Lecture 07

Autoencoders
Variational Autoencoders
Manifold Hypothesis

cscie-89 Deep Learning, Fall 2024

Rahul Joglekar & Zoran B. Djordjević

Objectives and Reference

- We will learn techniques known as:
  - Autoencoders
  - Variational Autoencoders
  - Manifold Hypothesis

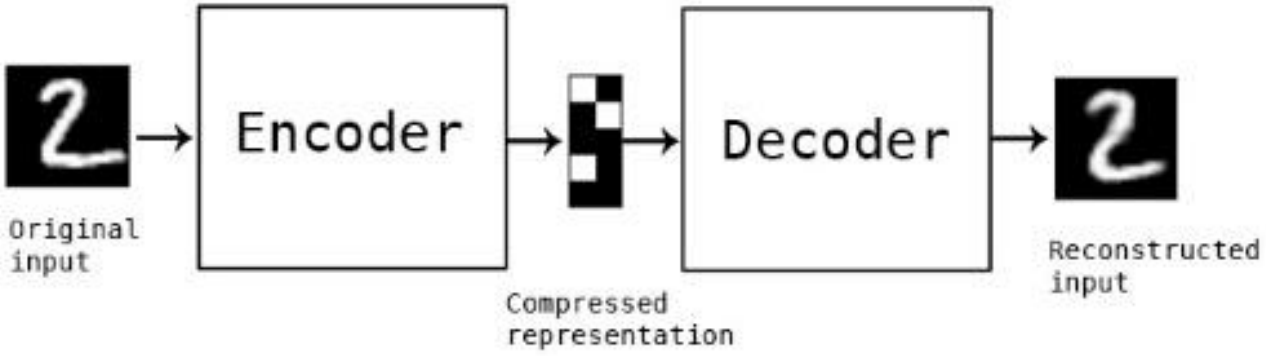
This lecture follows material presented in:

- Chapter 17, Hands on Machine Learning with Scikit-learn, Keras & TensorFlow, 2nd Edition, by Aurélien Géron, O'Reilly 2019
- "Building Autoencoders in Keras" by Francois Chollet (https://blog.keras.io/building-autoencoders-in-keras.html)

Autoencoders

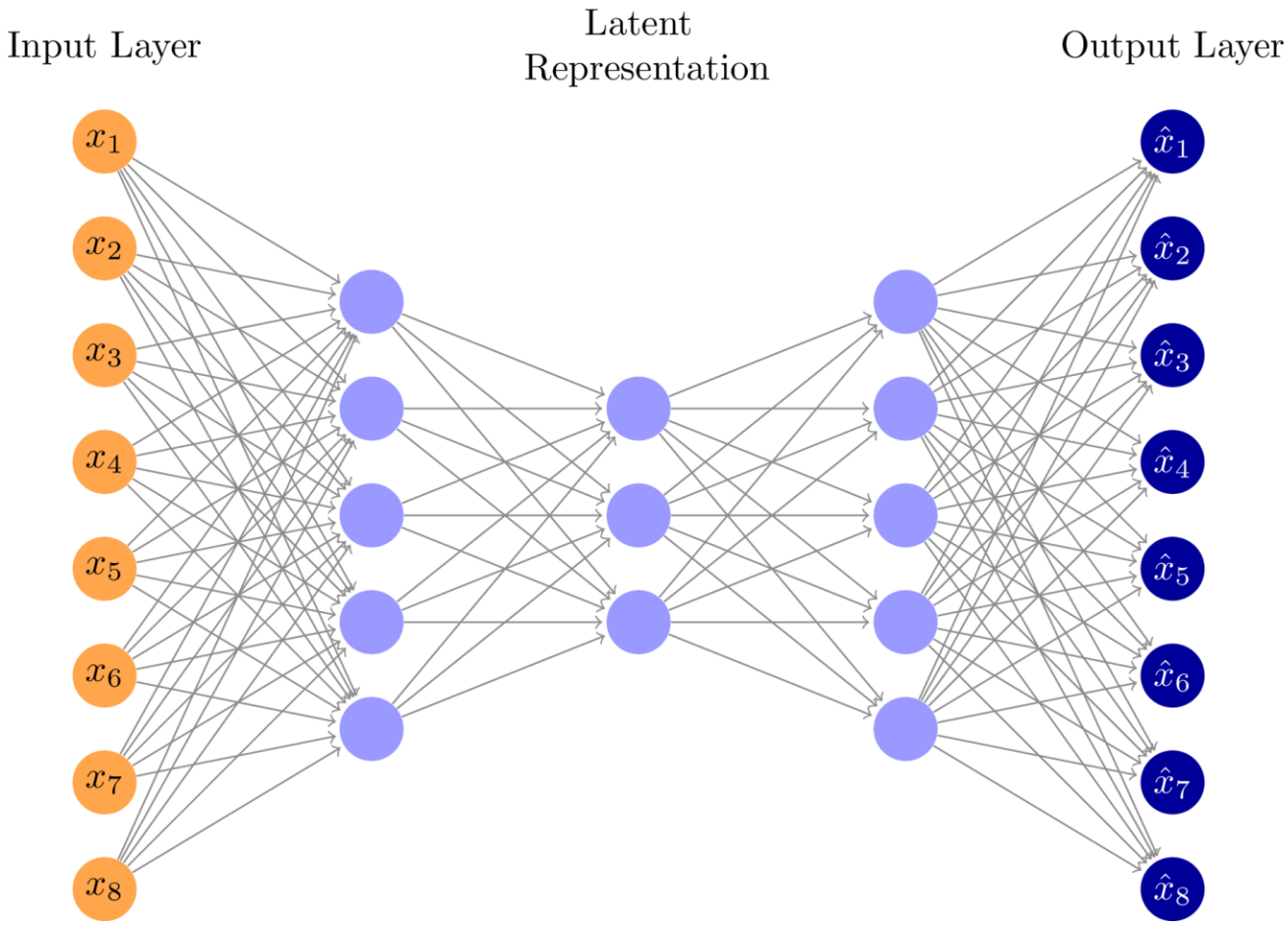
Autoencoders

- Autoencoders are artificial neural networks capable of learning efficient representations of the input data, called `codings`, without any supervision (the training set is unlabeled).
- These `codings` typically have a much lower dimensionality than the input data, making autoencoders useful for dimensionality reduction.
- Autoencoders act as powerful feature detectors, and they can be used for unsupervised pre-training of deep neural networks.
- Autoencoders generating new data that looks very similar to the training data; this is called a *generative model*. For example, you could train an autoencoder on pictures of faces, and it would then be able to generate new faces.



An Illustrative Autoencoder

- An autoencoder is always composed of two parts: an *encoder* (or *recognition network*) that converts the inputs into an internal representation, followed by a *decoder* (or *generative network*) that converts the internal representation to the outputs:



What do Autoencoders Do

- Autoencoders work by learning to copy their inputs to their outputs.
- This is not a trivial task if we constrain the network. For example, we could limit the size of the internal representation, or we can add noise to the inputs and train the network to recover the original inputs.
- These constraints prevent the autoencoder from trivially copying the inputs directly to the outputs, which forces it to learn efficient ways of representing the data.
- In short, the `codings` are byproducts of the autoencoder's attempt to learn the identity function under some constraints.

- Coding is performed in many of our regular tasks. Writing is a coding process. We replace complex sound waveforms with character symbols. Morse code replaces characters with simple sequences of dots and dashes, etc.
- An autoencoder looks at the inputs, converts them to an efficient internal representation, and then spits out something that (hopefully) looks very close to the inputs.

What is autoencoding

- "Autoencoding" is a data compression algorithm where the compression and decompression functions are 1) data-specific, 2) lossy, and 3) *learned automatically from examples* rather than engineered by a human. Additionally, in almost all contexts where the term "autoencoder" is used, the compression and decompression functions are implemented with neural networks.

  1) Autoencoders are data-specific, which means that they are only able to compress data like what they are trained on. This is different from, say, the MPEG-2 Audio Layer III (MP3) compression algorithm, which only holds assumptions about "sound" in general, but not about specific types of sounds.

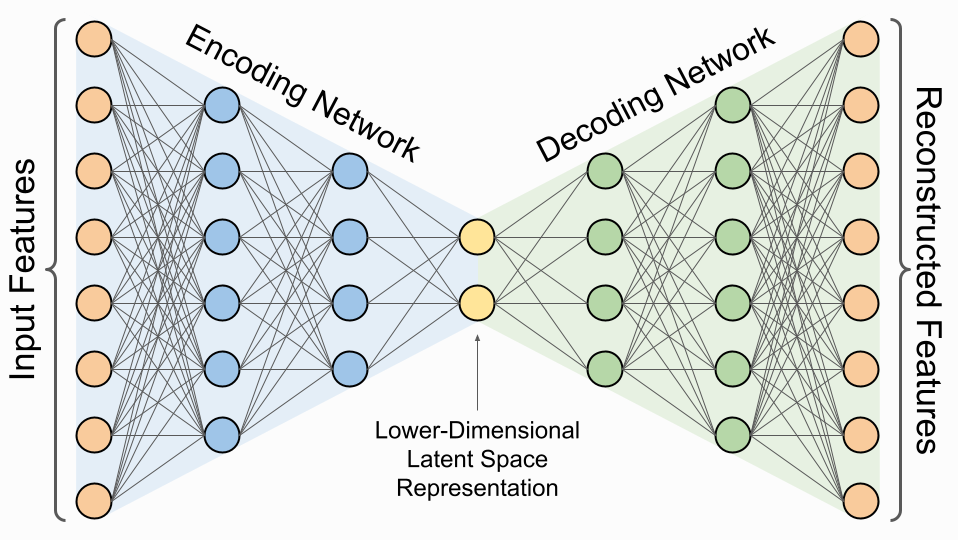
  – An autoencoder trained on pictures of faces does a rather poor job of compressing pictures of trees, because the features it learns are face-specific.

  2) **Autoencoders are lossy,** which means that the decompressed outputs will be degraded compared to the original inputs (similar to MP3 or JPEG compression). This differs from lossless arithmetic compression.

  3) **Autoencoders are trained automatically from data examples.** That is very useful in the sense that it is easy to train specialized instances of the algorithm that will perform well on a specific type of input. It does not require any new autoencoder architecture just appropriate training data.

Reconstructions & Undercomplete Autoencoders

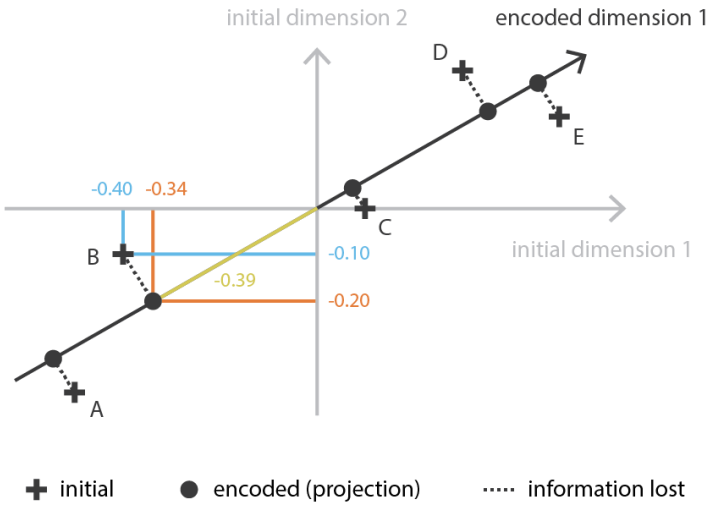
- The outputs are often called the *reconstructions* since the autoencoder tries to reconstruct the inputs, and the cost function contains a *reconstruction loss* that penalizes the model when the reconstructions are different from the inputs.
- Because the internal representation has a lower dimensionality than the input data (it is for example 2D instead of 3D), the autoencoder is said to be *undercomplete*.
- An undercomplete autoencoder cannot trivially copy its inputs to the codings, yet it must find a way to output a copy of its inputs. ***It is forced to learn the most important features in the input data (and drop the unimportant ones).***
- The above statement about finding the most efficient representation is apparently true for most NN.



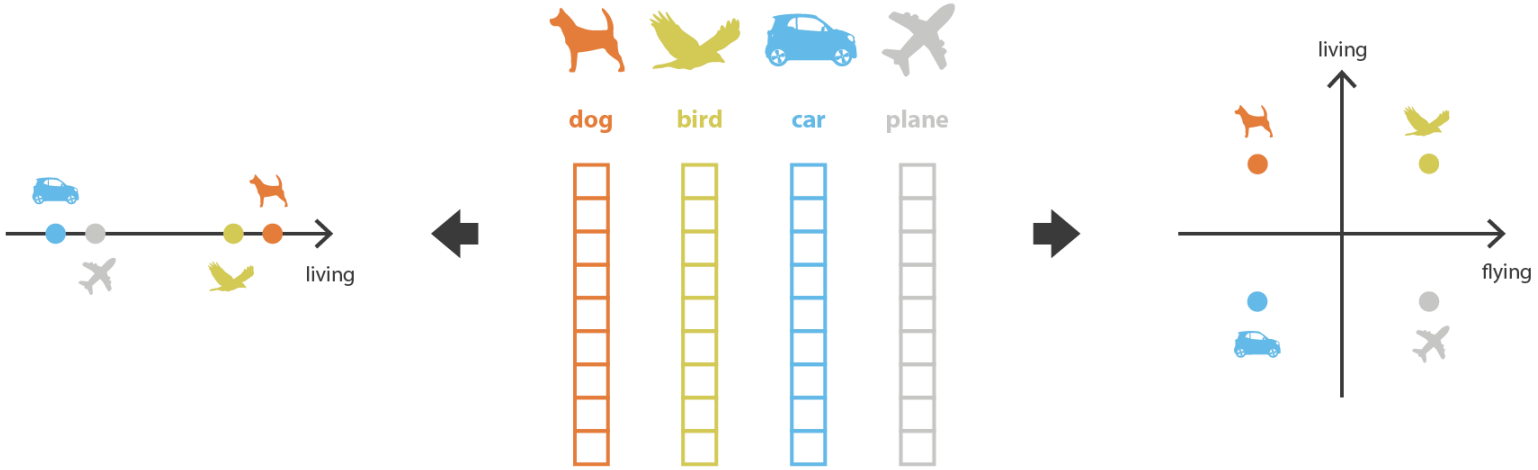
Ingredients

- To build an autoencoder, you need three functions:
  - **an encoding function,**
  - **a decoding function, and**
  - **a distance function** measuring the information loss between the compressed representation of your data and the decompressed representation (a "loss" function).
- The encoder and decoder are parametric functions (typically neural networks) and are given a distance function which is differentiable with respect to those parameters.
- We minimize the reconstruction loss, using Stochastic Gradient Descent and identify encoding/decoding functions that give optimized reconstruction loss.

