UNIVERSITAT DE BARCELONA

Jordi Vitrià

Deep Learning

Unsupervised Learning and Generative Models

Generative modeling and unsupervised learning

Given a dataset X of (**multidimensional**) 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.

Generative modeling and unsupervised learning

We distinguish *implicit* and e*xplicit* 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).

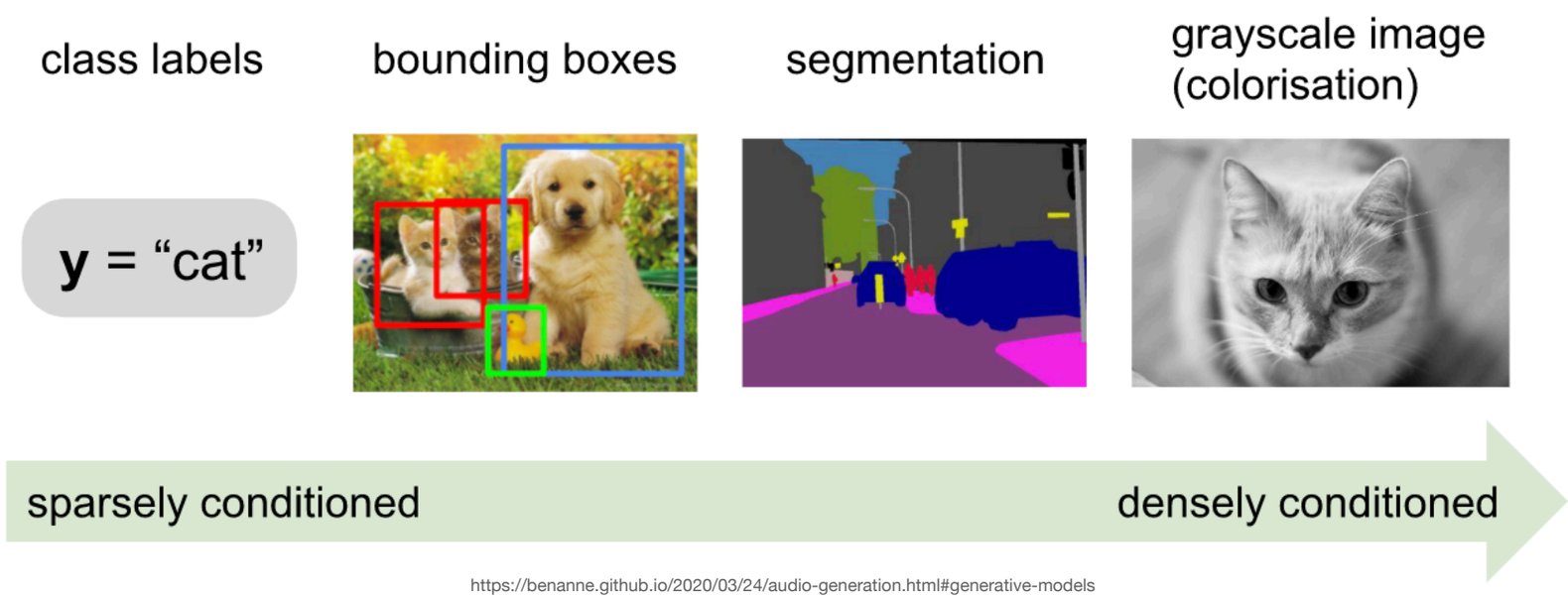
- If we have an **explicit model**, we can infer the likelihood of an example, though sometimes only up to an unknown normalizing constant.

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 (https://deepgenerativemodels.github.io/notes/flow/).
  - **Variational Autoencoders**.

- **Adversarial Models**.

Likelihood based models

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

The parameters θ are then fit by maximizing the (log) 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 the distribution of our **multidimensional examples** $\mathbf{x} \in X$ can be factorized into a product of conditionals, using the **chain rule of probability**:

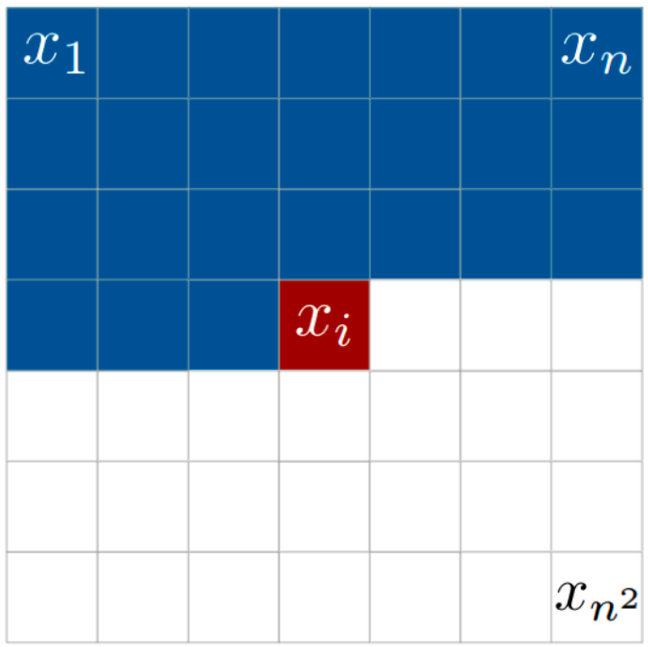
$$p_X(\mathbf{x}) = \prod_i p(x_i \,|\, x_{<i}) = p(x_0)p(x_1 \,|\, x_0)p(x_2 \,|\, x_0, x_1)\ldots p(x_n \,|\, x_0, \ldots x_{n-1})$$

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).



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

Everyone of this conditional probability distributions can be implemented by a NN (f.e. a RNN)

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

Autoregressive models

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.

Autoregressive models

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.

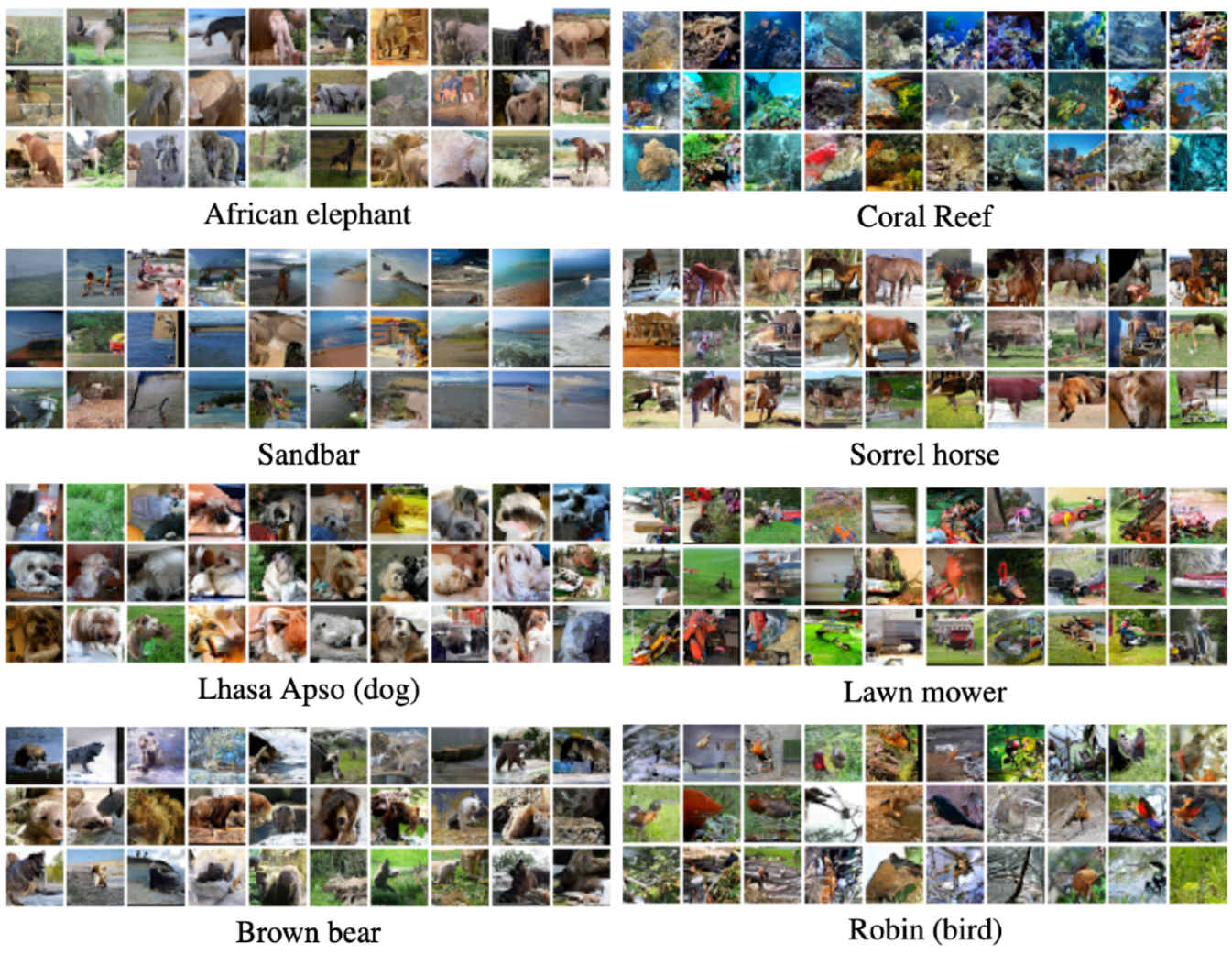


Figure 3: Class-Conditional samples from the Conditional PixelCNN.

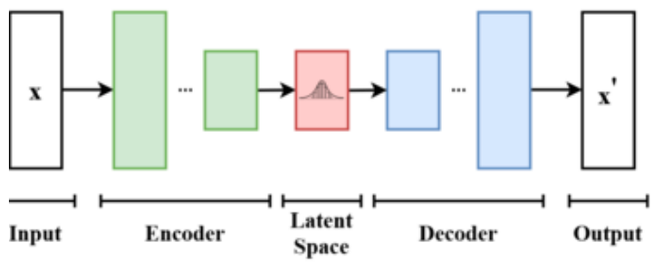
Variational Autoencoders

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

Variational Autoencoders learn two neural networks: an *inference network or encoder* $q(z|x)$ learns to probabilistically map examples x into a **latent space** Z, and a *generative network* or *decoder* $p(x|z)$ 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 $p(x|z)$, which limits the extent to which they can capture dependencies in the data.



Adversarial models

Generative Adversarial Networks (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:

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

Adversarial models

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

Probability that the discriminator is rightly classifying the real image

Probability that the discriminator is rightly classifying the fake image

The generator is trained to **minimize** this loss, whereas the discriminator attempts to **maximize** it.

Discriminator Loss

It penalizes itself for misclassifying a real instance as fake, or a fake instance (created by the generator) as real, by **maximizing** the function.

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

Adversarial models

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

Probability that the generator is rightly classifying the real image

Probability that the generator is rightly classifying the fake image

The generator is trained to **minimize** this loss, whereas the discriminator attempts to **maximize** it.

Generator Loss

It gets rewarded if it successfully fools the discriminator, and gets penalized otherwise:

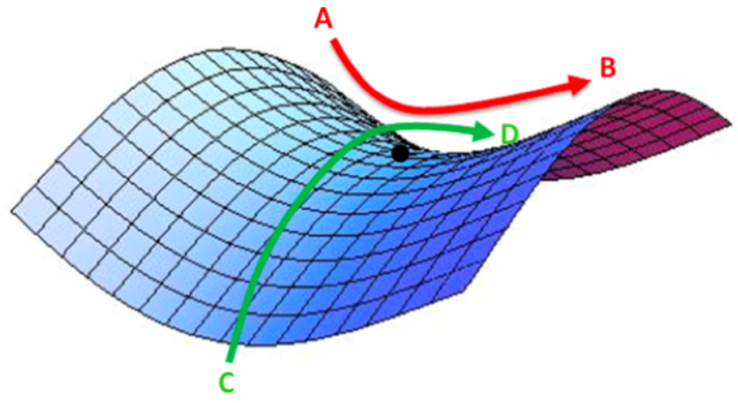
$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log \left(1 - D\left(G\left(z^{(i)}\right)\right)\right)$$

Adversarial models

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

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.

