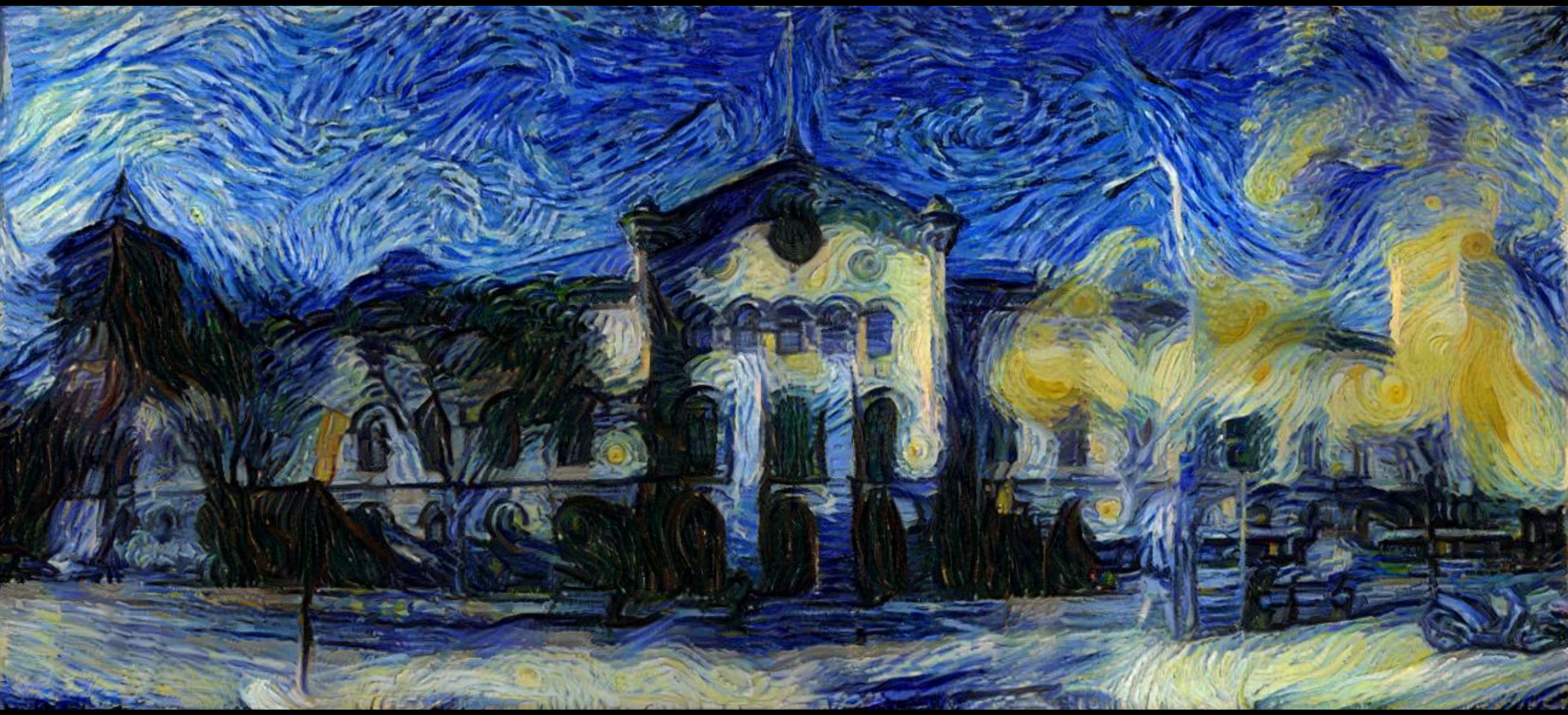
UNIVERSITAT DE BARCELONA

Jordi Vitrià



Deep Learning

Generative Models

Generative Models

Given a dataset X of examples $x \in X$, which we assume to have been drawn independently from some underlying distribution $p_X(x)$, a generative model can learn to **approximate** this distribution $p_X(x)$.

Such a model could be used to **generate** new samples that look like they could have been part of the original dataset and/or to infer the **likelihood** of an example.

We distinguish *implicit* and *explicit* generative models:

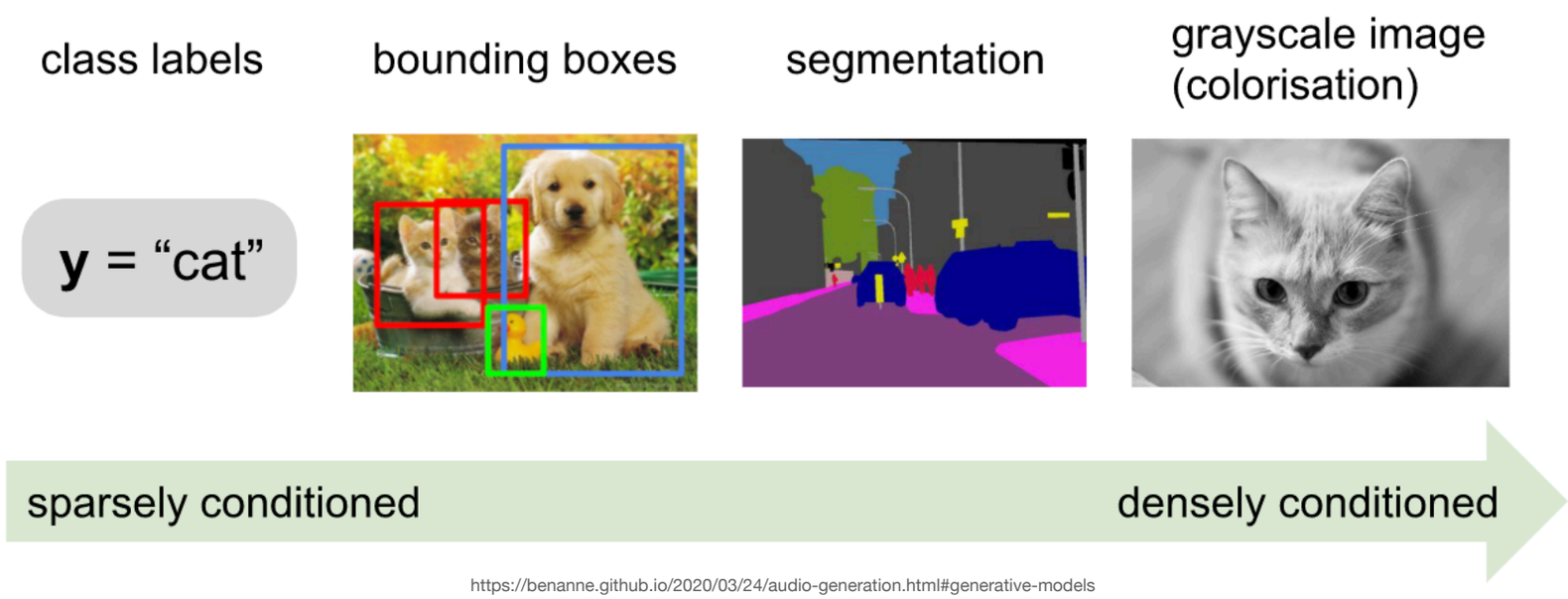
- An i**mplicit model** can produce new samples $x \sim p_X(x)$, but cannot be used to infer the likelihood of an example (i.e. we cannot tractably compute $p_X(x)$ given x).
- If we have an **explicit model**, we can do this, though sometimes only up to an unknown normalizing constant.

