Variational Autoencoders

April 29, 2025

This document is scanned and adapted from the original source by Auerlien Geron[1] .

Intro

An important category of autoencoders was introduced in 2013 by Diederik Kingma and Max Welling (https://homl.info/115) [6] and quickly became one of the most popular variants: variational autoencoders (VAEs).
VAEs are quite different from all the autoencoders we have discussed so far, in these particular ways:

- They are probabilistic autoencoders, meaning that their outputs are partly determined by chance, even after training (as opposed to denoising autoencoders, which use randomness only during training).
- Most importantly, they are generative autoencoders, meaning that they can generate new instances that look like they were sampled from the training set.

Both these properties make VAEs rather similar to RBMs, but they are easier to train, and the sampling process is much faster (with RBMs you need to wait for the network to stabilize into a "thermal equilibrium" before you can sample a new instance). As their name suggests, variational autoencoders perform variational Bayesian inference, which is an efficient way of carrying out approximate Bayesian inference. Recall that Bayesian inference means updating a probability distribution based on new data, using equations derived from Bayes' theorem. The original distribution is called the prior, while the updated distribution is called the posterior. In our case, we want to find a good approximation of the data distribution. Once we have that, we can sample from it.

Let's take a look at how VAEs work. Figure 17-11 (left) shows a variational autoencoder. You can recognize the basic structure of all autoencoders, with an encoder followed by a decoder (in this example, they both have two hidden layers), but there is a twist: instead of directly producing a coding for a given input, the encoder produces a mean coding $\boldsymbol{\mu}$ and a standard deviation $\boldsymbol{\sigma}$. The actual coding is then sampled randomly from a Gaussian distribution with mean μ and standard deviation σ. After that the decoder decodes the sampled coding normally. The right part of the diagram shows a training instance going through this autoencoder. First, the encoder produces $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$, then a coding is sampled randomly (notice that it is not

1 *Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow*, 3ed, ch. 17
6 Diederik Kingma and Max Welling, "Auto-Encoding Variational Bayes", arXiv preprint arXiv:1312.6114 (2013).

exactly located at μ), and finally this coding is decoded; the final output resembles the training instance.