Encoding/Latent Space/Embedding Space



Point	Initial	Encoded	Decoded
A	(-0.50, -0.40)	-0.63	(-0.54, -0.33)
B	(-0.40, -0.10)	-0.39	(-0.34, -0.20)
C	(0.10, 0.00)	0.09	(0.07 0.04)
D	(0.30, 0.30)	0.41	(0.35, 0.21)
E	(0.50, 0.20)	0.53	(0.46, 0.27)



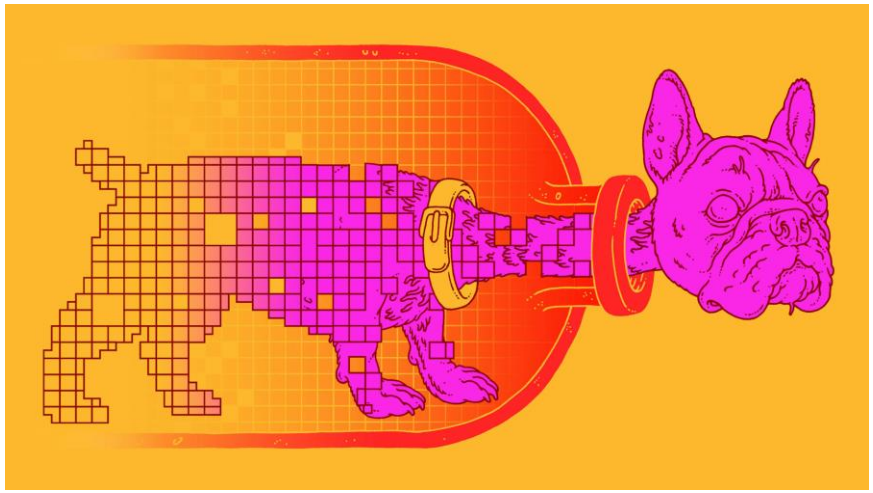
near optimal encoding
in one dimension
(too much information lost)

initial data with many features

near optimal encoding
in two dimensions
(less information lost)

Information Bottleneck Principle

- It is argued that deep neural networks learn using a procedure called the "information bottleneck." The concept was first described in 1999.
- The idea is that a network rids noisy input data of extraneous details as if by squeezing the information through a bottleneck, retaining only the features most relevant to general concepts.
- You could find the basic explanation and references here: https://www.quantamagazine.org/new-theory-cracks-open-the-black-box-of-deep-learning-20170921/



- In a nutshell, this theory tells you that neural networks, just like us, learn to represent complex objects with a few symbols. An image of a dog could have millions of pixels and yet we describe it with three letters: "d, o, and g". The dog version of the theory is done. Cat version of the theory is in development.

Use for Compression

- One is tempted to use autoencoders as compression tool.
- In the image compression, for instance, it is pretty difficult to train an autoencoder that does a better job than a basic algorithm like JPEG.
- Typically, the only way an efficient compression could be achieved with autoencoders is by restricting autoencoders to a very specific types of images.
- The fact that autoencoders are data-specific makes them generally impractical for real-world data compression problems.
- We can only use autoencoders on data that is similar to what they were trained on.
- Making autoencoders more general would requires *large volumes* of training data.
- Future scientific advances might change this.

Practical Use Cases

- Today, two most interesting practical applications of autoencoders are
  - **data denoising**
  - **Anomaly detection**
  - **dimensionality reduction for data visualization**.
- With appropriate dimensionality and sparsity constraints, autoencoders could learn data projections that are more interesting than PCA or other basic techniques.
- For 2D visualization specifically, t-SNE is probably the best algorithm around, but it typically requires relatively low-dimensional data.
- A good strategy for visualizing similarity relationships in high-dimensional data is to start by using an autoencoder to compress your data into a low-dimensional space (e.g., 32 dimensional), then use t-SNE for mapping the compressed data to a 2D plane.

Anomaly Detection

Practical Use Cases

Autoencoders in Practice

Autoencoders Claim to Fame

- One reason why autoencoders have attracted so much research and attention is because they have long been thought to be a potential avenue for solving the problem of unsupervised learning, i.e., learning of useful representations without the need for labels.
- Then again, autoencoders are not a true unsupervised learning technique. They are rather a *self-supervised* technique. Autoencoders are a specific type of *supervised learning* where the targets are generated from the input data (or are identical to the input data).
- To get self-supervised models to learn interesting features, you need to come up with an interesting synthetic target and loss function, and that's where problems arise: merely learning to reconstruct your input in minute detail might not be the right choice.
- There is a significant evidence that focusing on the reconstruction of a picture at the pixel level, for instance, is not conductive to learning interesting, abstract features of the kind that labeled (supervised) learning identifies.
- We want to learn abstract concepts "invented" by humans such as "dog", "car“, etc.

Performing PCA with a Linear Autoencoder

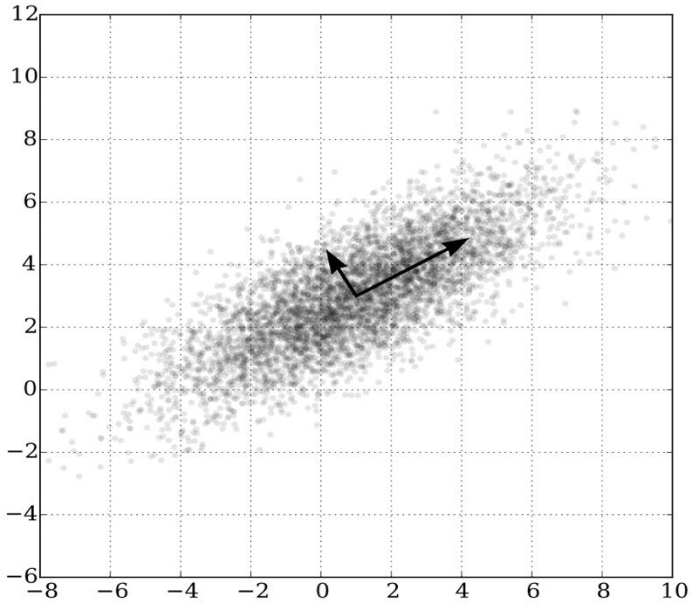
Common Issues

- Quite often we are dealing with large numbers (N) of samples represented as vectors in a highly n-dimensional (vector) space. Typically, the dimensionality n is a bigger cause of trouble than the number of samples N. Typically, n is large, but $n << N$.
- The issues are twofold:
  - We can not readily visualize the spread of samples in the n-dimensional space.
  - To speed up calculations we prefer to substitute true n-dimensional vectors with approximate vectors of a considerably smaller dimension $n_0 << n$.

- A technique referred to as the Principal Component Analysis (PCA) could be used in both cases

Principal component analysis (PCA)

- **Principal component analysis** (**PCA**) is a linear dimensionality reduction technique with applications in exploratory data analysis, visualization and data preprocessing.
- The data is linearly transformed onto a new coordinate system such that the directions (principal components) capturing the largest variation in the data can be easily identified.



- PCA of a multivariate Gaussian distribution centered at (1,3) with a standard deviation of 3 in roughly the (0.866, 0.5) direction and of 1 in the orthogonal direction.
- The vectors shown are the eigenvectors of the covariance matrix scaled by the square root of the corresponding eigenvalue, and shifted so their tails are at the mean.

Variance

- The variance of a random variable X is the expected value of the squared deviation form the mean of X, $\mu = E[X]$:

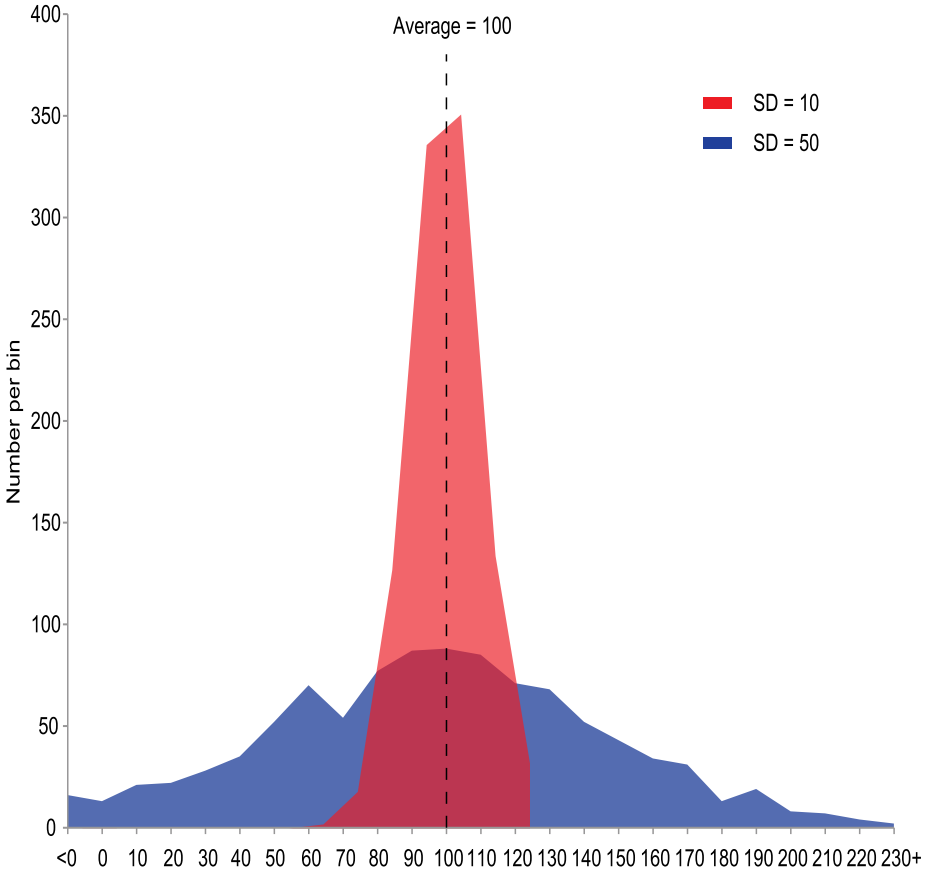
$$Var(X) = E[(X - \mu)^2]$$

- The variance is typically designated as

$$Var(X) = \sigma_X^2$$

(pronounced "sigma squared“ or “standard deviation squared”.).

- Variance measures the width of the probability distribution. On the right are samples from two distributions with the same mean but different variances.
- The red population has mean 100 and variance 100 and standard deviation, SD $= \sigma$=10. The blue population has mean 100 and variance 2500 (SD $= \sigma$ =50).

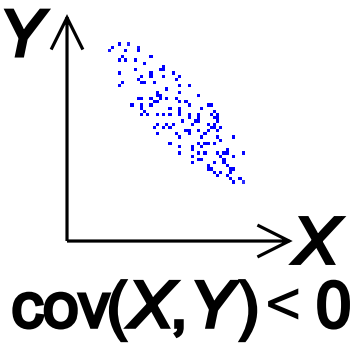


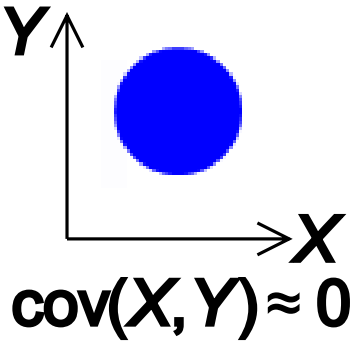
Covariance

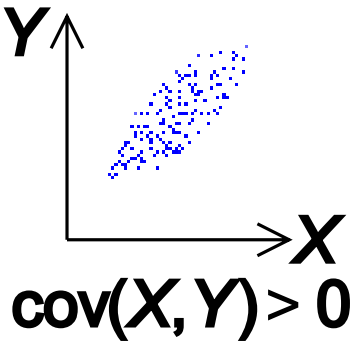
- The covariance is a measure of the joint variability of two random variables.
- For two jointly distributed real-valued random variables X and Y with finite second moments, the covariance is defined as the expected value (or mean) of the product of their deviations from their individual expected values:

$$Cov(X,Y) = E[(X - E[X])(Y - E[Y])]$$

- Here $E[X]$ is the expected value of X, the mean of X. The covariance is sometimes denoted as σ_{XY} or $\sigma(X,Y)$.
- The sign of the covariance shows the tendency in the relationship between the variables. If greater values of one variable mainly correspond with greater values of the other variable the covariance is positive. When greater values of one variable mainly correspond to lesser values of the other (that is, the variables tend to show opposite behavior), the covariance is negative.







Covariance Matrix

- A generalization of those two concepts, variance and covariance, in the case of stochastic variables living in n-dimensional space, is the covariance matrix.
- To calculate covariance matrix, we arrange N vectors of length n in a matrix X with n rows and N columns.
- $$X = \begin{bmatrix} x_1^1 & \cdots & x_1^N \\ \vdots & \ddots & \vdots \\ x_n^1 & \cdots & x_n^N \end{bmatrix}$$
- We are typically interested in the variations of coordinates of sample vectors, we subtract, from matrix X, its mean value, calculated along every one of n coordinates. Resulting "centered" matrix $X - E[X]$ is still a matrix with n rows and N columns.
- Next, we multiply from the right this "centered" matrix with its transpose. The transpose matrix has N rows and n columns. The resulting matrix of dimensions $n \times n$ is a symmetric, real, positive defined, matrix we refer to as the covariance matrix:

$$C(X) = E[(X - E[X]) \times (X - E[X])^T]$$

- The covariance matrix $C(X)$ measures variance and covariance in different dimensions. The E operators introduce factor $^1/_{N-1}$

Eigen Vectors and Eigne Values of $C(X)$

- Every real symmetric matrix of dimension $n \times n$ has n real eigenvalues $\{\lambda_i, i = 1, .., n\}$ and n corresponding eigenvectors : $\{e_i, i = 1, .., n\}$
- Eigenvectors $\{e_i\}$ are mutually orthogonal vectors, which, when multiplied by a matrix, in our case covariance matrix $C(X)$, do not change their direction, but rather only change their length, like in the following equation:

$$C(X)e_i = \lambda_i e_i$$

- Typically, the vector with the largest positive eigenvalue λ_1, is denoted as e_1. The vectors with progressively smaller eigenvalues are e_2 with λ_2 , e_3 with λ_3 , and so on. All those vectors : $\{e_i, i = 1, .., n\}$ are vectors in the n-dimensional space in which our samples (events) are defined.
- Since the covariance matrix $C(X)$ measures variances and covariances in certain dimensions and between different dimensions, vectors e_1, e_2, e_3 and so forth, specify the directions of the largest variation, next largest and so on.
- This is where the image on slide 16 comes from.
- Vectors $\{e_i, i = 1, .., n\}$,with the largest eigenvalues, 2, 3 or more of them are selected as the basis of a new coordinate system and all N points in the sample set are projected from the original n-dimensional space into this 2,3 or “small” dimensional space. Such projections can be viewed in our 2 or 3 dim renderings.

Performing PCA with a Linear Autoencoder

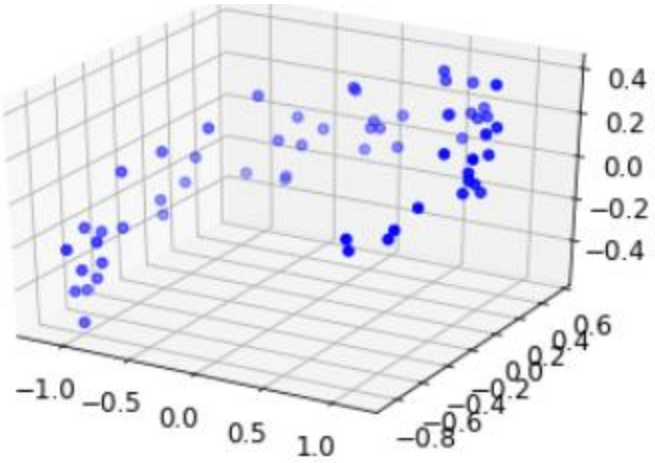
- If an autoencoder uses only linear activations and the cost function is the Mean Squared Error (MSE), then it can be shown that it ends up performing Principal Component Analysis (PCA).
- To demonstrate this capability, we create a cloud of 60 simulated random data points in 3-d. We could visualize such data.

```
import numpy as np
np.random.seed(4)

def generate_3d_data(m, w1=0.1, w2=0.3, noise=0.1):
    angles = np.random.rand(m) * 3 * np.pi / 2 - 0.5
    data = np.empty((m, 3))
    data[:, 0] = np.cos(angles) + np.sin(angles)/2 + noise *
np.random.randn(m) / 2
    data[:, 1] = np.sin(angles) * 0.7 + noise * np.random.randn(m) / 2
    data[:, 2] = data[:, 0] * w1 + data[:, 1] * w2 + noise *
np.random.randn(m)
    return data

X_train = generate_3d_data(60)
X_train = X_train - X_train.mean(axis=0, keepdims=0)
```

Visualize the data and Normalize them

- We visualize our 3-d numpy array using `scatter()` function from `Axes3D` package

```
# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X_train[:,0],X_train[:,1],X_train[:,2], c='b',marker='o')
plt.show()
```



Autoencoder in Keras

- Our encoder has 3 inputs (`n_input = 3`), and two dense layers.
- "`codings`" are desired lower dimensional codes, representing inputs in a lower dimensional (2-d) space

```
import tensorflow as tf
from tensorflow import keras
import numpy as np


np.random.seed(42)
tf.random.set_seed(42)

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=1.5))


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

Structure of the Model

- We organized the autoencoder into two subcomponents: the encoder and the decoder.
- Both are regular Sequential models with a single Dense layer each. The autoencoder is a Sequential model containing the encoder followed by the decoder. Remember that a model, of class `tf.keras.models.Model`, can be used as a layer in another model.
- The autoencoder's number of outputs is equal to the number of inputs (i.e., 3).
- To perform simple PCA, we do not use any activation function (i.e., all neurons are linear), and the cost function is the MSE. We will see more complex autoencoders shortly.

`codings`, outputs of the first hidden layer

- Note that dense layer is not followed by a non-linear unit, sigmoids or ReLU.
- We are training the network with 60 vectors of dimension 3. As the outputs of the first hidden layer, encoder, we are producing 60 vectors of dimension 2.

