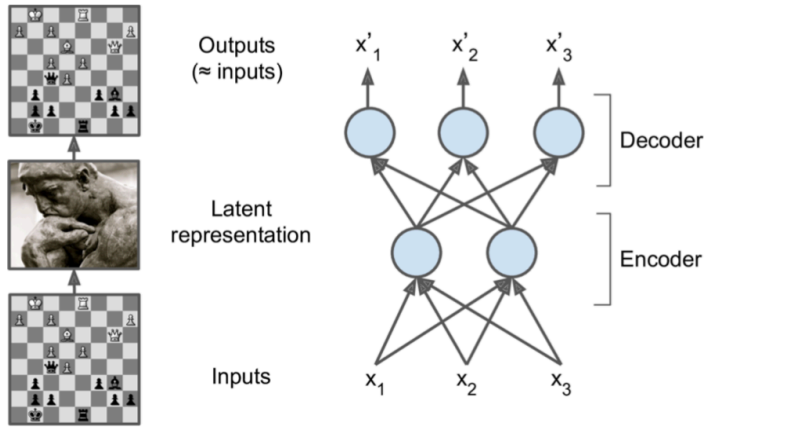
Autoencoders (Chap 17)

- We can think of autoencoders as a form of self-supervised learning – one where supervised learning is used with automatically generated labels.
- For autoencoders, the labels are just equal to the inputs.
- The autoencoders learn useful codings (more economical representation of the data based on discovered patterns)

$$40, 27, 25, 36, 81, 57, 10, 73, 19, 68$$
$$50, 48, 46, 44, 42, 40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14$$



The chess memory experiment (left) and a simple autoencoder (right)

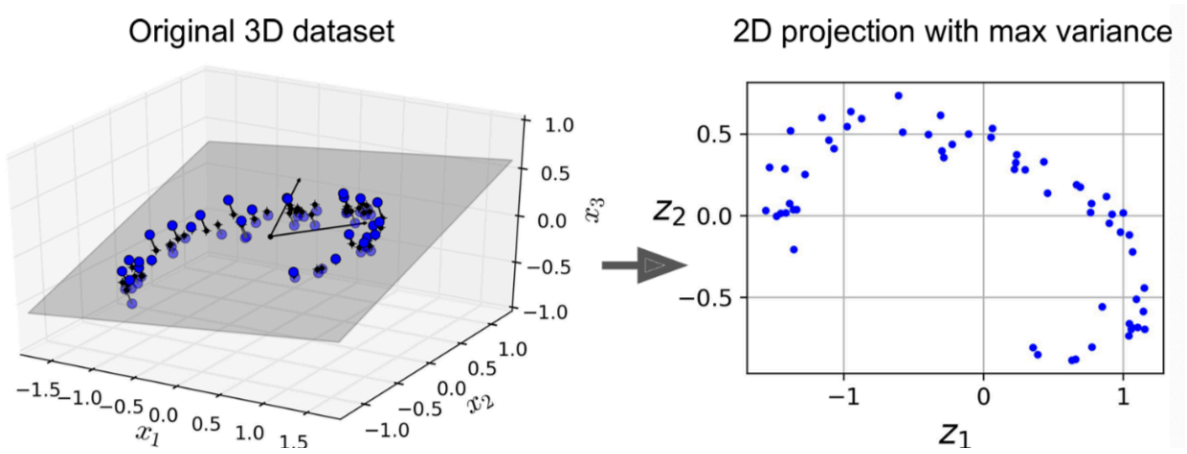
PCA with an Undercomplete Linear Autoencoder

- In this case the activations are linear and the cost function is MSE
- 2 parts – encoder & decoder with 1 Dense layer
- # Outputs = # Inputs

```
encoder = keras.models.Sequential([keras.layers.Dense(2, input_shape=[3])])
decoder = keras.models.Sequential([keras.layers.Dense(3, input_shape=[2])])
autoencoder = keras.models.Sequential([encoder, decoder])

autoencoder.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=0.1))
```

```
history = autoencoder.fit(X_train, X_train, epochs=20)
codings = encoder.predict(X_train)
```