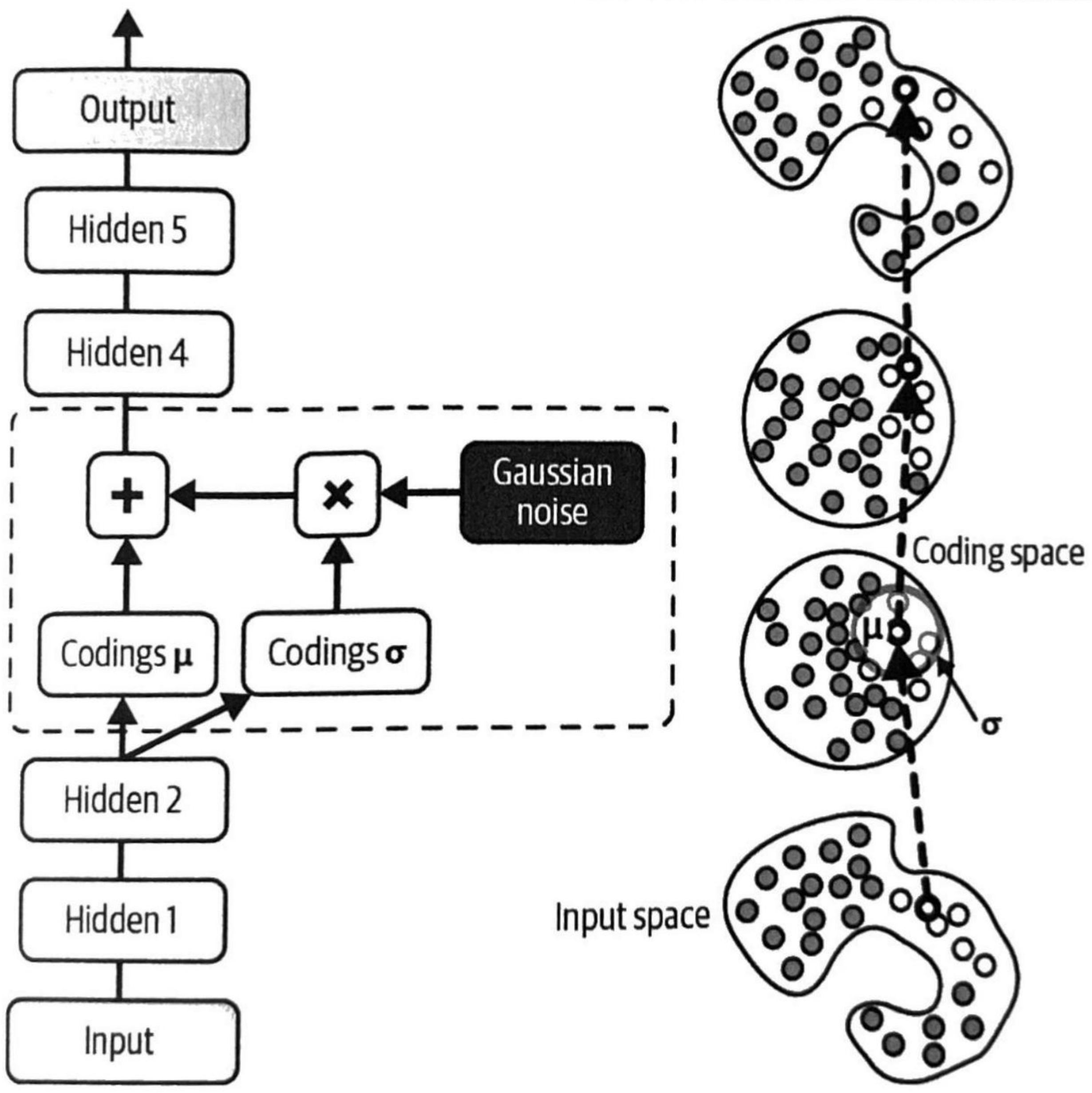

Figure 17-11. A variational autoencoder (left) and an instance going through it (right)
As you can see in the diagram, although the inputs may have a very convoluted distribution, a variational autoencoder tends to produce codings that look as though they were sampled from a simple Gaussian distribution:7 during training, the cost function (discussed next) pushes the codings to gradually migrate within the coding space (also called the latent space) to end up looking like a cloud of Gaussian points. One great consequence is that after training a variational autoencoder, you can very easily generate a new instance: just sample a random coding from the Gaussian distribution, decode it, and voilà!

Now, let's look at the cost function. It is composed of two parts. The first is the usual reconstruction loss that

7 Variational autoencoders are actually more general; the codings are not limited to Gaussian distributions.

pushes the autoencoder to reproduce its inputs. We can use the MSE for this, as we did earlier. The second is the latent loss that pushes the autoencoder to have codings that look as though they were sampled from a simple Gaussian distribution: it is the KL divergence between the target distribution (i.e., the Gaussian distribution) and the actual distribution of the codings. The math is a bit more complex than with the sparse autoencoder, in particular because of the Gaussian noise, which limits the amount of information that can be transmitted to the coding layer. This pushes the autoencoder to learn useful features. Luckily, the equations simplify, so the latent loss can be computed using Equation 17-3. [8]

Variational autoencoder's latent loss

$$\mathscr{L} = -\frac{1}{2}\sum_{i=1}^{n}\left[1+\log\left(\sigma_i^2\right)-\sigma_i^2-\mu_i^2\right]$$

In this equation, $\mathscr{L}$ is the latent loss, n is the codings' dimensionality, and μ_{i} and σ_{i} are the mean and standard deviation of the i^{th} component of the codings. The vectors μ and $\boldsymbol{\sigma}$ (which contain all the μ_{i} and σ_{i}) are output by the encoder, as shown in Figure 17-11 (left).

A common tweak to the variational autoencoder's architecture is to make the encoder output $\gamma = \log\left(\boldsymbol{\sigma}^2\right)$ rather than $\boldsymbol{\sigma}$. The latent loss can then be computed as shown in Equation 17-4. This approach is more numerically stable and speeds up training.