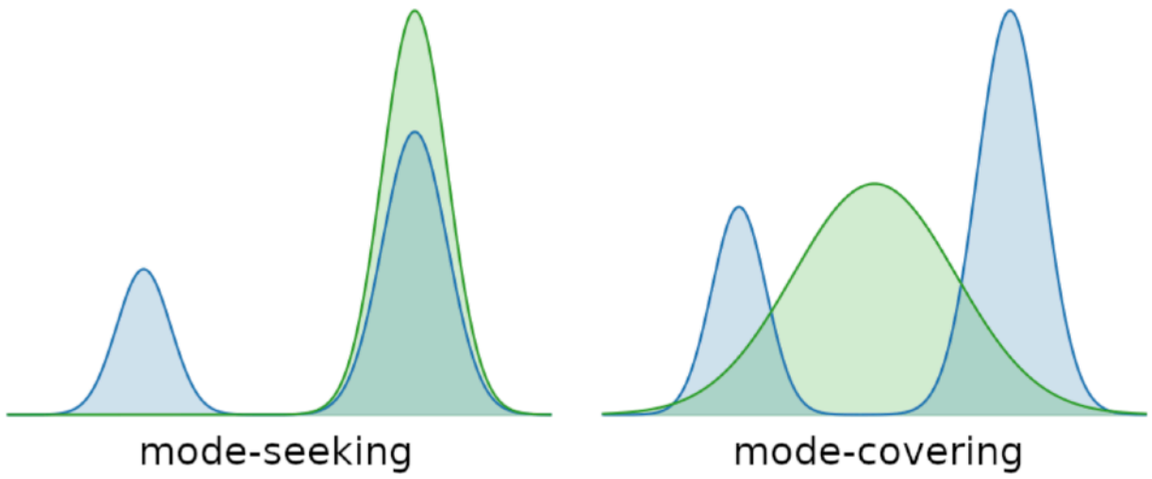
An image generated by a StyleGAN that looks deceptively like a photograph of a real person.

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**).

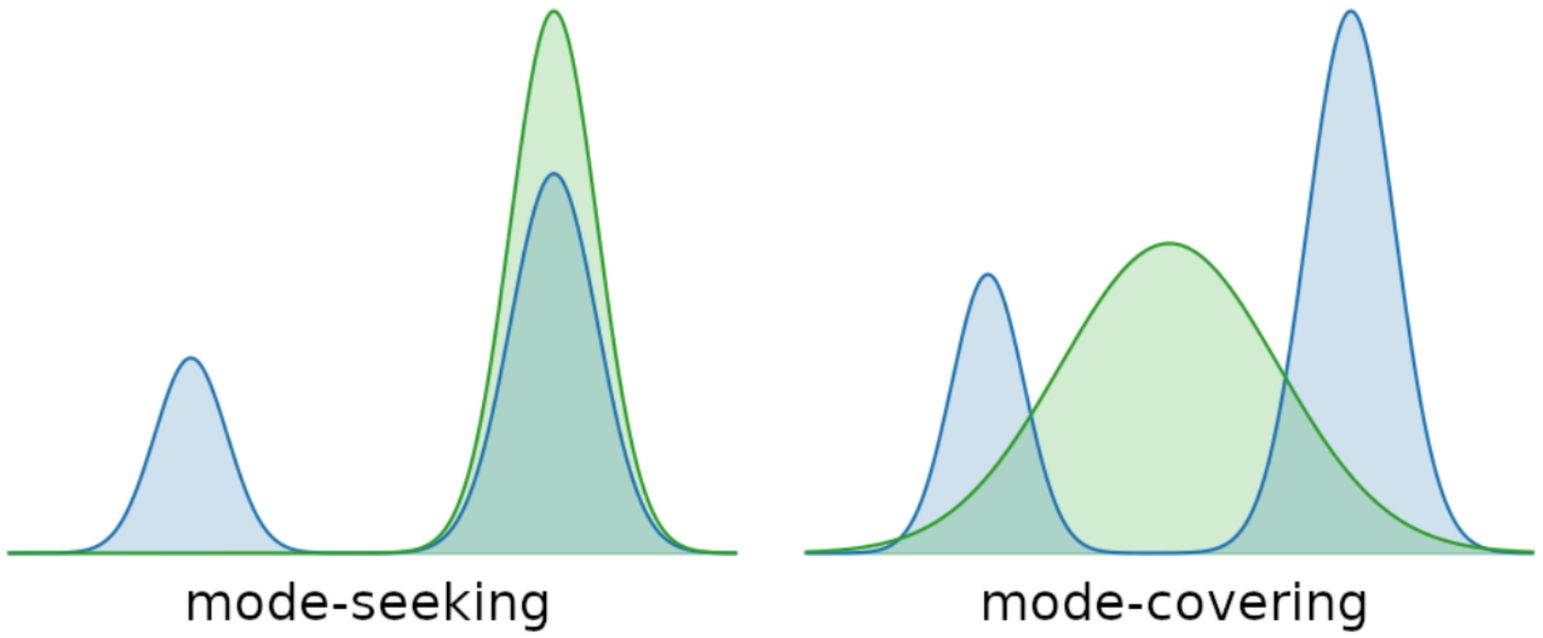
If there is no such requirement, the probability mass can be focused on a subset of examples, but then some parts of the distribution will be ignored by the model (**mode-seeking**).



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.