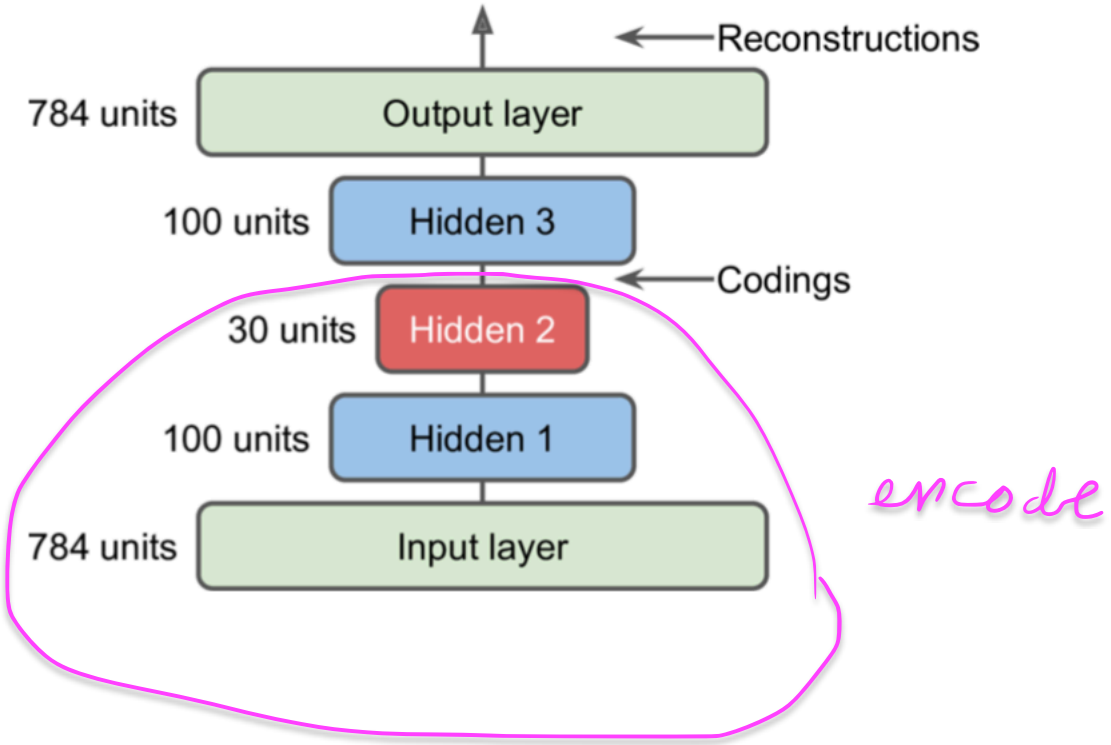


Stacked Autoencoders

- Called also deep autoencoders - can have multiple hidden layers.
- Adding more layers helps the autoencoder learn more complex codings but can make the autoencoder too “powerful”. E.g. an encoder which learns to map each input to a single arbitrary number (and the decoder learns the reverse mapping). Such an autoencoder will reconstruct the training data perfectly, but it will not have learned any useful data representation in the process (and it is unlikely to generalize well to new instances).
- The architecture of a stacked autoencoder is typically symmetrical with regard to the central hidden layer (the coding layer).

28 x 28

- For example, an autoencoder for MNIST may have 784 inputs, followed by a hidden layer with 100 neurons, then a central hidden layer of 30 neurons, then another hidden layer with 100 neurons, and an output layer with 784 neurons.



- When compiling the stacked autoencoder, the binary cross-entropy loss is used instead of the MSE. The reconstruction task is considered as a multilabel binary classification problem: each pixel intensity represents the probability that the pixel should be black. The model converges faster.