Conditional Generative Models

Generative models become more practically useful when we can exert some influence over the samples we draw from them.

We can do this by providing a **conditioning signal** c, which contains side information about the kind of samples we want to generate. The model is then fit to the conditional distribution $p_X(x \mid c)$ instead of $p_X(x)$.

Conditioning signals can take many shapes or forms, and it is useful to distinguish different levels of information content. The generative modeling problem becomes easier if the conditioning signal c is richer, because it reduces uncertainty about x.



https://benanne.github.io/2020/03/24/audio-generation.html#generative-models

Generative Models Families

- Likelihood-based models.
  - **Autoregressive models.**
  - Normalizing Flows.
  - **Variational Autoencoders.**
- Adversarial Models.

Likelihood based models

Likelihood-based models directly parameterize $p_X(x)$.

The parameters θ are then fit by maximizing the likelihood of the data under the model:

$$\mathcal{L}_\theta(x) = \sum_{x \in X} \log p_X(x|\theta) \qquad \theta^* = \arg\max_\theta \mathcal{L}_\theta(x).$$

Note that this is typically done in the log-domain because it simplifies computations and improves numerical stability. Because the model directly parameterizes $p_X(x)$, we can easily infer the likelihood of any x, so we get an explicit model.

Three popular flavors of likelihood-based models are:
- **autoregressive models**,
- **flow-based models**, and
- **variational autoencoders**.

Autoregressive models

In an autoregressive model, we assume that our multidimensional examples $\mathbf{x} \in X$ can be treated as sequences $\{x_1, x_2, \ldots, x_n\}$.

We then factorize the distribution into a product of conditionals, using the **chain rule of probability**:

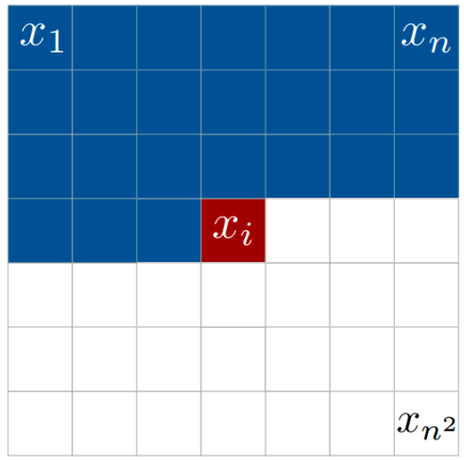
$$p_X(\mathbf{x}) = \prod_i p(x_i \mid x_{<i}) = p(x_0)p(x_1 \mid x_0)p(x_2 \mid x_0, x_1)\ldots p(x_n \mid x_0, \ldots x_{n-1})$$

These conditional distributions, $p(\cdot)$, are typically scalar-valued and much easier to model.

Autoregressive models

For time-series, audio signals, NLP…, AR models are a very natural thing to do.

But we can also do this for other types of structured data by arbitrarily choosing an order (e.g. raster scan order for images, as in *PixelRNN*2 and *PixelCNN*3).



$$p(\mathbf{x}) = \prod_{i=1}^{n^2} p(x_i|x_1, ..., x_{i-1})$$

https://towardsdatascience.com/auto-regressive-generative-models-pixelrnn-pixelcnn-32d192911173

Autoregressive models are attractive because they are able to **accurately capture correlations between the different elements** x_i in a sequence, and they allow for **fast inference** (i.e. computing $p_X(x)$ given x).

Unfortunately they tend to be **slow to sample from**, because samples need to be drawn sequentially from the conditionals for each position in the sequence.

Autoregressive models

In the case of color images, the conditional probability of i^{th} pixel becomes:

$$p(x_{i,R} \mid \mathbf{x}_{<i}) p(x_{i,G} \mid \mathbf{x}_{<i}, x_{i,R}) p(x_{i,B} \mid \mathbf{x}_{<i}, x_{i,R}, x_{i,G})$$

Thus, each color is conditioned on other colors as well as the previously generated pixels.

To get the appropriate pixel value we use a 256-way softmax layer.

We could use LSTMs to model conditional probabilities in a natural way.

In order to use convolutional layers when modeling conditional probabilities, **masked convolutions** can be used:

1	1	1	1	1
1	1	1	1	1
1	1	0	0	0
0	0	0	0	0
0	0	0	0	0

In this way we can build functions that restrict information flow from '*future*' pixels into the one we're predicting.

Conditional autoregressive models

$$p_X(\mathbf{x}, C) = \prod_i p(x_i \mid x_{<i}, C) \text{ where each } p(x_i \mid x_{<i}, C) \text{ is a neural network}$$

that predicts a probability distribution over pixel values.

Figure 3: Class-Conditional samples from the Conditional PixelCNN.

Flow-based models

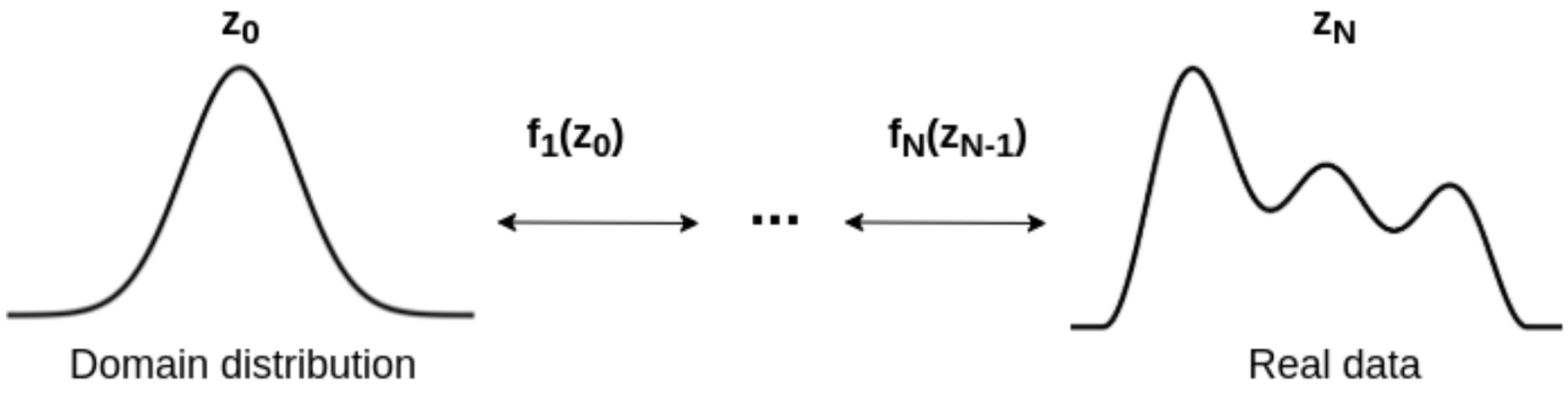
Another strategy for constructing a likelihood-based model is to use the **change of variables theorem** to transform $p_X(x)$ into a simple, factorized distribution $p_Z(z)$ (standard Gaussian is a popular choice) using an invertible mapping $x = g(z)$:

$$p_X(x) = p_Z(z) \cdot |\det J|^{-1} \qquad J = \frac{dg(z)}{dz}.$$

Here, J is the Jacobian of $g(z)$.