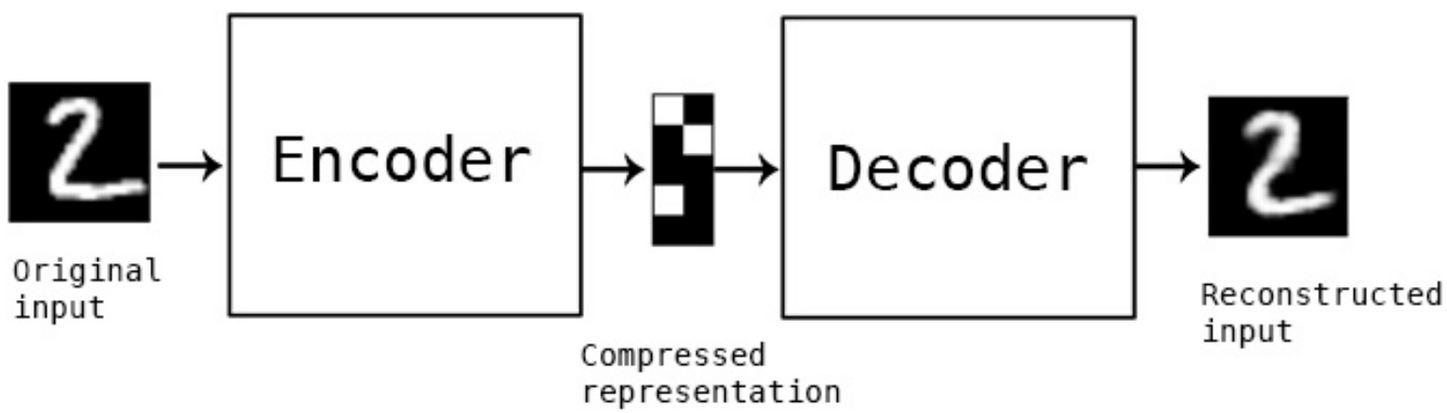
Algorithms

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

Simple 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

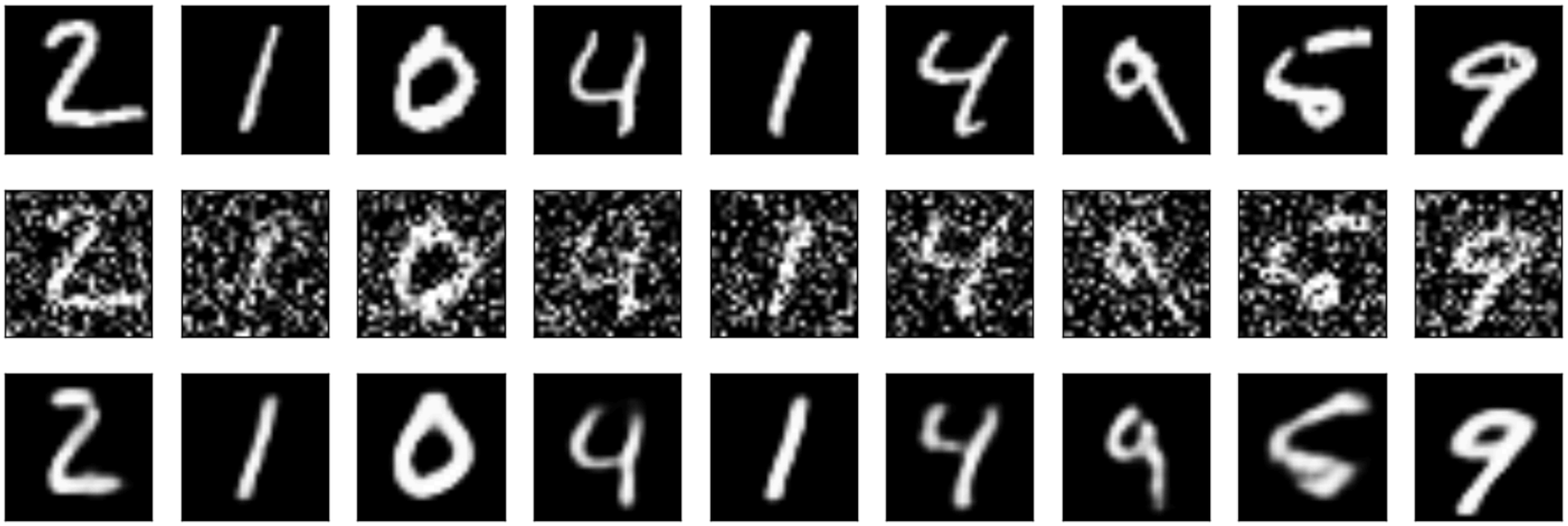
```
from tensorflow.keras import regularizers
from tensorflow.keras import optimizers
from tensorflow.keras.regularizers import l2, activity_l1
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model

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**.

Variational Autoencoders

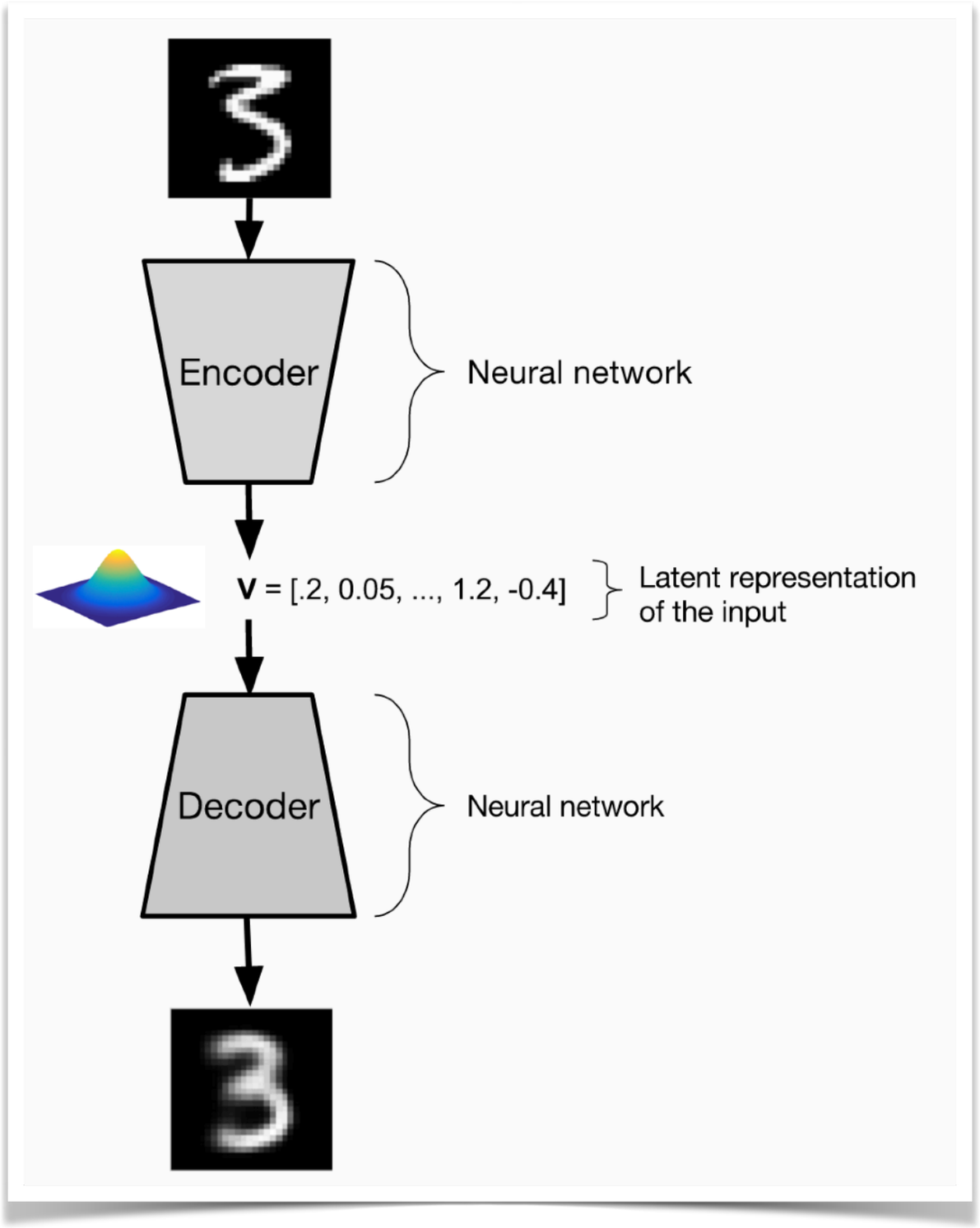
Why?

> 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** ($\mathbf{V}$) to represent the distribution of inputs the decoder will receive.

Variational Autoencoders

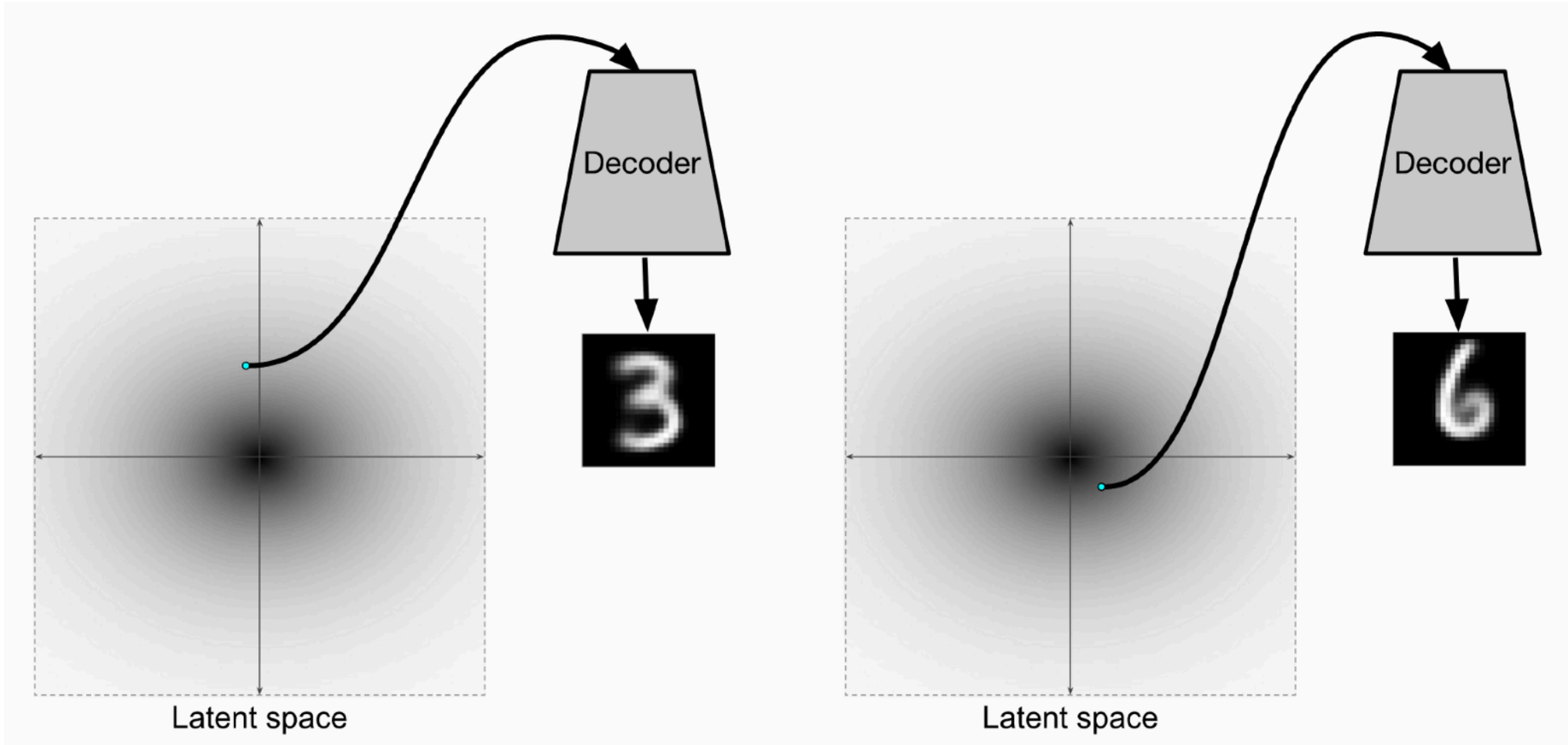
Architecture of a variational autoencoder (VAE)



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.

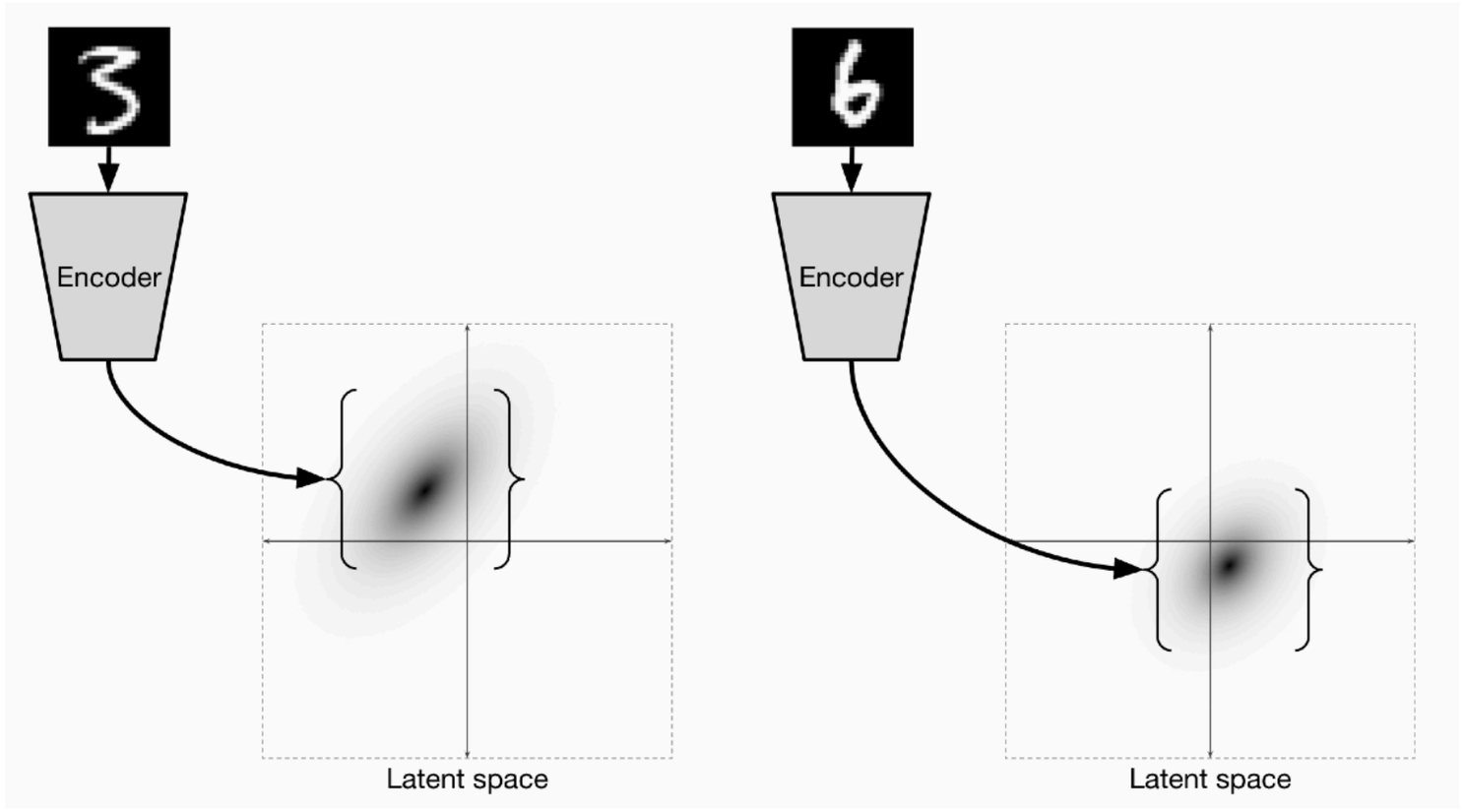


http://ijdykeman.github.io/ml/2016/12/21/cvae.html

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

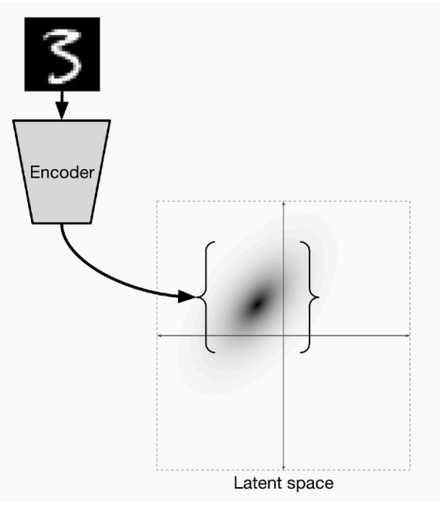
Variational Autoencoders

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)
```

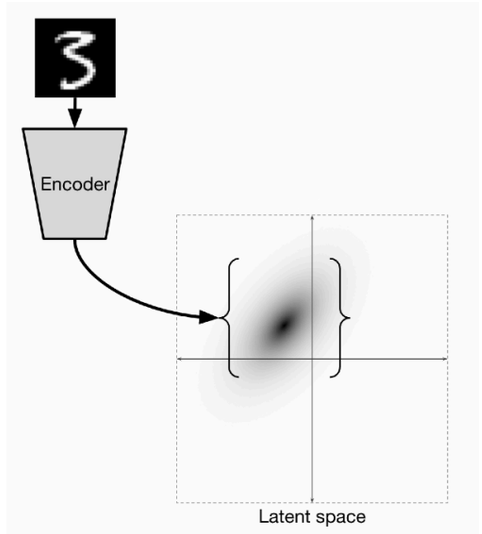
Diagonal covariance matrix

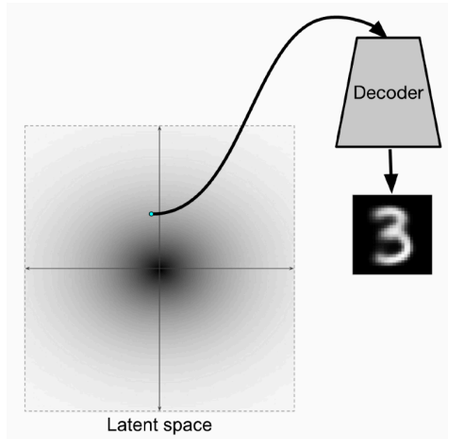


Variational Autoencoders

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.