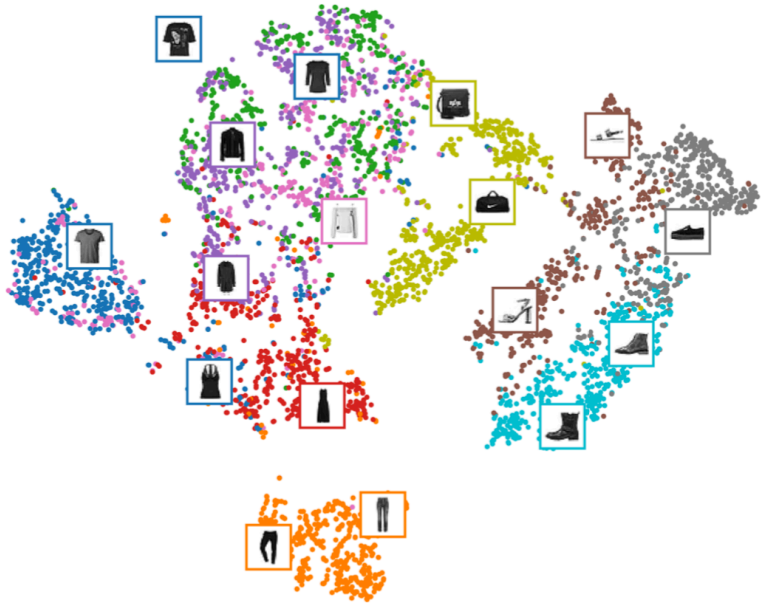
- We can check the quality of training by comparing plots of the input and the reconstructed inputs (the outputs). They should look similar

Original images (top) and their reconstructions (bottom)

The blur at the bottom indicates the need for longer training, a deeper model or higher codings dimension.

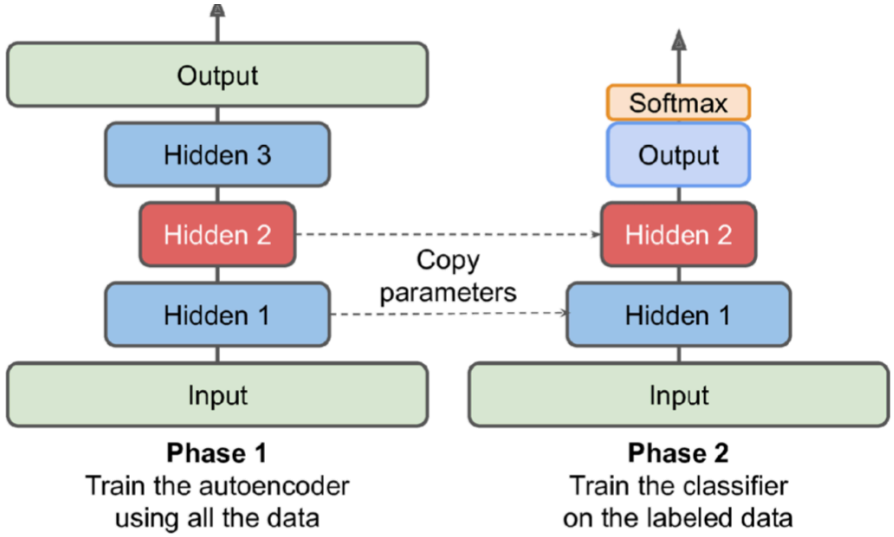
Visualizing the Reconstructions – MNIST example

- One big advantage of autoencoders is that they can handle large datasets, with many instances and many features.
- One visualization strategy use an autoencoder to reduce the dimensionality to a reasonable level, then visualize the result e.g. by t-SNE



Unsupervised Pretraining Using Stacked Autoencoders

- For a complex supervised task, when you do not have a lot of labeled training data, one solution is to find a neural network that performs a similar task and reuse its lower layers. (transfer learning)
- Similarly, with a large dataset but most of it is unlabeled, first train a stacked autoencoder using all the data, then reuse the lower layers to create a NN for your actual task and train it using the labeled data



