Variational Autoencoder

Contents

Introduction

Variational autoencoders, simultaneously discovered by Kingma & Welling in December 2013, and Rezende, Mohamed & Wierstra in January 2014, are a kind of generative model that is especially appropriate for the task of image editing via concept vectors. They are a modern take on autoencoders – a type of network that aims to "encode" an input to a low-dimensional latent space then "decode" it back – that mixes ideas from deep learning with Bayesian inference.

Autoenconder (AE)

A classical image autoencoder takes an image, maps it to a latent vector space via an "encoder" module, then decode it back to an output with the same dimensions as the original image, via a "decoder" module. It is then trained by using as target data the *same images* as the input images, meaning that the autoencoder learns to reconstruct the original inputs. By imposing various constraints on the "code", i.e. the output of the encoder, one can get the autoencoder to learn more or less interesting latent representations of the data. Most commonly, one would constraint the code to be very low-dimensional and sparse (i.e. mostly zeros), in which case the encoder acts as a way to compress the input data into fewer bits of information.

In practice, such classical autoencoders don't lead to particularly useful or well-structured latent spaces. They're not particularly good at compression, either. For these reasons, they have largely fallen out of fashion over the past years. Variational autoencoders, however, augment autoencoders with a little bit of statistical magic that forces them to learn continuous, highly structured latent spaces. They have turned out to be a very powerful tool for image generation.

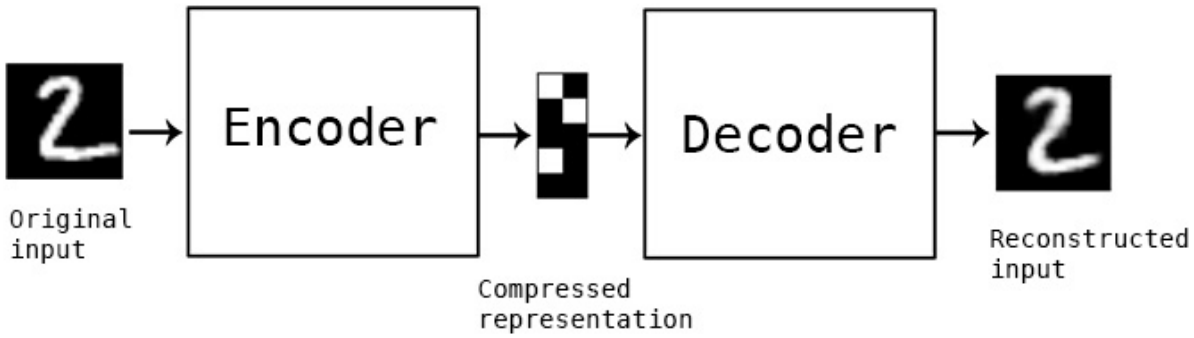


Figure 1: Autoencoder

Variational Autoencoder (VAE)

A variational autoencoder, instead of compressing its input image into a fixed "code" in the latent space, turns the image into the parameters of a statistical distribution: a mean and a variance. Essentially, this means that we are assuming that the input image has been generated by a statistical process, and that the *randomness of this process should be taken into accounting during encoding and decoding*. The VAE then uses the mean and variance parameters to randomly sample one element of the distribution, and decodes that element back to the original input. The stochasticity of this process *improves robustness and forces the latent space to encode meaningful representations everywhere*, i.e. every point sampled in the latent will be decoded to a valid output.