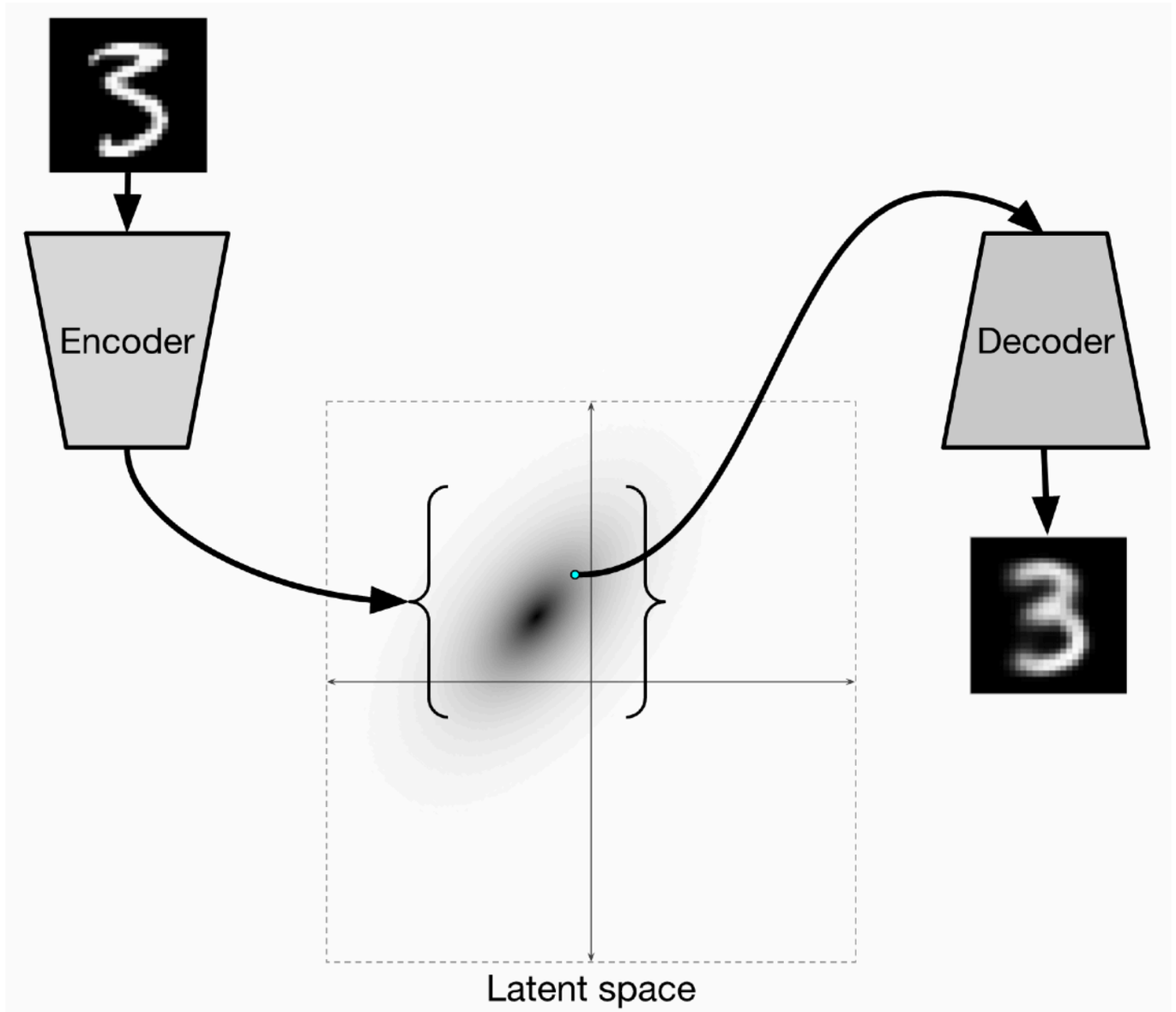




How do we connect both models?

By sampling from the produced distribution!

Variational Autoencoders



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

Variational Autoencoders

Sampling from the produced distribution

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*).

Variational Autoencoders

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

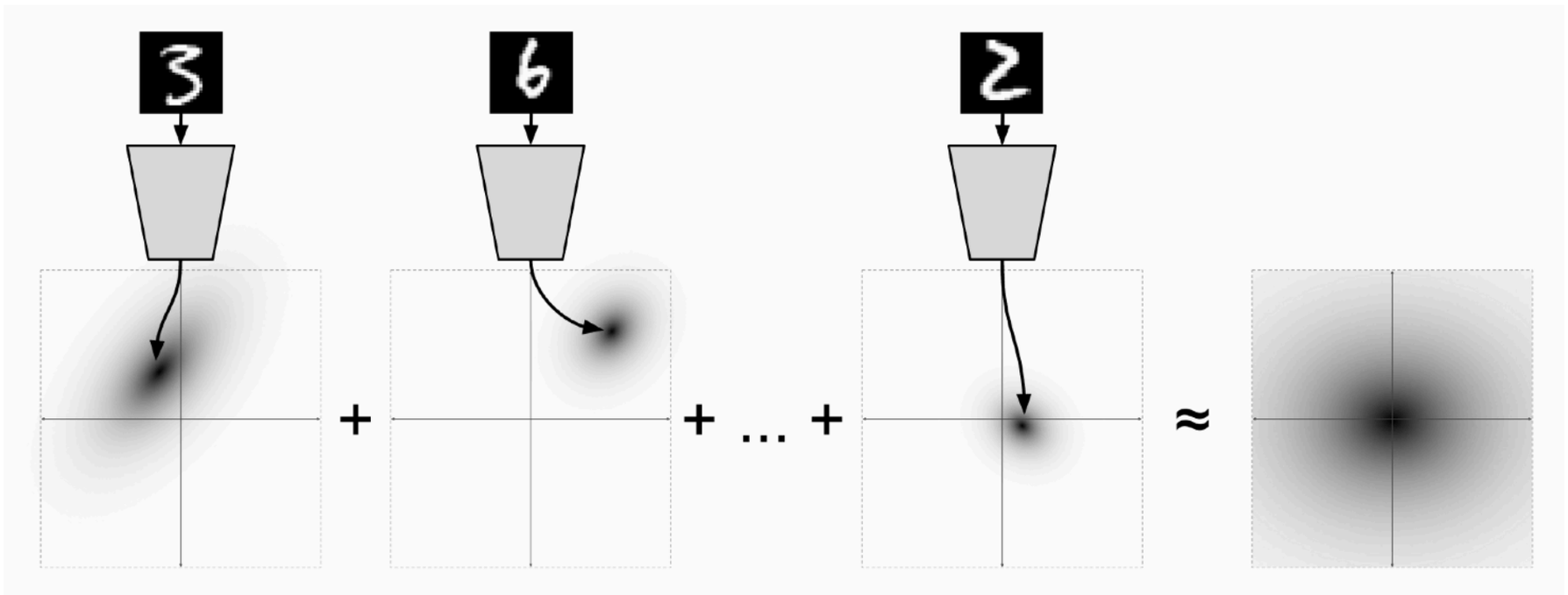
- Generate $\epsilon \sim \mathcal{N}(0; 1)$.
- Compute a sample from $\mathcal{N}(\mu; RR^T)$ as $\mu + R\epsilon$.

```
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:

VAE's loss function comprises a **Reconstruction** error, and a **KL-divergence** error used to model the networks' objectives: $L_{MSE} + L_{KL}$.

The final loss is a weighted sum of both losses.

The reconstruction loss in VAE is similar to the Loss we used in the traditional Autoencoder i.e. **Mean-Squared-Error**

$$L_{MSE}(\theta, \phi) = \frac{1}{N} \sum_{i=1}^{N} (x_i - D_\phi(E_\theta(x_i))^2$$

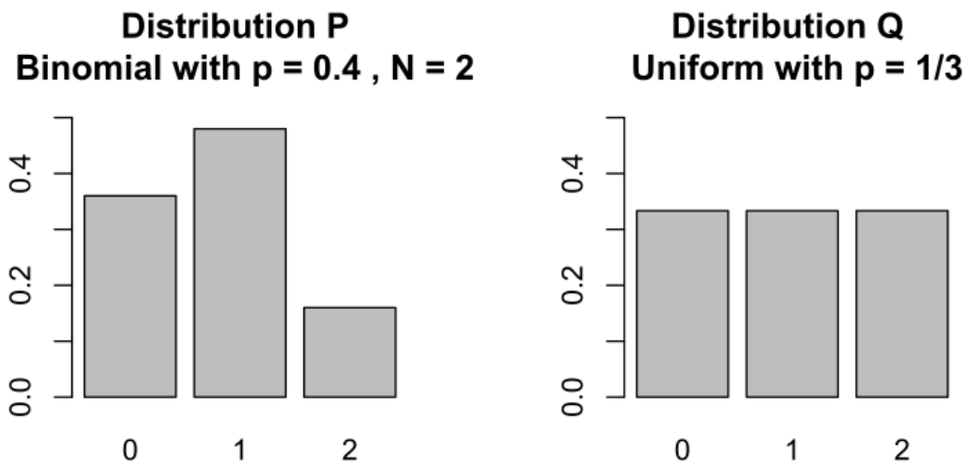
where θ and ϕ are the parameters of the encoder and decoder, respectively.

Variational Autoencoders

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

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



x	0	1	2
Distribution $P(x)$	$\frac{9}{25}$	$\frac{12}{25}$	$\frac{4}{25}$
Distribution $Q(x)$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$

$$\begin{aligned}
D_{\mathrm{KL}}(P \parallel Q) &= \sum_{x \in \mathcal{X}} P(x) \ln\left(\frac{P(x)}{Q(x)}\right) \\
&= \frac{9}{25} \ln\left(\frac{9/25}{1/3}\right) + \frac{12}{25} \ln\left(\frac{12/25}{1/3}\right) + \frac{4}{25} \ln\left(\frac{4/25}{1/3}\right) \\
&= \frac{1}{25}\left(32\ln(2) + 55\ln(3) - 50\ln(5)\right) \approx 0.0852996 \\
D_{\mathrm{KL}}(Q \parallel P) &= \sum_{x \in \mathcal{X}} Q(x) \ln\left(\frac{Q(x)}{P(x)}\right) \\
&= \frac{1}{3} \ln\left(\frac{1/3}{9/25}\right) + \frac{1}{3} \ln\left(\frac{1/3}{12/25}\right) + \frac{1}{3} \ln\left(\frac{1/3}{4/25}\right) \\
&= \frac{1}{3}\left(-4\ln(2) - 6\ln(3) + 6\ln(5)\right) \approx 0.097455
\end{aligned}$$

A relative entropy of 0 indicates that the two distributions in question have identical quantities of information.

Variational Autoencoders

We train our VAE to minimize the KL divergence between the encoder's distribution $Q_\phi(Z|X)$ and $P(Z)$.

In VAE, $P(Z)$ follows a standard or unit $P(Z) = \mathcal{N}(0,1)$. If the encoder outputs encoding Z far from a standard normal distribution, KL-divergence loss will penalize it more.

Trick: While calculating the KL-divergence we choose to map the variance to the logarithm of the variance. By taking the **logarithm** of the **variance**, we force the network to have the output range of the natural numbers rather than just positive values (**variances** would only have positive values). This allows for smoother representations for the latent space.

In this special case, the KL divergence has the closed form:

$$L_{KL}(\theta, \phi) = -0.5 \sum_{i=1}^{N} 1 + \log \sigma_i^2 - \mu_i^2 - e^{\log \sigma_i^2}$$