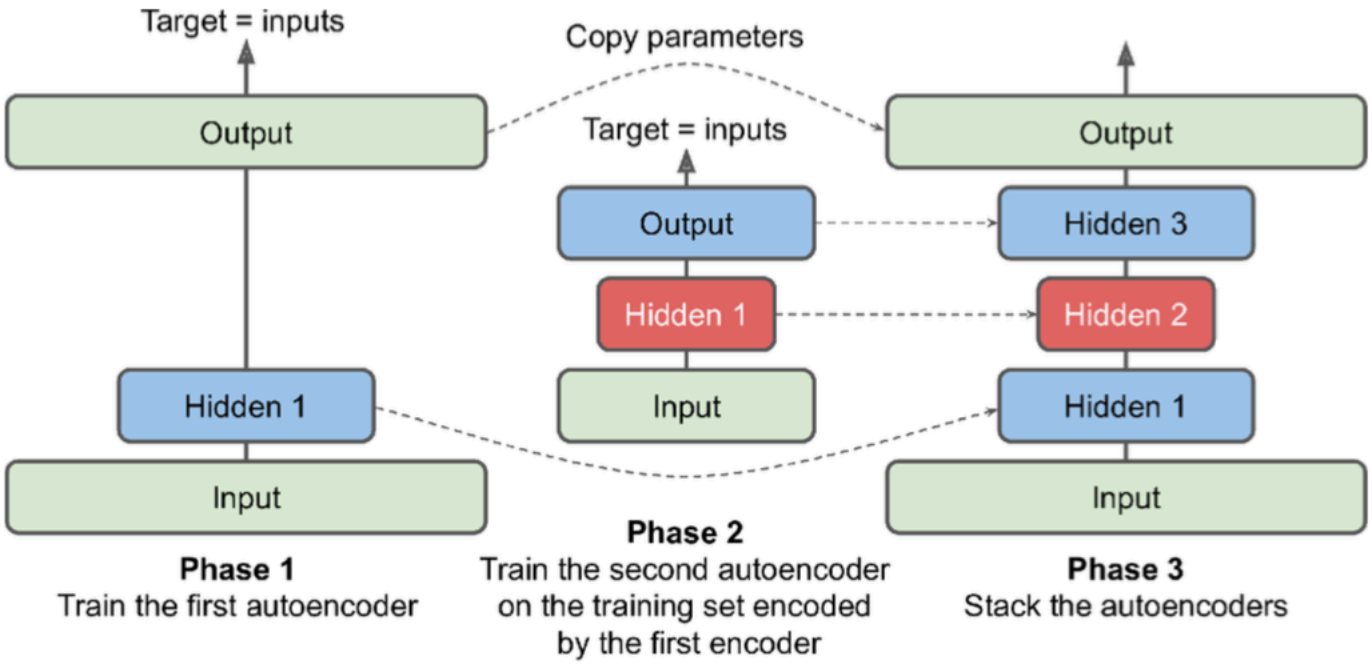
Techniques to Train Stacked Autoencoders - Tying Weights

- For symmetrical autoencoder, a common technique is to tie the weights of the decoder layers to the weights of the encoder layers. This halves the number of weights in the model, speeding up training and limiting the risk of overfitting.

- Specifically, if the autoencoder has a total of *N* layers (not counting the input layer), and $\boldsymbol{W}_L$ represents the connection weights of the Lth layer (e.g., layer 1 is the 1st hidden layer, layer *N*/2 is the coding layer, and layer *N* is the output layer), then the decoder layer weights can be defined simply as: $\boldsymbol{W}_{N-L+1} = \boldsymbol{W}_L^T$ (with $L = 1, 2, \ldots, N/2$).

Technique 2 - Training One Autoencoder at a Time

- Rather than training the whole stacked autoencoder in one go like we just did, it is possible to train one shallow autoencoder at a time, then stack all of them into a single stacked autoencoder. This “greedy layer-wise training” is not used much anymore.



Convolutional Autoencoder

- Use it if you want to build an autoencoder for images (e.g., for unsupervised pretraining or dimensionality reduction.
- The encoder is a regular CNN composed of convolutional layers and pooling layers. It typically reduces the spatial dimensionality of the inputs (i.e., height and width) while increasing the depth (i.e., the number of feature maps).
- The decoder must do the reverse (upscale the image and reduce its depth back to the original dimensions), and for this you can use transpose convolutional layers.

```
conv_encoder = keras.models.Sequential([
    keras.layers.Reshape([28, 28, 1], input_shape=[28, 28]),
    keras.layers.Conv2D(16, kernel_size=3, padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2),
    keras.layers.Conv2D(32, kernel_size=3, padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2),
    keras.layers.Conv2D(64, kernel_size=3, padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2)
])
```

etc.