Models that use this approach are referred to as **normalising flows** or flow-based models.

Flow-based models



They are fast both for inference and sampling, but the **requirement for** $g(z)$ **to be invertible significantly constrains the model architecture**, and it makes them less parameter-efficient. In other words: flow-based models need to be quite large to be effective.

Variational Autoencoders

By far the most popular class of likelihood-based generative models,

Variational Autoencoders learn two neural networks: an *inference network* $q(z \mid x)$ learns to probabilistically map examples x into a latent space, and a *generative network* $p(x \mid z)$ learns the distribution of the data conditioned on a latent representation z.

These are trained to maximise a lower bound on $p_X(x)$, called the **ELBO** (Evidence Lower BOund), because computing $p_X(x)$ given x (exact inference) is not tractable.

Typical VAEs assume a factorised distribution for $p(x \mid z)$, which limits the extent to which they can capture dependencies in the data.

Adversarial models

Generative Adversarial Networks[9] (GANs) take a **very different approach** to capturing the data distribution.

Two networks are trained simultaneously: a ***generator*** G attempts to produce examples according to the data distribution $p_X(x)$, given latent vectors z, while a ***discriminator*** D attempts to tell apart generated examples and real examples. In doing so, the discriminator provides a learning signal for the generator which enables it to better match the data distribution.

In the original formulation, the loss function is as follows:

$$\mathcal{L}(x) = \mathbb{E}_x[\log D(x)] + \mathbb{E}_z[log(1 - D(G(z)))].$$

The generator is trained to minimize this loss, whereas the discriminator attempts to maximize it. This means the training procedure is a **two-player minimax game**, rather than an optimization process, as it is for most machine learning models. Balancing this game and keeping training stable has been one of the main challenges for this class of models. Many alternative formulations have been proposed to address this.

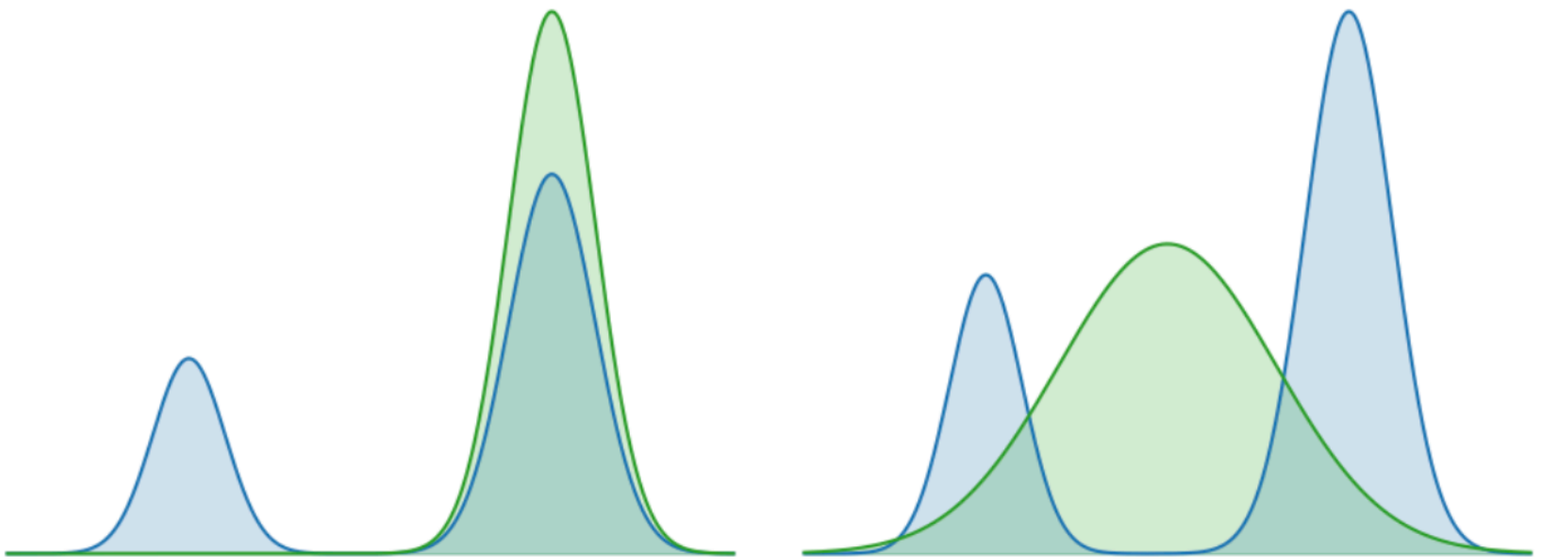
Adversarial models

While adversarial and likelihood-based models are both ultimately trying to model $p_X(x)$, they approach this target from very different angles. As a result, GANs tend to be **better at producing realistic examples**, but worse at capturing the full diversity of the data distribution, compared to likelihood-based models.

An image generated by a StyleGAN that looks deceptively like a photograph of a real person.

Mode-covering vs. mode-seeking behaviour

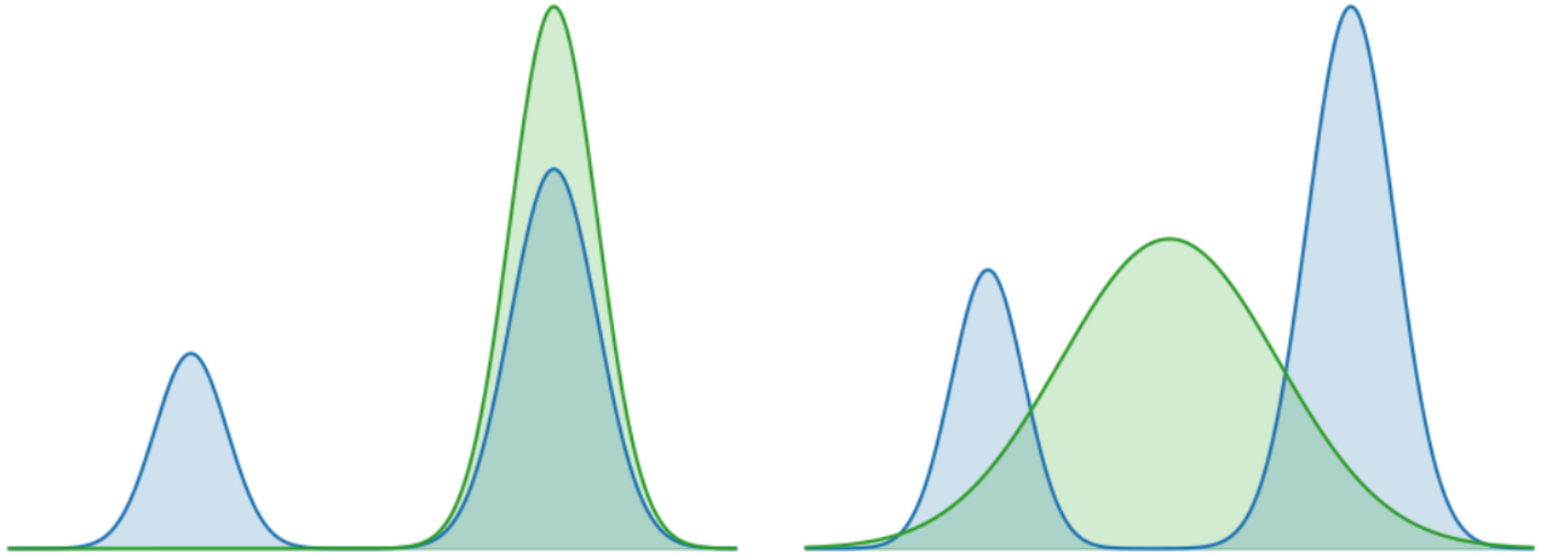
An important consideration when determining which type of generative model is appropriate for a particular application, i**s the degree to which it is *mode-covering* or *mode-seeking***. When a model does not have enough capacity to capture all the variability in the data, different compromises can be made. If all examples should be reasonably likely under the model, it will have to overgeneralise and put probability mass on interpolations of examples that may not be meaningful (mode-covering). If there is no such requirement, the probability mass can be focused on a subset of examples, but then some parts of the distribution will be ignored by the model (mode-seeking).