```
codings = encoder.predict(X_train)
```

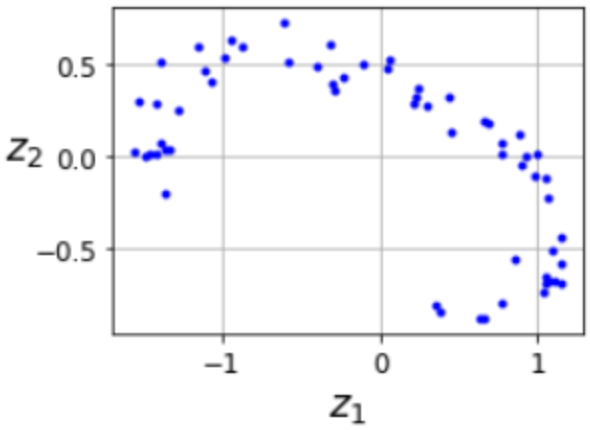
- Input training tensor `X_train`, results in tensor `codings`, the output of the first hidden layer (with 2 neurons).

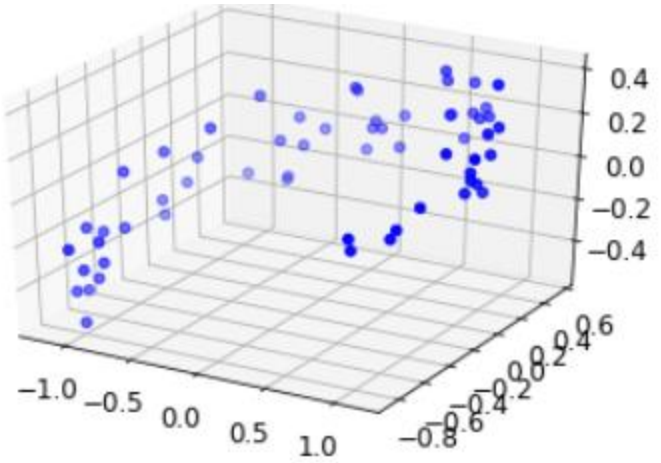
Visualize `codings` Values

- In our case, with 60 3-d vectors in `X_train`, `codings` is a (60,2) `numpy array` which we could present as a 2-dim plot.

```
fig = plt.figure(figsize=(4,3))
plt.plot(codings[:,0], codings
```

```
[:, 1], "b.")
plt.xlabel("$z_1$", fontsize=18)
plt.ylabel("$z_2$", fontsize=18, rotation=0)
save_fig("linear_autoencoder_pca_plot")
plt.show()
```





- `Data projected onto PCA plane` `The original 3D data`
- While this might not be entirely obvious that this is the PCA projection, one can prove that this autoencoder actually found the best 2D plane to project the data onto. That plane exposes the largest variance present in the data.

Scaling the Input Data

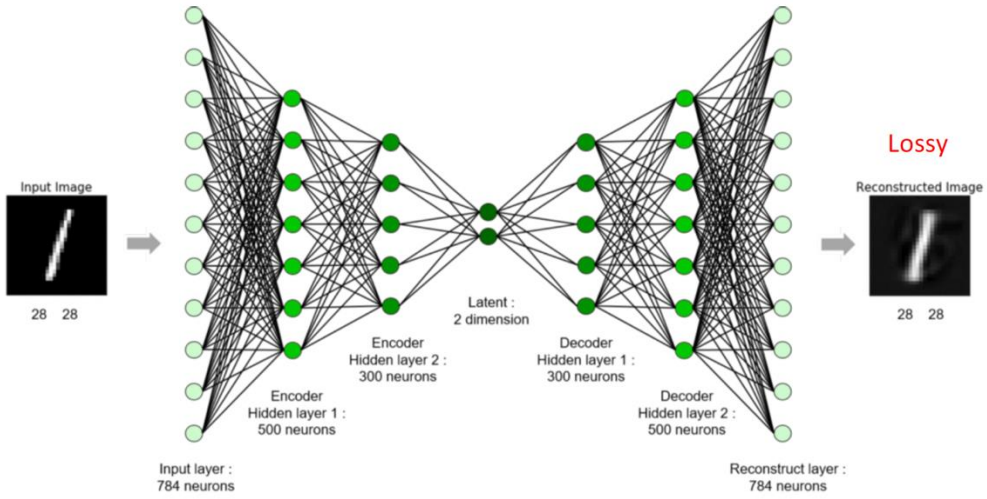
- In some earlier models we used SciKit Learn's `StandardScaler()` scaler object.
- Neural networks work best if all the data is between zero and one.
- Most practitioners use SciKit Learn's `MinMaxScaler` object for that purpose.
- Method `fit()` scales `train_dataset` first and determines scaling parameters.
- Those parameters are used by method `transform()` of object `scaler` on any other data (`test` or `validation`) . The code reads:

```
# Scale all the training data to the range [0,1].
scaler = MinMaxScaler(copy=False)
scaler.fit(train_dataset)
scaler.transform(train_dataset)
scaler.transform(valid_dataset)
scaler.transform(test_dataset)
scaler.transform(final_row)
```

- The `copy=False` parameter says: "do the conversion in place rather than producing a new copy of the data".
- SciKit Learn has a lot of useful utility functions that can help while working with TensorFlow, Keras or PyTorch.
- The syntax of SciKit Learn API is rather convoluted but is worth learning even though you are not using a SciKit Learn Machine Learning functions.
- Writing and optimizing all the functions contained in SkiKit Learn API is quite a task.

Vanilla AutoEncoder summarized

Compress MNIST (28x28x1) to the latent code with only 2 variables



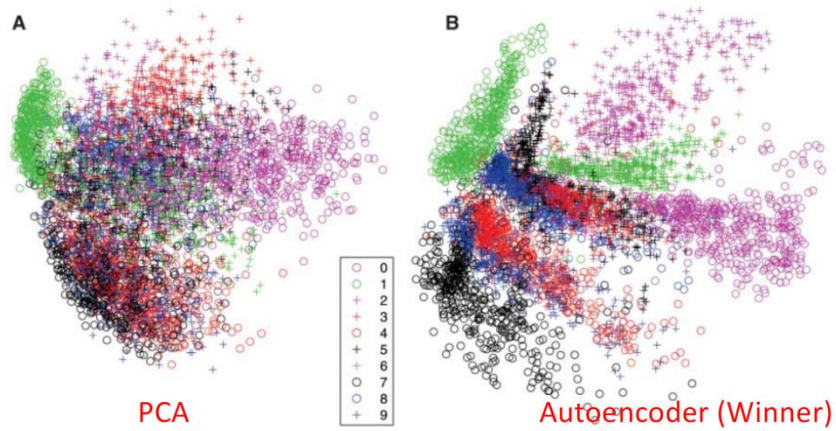


Fig. 3. **(A)** The two-dimensional codes for 500 digits of each class produced by taking the first two principal components of all 60,000 training images. **(B)** The two-dimensional codes found by a 784-1000-500-250-2 autoencoder. For an alternative visualization, see (*8*).

2006 Science paper by Hinton and Salakhutdinov

Autoencoder is also a self-supervised (self-taught) learning method where the training labels are determined by the input data.

- Latent Space holds powerful information
- Non-Linearity is captured
- O/P is always Lossy
- Embeddings can be saved and used for other purposes

AutoEncoder Types

Vanilla Autoencoder

Denoising Autoencoder

Sparse Autoencoder

Contractive Autoencoder

Stacked Autoencoder

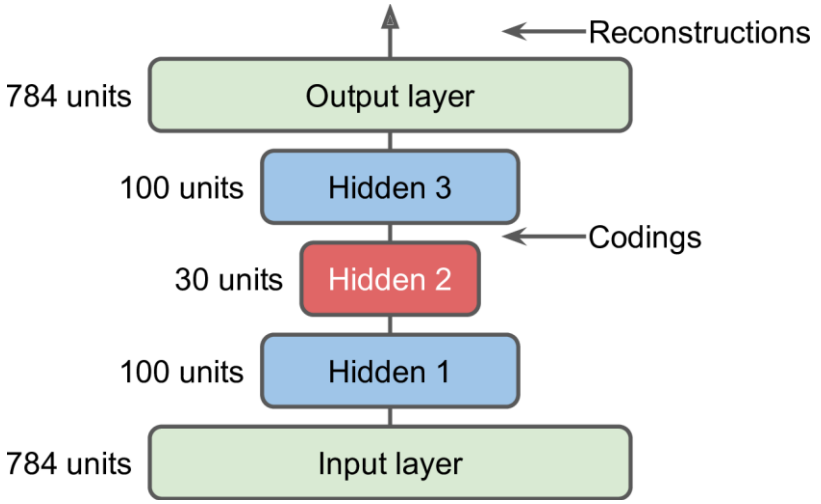
Variational Autoencoder (VAE)

Deep or Stacked Autoencoders

Deep Autoencoders

- Like other neural networks, autoencoders can have multiple hidden layers. In that case they are called *stacked autoencoders* (or *deep autoencoders*).
- Adding more layers helps the autoencoder learn more complex `codings`.
- One must be careful not to make the autoencoder too powerful. With too many layers and too many trainable parameters, autoencoders would easily learn to exactly map every input to an identical output and would not have learned any useful (reduced) data representation.
- The architecture of a deep autoencoder is typically symmetrical with regards to the central hidden layer (the `codings` layer). Stacked autoencoder looks like a sandwich.

- For example, an autoencoder for MNIST images 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.



Deep Autoencoder for MNIST Images

- We will load MNIST examples from `tensorflow.examples.tutorials.mnist`

```
(X_train_full, y_train_full), (X_test, y_test) =
keras.datasets.mnist.load_data()
X_train_full = X_train_full.astype(np.float32) / 255
X_test = X_test.astype(np.float32) / 255
X_train, X_valid = X_train_full[:-5000], X_train_full[-5000:]
y_train, y_valid = y_train_full[:-5000], y_train_full[-5000:]
```

We will implement a stacked autoencoder using 4 dense layers. We will use so called *He* initialization, the SELU activation function, and L2 regularization.

- There are no labels (no y-s).
- The dimension of inputs, $\mathrm{n}_{\mathrm{inputs}} = 28 * 28 = 784$ (number of pixels in a MNIST image)

```
n_inputs = 28 * 28
n_hidden1 = 100
n_hidden2 = 30  # codings (the dimension of the coding vector)
n_hidden3 = n_hidden1
n_outputs = n_inputs
```

Deep Autoencoder

- The autoencoder has two submodels: the encoder and the decoder.
- The encoder takes 28 × 28–pixel grayscale images, flattens them so that each image is represented as a vector of size 784, then processes these vectors through two layers of diminishing sizes (100 units then 30 units), both using the SELU activation function. For each input image, the encoder outputs a vector of size 30.
- The decoder takes `codings` of size 30 (output of the encoder) and processes them through two `Dense` layers of increasing sizes (100 units then 784 units), and reshapes the final vectors into 28 × 28 arrays so the decoder's outputs have the same shape as the encoder's inputs.
- When compiling the stacked autoencoder, we use the binary cross-entropy loss instead of the mean squared error. We are treating the reconstruction task as a multilabel binary classification problem: each pixel intensity represents the probability that the pixel should be black. Framing it this way (rather than as a regression problem) tends to make the model converge faster.
- Finally, we train the model using `X_train` as both the inputs and the targets (and similarly, we use `X_valid` as both the validation inputs and targets).

Deep MNIST Autoencoder

```
def rounded_accuracy(y_true, y_pred):
    return keras.metrics.binary_accuracy(tf.round(y_true), tf.round(y_pred))

tf.random.set_seed(42)
np.random.seed(42)

stacked_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(30, activation="selu"), ← coding
])
stacked_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])
])
stacked_ae = keras.models.Sequential([stacked_encoder, stacked_decoder])
stacked_ae.compile(loss="binary_crossentropy",
                   optimizer=keras.optimizers.SGD(lr=1.5),
metrics=[rounded_accuracy])
history = stacked_ae.fit(X_train, X_train, epochs=20,
                         validation_data=[X_valid, X_valid])
```

Training Deep MNIST Autoencoder

- Train on 55000 samples, validate on 5000 samples

```
Epoch 1/20
55000/55000 [==============================] - 5s 91us/sample - loss:
0.1844 - rounded_accuracy: 0.9184 - val_loss: 0.1423 -
val_rounded_accuracy: 0.9417
Epoch 2/20
55000/55000 [==============================] - 5s 84us/sample - loss:
0.1322 - rounded_accuracy: 0.9477 - val_loss: 0.1264 -
val_rounded_accuracy: 0.9514
. . . . .
Epoch 19/20
55000/55000 [==============================] - 5s 85us/sample - loss:
0.0953 - rounded_accuracy: 0.9688 - val_loss: 0.0960 -
val_rounded_accuracy: 0.9688
Epoch 20/20
55000/55000 [==============================] - 5s 87us/sample - loss:
0.0946 - rounded_accuracy: 0.9693 - val_loss: 0.0944 -
val_rounded_accuracy: 0.9698
```

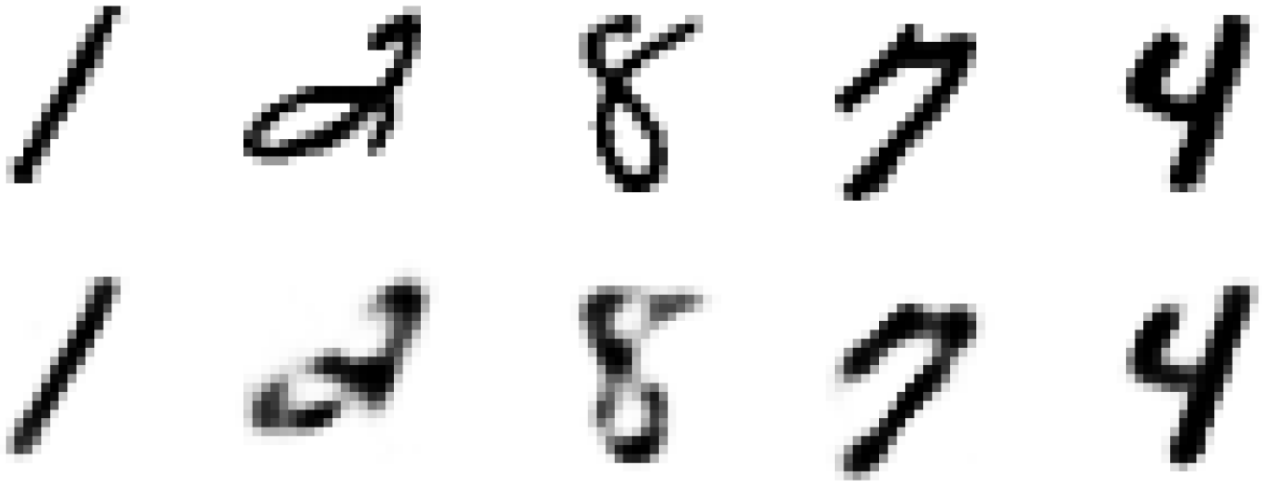
Visualizing the Reconstructions

- One way to ensure that an autoencoder is properly trained is to compare the inputs and the outputs: the differences should not be too significant. Let's plot a few images from the validation set, as well as their reconstructions