```
# D_KL(Q(z|X) || P(z|X)); calculate in closed form as both dist. are Gaussian
kl_loss = 0.5 * tf.reduce_sum(tf.exp(z_logvar) + z_mu**2 - 1. - z_logvar, 1)
# VAE loss
vae_loss = tf.reduce_mean(recon_loss + kl_loss)
```

How do we train a Variational Autoencoder?

We have a sampling operation (stochastic node) in the computational graph!

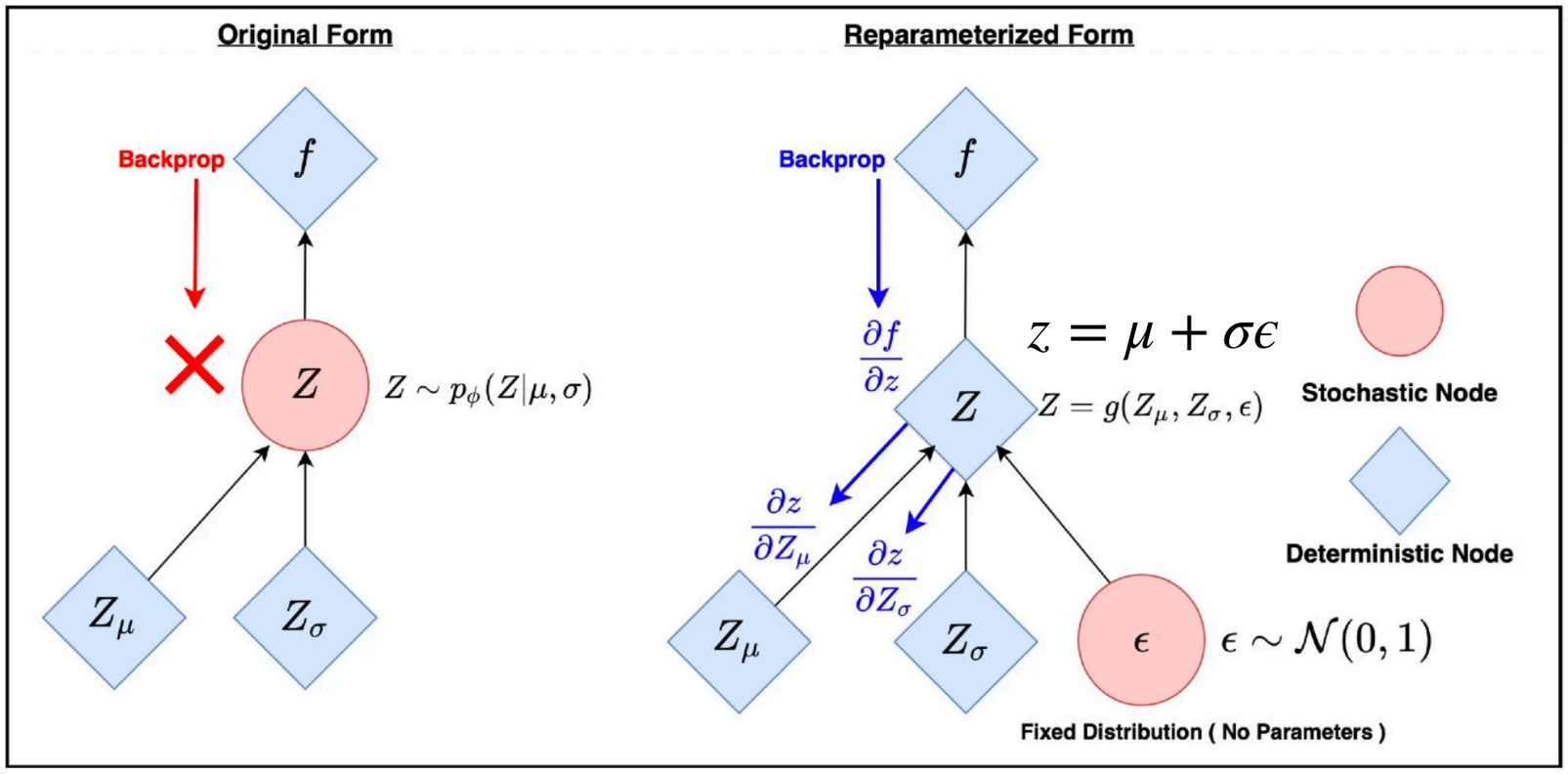
In fact this is not a problem!

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.

Reparametrization trick

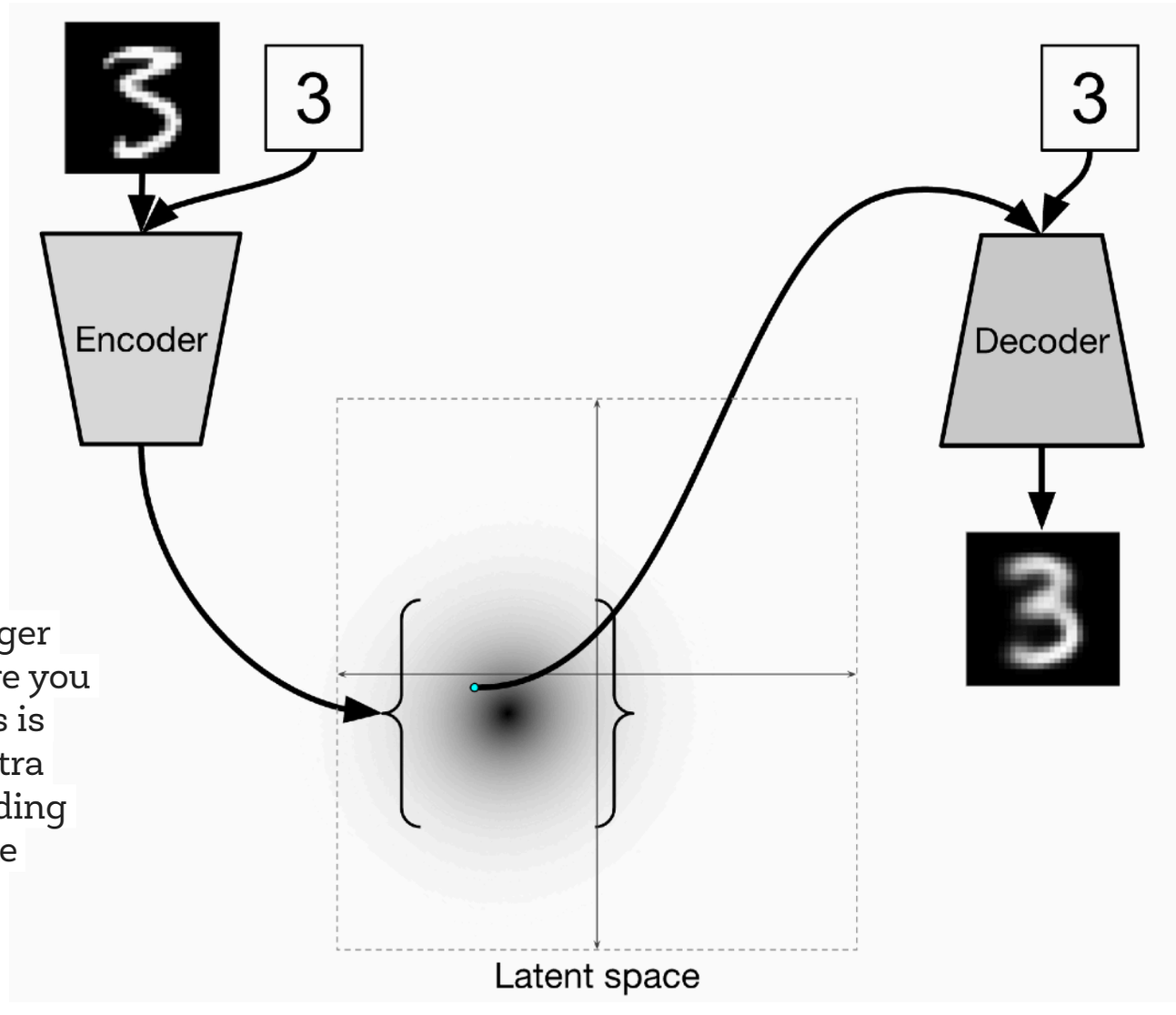



```python
def sample_z(args):
    mu, log_sigma = args
    eps = K.random_normal(shape=(m, n_z), mean=0., std=1.)
    return mu + K.exp(log_sigma / 2) * eps