Mode-covering vs. mode-seeking behaviour

Likelihood-based models are usually mode-covering. This is a consequence of the fact that they are fit by maximising the joint likelihood of the data.

Adversarial models on the other hand are typically mode-seeking. A lot of ongoing research is focused on making it possible to control the trade-off between these two behaviours directly, without necessarily having to switch the class of models that are used.

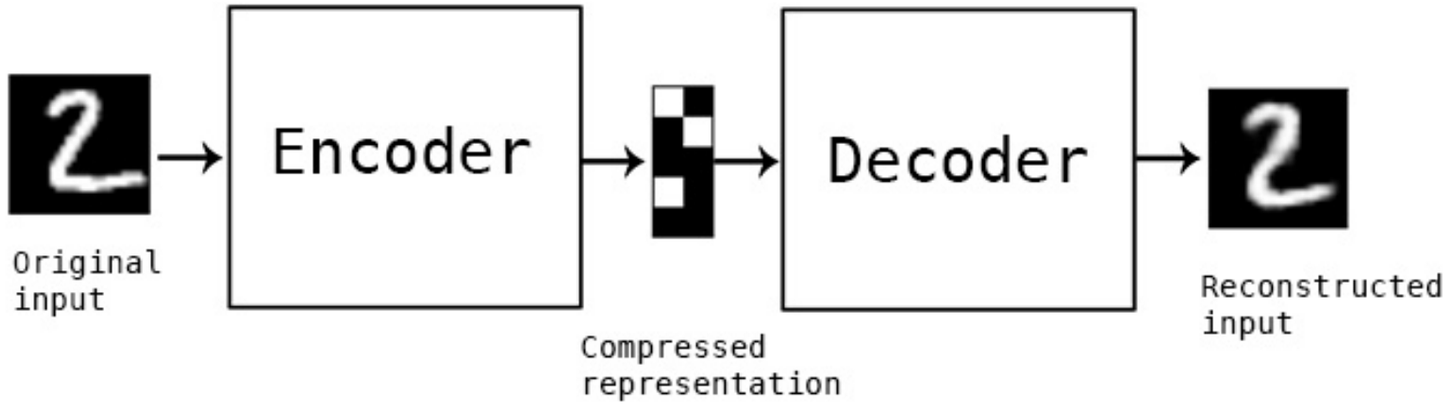


Autoencoders

Suppose we have only a set of unlabeled training examples $x_1, x_2, x_3, \ldots$, where $x_i \in \Re^n$.

An autoencoder neural network is an **unsupervised** learning algorithm that applies backpropagation and uses a loss function that is optimal when setting the target values to be equal to the inputs, $y_i = x_i$.

To build an autoencoder, you need three things: an e**ncoding function**, a **decoding function**, and a **distance function** between the amount of information loss between the compressed representation of your data and the decompressed representation.



https://blog.keras.io/building-autoencoders-in-keras.html

Simplest Autoencoder for MNIST

```python
# Source: Adapted from https://blog.keras.io/building-autoencoders-in-keras.html

from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model

# this is the size of our encoded representations
encoding_dim = 32  # 32 floats -> compression of factor 24.5,
                   # assuming the input is 784 floats

input_img = Input(shape=(784,))

# encoded representation of the input
encoding_layer = Dense(encoding_dim, activation='relu')
encoded = encoding_layer(input_img)

# lossy reconstruction of the input
decoding_layer = Dense(784,
                       activation='sigmoid')
decoded = decoding_layer(encoded)

# this model maps an input to its reconstruction
autoencoder = Model(input_img, decoded)
```

Input

Output

Adding depth and sparsity constraints

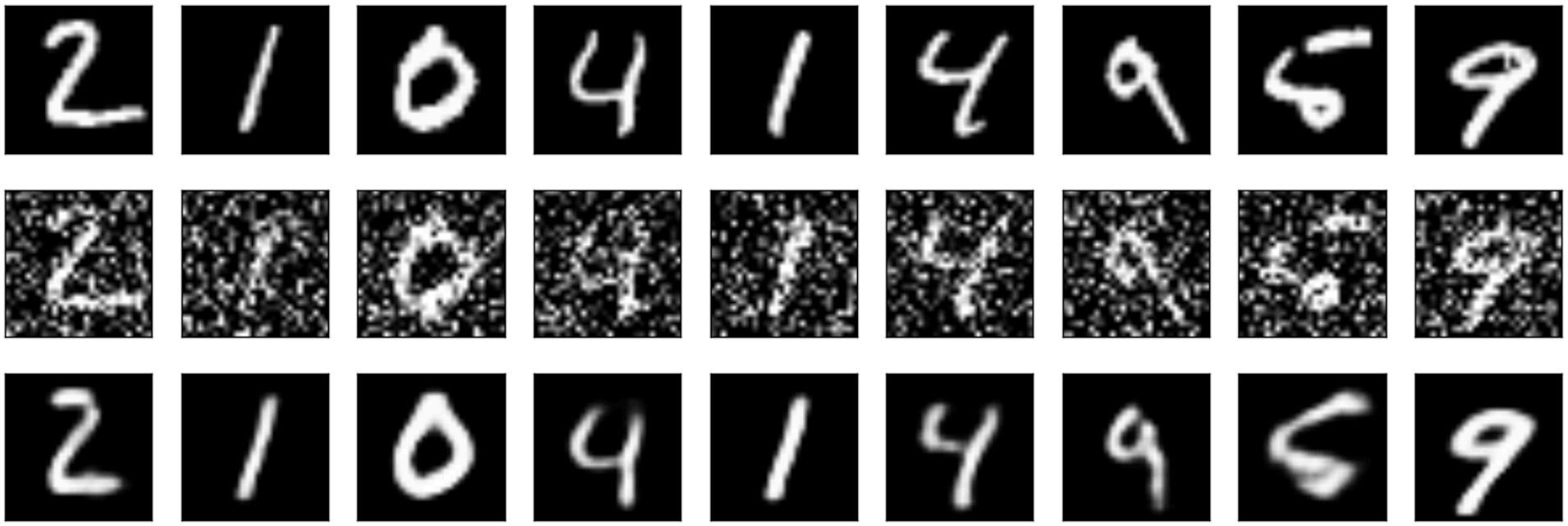
```
from tensorflow.keras import regularizers
from tensorflow.keras import optimizers
from tensorflow.keras.regularizers import l2, activity_l1
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model

input_img = Input(shape=(784,))
encoded = Dense(128, activation='relu')(input_img)
encoded = Dense(64, activation='relu')(encoded)
encoded = Dense(32, activation='relu')(encoded)
decoded = Dense(64, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(decoded)
decoded = Dense(784, activation='sigmoid')(decoded)

autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adadelta',
                    loss='binary_crossentropy',
                    activity_regularizer=regularizers.l1(10e-5))
```