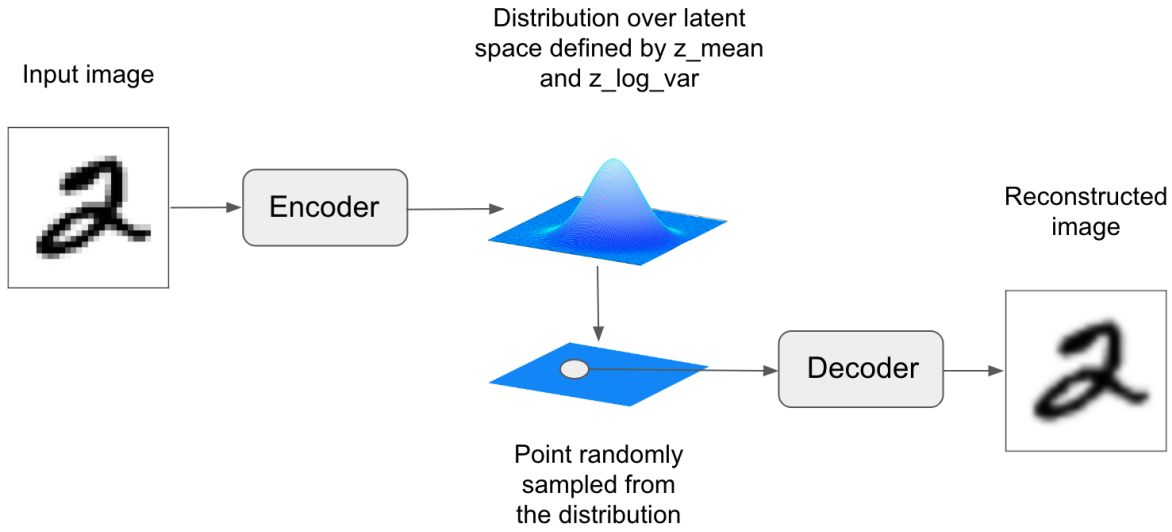


Figure 2: VAE

In technical terms, here is how a variational autoencoder works. First, an encoder module turns the input samples `input_img` into two parameters (possibly multivariate) in a latent space of representations, which we will note `z_mean` and `z_log_variance`. Then, we randomly sample a point `z` from the latent normal distribution that is assumed to generate the input image, via `z = z_mean + exp(z_log_variance) * epsilon`,

where epsilon is a random tensor of small values. Finally, a decoder module will map this point in the latent space back to the original input image. Because `epsilon` is random, the process ensures that every point that is close to the latent location where we encoded `input_img` (`z-mean`) can be decoded to something similar to `input_img`, thus forcing the latent space to be continuously meaningful. Any two close points in the latent space will decode to highly similar images. Continuity, combined with the low dimensionality of the latent space, forces every direction in the latent space to encode a meaningful axis of variation of the data, making the latent space very structured and thus highly suitable to manipulation via concept vectors.

The parameters of a VAE are trained via two loss functions: first, a **reconstruction loss** that forces the decoded samples to match the initial inputs, and a **regularization loss**, which helps in learning well-formed latent spaces and reducing overfitting to the training data.

Let's quickly go over a Keras implementation of a VAE. Schematically, it looks like this:

```
# Encode the input into a mean and variance parameter
c(z_mean, z_log_variance) %<% encoder(input_img)

# Draws a latent point using a small random epsilon
z <- z_mean + exp(z_log_variance) * epsilon

# Decodes z back to an image
reconstructed_img <- decoder(z)

# Creates a model
model <- keras_model(input_img, reconstructed_img)

# Then train the model using 2 losses:
# a reconstruction loss and a regularization loss
```

Variational autoencoder to generate handwritten digits

This VAE example is to generate the handwritten digits from MNIST database. Here is the encoder network we will use: a very simple convnet which maps the input image `x` to two vectors, `z_mean` and `z_log_variance`.

```
# With TF-2, you can still run this code due to the following line:
if (tensorflow::tf$executing_eagerly())
  tensorflow::tf$compat$v1$disable_eager_execution()

library(keras)

img_shape <- c(28, 28, 1)
batch_size <- 16
latent_dim <- 2L  # Dimensionality of the latent space: a plane

input_img <- layer_input(shape = img_shape)

x <- input_img %>%
  layer_conv_2d(filters = 32, kernel_size = 3, padding = "same",
                activation = "relu") %>%
  layer_conv_2d(filters = 64, kernel_size = 3, padding = "same",
                activation = "relu", strides = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = 3, padding = "same",
                activation = "relu") %>%
```

```
  layer_conv_2d(filters = 64, kernel_size = 3, padding = "same",
                activation = "relu")

shape_before_flattening <- k_int_shape(x) #

x <- x %>%
  layer_flatten() %>%
  layer_dense(units = 32, activation = "relu")

z_mean <- x %>%
  layer_dense(units = latent_dim)

z_log_var <- x %>%
  layer_dense(units = latent_dim)
```