```
def plot_image(image):
    plt.imshow(image, cmap="binary")
    plt.axis("off")

def show_reconstructions(model, n_images=5):
    reconstructions = model.predict(X_valid[:n_images])
    fig = plt.figure(figsize=(n_images * 1.5, 3))
    for image_index in range(n_images):
        plt.subplot(2, n_images, 1 + image_index)
        plot_image(X_valid[image_index])
        plt.subplot(2, n_images, 1 + n_images + image_index)
        plot_image(reconstructions[image_index])

show_reconstructions(stacked_ae)
```

Passage of images through Autoencoder

- We invoke `show_reconstructions()` and let it display two samples of input images and images generated by the autoencoder. We are using previously stored model parameters.



- We see that the autoencoder was not able to perfectly reconstruct input images.
- In a way that is desired.

Visualizing the MNIST Dataset

- Now that we have trained a stacked autoencoder, we can use it to reduce the dataset's dimensionality.
- For visualization, this does not give great results compared to other dimensionality reduction algorithms, but one big advantage of autoencoders is that they can handle large datasets, with many instances and many features.
- So, one strategy is to use an autoencoder to reduce the dimensionality down to a reasonable level, then use another dimensionality reduction algorithm for visualization.
- We will use this strategy to visualize MNIST dataset. First, we use the encoder from our stacked autoencoder to reduce the dimensionality down to 30, then we use Scikit-Learn's implementation of the t-SNE algorithm to reduce the dimensionality down to 2 for visualization. The result is presented on the following slides.

t-SNE Dimension Reduced Display

t-distributed Stochastic Neighbor Embedding (TSNE)

- Please visit:

https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html
https://scikit-learn.org/stable/modules/manifold.html#t-sne

Visualizing Data using t-SNE, Journal of Machine Learning Research 9 (2008) 2579-2605, by Laurens van der Maaten and Geoffrey Hinton, available at:
https://jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf:

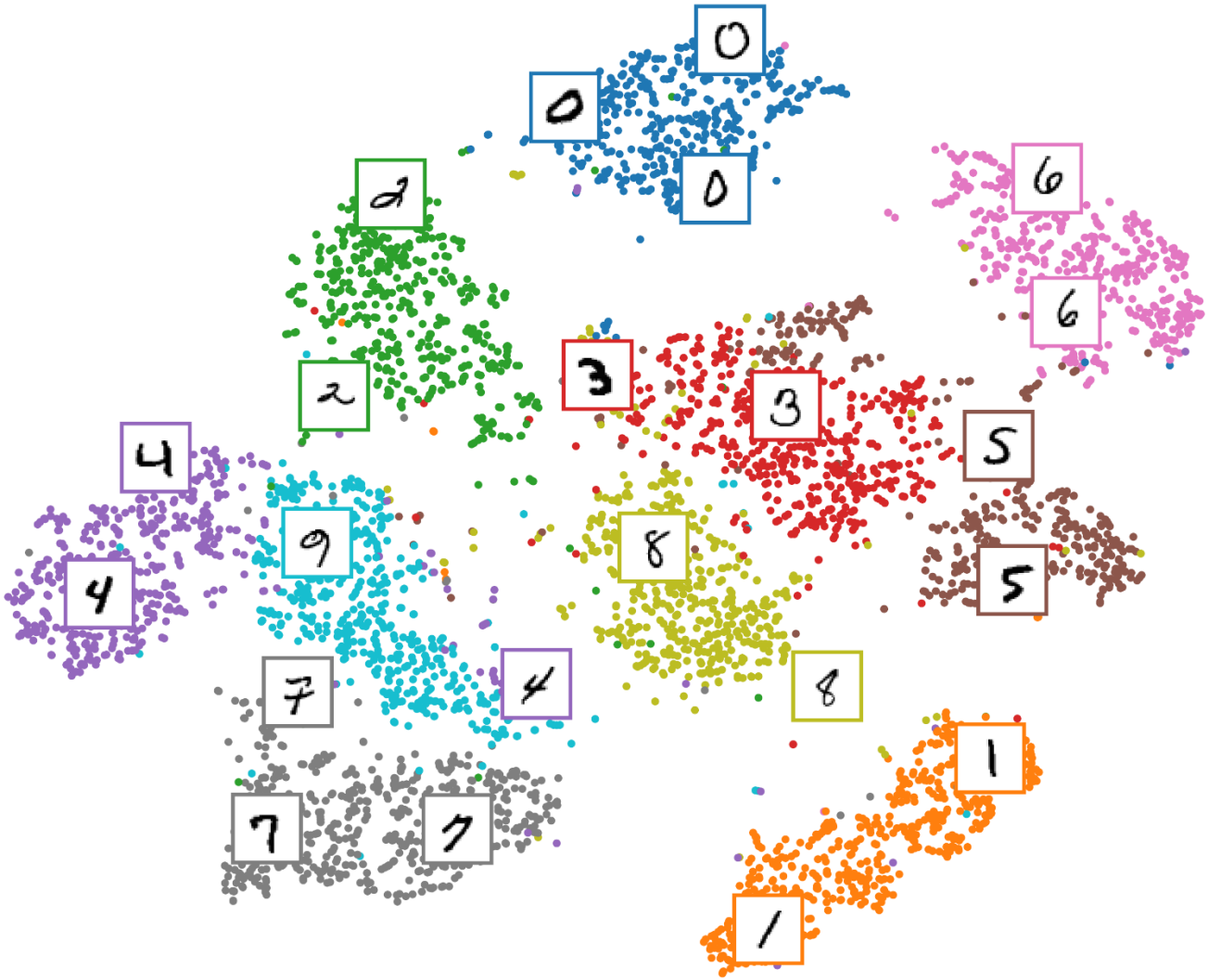
Slightly reworded Abstract:

- "t-SNE" is a technique that visualizes high-dimensional data by giving each datapoint a location in a two or three-dimensional map.
- The technique is a variation of Stochastic Neighbor Embedding (Hinton and Roweis, 2002) but is much easier to optimize, and produces significantly better visualizations by reducing the tendency to crowd points together in the center of the map.
- t-SNE is better than many other techniques at creating a single map that reveals structure at many different scales. This is particularly important for high-dimensional data that lie on several different, but related, low-dimensional manifolds, such as images of objects from multiple classes seen from multiple viewpoints.
- For visualizing the structure of very large data sets, t-SNE can use random walks on neighborhood graphs to allow the implicit structure of all the data to influence the way in which a subset of the data is displayed.
- The visualizations produced by t-SNE are significantly better than those produced by the other techniques on almost all data sets.

Code for t-SNE display

```
# adapted from
# https://scikit-learn.org/stable/auto_examples/manifold/plot_lle_digits.html

plt.figure(figsize=(10, 8))
cmap = plt.cm.tab10
plt.scatter(X_valid_2D[:, 0], X_valid_2D[:, 1], c=y_valid, s=10, cmap=cmap)
image_positions = np.array([[1., 1.]])
for index, position in enumerate(X_valid_2D):
    dist = np.sum((position - image_positions) ** 2, axis=1)
    if np.min(dist) > 0.02: # if far enough from other images
        image_positions = np.r_[image_positions, [position]]
        imagebox = mpl.offsetbox.AnnotationBbox(
            mpl.offsetbox.OffsetImage(X_valid[index], cmap="binary"),
            position, bboxprops={"edgecolor": cmap(y_valid[index]), "lw": 2})
        plt.gca().add_artist(imagebox)
plt.axis("off")
save_fig("fashion_mnist_visualization_plot")
plt.show()
```

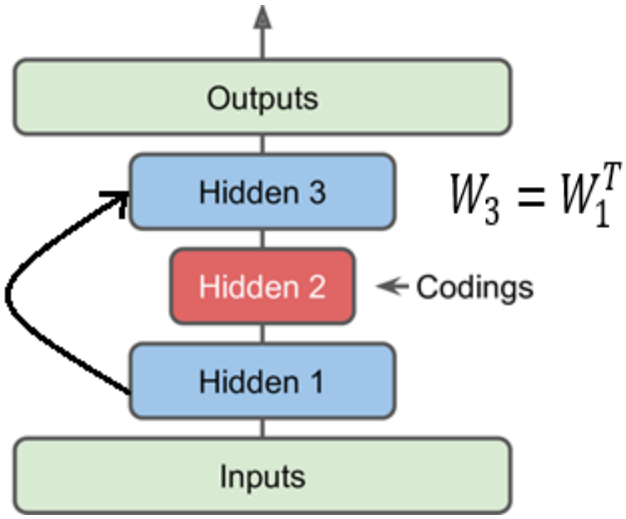
T-SNE Representation of MNIST in 2d



Tied Autoencoder

Tying Weights

- When an autoencoder is neatly symmetrical, a common technique is to *tie the weights* of the decoder layers to the weights of the encoder layers.
- This almost halves the number of weights in the model, speeding up training and limiting the risk of overfitting.
- If the autoencoder has a total of *N* layers (not counting the input layer), and $\mathbf{W}_L$ represents the connection weights of the *L*th layer (e.g., layer 1 is the first 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:



$$W_{N-L+1} = W_L^T, \text{ where } L = 1,2,..,{}^N/_2.$$

T stands for Transpose

- To understand why is the transpose needed, look at the size of inputs and outputs in the bottom layers and the top layers. Bottom layers have larger inputs and smaller outputs. Top layers the other way around, small inputs and large outputs.
- Implementing tied weights in TensorFlow using the `Dense()` layer is a bit cumbersome. It is easier to just define custom layer: `DenseTranspose`

Custom Layer: `DenseTranspose()`

```
class DenseTranspose(keras.layers.Layer):
    def __init__(self, dense, activation=None, **kwargs):
        self.dense = dense
        self.activation = keras.activations.get(activation)
        super().__init__(**kwargs)
    def build(self, batch_input_shape):
        self.biases = self.add_weight(name="bias",
                                      shape=[self.dense.input_shape[-
1]],
                                      initializer="zeros")
        super().build(batch_input_shape)
    def call(self, inputs):
        z = tf.matmul(inputs, self.dense.weights[0], transpose_b=True)
        return self.activation(z + self.biases)
```

- This custom layer acts like a regular `Dense` layer, but it uses another `Dense` layer's weights, transposed.
- Setting `transpose_b=True` is equivalent to transposing the second argument, but it's more efficient as it performs the transposition on the fly within the `matmul()` operation). However, it uses its own bias vector.

Autoencoder with Tied Weights

```
keras.backend.clear_session()
tf.random.set_seed(42)
np.random.seed(42)

dense_1 = keras.layers.Dense(100, activation="selu")
dense_2 = keras.layers.Dense(30, activation="selu")

tied_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    dense_1,
    dense_2
])

tied_decoder = keras.models.Sequential([
    DenseTranspose(dense_2, activation="selu"),
    DenseTranspose(dense_1, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])

tied_ae = keras.models.Sequential([tied_encoder, tied_decoder])

tied_ae.compile(loss="binary_crossentropy",
                optimizer=keras.optimizers.SGD(lr=1.5),
metrics=[rounded_accuracy])
history = tied_ae.fit(X_train, X_train, epochs=10,
                      validation_data=[X_valid, X_valid])
```

A few important things to note:

- The decoder's Dense layers are tied to the encoder's Dense layers.
- Biases are never tied, and never regularized.

Train Tied autoencoder

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/10
55000/55000 [==============================] - 5s 87us/sample - loss:
0.1527 - rounded_accuracy: 0.9364 - val_loss: 0.1170 -
val_rounded_accuracy: 0.9572
Epoch 2/10
55000/55000 [==============================] - 5s 84us/sample - loss:
0.1108 - rounded_accuracy: 0.9601 - val_loss: 0.1069 -
val_rounded_accuracy: 0.9621
. . . .
Epoch 10/10
55000/55000 [==============================] - 5s 86us/sample - loss:
0.0940 - rounded_accuracy: 0.9696 - val_loss: 0.0944 -
val_rounded_accuracy: 0.9695
```

- This training is tiny bit faster than the one on slide 28, when we had different weights in hidden layers 1 and 3.
- In the case of tied autoencoders, the loss is usually somewhat higher than in the non-tied case.

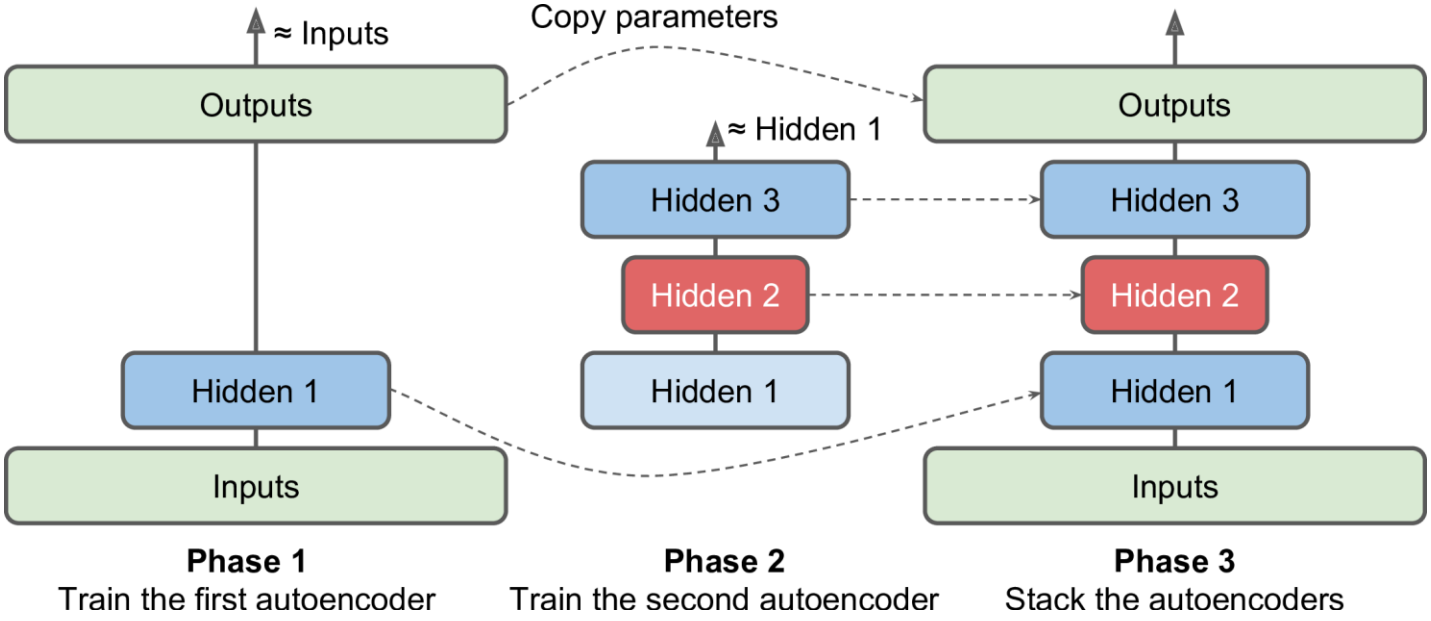
Comparison of Reconstructed Images

- Again, we can compare output images of the simple stacked autoencoder (top) and tied autoencoder bottom.

- Since tied autoencoder overall has fewer (one half) parameters, images on the bottom, with tied hidden layers, are a bit more blurry!

Training Composite Autoencoders, One at a Time

- Encoders could be quite complex and one is free to use techniques similar to those used with large CNN models. We can separately train a portion of the network, add new layers and train those on their own.
- Rather than training the whole stacked autoencoder in one go, it is often much faster to train one shallow autoencoder at a time, then stack all of them into a single stacked autoencoder, as shown on figure below. This is especially useful for very deep autoencoders.