Convolutional Autoencoders

```
input_img = Input(shape=(28, 28, 1))

x = Conv2D(16, 3, 3, activation='relu', border_mode='same')(input_img)
x = MaxPooling2D((2, 2), border_mode='same')(x)
x = Conv2D(8, 3, 3, activation='relu', border_mode='same')(x)
x = MaxPooling2D((2, 2), border_mode='same')(x)
x = Conv2D(8, 3, 3, activation='relu', border_mode='same')(x)
encoded = MaxPooling2D((2, 2), border_mode='same')(x)

# at this point the representation is (4, 4, 8) i.e. 128-dimensional

x = Conv2D(8, 3, 3, activation='relu', border_mode='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(8, 3, 3, activation='relu', border_mode='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(16, 3, 3, activation='relu')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, 3, 3, activation='sigmoid', border_mode='same')(x)

# at this point the representation is (28, 28, 1) i.e. 784-dimensional

autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')
```

Denoising Autoencoders

Noise as a regularization strategy...



Variational Autoencoders

A **variational autoencoder** is an autoencoder that adds probabilistic constraints on the representations being learned.

> When using probabilistic models, compressed representation is called **latent variable model**.

So, instead of learning a function this model is **learning a probabilistic distribution function that models your data**.

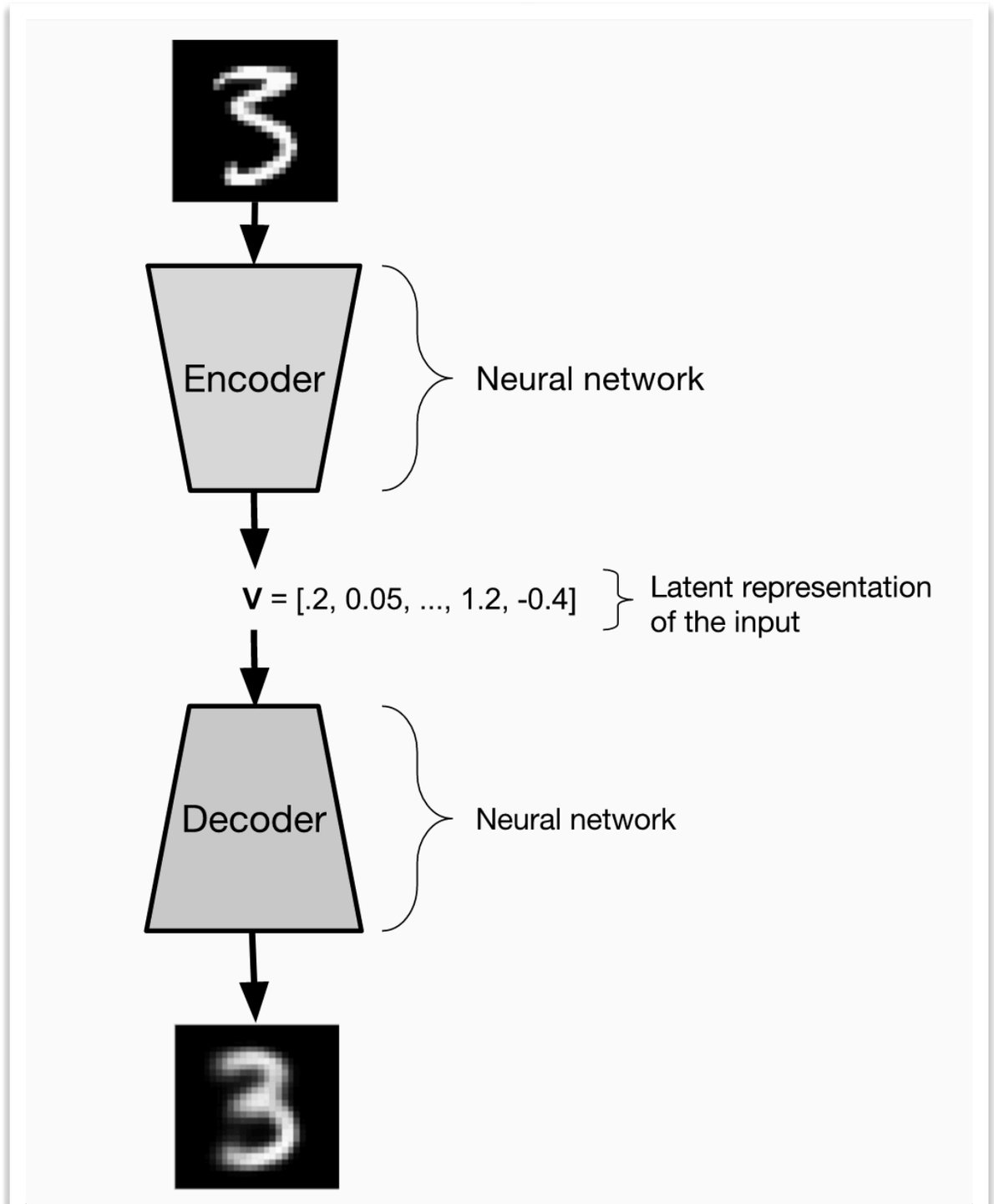
Why?

> Standard autoencoders are not suited to work as a generative model. **If you pick a random value for your decoder you won't get necessarily a good reconstruction**: the value can far away from any previous value the network has seen before! That's why attaching a probabilistic model to the compressed representation is a good idea!

For sake of simplicity, let's use a **standard normal distribution** to define the distribution of inputs ($\mathbf{V}$) the decoder will receive.