Next is the code for using `z_mean` and `z_log_var`, the parameters of the statistical distribution assumed to have produced `input_img`, to generate a latent space point `z`. Here, you wrap some arbitrary code (built on top of Keras backend primitives) into a `layer_lambda()`, which wraps an R function into a layer. In Keras, everything needs to be a layer, so code that isn't part of a built-in layer should be wrapped in a `layer_lambda()` (or in a custom layer).

```
# Sampling function
sampling <- function(args) {
  c(z_mean, z_log_var) %<-% args
  epsilon <- k_random_normal(shape = list(k_shape(z_mean)[1], latent_dim),
                             mean = 0, stddev = 1)
  z_mean + k_exp(z_log_var) * epsilon
}

# Point randomly sampled
z <- list(z_mean, z_log_var) %>%
  layer_lambda(sampling)
```

This is the decoder implementation: we reshape the vector `z` to the dimensions of an image, then we use a few convolution layers to obtain a final image output that has the same dimensions as the original `input_img`.

```
# This is the input where we will feed `z`.
decoder_input <- layer_input(k_int_shape(z)[-1])

x <- decoder_input %>%
  # Upsample to the correct number of units
  layer_dense(units = prod(as.integer(shape_before_flattening[-1])),
              activation = "relu") %>%
  # Reshapes into an image of the same shape as before the last flatten layer
  layer_reshape(target_shape = shape_before_flattening[-1]) %>%
  # Applies and then reverses the operation to the initial stack of
  # convolution layers
  layer_conv_2d_transpose(filters = 32, kernel_size = 3, padding = "same",
                          activation = "relu", strides = c(2, 2)) %>%
  layer_conv_2d(filters = 1, kernel_size = 3, padding = "same",
                activation = "sigmoid")
  # We end up with a feature map of the same size as the original input.
```

```
# This is our decoder model.
decoder <- keras_model(decoder_input, x)

# We then apply it to `z` to recover the decoded `z`.
z_decoded <- decoder(z)
```

The dual loss of a VAE doesn't fit the traditional expectation of a sample-wise function of the form `loss(input, target)`. Thus, we set up the loss by writing a custom layer with internally leverages the built-in `add_loss` layer method to create an arbitrary loss.