Equation 17-4. Variational autoencoder's latent loss, rewritten using $\gamma = \log\left(\sigma^2\right)$

$$\mathscr{L} = -\frac{1}{2}\sum_{i=1}^{n}\left[1+\gamma_i-\exp\left(\gamma_i\right)-\mu_i^2\right]$$

Let's start building a variational autoencoder for Fashion MNIST (as shown in Fig. ure 17-11, but using the γ tweak). First, we will need a custom layer to sample the codings, given μ and γ :

```
class Sampling(tf.keras.layers.Layer):
    def call(self, inputs):
        mean, log_var = inputs
        return tf.random.normal(tf.shape(log_var)) * tf.exp(log_var / 2) + mean
```

This Sampling layer takes two inputs: mean ($\boldsymbol{\mu}$) and log_var ($\boldsymbol{\gamma}$). It uses the function tf.random.normal() to sample a random vector (of the same shape as γ) from the Gaussian distribution, with mean 0 and standard deviation 1 . Then it multiplies it by exp(γ/2) (which is equal to $\boldsymbol{\sigma}$, as you can verify mathematically), and finally it adds $\boldsymbol{\mu}$ and returns the result. This samples a codings vector from the Gaussian distribution with mean μ and standard deviation $\boldsymbol{\sigma}$.

Next, we can create the encoder, using the functional API because the model is not entirely sequential:

```
codings_size = 10
```

8 For more mathematical details, check out the original paper on variational autoencoders, or Carl Doersch's great tutorial (https://homl.info/116) (2016).

```
inputs = tf.keras.layers.Input(shape=[28, 28])
z tf.keras.layers.Flatten()(inputs)
z = tf.keras.layers.Dense(150, activation="relu")(Z)
z tf.keras.layers.Dense(100, activation="relu")(Z)
codings_mean = tf.keras.layers.Dense(codings_size)(Z) # \mu
codings_log_var = tf.keras.layers.Dense(codings_size)(Z) # v
codings = Sampling()([codings_mean, codings_log_var])
variational_encoder = tf.keras.Model(
    inputs=[inputs], outputs=[codings_mean, codings_log_var, codings])
```

Note that the Dense layers that output codings_mean ($\boldsymbol{\mu}$) and codings_log_var (γ) have the same inputs (i.e., the outputs of the second Dense layer). We then pass both codings_mean and codings_log_var to the Sampling layer. Finally, the variational_encoder model has three outputs. Only the codings are required, but we add codings_mean and codings_log_var as well, in case we want to inspect their ralues. Now let's build the decoder:

```
decoder_inputs = tf.keras.layers.Input(shape=[codings_size])
x = tf.keras.layers.Dense(100, activation="relu")(decoder_inputs)
x = tf.keras.layers.Dense(150, activation="relu")(x)
x = tf.keras.layers.Dense(28 * 28)(x)
outputs = tf.keras.layers.Reshape([28, 28])(x)
variational_decoder = tf.keras.Model(inputs=[decoder_inputs], outputs=[outputs])
```

For this decoder, we could have used the sequential API instead of the functional API, since it is really just a simple stack of layers, virtually identical to many of the decoders we have built so far. Finally, let's build the variational autoencoder model:

```
_, _, codings = variational_encoder(inputs)
reconstructions = variational_decoder(codings)
variational_ae = tf.keras.Model(inputs=[inputs], outputs=[reconstructions])
```

We ignore the first two outputs of the encoder (we only want to feed the codings to the decoder). Lastly, we must add the latent loss and the reconstruction loss:

```
latent_loss = -0.5 * tf.reduce_sum(
    1 + codings_log_var - tf.exp(codings_log_var) - tf.square(codings_mean),
    axis=-1)
variational_ae.add_loss(tf.reduce_mean(latent_loss) / 784.)
```

We first apply Equation 17-4 to compute the latent loss for each instance in the batch, summing over the last axis. Then we compute the mean loss over all the instances in the batch, and we divide the result by 784 to ensure it has the appropriate scale compared to the reconstruction loss. Indeed, the variational autoencoder's reconstruction loss is supposed to be the sum of the pixel reconstruction errors, but when Keras computes the "mse" loss it computes the mean over all 784 pixels, rather than the sum. So, the reconstruction loss is 784 times smaller than we need it to be. We could define a custom loss to compute the sum rather than the mean, but it is simpler to divide the latent loss by 784 (the final loss will be 784 times smaller than it should be, but this just means that we should use a larger learning rate).