Variational Autoencoders

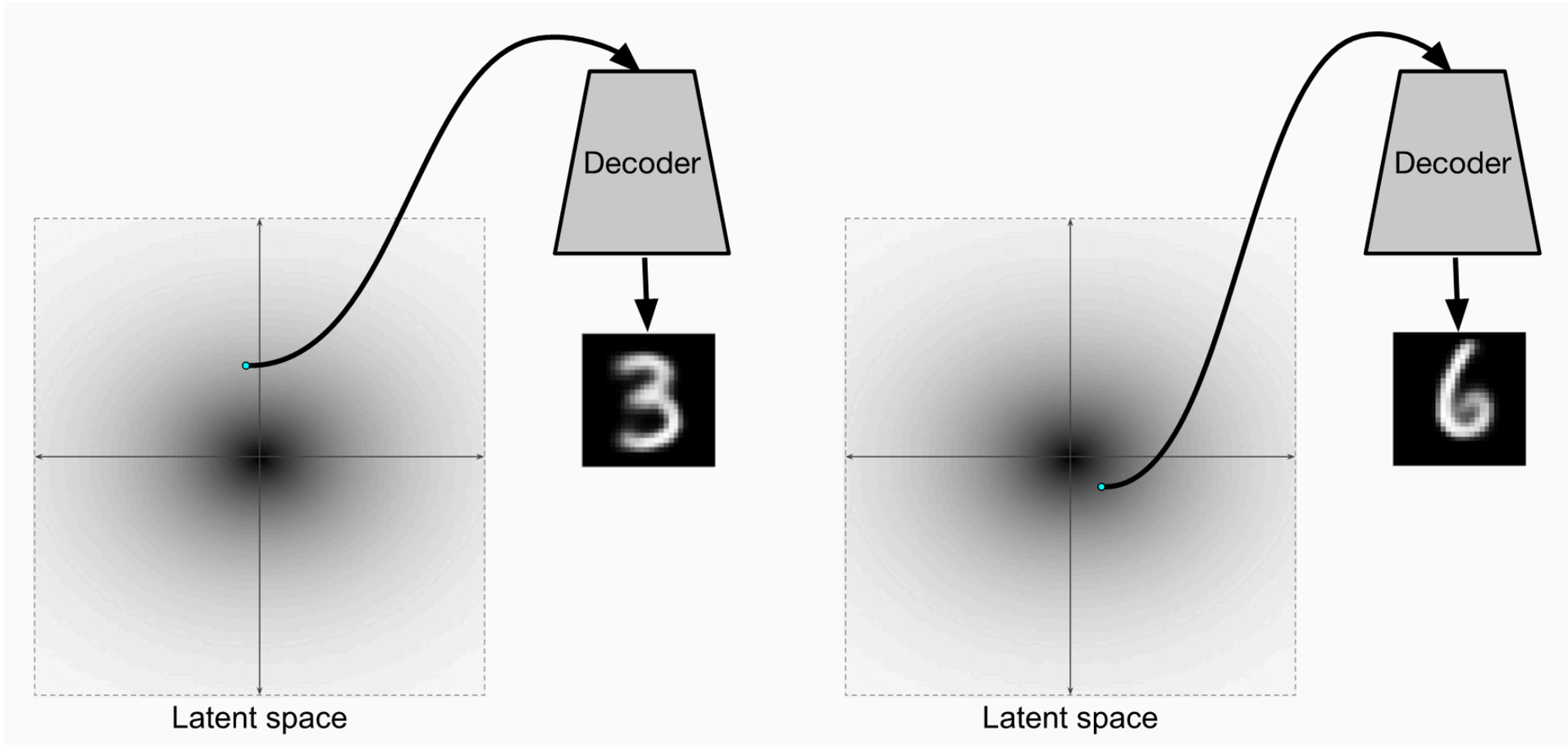
Architecture of a variational autoencoder (VAE)



http://ijdykeman.github.io/ml/2016/12/21/cvae.html

Variational Autoencoders

We want the decoder to take any point taken from a standard normal distribution to return a reasonable element of our dataset.

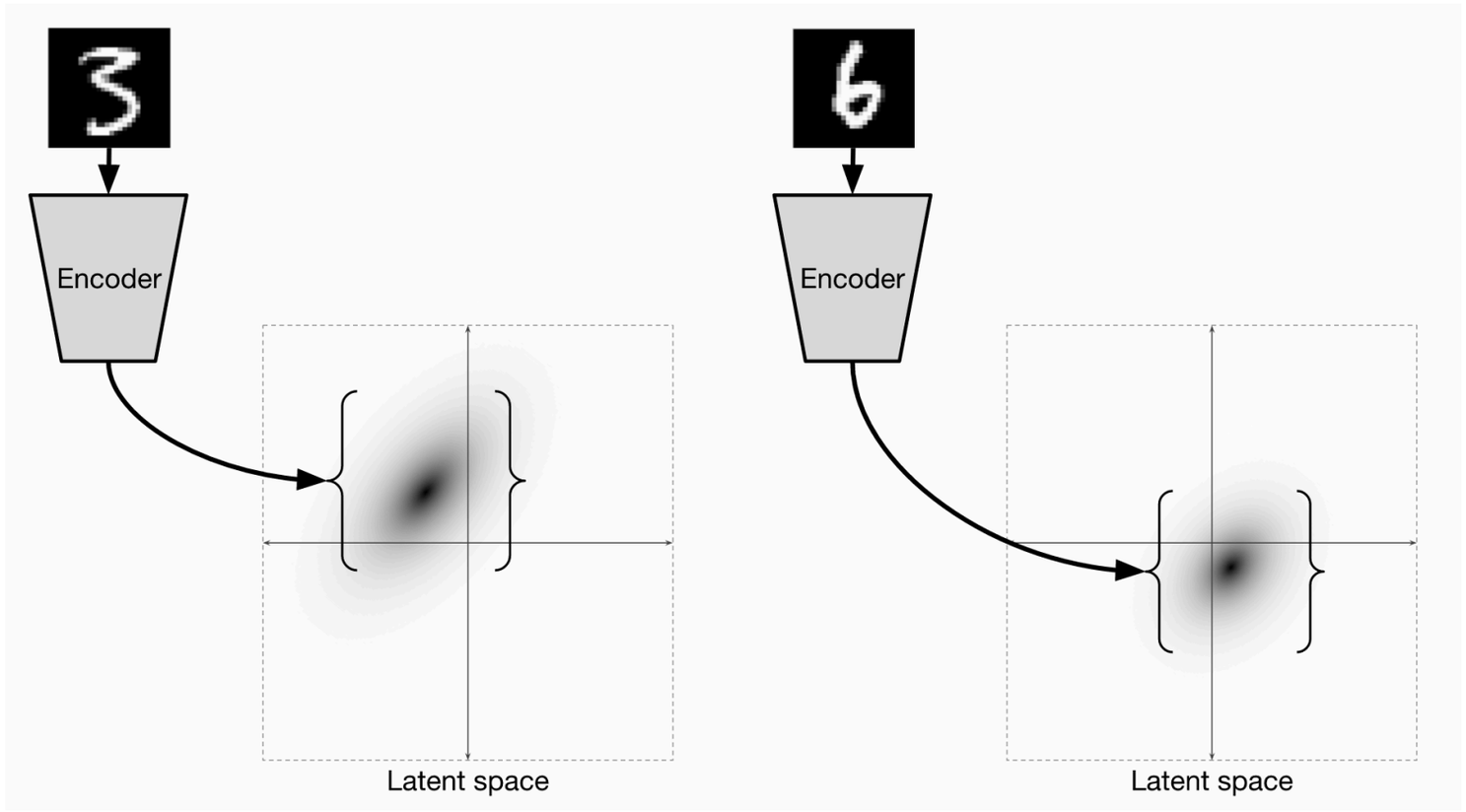


http://ijdykeman.github.io/ml/2016/12/21/cvae.html

Variational Autoencoders

Let's consider the encoder role in this architecture.

> In a traditional autoencoder, the encoder model takes a sample from data and returns a single point in the latent space, which is then passed to the decoder.
>
> In VAE the encoder instead produces (the parameters of) a probability distribution in the latent space:



http://ijdykeman.github.io/ml/2016/12/21/cvae.html

Variational Autoencoders

First, let's implement the encoder net, which takes input X and outputs two things: $\mu(X)$ and $\Sigma(X)$, the parameters of the Gaussian. Our encoder will be a neural net with one hidden layer.