Recurrent Autoencoder

- The encoder is typically a sequence-to-vector RNN which compresses the input sequence down to a single vector. The decoder is a vector-to-sequence RNN that does the reverse. E.g. this code can be used on Fashion MNIST images

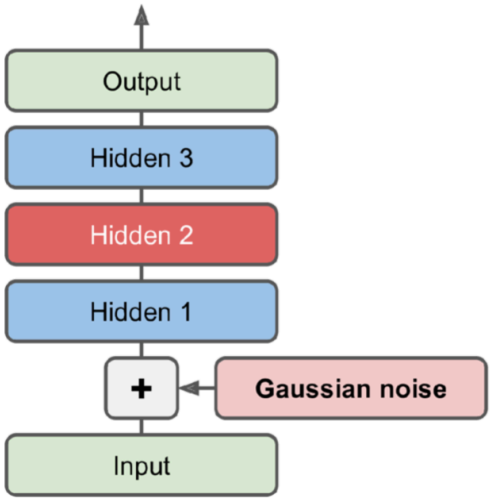
```
recurrent_encoder = keras.models.Sequential([
    keras.layers.LSTM(100, return_sequences=True, input_shape=[None, 28]),
    keras.layers.LSTM(30)
])
recurrent_decoder = keras.models.Sequential([
    keras.layers.RepeatVector(28, input_shape=[30]),
    keras.layers.LSTM(100, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(28, activation="sigmoid"))
])
recurrent_ae = keras.models.Sequential([recurrent_encoder, recurrent_decoder])
```

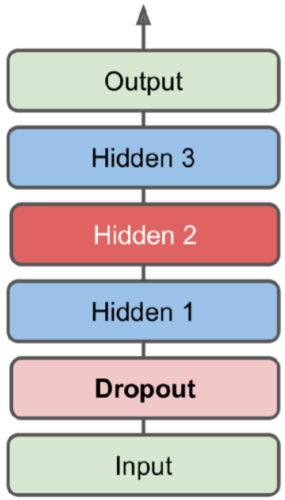
- At each time step, the RNN will process a single row of 28 pixels. Also RepeatVector layer as the first layer of the decoder, to ensure that its input vector gets fed to the decoder at each time step.

All the models considered so far were undercomplete. That is not always the case as can be seen in the following examples

Denoising Autoenconders

- Noise is added to the inputs. Train to recover the original, noise-free inputs. An old idea improved in 2010 with its stacked version






```
dropout_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(30, activation="selu")
])
dropout_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[30]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
dropout_ae = keras.models.Sequential([dropout_encoder, dropout_decoder])
```

Sparse Autoencoders

- Another kind of constraint that often leads to good feature extraction is sparsity: by adding an appropriate term to the cost function, the autoencoder is pushed to reduce the number of active neurons in the coding layer.

- One approach is to use l_1 or l_2 regularization in the coding level – see text

- Another one is to measure the actual sparsity of the coding layer at each training iteration, and penalize the model when the (average) measured sparsity differs from a target sparsity.

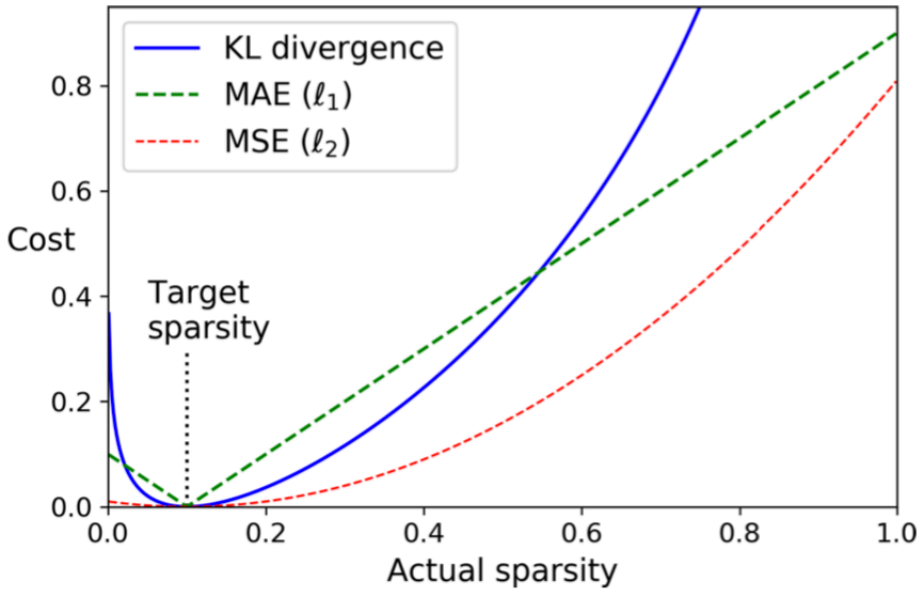
- Then we want to penalize the neurons that are too active, or not active enough, by adding a sparsity loss to the cost function. We can use squared error added to the cost function or the Kullback-Liebler (KL) divergence which leads to faster/better results