One Autoencoder at a Time

- During the first phase of training, the first autoencoder learns to reconstruct the inputs.
- During the second phase, the second autoencoder learns to reconstruct the output of the first autoencoder's hidden layer.
- At the end, we just build a big sandwich using all these autoencoders, as shown on previous slide (i.e., you first stack the hidden layers of each autoencoder, then the output layers in reverse order). This gives us the final stacked autoencoder. We could easily train more autoencoders this way, building a very deep stacked autoencoder.
- After training an autoencoder, we just run the training set through it and capture the output of the hidden layer. This output then serves as the training set for the next autoencoder.
- Once all autoencoders have been trained this way, we simply copy the weights and biases from each autoencoder and use them to build the stacked autoencoder.
- Implementing this approach could be quite straightforward or somewhat confusing depending on your skills and patience.

Training one Autoencoder at a Time

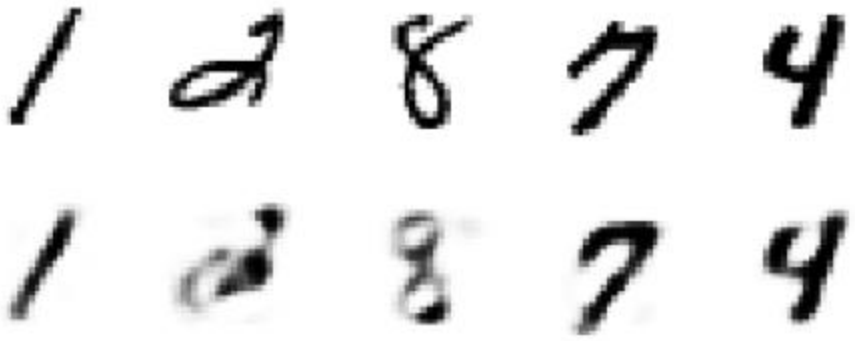
```
def train_autoencoder(n_neurons, X_train, X_valid, loss, optimizer,
                      n_epochs=10, output_activation=None, metrics=None):
    n_inputs = X_train.shape[-1]
    encoder = keras.models.Sequential([
        keras.layers.Dense(n_neurons, activation="selu",
input_shape=[n_inputs])
    ])
    decoder = keras.models.Sequential([
        keras.layers.Dense(n_inputs, activation=output_activation),
    ])
    autoencoder = keras.models.Sequential([encoder, decoder])
    autoencoder.compile(optimizer, loss, metrics=metrics)
    autoencoder.fit(X_train, X_train, epochs=n_epochs,
                    validation_data=[X_valid, X_valid])
    return encoder, decoder, encoder(X_train), encoder(X_valid)
```

First part of Training, Outer Autoencoder

```
tf.random.set_seed(42); np.random.seed(42); K = keras.backend
X_train_flat = K.batch_flatten(X_train) # equivalent to .reshape(-1, 28 * 28)
X_valid_flat = K.batch_flatten(X_valid)
enc1, dec1, X_train_enc1, X_valid_enc1 = train_autoencoder(
    100, X_train_flat, X_valid_flat, "binary_crossentropy",
    keras.optimizers.SGD(lr=1.5), output_activation="sigmoid",
    metrics=[rounded_accuracy])
enc2, dec2, _, _ = train_autoencoder(
    30, X_train_enc1, X_valid_enc1, "mse", keras.optimizers.SGD(lr=0.05),
    output_activation="selu"))

Train on 55000 samples, validate on 5000 samples
Epoch 1/10
55000/55000 [==============================] - 4s 80us/sample - loss: 0.1866
- rounded_accuracy: 0.9228 - val_loss: 0.1346 - val_rounded_accuracy: 0.9500
Epoch 2/10
55000/55000 [==============================] - 4s 74us/sample - loss: 0.1197
- rounded_accuracy: 0.9575 - val_loss: 0.1092 - val_rounded_accuracy: 0.9633
. . . .
Epoch 10/10
55000/55000 [==============================] - 3s 61us/sample - loss: 1.1436
- val_loss: 1.0149
```

Results of the First part of the training

```
stacked_ae_1_by_1 = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    enc1, enc2, dec2, dec1,
    keras.layers.Reshape([28, 28])
])
show_reconstructions(stacked_ae_1_by_1)
plt.show()
```



Second part of Training, with the Inner Layer

```
stacked_ae_1_by_1.compile(loss="binary_crossentropy",
                          optimizer=keras.optimizers.SGD(lr=0.1),
metrics=[rounded_accuracy])
history = stacked_ae_1_by_1.fit(X_train, X_train, epochs=10,
                                validation_data=[X_valid, X_valid])


Train on 55000 samples, validate on 5000 samples
Epoch 1/10
55000/55000 [==============================] - 5s 88us/sample - loss:
0.1177 - rounded_accuracy: 0.9556 - val_loss: 0.1156 -
val_rounded_accuracy: 0.9569
….
Epoch 10/10
55000/55000 [==============================] - 5s 83us/sample - loss:
0.1101 - rounded_accuracy: 0.9603 - val_loss: 0.1098 -
val_rounded_accuracy: 0.9606
```

- The output of the second training shows improvement over the output of the first training on the previous slide

Unsupervised Training with Stacked Autoencoders

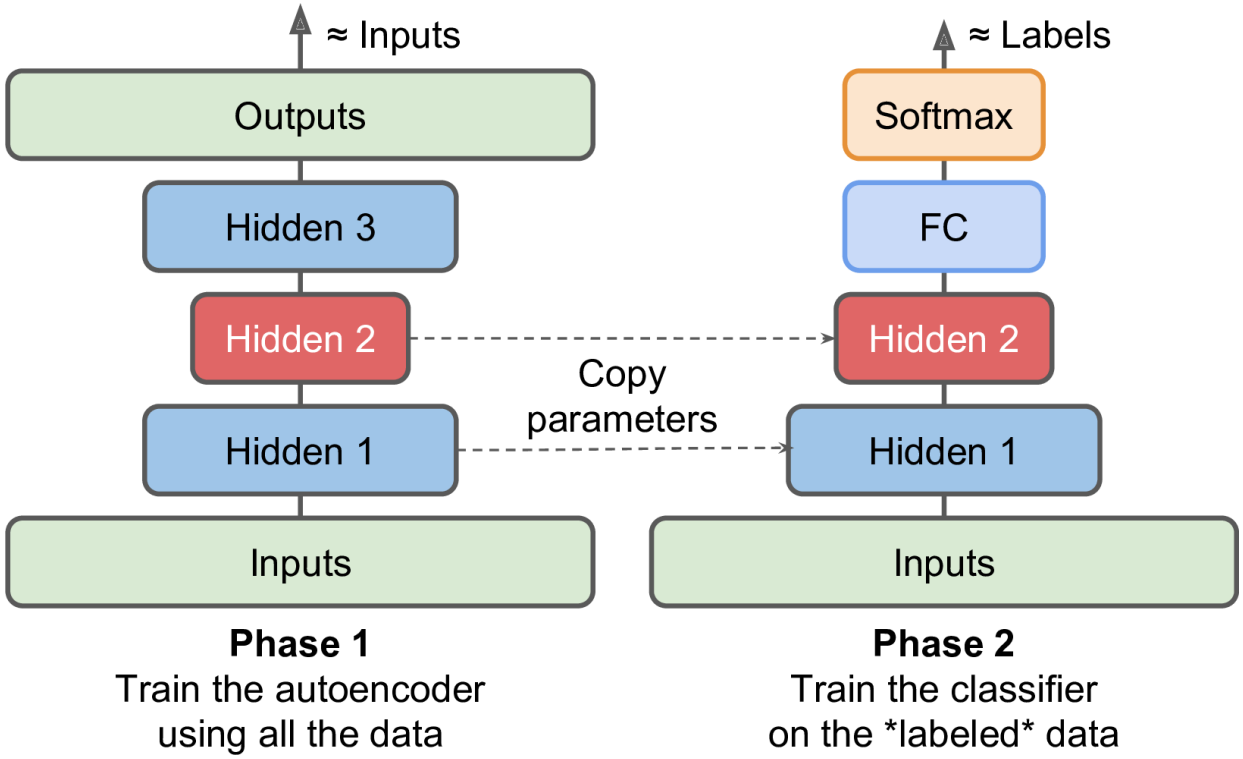
Unsupervised Pre-training Using Stacked Autoencoders

- If you are dealing with a complex supervised task but you do not have a lot of labeled training data, one solution is to find a neural network that performs a similar task, and then reuse its lower layers. This makes it possible to train a high-performance model using only little training data because your neural network won't have to learn all the low-level features; it will just reuse the feature detectors learned by the existing net.
- Similarly, if you have a large dataset but most of it is unlabeled, you can first train a stacked auto-encoder using all the data, then reuse the lower layers to create a neural network for your actual task and train it using the labeled data.
- Figure on the next slide shows how to use a stacked autoencoder to perform unsupervised pre-training for a classification neural network.
- The stacked auto-encoder itself is typically trained one autoencoder at a time. When training the classifier, if you really don't have much labeled training data, you may want to freeze the pre-trained layers (at least the lower ones).

Pretraining Using Stacked Autoencoders

- This situation is quite common, because building a large unlabeled dataset is often cheap (e.g., a simple script can download millions of images off the internet). However, labeling samples can only be done reliably by humans (e.g., classifying images as cute or not).
- Labeling instances is time-consuming and costly, so it is quite common to have only a few thousand labeled instances.

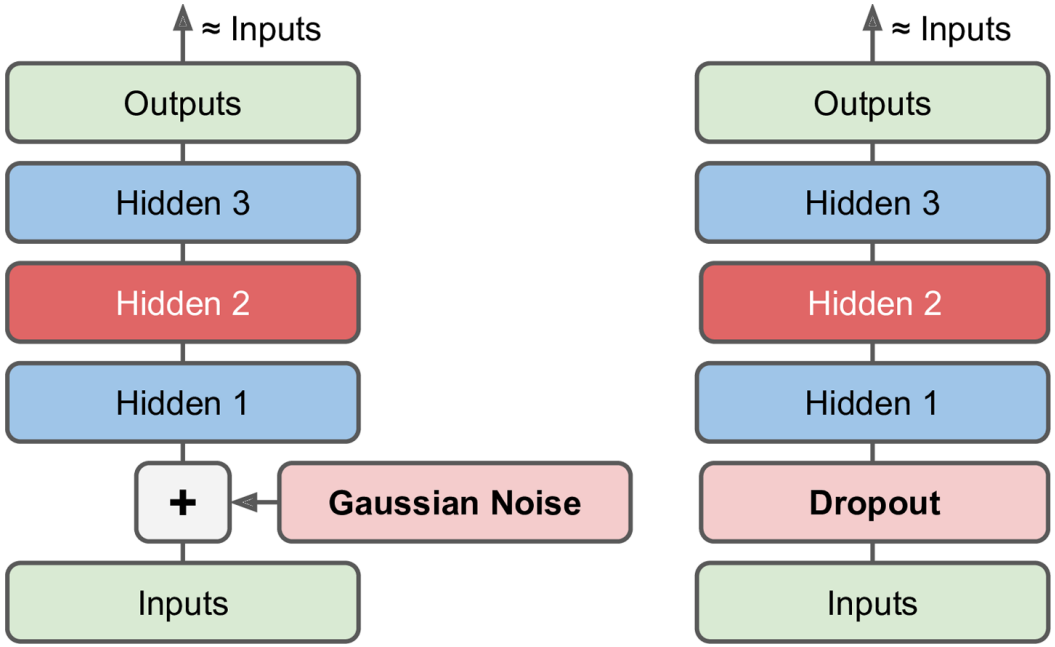
- Keras implementation: just train an autoencoder using all the training data, then reuse its encoder layers to create a new neural network.
- The encoder portion is subsequently frozen



Denoising Autoencoders

Denoising Autoencoders

- One important use case for autoencoders is removing noise from images and other data. One way to force an autoencoder to learn this useful mechanism is to add noise to autoencoder's inputs, and train it to recover the original, noise-free inputs. This prevents the autoencoder from trivially copying its inputs to its outputs, so it ends up having to find patterns in the data.
- The noise can be pure Gaussian noise added to the inputs, or it can be randomly switched off inputs, just like in the `Dropout` layer.



- Denoising autoencoders with Gaussian noise (left) and dropout (right).
- The Dropout (or Gaussian) layer is only active during training

Denoising Autoencoder with Gaussian Noise

- When adding the Gaussian noise, we start with a regular autoencoder, and add noise to the inputs or add a `keras.layers.GaussianNoise()` as the first layer.
- The reconstruction loss is calculated based on the original inputs:

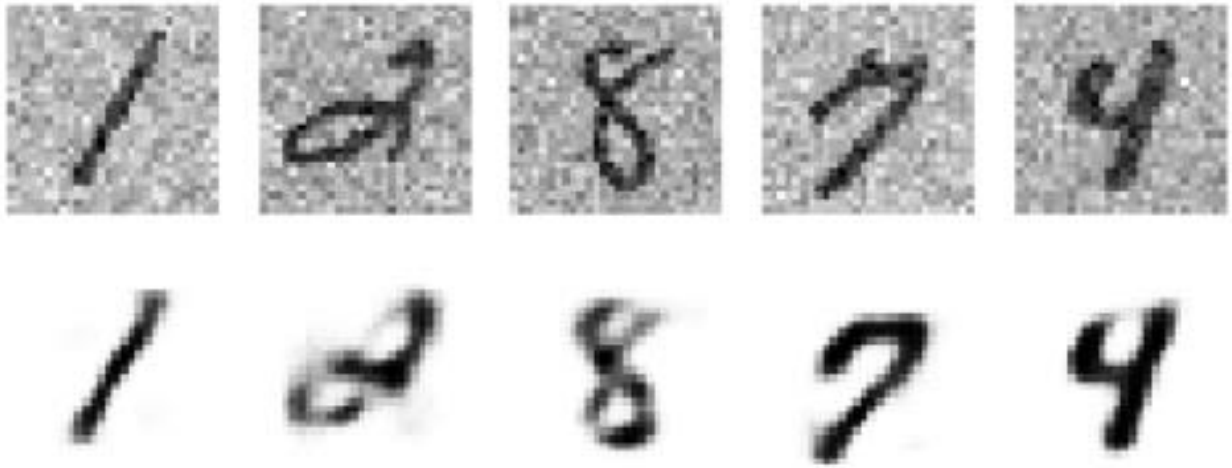
```
tf.random.set_seed(42)
np.random.seed(42)
denoising_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.GaussianNoise(0.2),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(30, activation="selu")
])
denoising_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[30]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
denoising_ae = keras.models.Sequential([denoising_encoder,
denoising_decoder])
denoising_ae.compile(loss="binary_crossentropy",
optimizer=keras.optimizers.SGD(lr=1.0),
                     metrics=[rounded_accuracy])
```

Training

```
history = denoising_ae.fit(X_train, X_train, epochs=10,
                           validation_data=[X_valid, X_valid])

Train on 55000 samples, validate on 5000 samples
Epoch 1/10
55000/55000 [==============================] - 5s 94us/sample - loss: 0.2046 -
rounded_accuracy: 0.9070 - val_loss: 0.1568 - val_rounded_accuracy: 0.9340
Epoch 2/10
55000/55000 [==============================] - 5s 89us/sample - loss: 0.1464 -
rounded_accuracy: 0.9397 - val_loss: 0.1335 - val_rounded_accuracy: 0.9474
Epoch 3/10
55000/55000 [==============================] - 5s 89us/sample - loss: 0.1339 -
rounded_accuracy: 0.9466 - val_loss: 0.1276 - val_rounded_accuracy: 0.9506
Epoch 4/10
55000/55000 [==============================] - 5s 91us/sample - loss: 0.1295 -
rounded_accuracy: 0.9489 - val_loss: 0.1240 - val_rounded_accuracy: 0.9525
Epoch 5/10
55000/55000 [==============================] - 5s 87us/sample - loss: 0.1265 -
rounded_accuracy: 0.9505 - val_loss: 0.1212 - val_rounded_accuracy: 0.9538
. . . . .

Epoch 9/10
55000/55000 [==============================] - 5s 87us/sample - loss: 0.1189 -
rounded_accuracy: 0.9547 - val_loss: 0.1130 - val_rounded_accuracy: 0.9588
Epoch 10/10
55000/55000 [==============================] - 5s 88us/sample - loss: 0.1168 -
rounded_accuracy: 0.9560 - val_loss: 0.1112 - val_rounded_accuracy: 0.9598
```

show_reconstructions(), Gaussian Noise

```
tf.random.set_seed(42)
np.random.seed(42)

noise = keras.layers.GaussianNoise(0.2)
show_reconstructions(denoising_ae, noise(X_valid, training=True))
plt.show()
```