# Sample z
z = Lambda(sample_z)([mu, log_sigma])
```

Conditional Variational Autoencoder

What about producing specific number instances on demand?

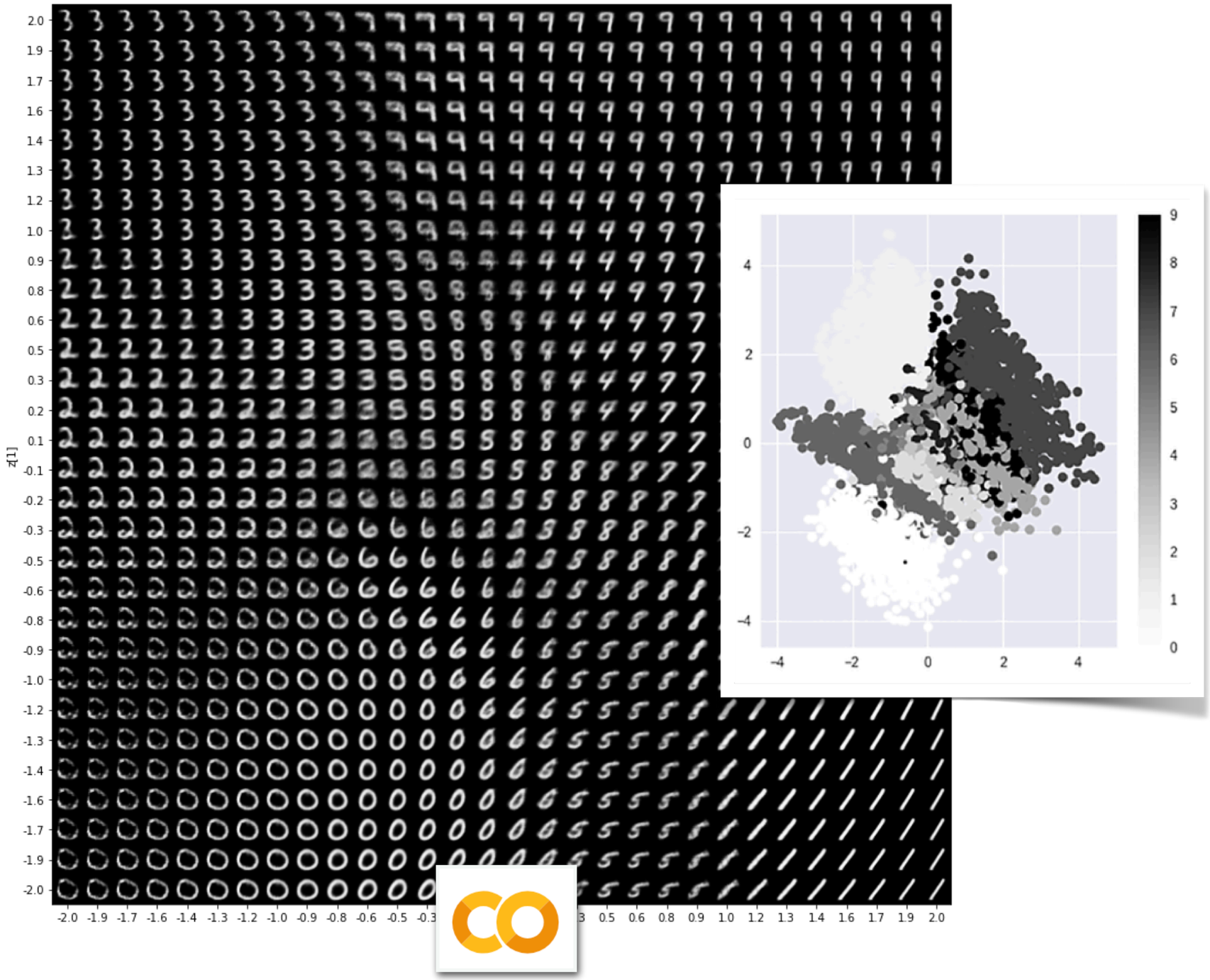
We can do this by adding an extra input (as a one-hot encoding) to both the encoder and the decoder:



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)



Is it enough?

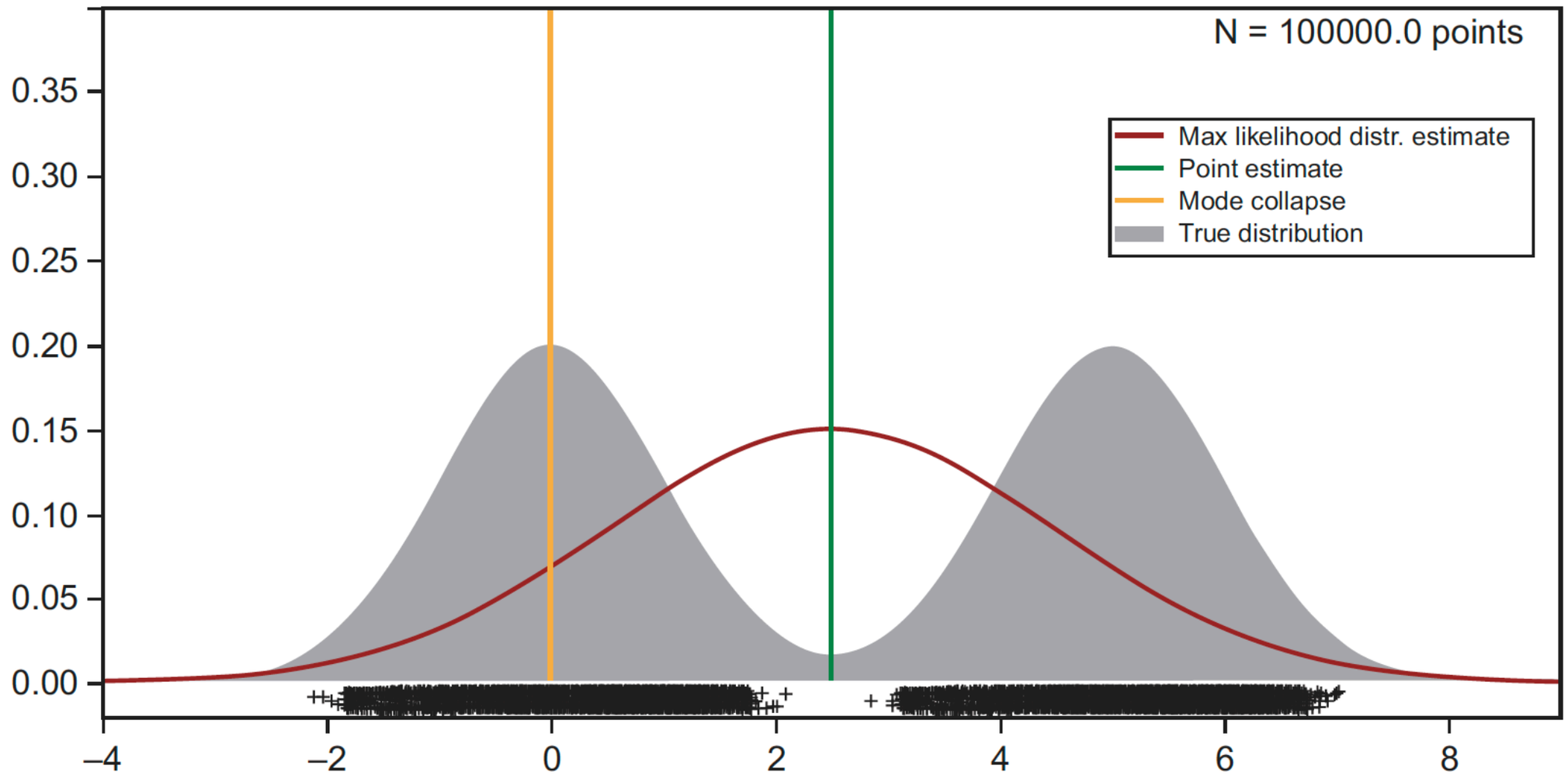
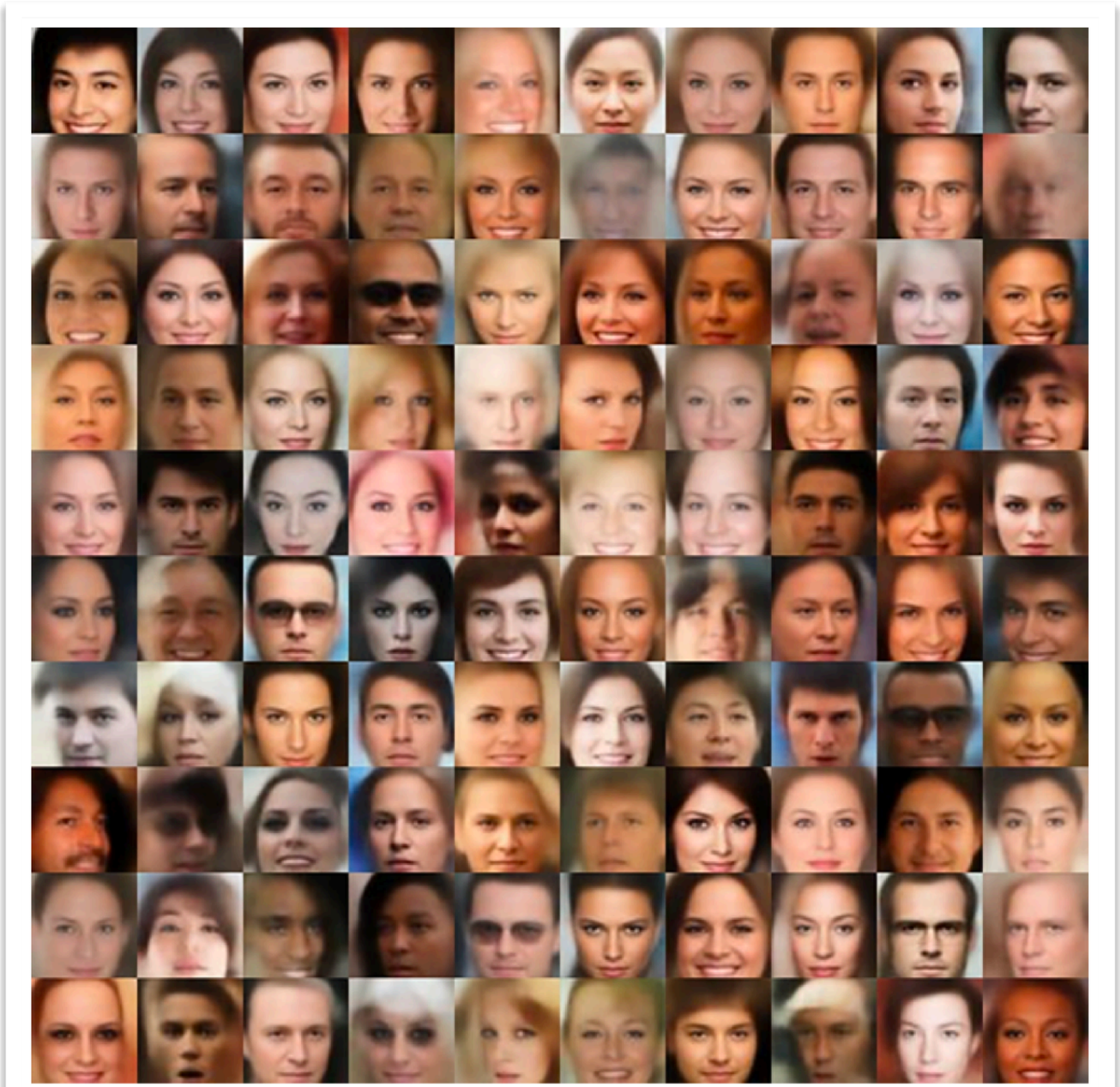


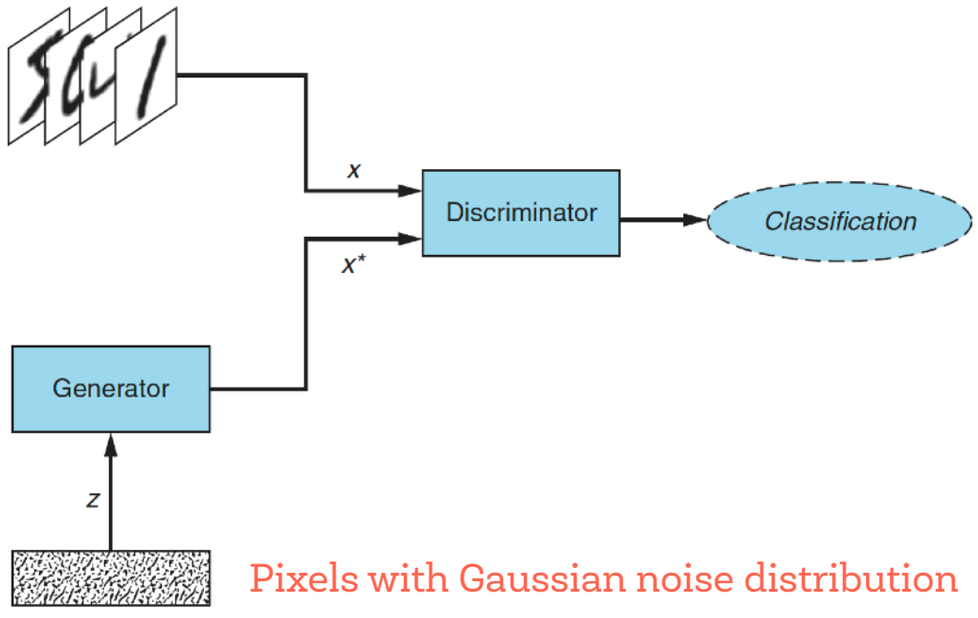
Figure 2.8 Maximum likelihood, point estimates, and true distributions. The gray (theoretical) distribution is bimodal rather than having a single mode. But because we have assumed this, our model is catastrophically wrong. Alternatively, we can get mode collapse, which is worth keeping in mind for chapter 5. This is especially true when we are using flavors of the KL, such as the VAE or early GANs.

Is it enough?



The VAE picks the safe path and makes the background blurry by choosing a “safe” pixel value, which minimizes the loss, but does not provide good images.

Generative Adversarial Networks



Pixels with Gaussian noise distribution

GAN training algorithm

For each training iteration ***do***

1. Train the Discriminator:
   a. Take a random mini-batch of real examples: x.
   b. Take a mini-batch of random noise vectors z and generate a mini-batch of fake examples: $G(z) = x^*$.
   c. Compute the classification losses for $D(x)$ and $D(x^*)$, and backpropagate the total error to update $\theta^{(D)}$ to *minimize* the classification loss.
2. Train the Generator:
   a. Take a mini-batch of random noise vectors z and generate a mini-batch of fake examples: $G(z) = x^*$.
   b. Compute the classification loss for $D(x^*)$, and backpropagate the loss to update $\theta^{(G)}$ to *maximize* the classification loss.

End for

Generative Adversarial Networks