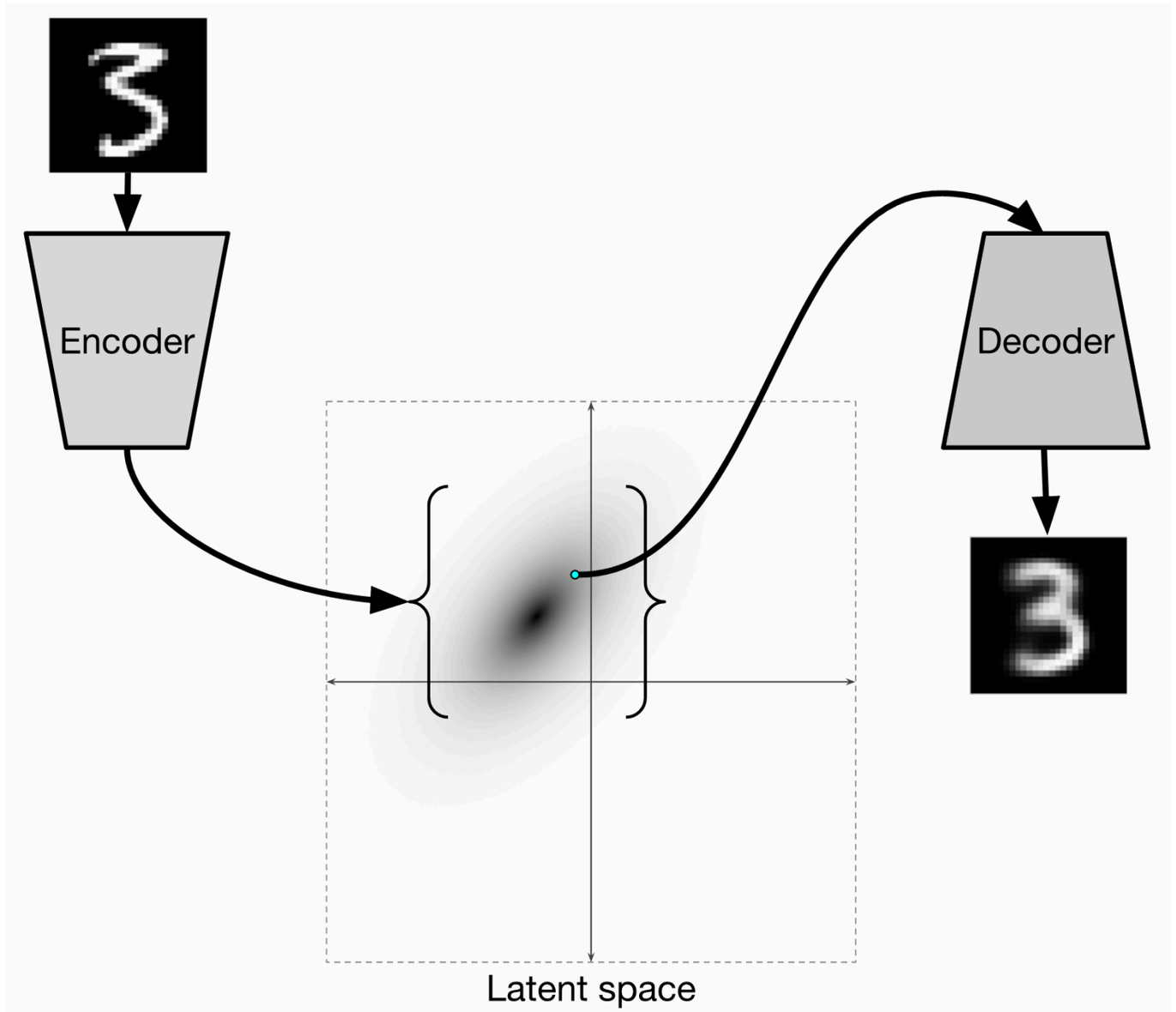
Our latent variable is two dimensional, so that we could easily visualize it.

```
18 # encoder
19 inputs = Input(shape=(784,))
20 h_q = Dense(512, activation='relu')(inputs)
21 mu = Dense(2, activation='linear')(h_q)
22 log_sigma = Dense(2, activation='linear')(h_q)
```

Up to now we have an encoder that takes images and produce (the parameters of) a pdf in the latent space. The decoder takes points in the latent space and return reconstructions.

How do we connect both models? By sampling from the produced distribution!

Variational Autoencoders



http://ijdykeman.github.io/ml/2016/12/21/cvae.html

Variational Autoencoders

To this end we will implement a random variate reparameterization: the substitution of a random variable by a deterministic transformation of a simpler random variable.

> There are several methods by which non-uniform random numbers, or random variates, can be generated. The most popular methods are the *one-liners*, which give us the simple tools to generate random variates in one line of code, following the classic paper by Luc Devroye (*Luc Devroye, Random variate generation in one line of code, Proceedings of the 28th conference on Winter simulation, 1996*).

In the case of a Gaussian, we can use the following algorithm:

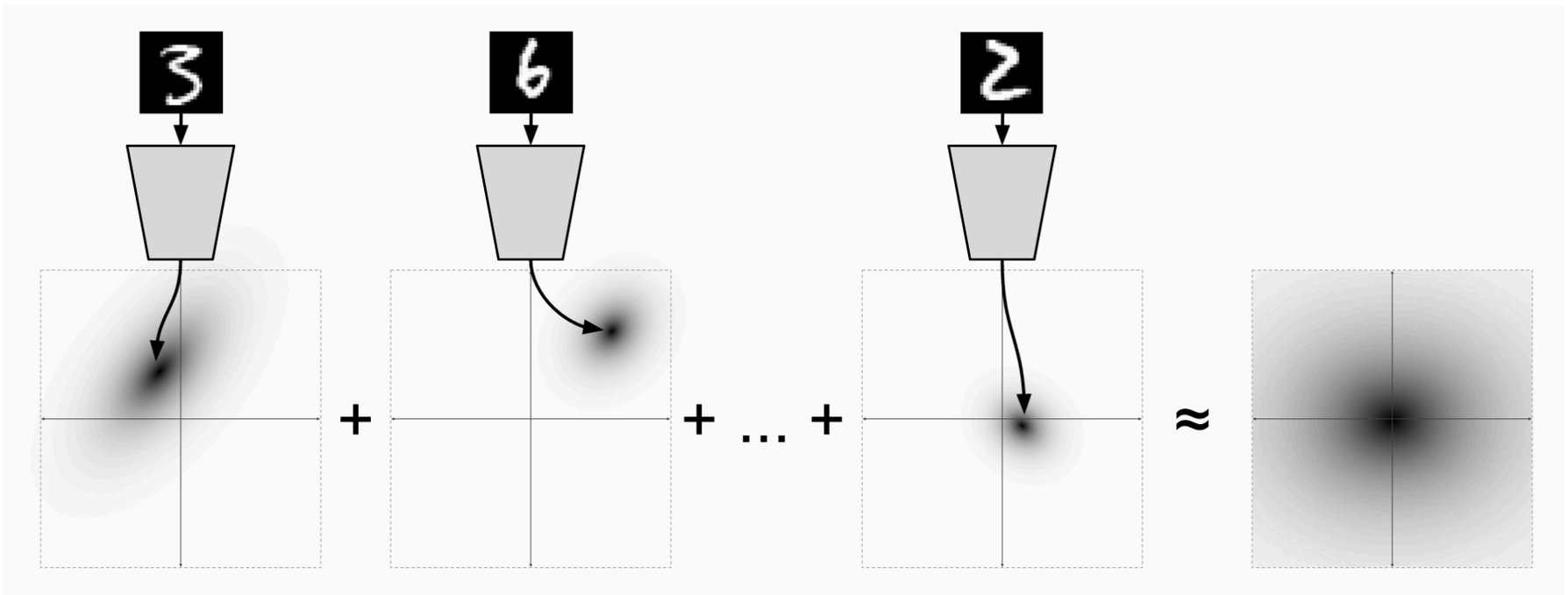
- Generate $\epsilon \sim \mathcal{N}(0; 1)$.
- Compute a sample from $\mathcal{N}(\mu; RR^T)$ as $\mu + R\epsilon$.

```
1 def sample_z(args):
2     mu, log_sigma = args
3     eps = K.random_normal(shape=(m, n_z), mean=0., std=1.)
4     return mu + K.exp(log_sigma / 2) * eps
5
6 # Sample z
7 z = Lambda(sample_z)([mu, log_sigma])
```