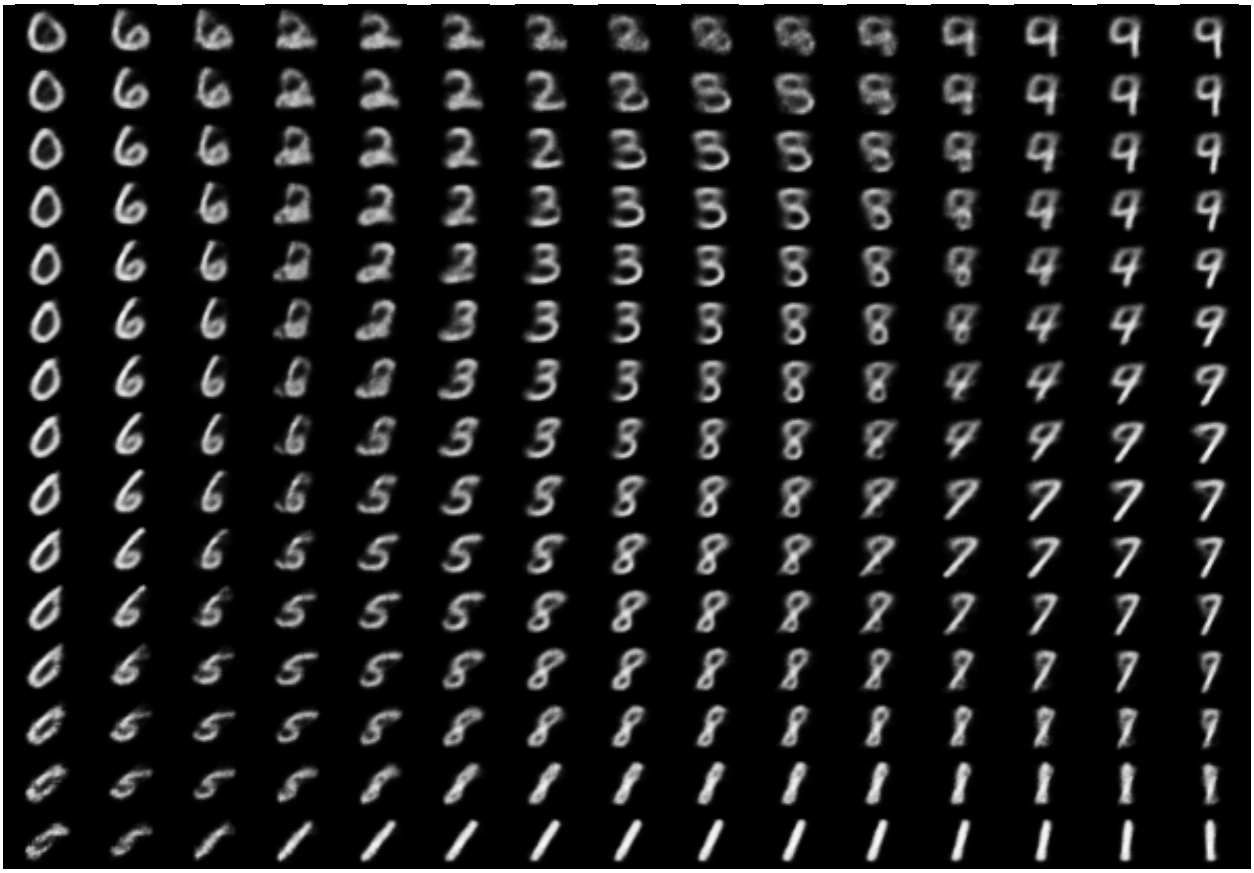
```
library(R6)

CustomVariationalLayer <- R6Class("CustomVariationalLayer",

  inherit = KerasLayer,

  public = list(

    vae_loss = function(x, z_decoded) {
      x <- k_flatten(x)
      z_decoded <- k_flatten(z_decoded)
      xent_loss <- metric_binary_crossentropy(x, z_decoded)
      kl_loss <- -5e-4 * k_mean(
        1 + z_log_var - k_square(z_mean) - k_exp(z_log_var),
        axis = -1L
      )
      k_mean(xent_loss + kl_loss)
    },

    call = function(inputs, mask = NULL) {
      x <- inputs[[1]]
      z_decoded <- inputs[[2]]
      loss <- self$vae_loss(x, z_decoded)
      self$add_loss(loss, inputs = inputs)
      x
    }
  )
)

layer_variational <- function(object) {
  create_layer(CustomVariationalLayer, object, list())
}

# Call the custom layer on the input and the decoded output to obtain
# the final model output
y <- list(input_img, z_decoded) %>%
  layer_variational()
```