```
%matplotlib inline

import matplotlib.pyplot as plt
import numpy as np

from keras.datasets import mnist
from keras.layers import Dense, Flatten, Reshape
from keras.layers.advanced_activations import LeakyReLU
from keras.models import Sequential
from keras.optimizers import Adam
```

```
img_rows = 28
img_cols = 28
channels = 1

img_shape = (img_rows, img_cols, channels)
```

Input image dimensions

```
z_dim = 100
```

Size of the noise vector, used as input to the Generator

Generative Adversarial Networks

```
def build_generator(img_shape, z_dim):
    model = Sequential()

    model.add(Dense(128, input_dim=z_dim))

    model.add(LeakyReLU(alpha=0.01))

    model.add(Dense(28 * 28 * 1, activation='tanh'))

    model.add(Reshape(img_shape))

    return model
```

Fully connected layer

Leaky ReLU activation

Output layer with tanh activation

Reshapes the Generator output to image dimensions

The reason for using tanh (as opposed to, say, sigmoid,) is that *tanh* tends to produce crisper images.

Generative Adversarial Networks

```
def build_discriminator(img_shape):

    model = Sequential()

    model.add(Flatten(input_shape=img_shape))

    model.add(Dense(128))

    model.add(LeakyReLU(alpha=0.01))

    model.add(Dense(1, activation='sigmoid'))

    return model
```

Flattens the input image

Fully connected layer

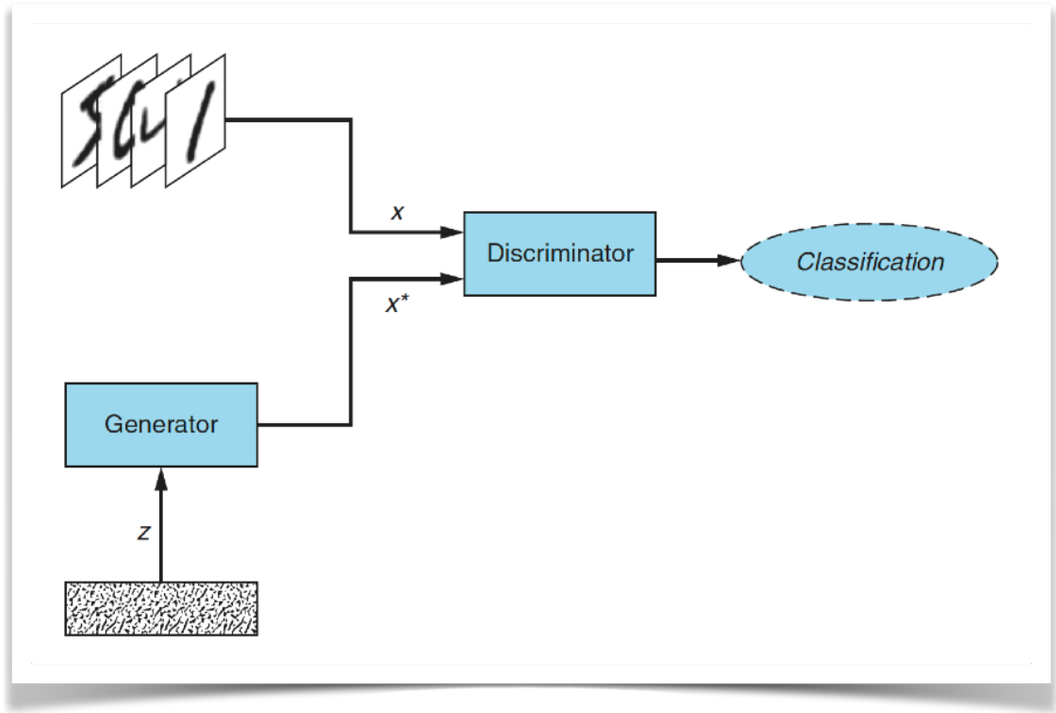
Leaky ReLU activation

Output layer with sigmoid activation

Generative Adversarial Networks

```
def build_gan(generator, discriminator):

    model = Sequential()

    model.add(generator)
    model.add(discriminator)

    return model
```

Combined Generator + Discriminator model



Generative Adversarial Networks

```
discriminator = build_discriminator(img_shape)
discriminator.compile(loss='binary_crossentropy',
                      optimizer=Adam(),
                      metrics=['accuracy'])

generator = build_generator(img_shape, z_dim)

discriminator.trainable = False

gan = build_gan(generator, discriminator)
gan.compile(loss='binary_crossentropy', optimizer=Adam())
```

Builds and compiles the Discriminator

Builds the Generator

Keeps Discriminator's parameters constant for Generator training

Builds and compiles GAN model with fixed Discriminator to train the Generator

GAN training loop

```
losses = []
accuracies = []
iteration_checkpoints = []


def train(iterations, batch_size, sample_interval):

    (X_train, _), (_, _) = mnist.load_data()

    X_train = X_train / 127.5 - 1.0
    X_train = np.expand_dims(X_train, axis=3)

    real = np.ones((batch_size, 1))

    fake = np.zeros((batch_size, 1))

    for iteration in range(iterations):


        idx = np.random.randint(0, X_train.shape[0], batch_size)
        imgs = X_train[idx]

        z = np.random.normal(0, 1, (batch_size, 100))
        gen_imgs = generator.predict(z)

        d_loss_real = discriminator.train_on_batch(imgs, real)
        d_loss_fake = discriminator.train_on_batch(gen_imgs, fake)
        d_loss, accuracy = 0.5 * np.add(d_loss_real, d_loss_fake)



        z = np.random.normal(0, 1, (batch_size, 100))
        gen_imgs = generator.predict(z)

        g_loss = gan.train_on_batch(z, real)

        if (iteration + 1) % sample_interval == 0:

            losses.append((d_loss, g_loss))
            accuracies.append(100.0 * accuracy)
            iteration_checkpoints.append(iteration + 1)


        print("%d [D loss: %f, acc.: %.2f%%] [G loss: %f]" %
              (iteration + 1, d_loss, 100.0 * accuracy, g_loss))

        sample_images(generator)
```

Loads the MNIST dataset

Rescales [0, 255] grayscale pixel values to [–1, 1]

Labels for real images: all 1s

Labels for fake images: all 0s

Gets a random batch of real images

Generates a batch of fake images

Trains the Discriminator

Generates a batch of fake images

Trains the Generator

Saves losses and accuracies so they can be plotted after training

Outputs training progress

Outputs a sample of generated images

GAN training loop

```
def sample_images(generator, image_grid_rows=4, image_grid_columns=4):

    z = np.random.normal(0, 1, (image_grid_rows * image_grid_columns, z_dim))

    gen_imgs = generator.predict(z)

    gen_imgs = 0.5 * gen_imgs + 0.5

    fig, axs = plt.subplots(image_grid_rows,
                            image_grid_columns,
                            figsize=(4, 4),
                            sharey=True,
                            sharex=True)

    cnt = 0
    for i in range(image_grid_rows):
        for j in range(image_grid_columns):
            axs[i, j].imshow(gen_imgs[cnt, :, :, 0], cmap='gray')
            axs[i, j].axis('off')
            cnt += 1
```

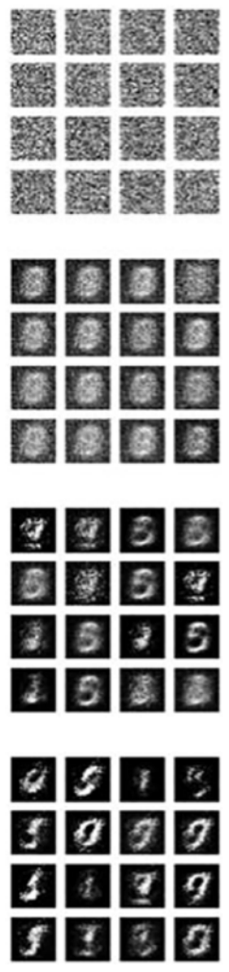
Sample random noise

Generates images from random noise

Rescales image pixel values to [0, 1]

Sets image grid

Outputs a grid of images