Variational Autoencoders

From this model, we can do three things: reconstruct inputs, encode inputs into latent variables, and generate data from latent variable.

In order to be coherent with our previous definitions, we must assure that points sampled from the latent space fit a standard normal distribution, but the encoder is producing non standard normal distributions. So, we must add a constraint for getting something like this:



http://ijdykeman.github.io/ml/2016/12/21/cvae.html

Variational Autoencoders

In order to impose this constraint we can add a term to the loss function: the **Kullback-Leibler divergence**.

The Kullback–Leibler divergence is a measure of how one probability distribution diverges from a second expected probability distribution.

For discrete probability distributions P and Q, the Kullback–Leibler divergence from Q to P is defined to be

$$D_{\mathrm{KL}}(P\|Q) = \sum_i P(i)\,\log\frac{P(i)}{Q(i)}$$

The rest of the loss function must take into account the "reconstruction" error.

```
def vae_loss(y_true, y_pred):
    """
    Calculate loss = reconstruction loss +
    KL loss for each data in minibatch
    """
    recon = K.sum(K.binary_crossentropy(y_pred, y_true), axis=1)
    # D_KL(Q(z|X) || P(z|X));
    # calculate in closed form as both dist. are Gaussian
    kl = 0.5 * K.sum(K.exp(log_sigma) + K.square(mu)
                     - 1. - log_sigma, axis=1)
    return recon + kl
```

How do we train a Variational Autoencoder?

In fact this is not a problem! By using the one-liner method for sampling we have expressed the latent distribution in a way that its parameters are factored out of the parameters of the random variable so that backpropagation can be used to find the optimal parameters of the latent distribution.

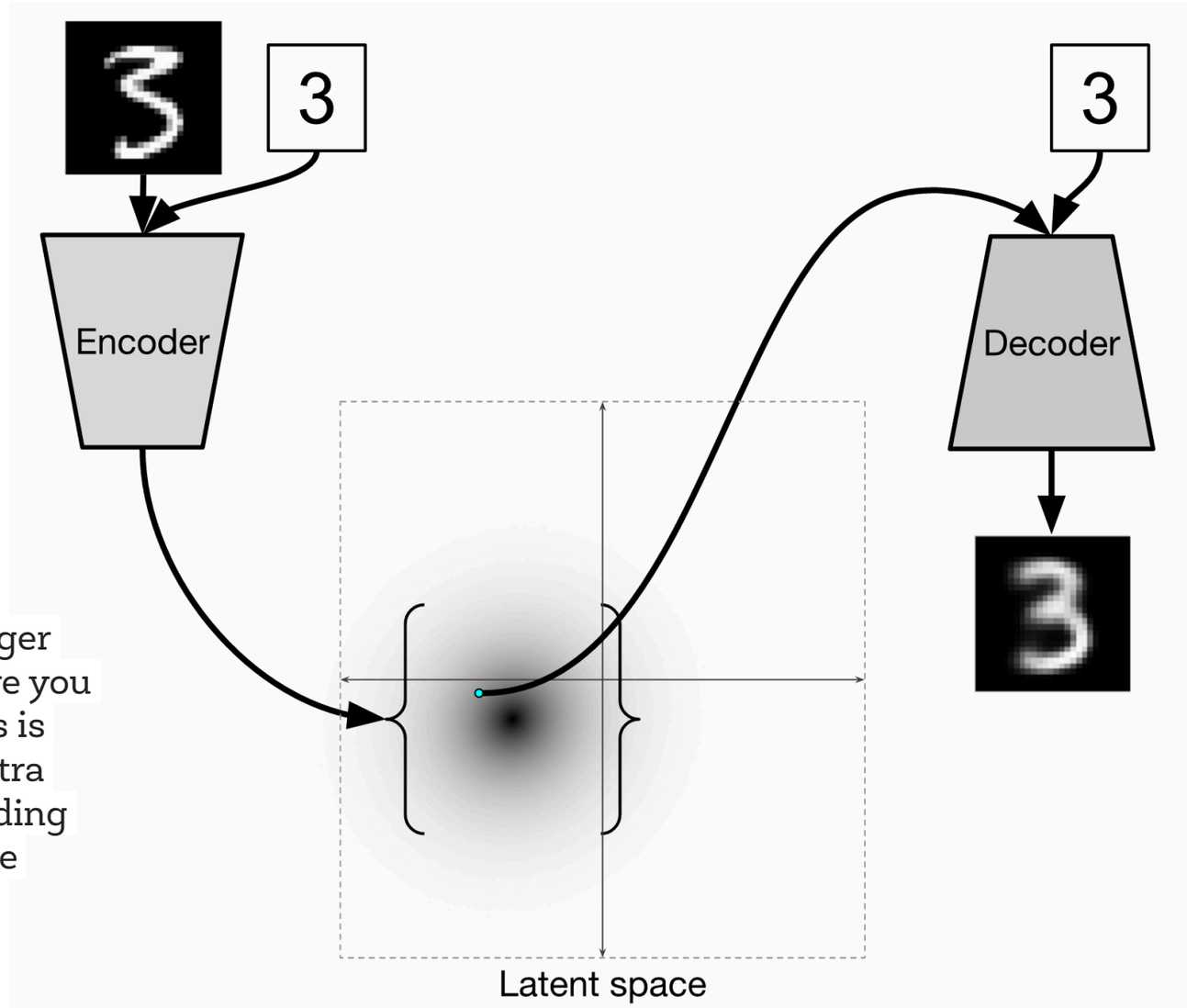
For this reason this method is called ***reparametrization trick***.

By using this trick we can train end-to-end a VAE with backpropagation.

Conditional Variational Autoencoder

What about producing specific number instances on demand?

We can do this by adding an extra input (as a one-hot encoding) to both the encoder and the decoder:



The latent space is no longer encoding what number are you dealing with (because this is already encoded in the extra input!). Instead, it is encoding information such as stroke width, angle, etc.

http://ijdykeman.github.io/ml/2016/12/21/cvae.html

Variational Autoencoder in Keras (d=2)