Equation 17-1. Kullback–Leibler divergence

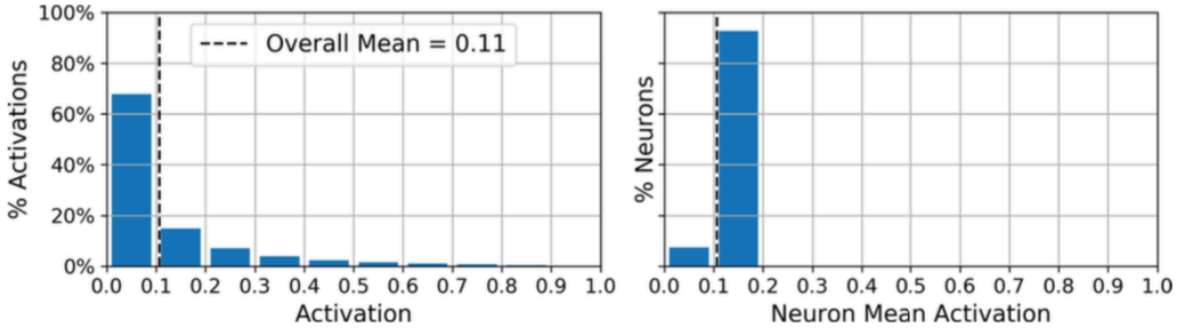
$$D_{\mathrm{KL}}(P \parallel Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

$$D_{\mathrm{KL}}(p \parallel q) = p \log \frac{p}{q} + (1-p) \log \frac{1-p}{1-q}$$

p – target, q- actual sparsity

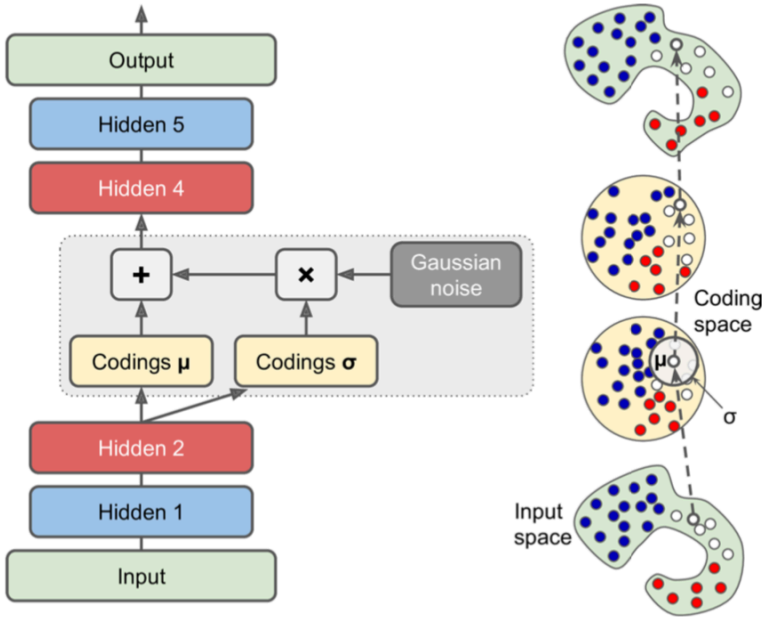


After training a sparse encoder on Fashion MNIST



Variational Autoencoders

- 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.



- You can recognize the basic structure - an encoder followed by a decoder but instead of directly producing a coding for a given input, the encoder produces a mean coding **μ** and a standard deviation **σ**.

- 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.

- Thus, the encoder produces **μ** and **σ**, then a coding is sampled randomly (notice that it is not exactly located at **μ**), and finally this coding is decoded; the final output resembles the training instance.

- Training pushes the codings to gradually migrate within the coding space (latent *space*) to end up looking like a cloud of Gaussian points.

- A 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, and decode it.