Denoising Autoencoder with Dropout

- Implementing the dropout version, which is more common, is not much harder

```
tf.random.set_seed(42)
np.random.seed(42)

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])
dropout_ae.compile(loss="binary_crossentropy",
optimizer=keras.optimizers.SGD(lr=1.0),
                   metrics=[rounded_accuracy])
```

Training of De-noising Autoencoder with Dropout

```
history = dropout_ae.fit(X_train, X_train, epochs=10,
                         validation_data=[X_valid, X_valid])
Train on 55000 samples, validate on 5000 samples
Epoch 1/10
55000/55000 [==============================] - 5s 95us/sample - loss: 0.2118 -
rounded_accuracy: 0.9025 - val_loss: 0.1615 - val_rounded_accuracy: 0.9314
Epoch 2/10
55000/55000 [==============================] - 5s 88us/sample - loss: 0.1599 -
rounded_accuracy: 0.9313 - val_loss: 0.1396 - val_rounded_accuracy: 0.9444
Epoch 3/10
55000/55000 [==============================] - 5s 88us/sample - loss: 0.1487 -
rounded_accuracy: 0.9375 - val_loss: 0.1336 - val_rounded_accuracy: 0.9478
Epoch 4/10
55000/55000 [==============================] - 5s 87us/sample - loss: 0.1429 -
rounded_accuracy: 0.9406 - val_loss: 0.1283 - val_rounded_accuracy: 0.9499
Epoch 5/10
. . . .
55000/55000 [==============================] - 5s 88us/sample - loss: 0.1389 -
rounded_accuracy: 0.9429 - val_loss: 0.1250 - val_rounded_accuracy: 0.9519
Epoch 9/10
55000/55000 [==============================] - 5s 88us/sample - loss: 0.1302 -
rounded_accuracy: 0.9476 - val_loss: 0.1180 - val_rounded_accuracy: 0.9561
Epoch 10/10
55000/55000 [==============================] - 5s 88us/sample - loss: 0.1288 -
rounded_accuracy: 0.9484 - val_lo
```

show_reconstructions(), Dropout

```
tf.random.set_seed(42)
np.random.seed(42)

dropout = keras.layers.Dropout(0.5)
show_reconstructions(dropout_ae, dropout(X_valid, training=True))
```

Variational Autoencoders

References

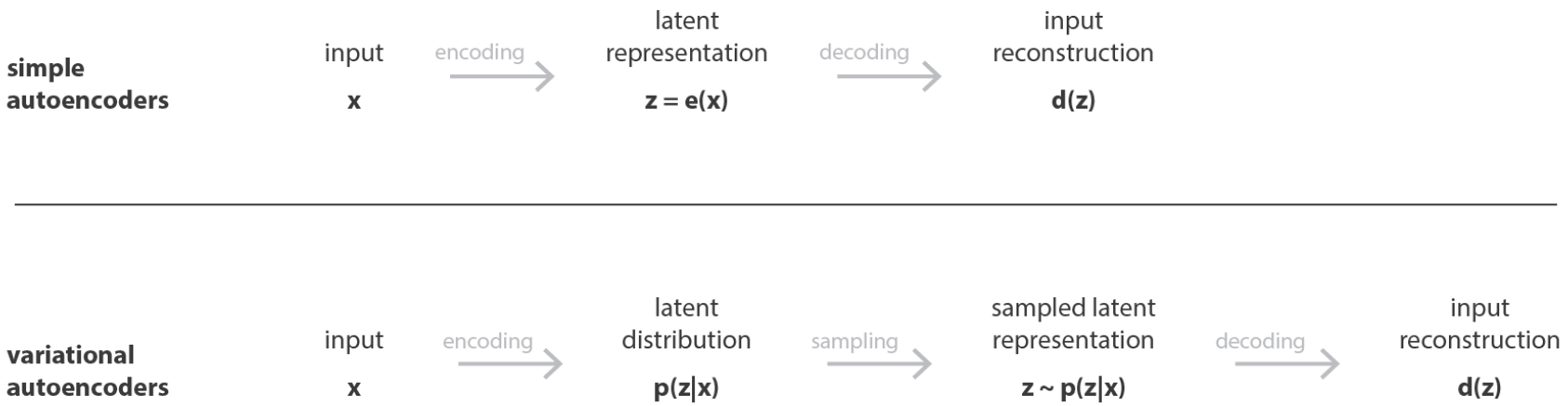
This lecture follows material presented in:

- Chapter 17, Hands on Machine Learning with Scikit-learn, Keras & TensorFlow, 2nd Edition, by Aurélien Géron, O'Reilly 2019
- Chapter 12, Deep Learning with Python, 2nd edition by Francois Chollet
- "Building Autoencoders in Keras" by Francois Chollet (https://blog.keras.io/building-autoencoders-in-keras.html)

The problem with standard autoencoders

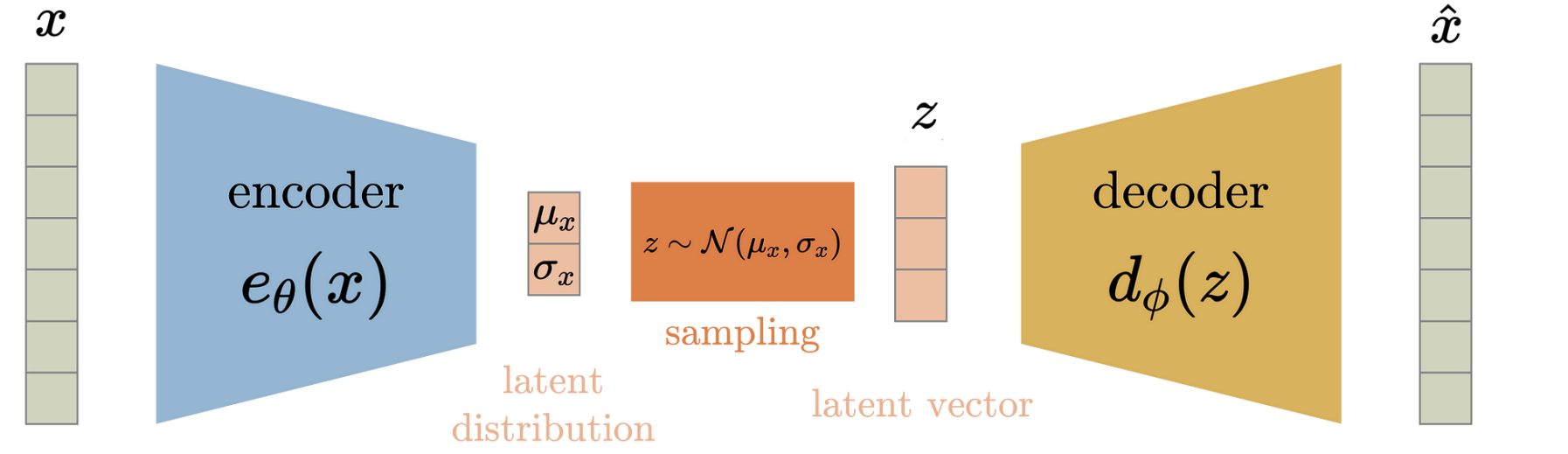
- Standard autoencoders learn to generate compact representations and reconstruct their inputs well.
- If we try to use autoencoders for generation, we have a serious problem. The latent space of `codings` may not be continuous or allow easy interpolation.
- Naïve attempt to generate new instances similar to those present in the training set by making linear interpolation between `codings` produced by autoencoders does not produce satisfactory images.

Variational Autoencoders



- ***Variational autoencoders*** are introduced in 2014 by Diederik Kingma and Max Welling (https://arxiv.org/pdf/1312.6114.pdf) and they appear to have found one solution to the interpolation problem.
- By using VAEs we generate images that appear to reside on the same manifold as the training images.
- *Variational autoencoders* are *probabilistic autoencoders*, meaning that their outputs are partly determined by chance, even after training (as opposed to de-noising autoencoders, which use randomness only during training).
- *Variational autoencoders* are *generative autoencoders*, meaning that they can generate new instances that look like they were sampled from the training set.

Variational Autoencoders



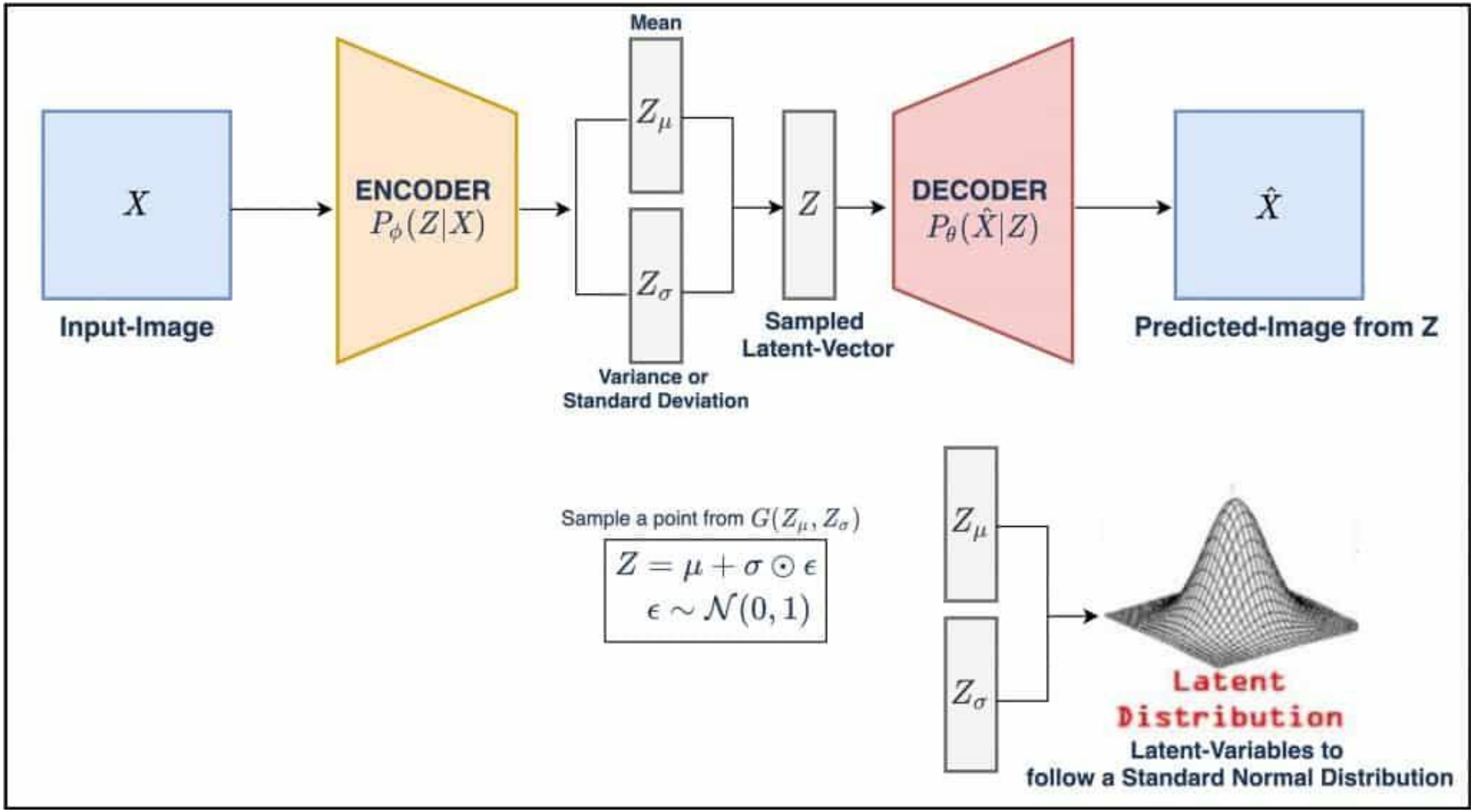
$$\text{reconstruction loss} = \|x - \hat{x}\|_2 = \|x - d_\phi(z)\|_2 = \|x - d_\phi(\mu_x + \sigma_x \epsilon)\|_2$$

$$\mu_x, \sigma_x = e_\theta(x), \quad \epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$$

$$similarity\ loss = KL\ Divergence = D_{KL}(\mathcal{N}(\mu_x, \sigma_x) \parallel \mathcal{N}(\mathbf{0}, \mathbf{I}))$$

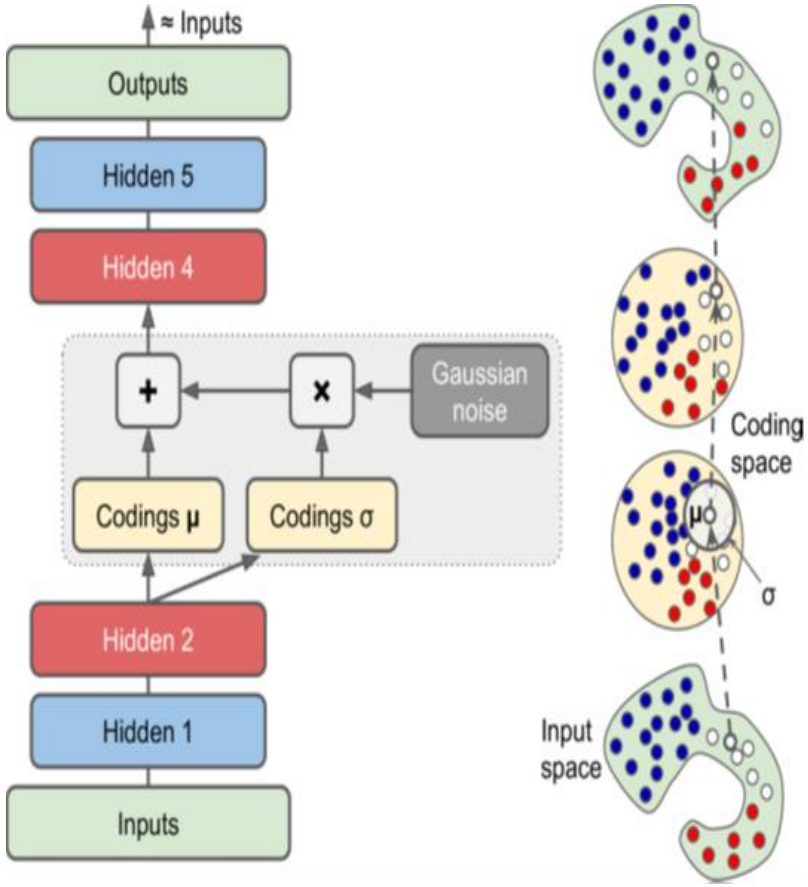
$$loss = reconstruction\ loss + similarity\ loss$$

Variational Autoencoders



Variational Autoencoders

- Figure on the right shows typical architecture of a variational autoencoder.
- Now, instead of directly producing a `codings` vector for a given input, the encoder produces two vectors: *mean coding* $\boldsymbol{\mu}$ and a vector of standard deviations $\boldsymbol{\sigma}$. The second vector $\boldsymbol{\sigma}$ is a vector and not a covariance matrix one would expect in a multidimensional space**.**
- We write $\boldsymbol{\mu} = (\mu_1, \mu_2, \ldots, \mu_K)$, and $\boldsymbol{\sigma} = (\sigma_1, \sigma_2, \ldots, \sigma_K)$. Number K is the dimension of the latent vector space, i.e., the dimensionality of the `codings` vectors.
- Subsequently, using Gaussian noise, we create K *normal random variables* $\{\varepsilon_i, i = 1, .., K\}$ with mean 0 and standard variation 1.
- An actual `coding` is then obtained as a vector of K values $(\mu_i + \sigma_i \varepsilon_i)$. This process of generating `codings` is usually describe as "sampling randomly from a Gaussian distribution with mean $\boldsymbol{\mu}$ and standard deviation $\boldsymbol{\sigma}$". The K-dimensional vector with elements $(\mu_i + \sigma_i \varepsilon_i)$ is passed to the decoder.