And finally, we can compile and fit the autoencoder!

```
variational_ae.compile(loss="mse", optimizer="nadam")
history = variational_ae.fit(X_train, X_train, epochs=25, batch_size=128,
    validation_data=(X_valid, X_valid))
```

Generating Fashion MNIST Images

Now let's use this variational autoencoder to generate images that look like fashion items. All we need to do is sample random codings from a Gaussian distribution and decode them:

```
codings = tf.random.normal(shape=[3 * 7, codings_size])
images = variational_decoder(codings).numpy()
```

Figure 17-12 shows the 12 generated images.

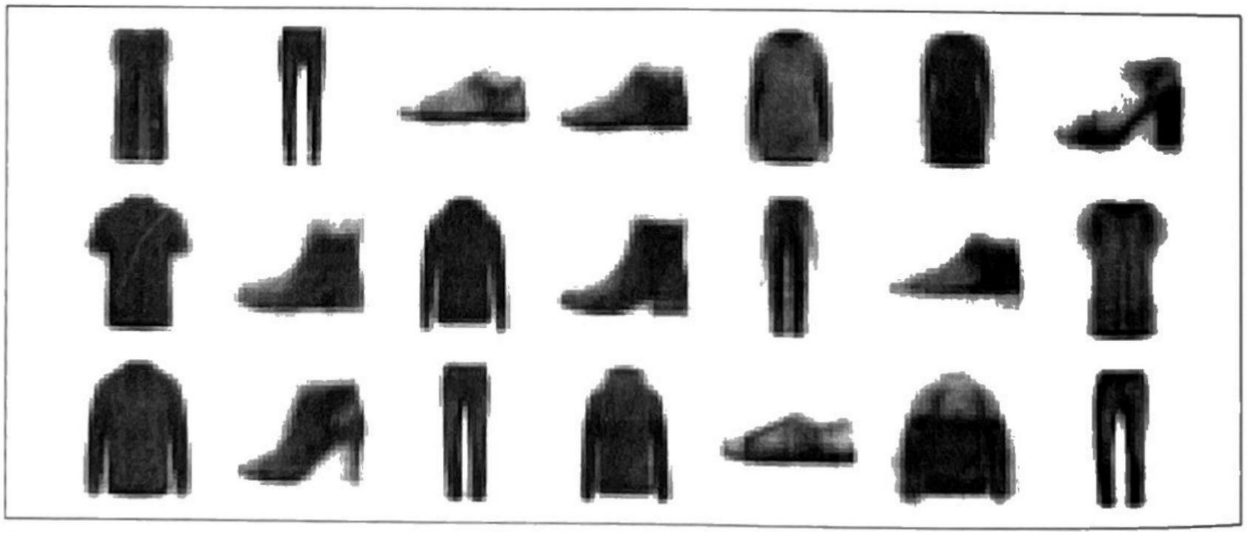

Figure 17-12. Fashion MNIST images generated by the variational autoencoder

The majority of these images look fairly convincing, if a bit too fuzzy. The rest are not great, but don't be too harsh on the autoencoder-it only had a few minutes to learn!

Variational autoencoders make it possible to perform semantic interpolation: instead of interpolating between two images at the pixel level, which would look as if the two images were just overlaid, we can interpolate at the codings level. For example, let's take a few codings along an arbitrary line in latent space and decode them. We get a sequence of images that gradually go from pants to sweaters (see Figure 17-13):

```
codings = np.zeros([7, codings_size])
codings[:, 3] = np.linspace(-0.8, 0.8, 7) # axis 3 looks best in this case
images = variational_decoder(codings).numpy()
```

Assignment

You should implement the code in this section and generate Fashion MNIST items as suggested. Vary some parameters, vary the model, see what happens. In particular,

- Obviously you can tweak the number of epochs, etc to improve the fit. Or maybe 25 is enough?
- Determine the effect of the latent loss. What do your generated images look like with the standard loss function?
- What effect does the sampling distribution on the input have? What happens if you sample from a uniform distribution instead of a Gaussian distribution?
- Any ideas for making the generated images less blurry?
- Now find your own favorite images dataset and make a VAE to generate samples from scratch.