_prob(params, x\_batch))

Almost there! With the loss at our fingertips, one task remains: optimising the loss.

**Model Training**

We optimise the loss using good ol’ gradient descent. First, let Haiku make a single pass through the network with a Gaussian to initialise the parameters params of our flow network.

# Random key iterator.  
key\_seq = hk.PRNGSequence(42)  
  
# Initialise parameters of flow network.  
n\_features = 2  
z\_forward = random.normal(next(key\_seq), shape=[n\_features])  
params = forward.init(next(key\_seq), z\_forward)

Next, we use JAX’s jax.value\_and\_grad to compute the loss and gradient in one go. The actual gradient updates are made using the off-the-shelf Adam optimiser from Optax (pip install optax). Given the weights and biases params, an Optax optimiser is initialised in two lines:

import optax  
  
# Compute loss and gradients in one go and jit to make it snappy.  
loss\_and\_grad = jax.jit(jax.value\_and\_grad(loss))  
  
# Initialise optimiser.  
optimizer = optax.adam(learning\_rate=5e-4)  
opt\_state = optimizer.init(params)

Finally, lets write down the training loop. We take 10k gradient updates — one per mini-batch of 256 examples — and use the optimiser to compute the updates. In each iteration, we renew the parameters with optax.apply\_updates.

for i in range(10\_000):  
 x\_batch = X[jax.random.choice(next(key\_seq), X.shape[0], [256])]  
 neg\_log\_likel, grads = loss\_and\_grad(params, x\_batch)  
 updates, opt\_state = optimizer.update(grads, opt\_state)  
 params = optax.apply\_updates(params, updates)

Congrats! 🎉 You have successfully implemented and trained a normalising flow — RealNVP — from scratch!

**Results**

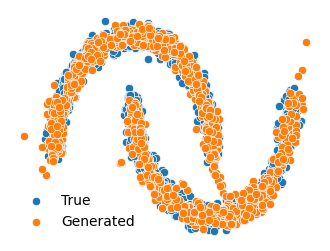


Fig. 7: Comparison of true and generated samples by RealNVP. Image by Author.

Let’s take a step back and appreciate what we’ve built. Fig. 7 compares the generated examples with the original dataset. They match quite well! In Fig. 1 and the accompanying [Colab notebook](https://colab.research.google.com/drive/1vXLAfsli2i09skbNsqQUVIng4AoTvfHq?usp=sharing" \t "_blank) with the complete code you can see how a Gaussian is sequentially transformed into the two moons dataset with the model we trained.

**Conclusion**

Generative AI is booming! 💥 In this post, we studied a specific class of generative models: normalising flows. 🌊 These unsupervised learners are characterised by (i) a tractable likelihood *p*(***x***) and (ii) a *deterministic* example ***x*** to latent space ***z*** mapping. Both aspects make normalising flows attractive for variational inference, representation learning, and anomaly detection. Normalising flows are surprisingly simple: with the help of [Haiku](https://github.com/deepmind/dm-haiku), implementing RealNVP [6] from scratch takes a few lines of JAX. This makes normalising flows a great addition to the data scientist’s toolbox.

**Acknowledgement**

I would like to thank Dorien Neijzen and Lee-Yen Wang for proofreading.

**References**

[1] Goodfellow, Ian, et al. “Generative adversarial networks.” *Communications of the ACM* 63.11 (2020): 139–144.

[2] Diederik P. Kingma and Max Welling. [“Auto-encoding variational Bayes.”](https://arxiv.org/abs/1312.6114) ICLR, 2014.

[3] Hinton, Geoffrey E. “Training products of experts by minimizing contrastive divergence.” *Neural computation* 14.8, pp. 1771–1800 (2002).

[4] LeCun, Yann, et al. “A tutorial on energy-based learning.” *Predicting structured data* 1.0 (2006).

[5] Salakhutdinov & Larochelle. [“Efficient learning of deep Boltzmann machines.”](http://proceedings.mlr.press/v9/salakhutdinov10a/salakhutdinov10a.pdf), *PLMR* (2010).

[6] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. [“Density estimation using Real NVP.”](https://arxiv.org/abs/1605.08803), *ICLR* (2017).

[7] George Papamakarios, Iain Murray, and Theo Pavlakou. [“Masked autoregressive flow for density estimation.”](https://arxiv.org/abs/1705.07057), *NIPS* (2017).

[8] Diederik P. Kingma, et al. [“Improved variational inference with inverse autoregressive flow.”](https://arxiv.org/abs/1606.04934), *NIPS* (2016).

[9] Frostig, Roy, Matthew James Johnson, and Chris Leary. “[Compiling machine learning programs via high-level tracing.](https://mlsys.org/Conferences/doc/2018/146.pdf)”, *Systems for Machine Learning* 4.9 (2018).

[10] Sharad Vikram, Matthew D. Hoffman, Alexey Radul, James Bradbury, Matthew Johnson, Sergio Guadarrama, “[Probabilistic Programming by Transformation in JAX](https://probprog.cc/2020/assets/posters/thu/52.pdf)”, *PROBPROG* poster (2020).