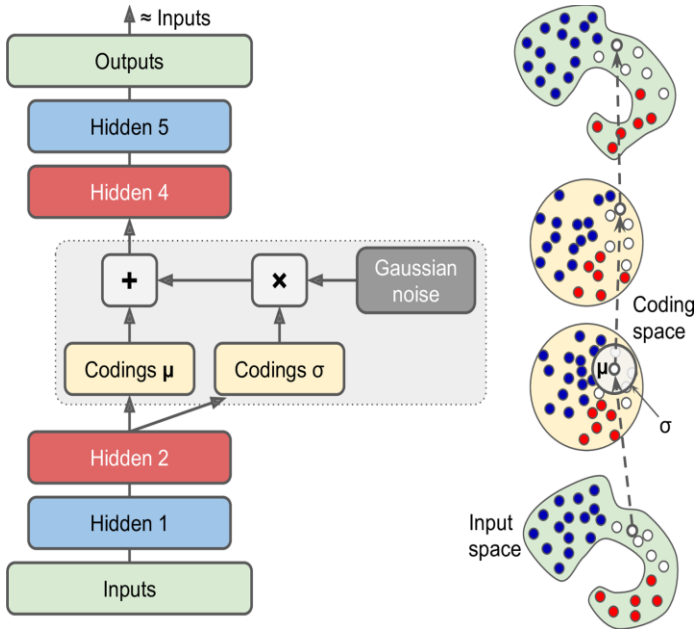


Variational Autoencoders

- Please note that we did not generate a normal random K-dimensional vector by invoking one of many routines in Scikit-learn or `tf.random` package. Only K normal random values generated are $\{\varepsilon_i\}$.
- After receiving the `codings` vector, the decoder just decodes that vector.
- Decoding produces a reconstructed image and we minimize the distance of that image from the input image using backpropagation and the gradient descent algorithm.
- The right part of the diagram on the next slide shows a training instance going through this autoencoder. First, the encoder produces μ and σ. Then elements of σ *are* multiplied by K *Gaussian random normal variables* and added to the elements of μ, $(\mu_i + \sigma_i \varepsilon_i)$, to produce new K-dimensional `codings` vector. Finally, this coding is decoded, and the final output (reconstructed image) is forced to resembles the training instance.

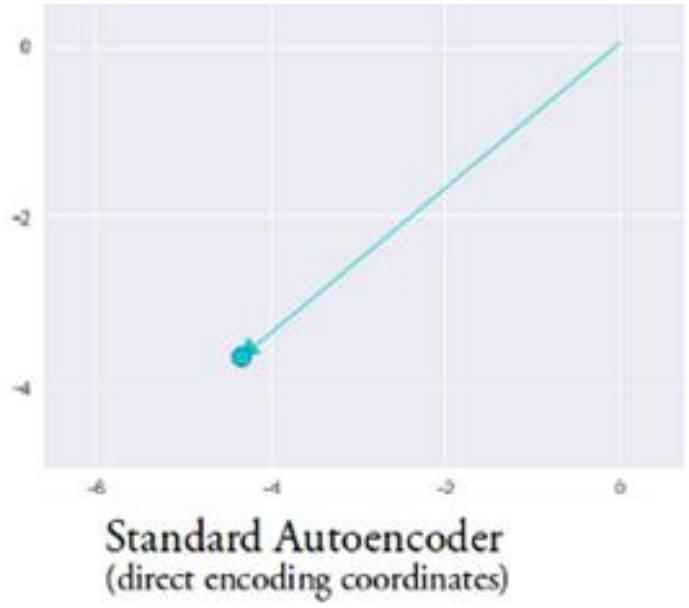
Variational autoencoder

- Any one of input samples (images) is run through the training process many times. For each pass of a training image through the encoder, we produce a different `codings` vector. Collection of those `codings` for each training image looks as though they were sampled from a Gaussian distribution.
- We can force a desired distribution in the latent space by using KL divergence between the actual distribution and the desired (Gaussian or other) distribution. As explained on the next slides our loss function contains standard binary cross entropy term to force the reconstructed and original images to be close and the KL divergence term in the latent space to enforce the desired distribution of points in the latent space.
- During training, the cost function pushes the `codings` to gradually migrate within the coding space (also called the *latent space*) to occupy a roughly (hyper)spherical region that looks like a cloud of Gaussian points. One consequence is that after training a variational autoencoder, you can very easily generate a new instance: sample a random coding from the Gaussian distribution, decode it, and you get an instance similar to the original among training samples.

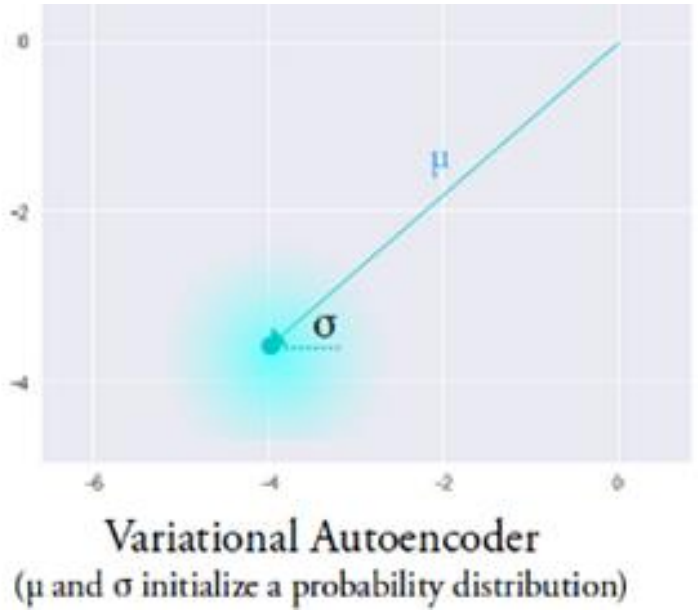


Stochastically Generated `codings` Vecors

- This stochastic generation means, that even for the same input, while the mean and standard deviations remain the same, the actual encoding will somewhat vary on every single pass simply due to sampling.
- In standard autoencoders the distribution of points in the latent space which belong to any one image is rather narrow, like in the image on the left.
- In variational autoencoder, points in the latent space which correspond to one image cover a broader distribution, like in the image on the right.



Standard Autoencoder
(direct encoding coordinates)



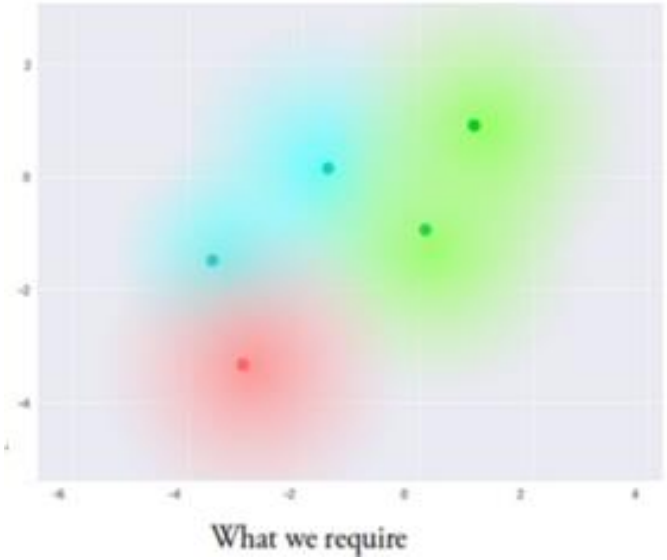
Variational Autoencoder
(μ and σ initialize a probability distribution)

Need for overlapping regions

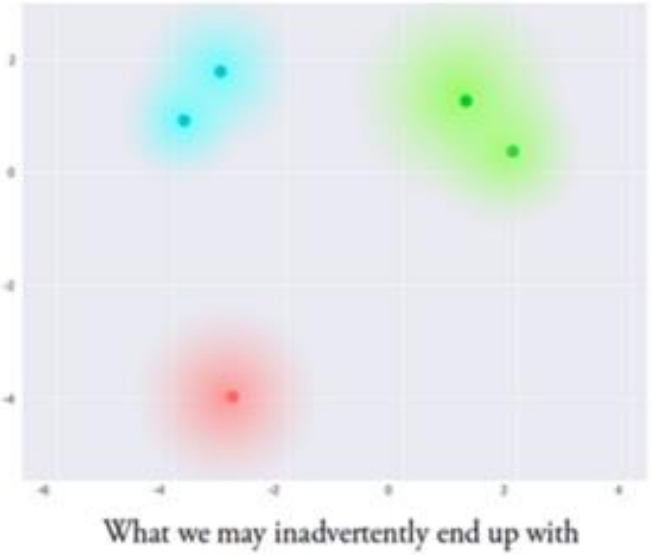
- Intuitively, the mean vector controls where the encoding of an input should be centered around, while the standard deviation controls the "area", how much from the mean the encodings can vary. *As encodings are generated at random from anywhere inside the "circle" (the distribution), the decoder learns that not only is a single point in latent space referring to a sample of that class, but all nearby points refer to the same sample as well.* This allows the decoder to not just decode individual, specific encodings in the latent space (leaving the decodable latent space discontinuous), but ones that slightly vary too, as the decoder is exposed to a range of variations of the en(`codings`) of the same input during training.
- The model is now exposed to a certain degree of local variation by varying the encoding of one sample, resulting in smooth latent spaces on a local scale, that is, for similar samples. Ideally, we want overlap between samples that are not very similar too, in order to interpolate *between* classes. However, since there are *no limits* on what values vectors $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ can take, the encoder can learn to generate very different $\boldsymbol{\mu}$ for different classes, clustering them apart, and minimize $\boldsymbol{\sigma}$, making sure the encodings themselves don't vary much for the same sample (that is, less uncertainty for the decoder). This allows the decoder to efficiently reconstruct the *training* data.

Need for overlapping regions

- What we ideally want are encodings, *all* of which are as close as possible to each other while still being distinct, allowing smooth interpolation, and enabling the construction of *new* samples.
- To force this, we introduce the Kullback–Leibler divergence (KL divergence) into the loss function.
- 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.



What we require

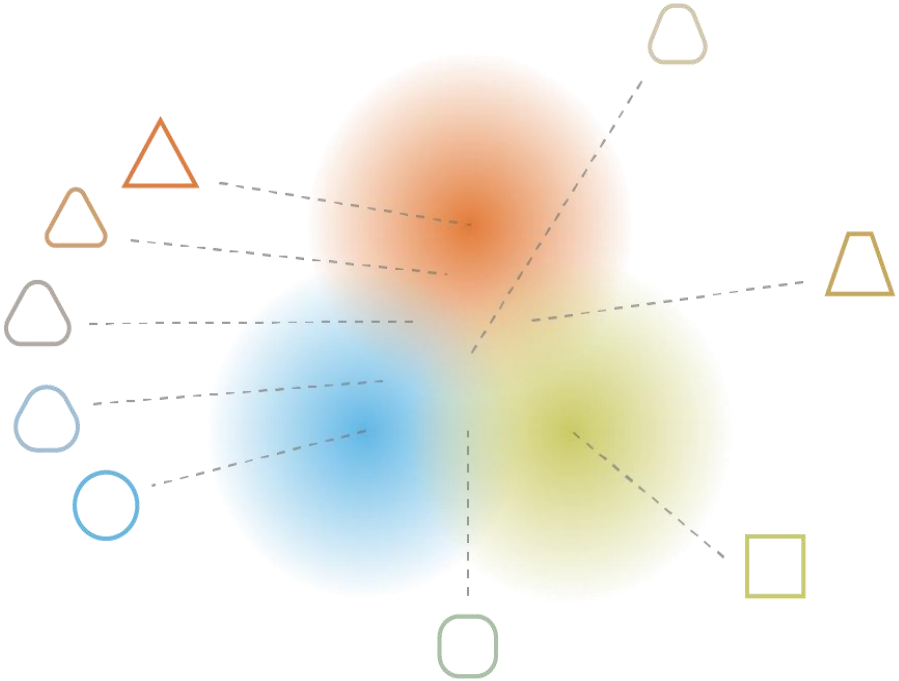


What we may inadvertently end up with

VAE & Generative properties

To generate *new* images, the VAE must be able to sample from anywhere in the latent space *between* the original data points. For this to be possible, the latent space must exhibit two types of regularity:

- **Continuity:** Nearby points in latent space should yield similar content when decoded.
- **Completeness:** Any point sampled from the latent space should yield meaningful content when decoded.



The Cost Function

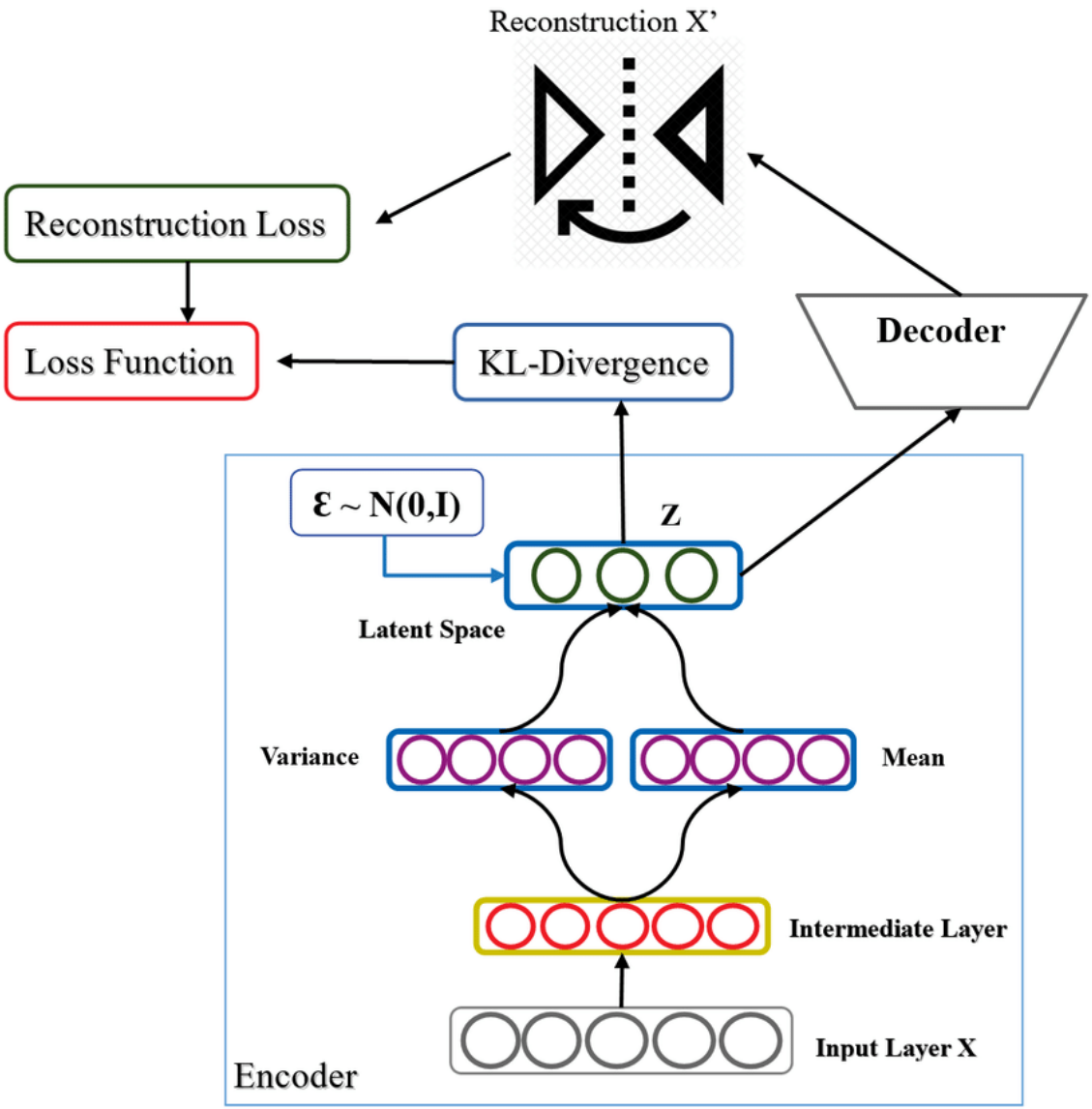
- The cost function of a typical VAE is composed of two parts. The first part is the usual reconstruction loss that pushes the autoencoder to reproduce its inputs (we use cross entropy loss function).
- The second part is the *latent loss* that pushes the autoencoder to have `codings` that look as though they were sampled from a simple Gaussian distribution, for which we use the KL divergence between the target distribution (the Gaussian distribution) and the actual distribution of the `codings`. Latent loss is

$$\mathcal{L} = -\frac{1}{2}\sum_{i=1}^{K} 1 + \log\left(\sigma_i^2\right) - \sigma_i^2 - \mu_i^2$$