```
iterations = 20000
batch_size = 128
sample_interval = 1000

train(iterations, batch_size, sample_interval)
```

Sets hyperparameters

Trains the GAN for the specified number of iterations

We can improve the quality of the generated images by using a more complex and powerful neural network architecture for the Generator and Discriminator (f.e. convolutional neural networks).

GAN problems

Many GAN models suffer the following major problems:

- **Non-convergence**: the model parameters oscillate, destabilize and never converge,
- **Mode collapse**: the generator collapses which produces limited varieties of samples,
- **Diminished gradient**: the discriminator gets too successful that the generator gradient vanishes and learns nothing,
- Unbalance between the generator and discriminator causing **overfitting**, &
- Highly sensitive to the **hyperparameter** selections.

Solutions:

- Advanced loss funtions.
- Advanced training strategies.
- Etc.

GAN problems

The objective evaluation of GAN generator models remains an open problem.

As such, GAN generator models are evaluated based on the **quality of the images generated**, often in the context of the target problem domain.

Twenty-four quantitative techniques for evaluating GAN generator models are listed below.

1. Average Log-likelihood
2. Coverage Metric
3. Inception Score (IS)
4. Modified Inception Score (m-IS)
5. Mode Score
6. AM Score
7. Frechet Inception Distance (FID)
8. Maximum Mean Discrepancy (MMD)
9. The Wasserstein Critic
10. Birthday Paradox Test
11. Classifier Two-sample Tests (C2ST)
12. Classification Performance
13. Boundary Distortion
14. Number of Statistically-Different Bins (NDB)
15. Image Retrieval Performance
16. Generative Adversarial Metric (GAM)
17. Tournament Win Rate and Skill Rating
18. Normalized Relative Discriminative Score (NRDS)
19. Adversarial Accuracy and Adversarial Divergence
20. Geometry Score
21. Reconstruction Error
22. Image Quality Measures (SSIM, PSNR and Sharpness Difference)
23. Low-level Image Statistics
24. Precision, Recall and F1 Score

https://machinelearningmastery.com/how-to-evaluate-generative-adversarial-networks/

Applications

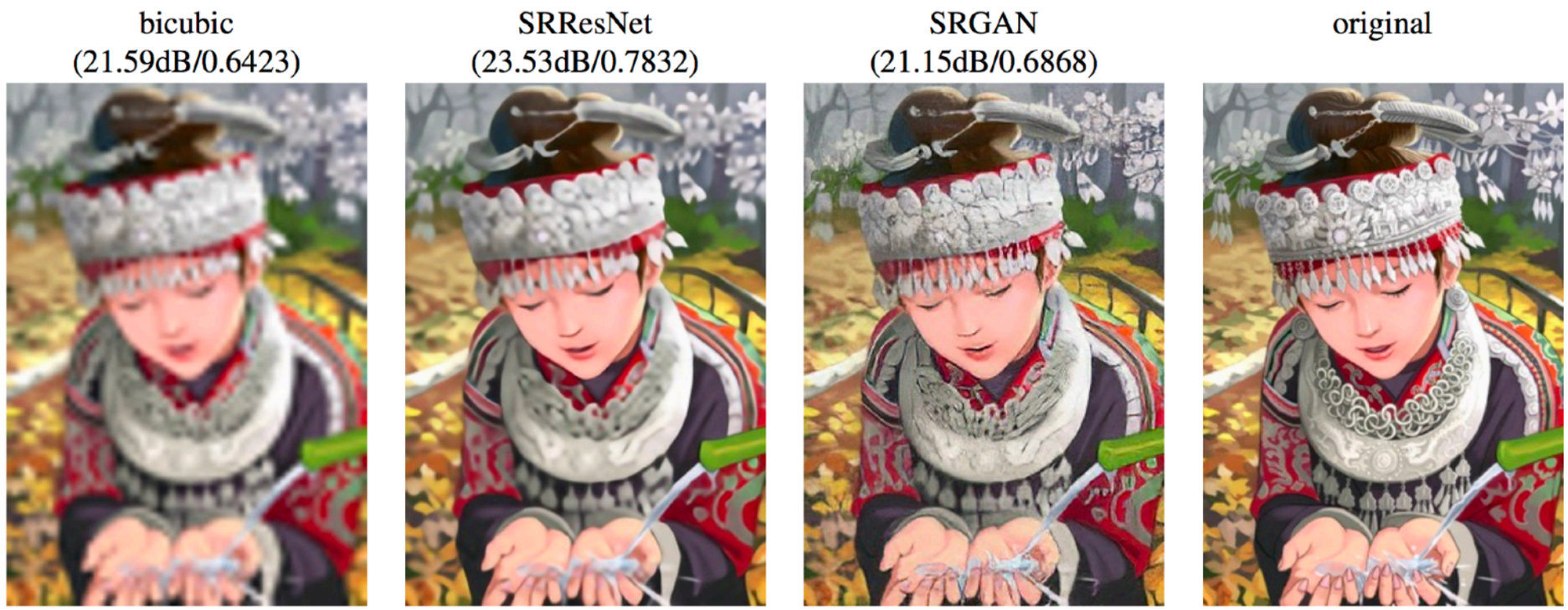
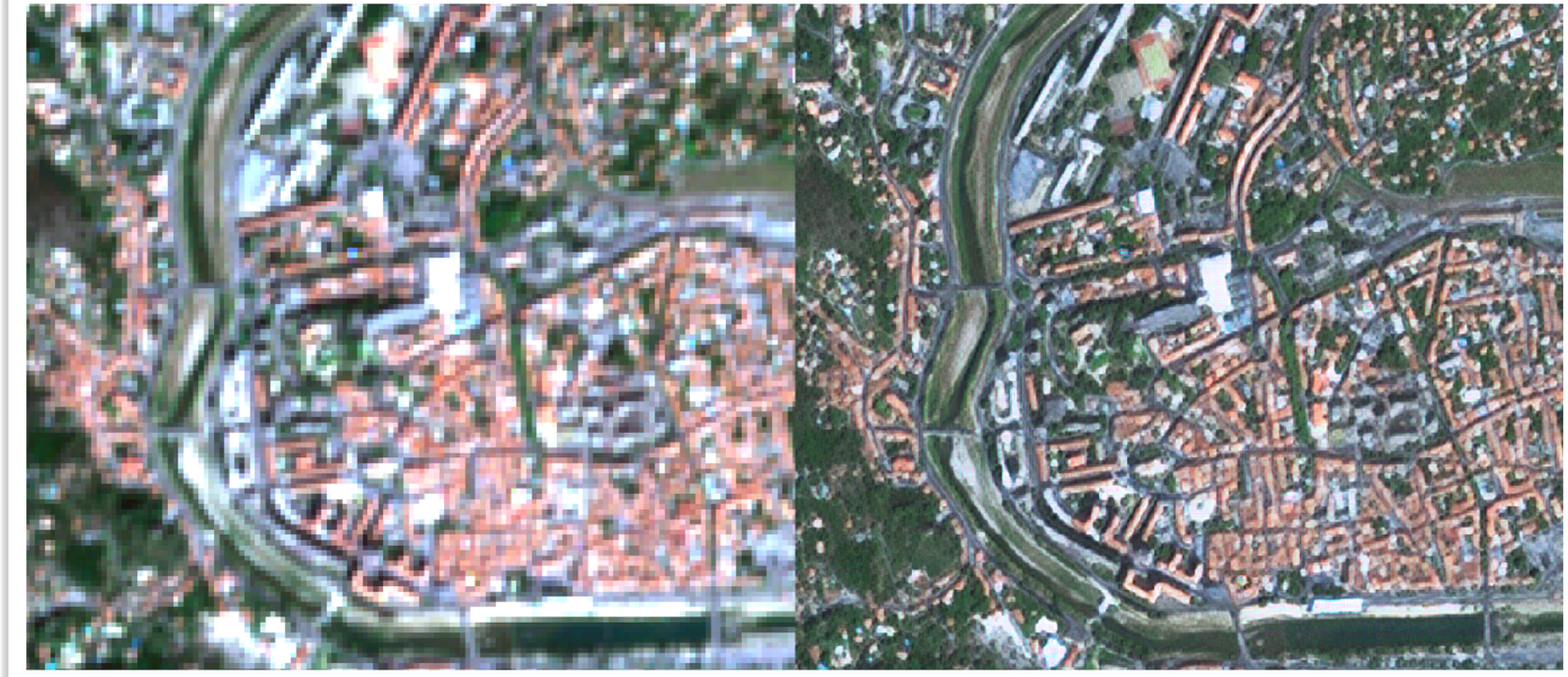


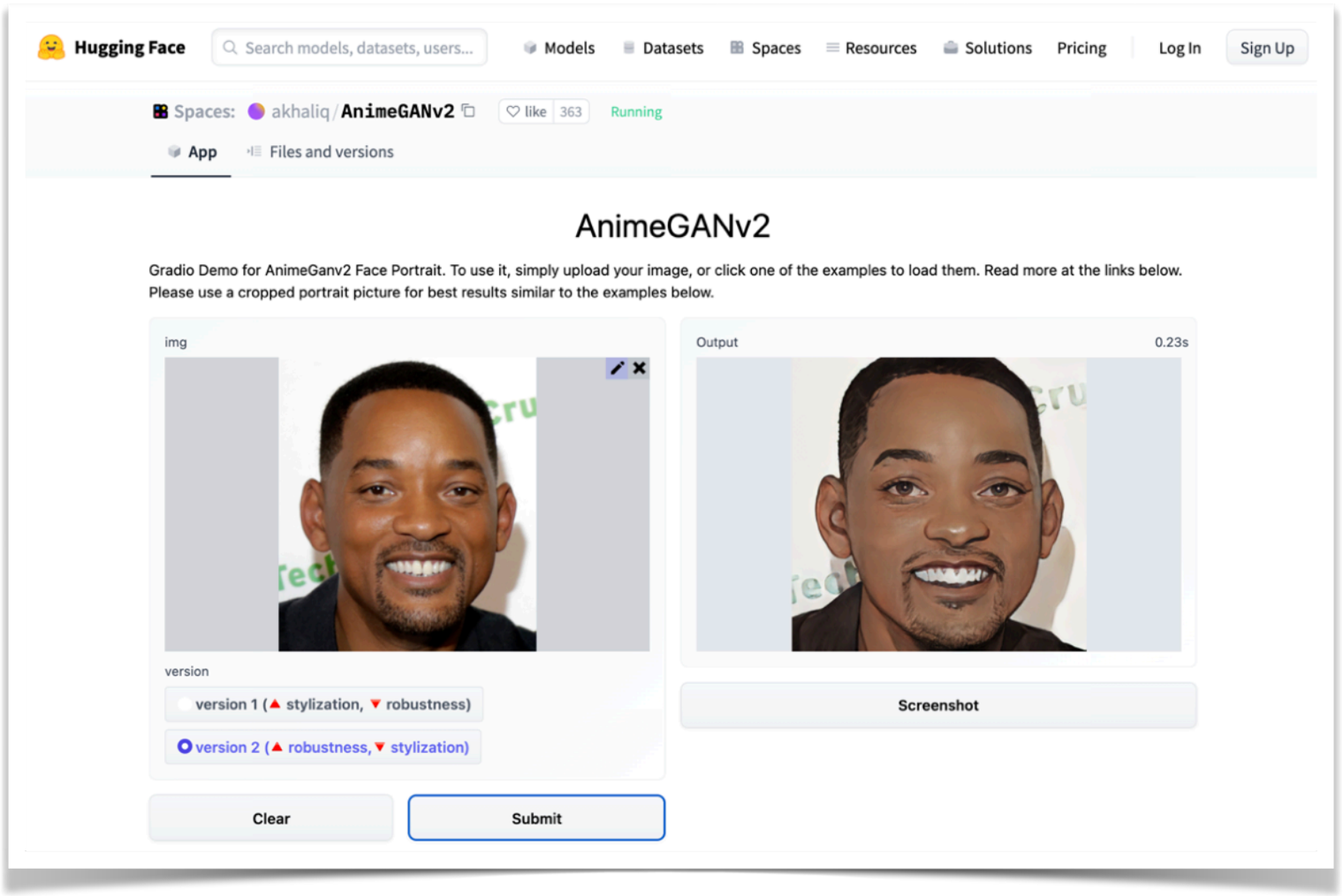
Figure 2: From left to right: bicubic interpolation, deep residual network optimized for MSE, deep residual generative adversarial network optimized for a loss more sensitive to human perception, original HR image. Corresponding PSNR and SSIM are shown in brackets. [4× upscaling]

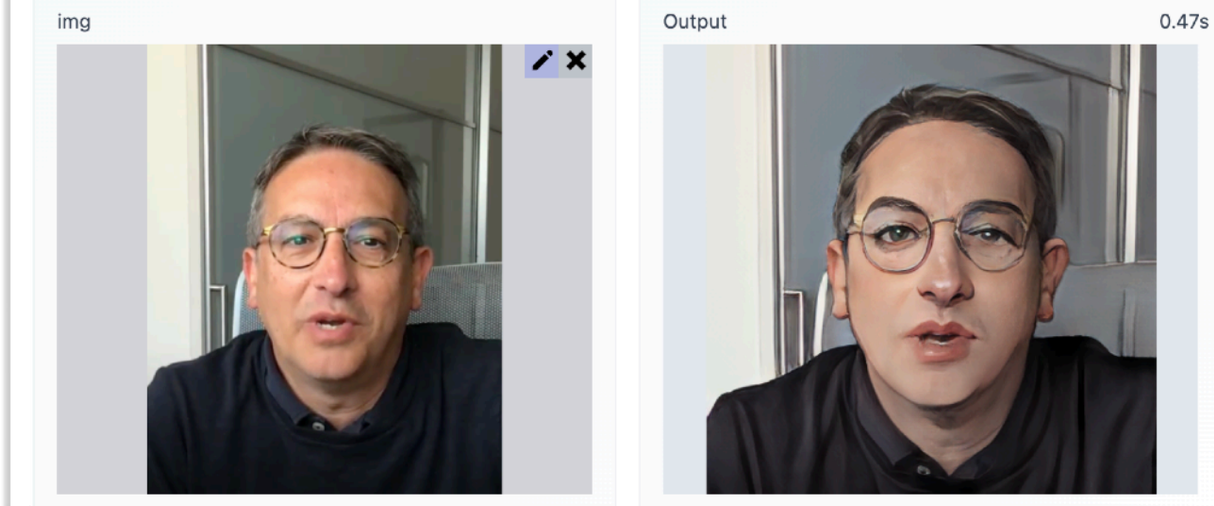
Applications



Enhancement of 10m Sentinel-2 images at 1.5m

Applications






That's all Folks!
Deep Learning 2021