Finally, we instantiate and train the model. Since the loss has been taken care of in our custom layer, we don't specify an external loss at compile time (`loss = NULL`), which in turns means that we won't pass target data during training (as you can see we only pass `x_train` to the model in `fit`).

```
vae <- keras_model(input_img, y)

vae %>% compile(
  optimizer = "rmsprop",
  loss = NULL
)

# Trains the VAE on MNIST digits
mnist <- dataset_mnist()
c(c(x_train, y_train), c(x_test, y_test)) %<-% mnist

x_train <- x_train / 255
x_train <- array_reshape(x_train, dim =c(dim(x_train), 1))

x_test <- x_test / 255
x_test <- array_reshape(x_test, dim =c(dim(x_test), 1))

vae %>% fit(
  x = x_train, y = NULL,
  epochs = 10,
  batch_size = batch_size,
  validation_data = list(x_test, NULL)
)
```

Once such a model is trained – e.g. on MNIST, in our case – we can use the `decoder` network to turn arbitrary latent space vectors into images:

```
n <- 15            # Number of rows / columns of digits
digit_size <- 28   # Height / width of digits in pixels

# Transforms linearly spaced coordinates on the unit square through the inverse
# CDF (ppf) of the Gaussian to produce values of the latent variables z,
# because the prior of the latent space is Gaussian
grid_x <- qnorm(seq(0.05, 0.95, length.out = n))
grid_y <- qnorm(seq(0.05, 0.95, length.out = n))

op <- par(mfrow = c(n, n), mar = c(0,0,0,0), bg = "black")
for (i in 1:length(grid_x)) {
  yi <- grid_x[[i]]
  for (j in 1:length(grid_y)) {
    xi <- grid_y[[j]]
    z_sample <- matrix(c(xi, yi), nrow = 1, ncol = 2)
    z_sample <- t(replicate(batch_size, z_sample, simplify = "matrix"))
    x_decoded <- decoder %>% predict(z_sample, batch_size = batch_size)
    digit <- array_reshape(x_decoded[1,,,], dim = c(digit_size, digit_size))
    plot(as.raster(digit))
  }
}
```



```
par(op)
```

The grid of sampled digits shows a completely continuous distribution of the different digit classes, with one digit morphing into another as you follow a path through latent space. Specific directions in this space have a meaning, e.g. there is a direction for “four-ness”, “one-ness”, etc.

References

Diederik P. Kingma and Max Welling, “AutoEncoding Variational Bayes,” https://arxiv.org/abs/1312.6114

Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra, “Stochastic Backpropagation and Approximate Inference in Deep Generative Models,” https://arxiv.org/abs/1401.4082.

Annex I

Complete loss function

In the VAE, our loss function is composed of two parts:

- *Reconstruction (or generative) loss* : This loss compares the model output with the model input. This can be the losses we used in the autoencoders, such as L2 loss.

- *Regularization (or latent) loss*: This loss compares the latent vector with a zero mean, unit variance Gaussian distribution. The loss we use here will be the Kullback-Leibler (KL) divergence loss. This loss term penalizes the VAE if it starts to produce latent vectors that are not from the desired distribution.

Mathematical expression:

$$\mathcal{L}(\Theta, \phi; \bar{x}, \bar{z}) = \mathbb{E}_{q_\phi(\bar{z}|\bar{x})}[\log p_\phi(\bar{x}|\bar{z})] - \mathcal{D}_{KL}(q_\phi(\bar{z}|\bar{x})||p(\bar{z}))$$

Kullback-Leibler divergence

The KL divergence between two probability distributions simply measures how much they diverge from each other. Minimizing the KL divergence here means optimizing the probability distribution parameters (μ and σ) to closely resemble that of the target distribution.

$$\frac{1}{2}\sum_{i}^{n}(\sigma_i^2 + \mu_i^2 - \log(\sigma_i) + 1)$$

For VAEs, the KL loss is equivalent to the sum of all the KL divergences between the component $X_i \sim N(\mu_i, \sigma_i^2)$ in X , and the standard normal. It's minimized when $\mu_i = 0$, $\sigma_i = 1$.

The reparametrization trick

The goal of reparametrization is to find a way to recast a statistical expression in a different way while preserving its meaning.

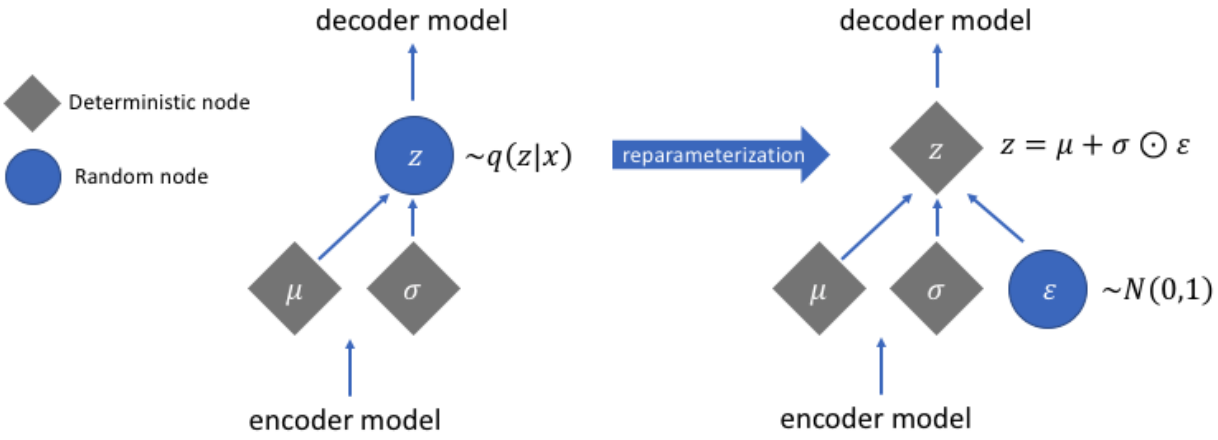


Figure 3: The reparametrization trick

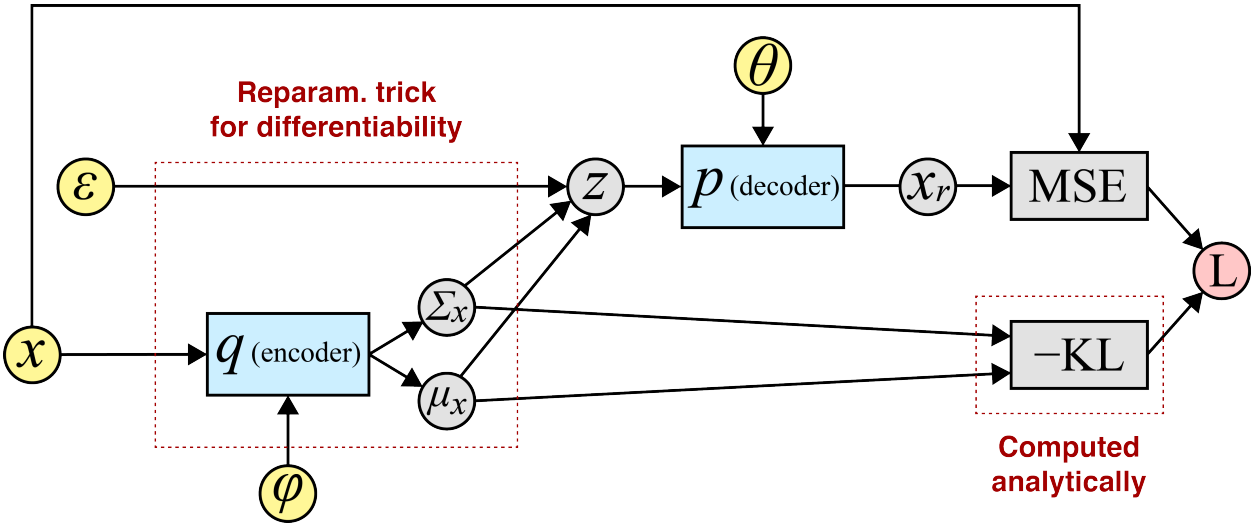


Figure 4: VAE process from point of statistical/loss view