- $\mathcal{L}$ is the latent loss, K 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 $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ (which contain all the μ_i and σ_i) are outputs of the encoder.
- *Variational autoencoder's latent loss, rewritten using* $\boldsymbol{\gamma} = \log(\boldsymbol{\sigma}^2)$, reads:

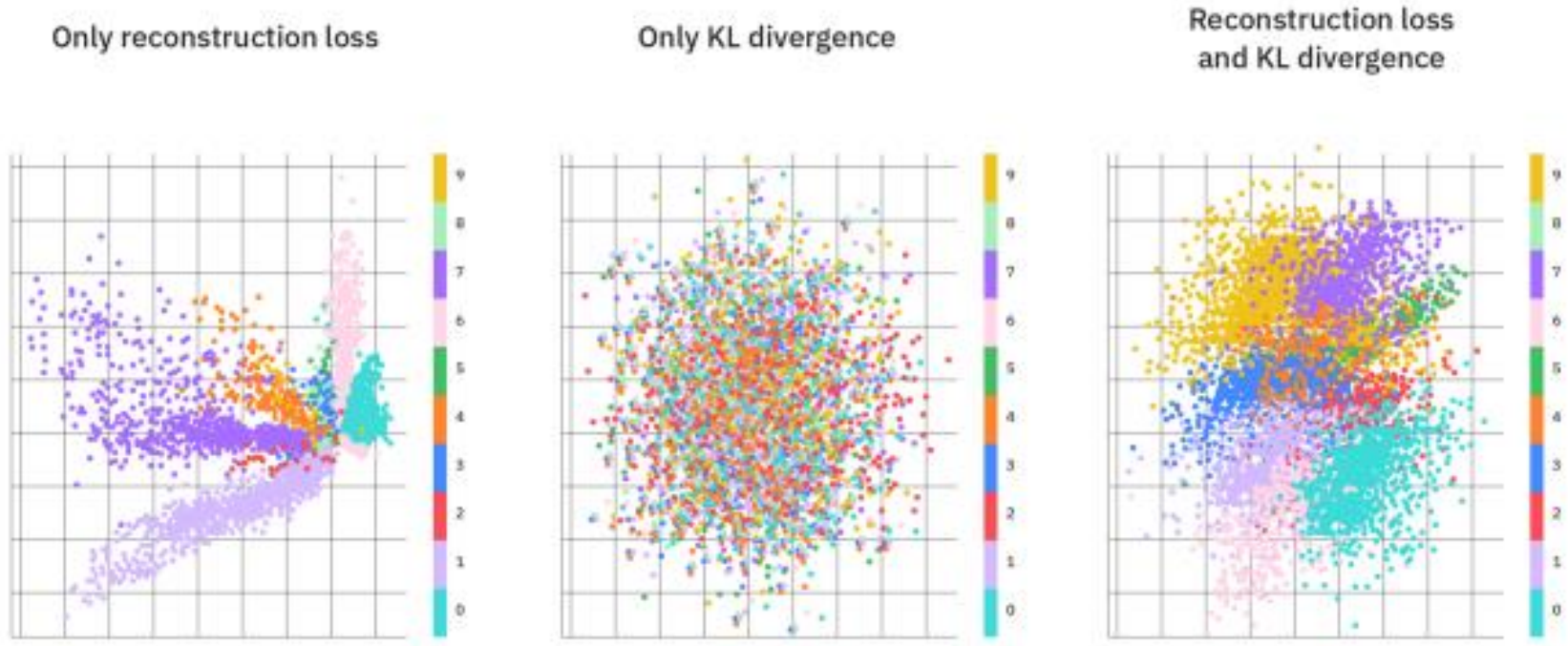
$$\mathcal{L} = -\frac{1}{2}\sum_{i=1}^{K} 1 + \gamma_i - \exp(\gamma_i) - \mu_i^2$$

The Cost Function



The Cost Function

Reconstruction Loss + Regularization term



The latent loss and the reconstruction loss:

- We first apply Equation from the previous slide to compute the latent loss for each instance in the batch (we sum 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 "binary_crossentropy" 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.

```
K = keras.backend
latent_loss = -0.5 * K.sum(
1 + codings_log_var - K.exp(codings_log_var) - K.square(codings_mean),
axis=-1)
variational_ae.add_loss(K.mean(latent_loss) / 784.)
variational_ae.compile(loss="binary_crossentropy", optimizer="rmsprop")
```

Common Version of Variational Autoencoder

- As indicated on the previous slide, one common approach is to train the encoder to produce $\gamma = \log(\sigma^2)$ rather than σ. Wherever we need σ, we can just compute $\sigma = \exp(\gamma/2)$
- This makes it a bit easier for the encoder to capture values of σ (*s*igmas) of different scales, and thus it helps speed up the convergence.
- We will need a custom layer to sample `codings`, given **μ** and **γ**:

```
class Sampling(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 Normal distribution, with mean 0 and standard deviation 1. Then it multiplies it by exp(**γ** / 2) (which is equal to **σ**, as you can verify), and finally it adds **μ** and returns the result. This samples a codings vector from the Normal distribution with mean **μ** and standard deviation **σ**.

Implementation, encoder

- Next, we create the encoder, using the Functional API, because the model is not entirely sequential:

```
tf.random.set_seed(42)
np.random.seed(42)
codings_size = 10
inputs = keras.layers.Input(shape=[28, 28])
z = keras.layers.Flatten()(inputs)
z = keras.layers.Dense(150, activation="selu")(z)
z = keras.layers.Dense(100, activation="selu")(z)
codings_mean = keras.layers.Dense(codings_size)(z)
codings_log_var = keras.layers.Dense(codings_size)(z)
codings = Sampling()([codings_mean, codings_log_var])
variational_encoder = keras.models.Model(
    inputs=[inputs], outputs=[codings_mean, codings_log_var, codings])
```

- Note that the `Dense` layers that output `codings_mean` (μ) 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, in case you want to inspect the values of `codings_mean` and `codings_log_var`.

Implementation, decoder

- The only output we will use in the decoder is the last one (`codings`). Now let's build the decoder:

```
decoder_inputs = keras.layers.Input(shape=[codings_size])

x = keras.layers.Dense(100, activation="selu")(decoder_inputs)
x = keras.layers.Dense(150, activation="selu")(x)
x = keras.layers.Dense(28 * 28, activation="sigmoid")(x)
outputs = keras.layers.Reshape([28, 28])(x)
variational_decoder = keras.models.Model(inputs=[decoder_inputs],
outputs=[outputs])
_, _, codings = variational_encoder(inputs)
reconstructions = variational_decoder(codings)
variational_ae = keras.models.Model(inputs=[inputs],
outputs=[reconstructions])
latent_loss = -0.5 * K.sum(
    1 + codings_log_var - K.exp(codings_log_var) - K.square(codings_mean),
    axis=-1)
variational_ae.add_loss(K.mean(latent_loss) / 784.)
variational_ae.compile(loss="binary_crossentropy", optimizer="rmsprop",
metrics=[rounded_accuracy])
```

Training

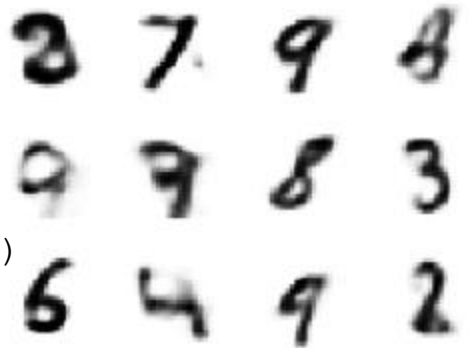
```
history = variational_ae.fit(X_train, X_train, epochs=25, batch_size=128,
                             validation_data=[X_valid, X_valid])

Train on 55000 samples, validate on 5000 samples
Epoch 1/25
55000/55000 [==============================] - 3s 55us/sample - loss: 0.2234 -
rounded_accuracy: 0.9055 - val_loss: 0.1834 - val_rounded_accuracy: 0.9289
Epoch 2/25
55000/55000 [==============================] - 2s 41us/sample - loss: 0.1773 -
rounded_accuracy: 0.9320 - val_loss: 0.1701 - val_rounded_accuracy: 0.9366
Epoch 3/25
55000/55000 [==============================] - 2s 41us/sample - loss: 0.1664 -
rounded_accuracy: 0.9385 - val_loss: 0.1631 - val_rounded_accuracy: 0.9405
Epoch 4/25
55000/55000 [==============================] - 2s 41us/sample - loss: 0.1609 -
rounded_accuracy: 0.9420 - val_loss: 0.1606 - val_rounded_accuracy: 0.9421
Epoch 5/25
55000/55000 [==============================] - 2s 41us/sample - loss: 0.1575 -
rounded_accuracy: 0.9442 - val_loss: 0.1554 - val_rounded_accuracy: 0.9458
Epoch 6/25
……. ….
Epoch 22/25
55000/55000 [==============================] - 2s 41us/sample - loss: 0.1425 -
rounded_accuracy: 0.9540 - val_loss: 0.1424 - val_rounded_accuracy: 0.9544
Epoch 23/25
55000/55000 [==============================] - 2s 41us/sample - loss: 0.1422 -
rounded_accuracy: 0.9543 - val_loss: 0.1435 - val_rounded_accuracy: 0.9538
Epoch 24/25
55000/55000 [==============================] - 2s 41us/sample - loss: 0.1419 -
rounded_accuracy: 0.9544 - val_loss: 0.1416 - val_rounded_accuracy: 0.9548
Epoch 25/25
55000/55000 [==============================] - 2s 43us/sample - loss: 0.1416 -
rounded_accuracy: 0.9546 - val_loss: 0.1427 - val_rounded_accuracy: 0.9543
```

Generate random Images

```
def plot_multiple_images(images, n_cols=None):
    n_cols = n_cols or len(images)
    n_rows = (len(images) - 1) // n_cols + 1
    if images.shape[-1] == 1:
        images = np.squeeze(images, axis=-1)
    plt.figure(figsize=(n_cols, n_rows))
    for index, image in enumerate(images):
        plt.subplot(n_rows, n_cols, index + 1)
        plt.imshow(image, cmap="binary")
        plt.axis("off")
```

- We generate a few random `codings`, decode them and plot the resulting images:

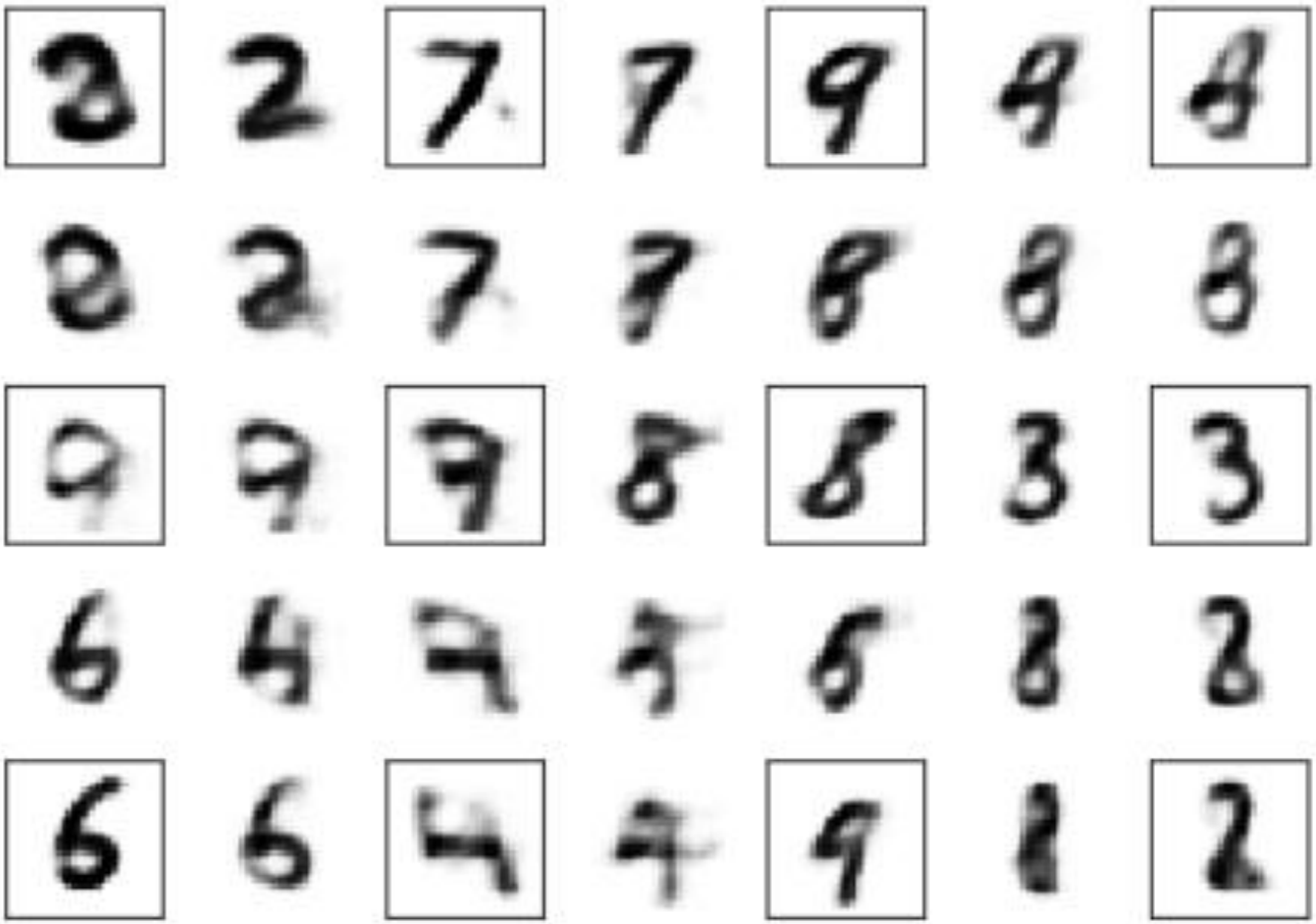
```
tf.random.set_seed(42)

codings = tf.random.normal(shape=[12, codings_size])
images = variational_decoder(codings).numpy()
plot_multiple_images(images, 4)
save_fig("vae_generated_images_plot", tight_layout=False)
```



Semantic interpolation between these images

```
tf.random.set_seed(42)
np.random.seed(42)

codings_grid = tf.reshape(codings, [1, 3, 4, codings_size])
larger_grid = tf.image.resize(codings_grid, size=[5, 7])
interpolated_codings = tf.reshape(larger_grid, [-1, codings_size])
images = variational_decoder(interpolated_codings).numpy()

plt.figure(figsize=(7, 5))
for index, image in enumerate(images):
    plt.subplot(5, 7, index + 1)
    if index%7%2==0 and index//7%2==0:
        plt.gca().get_xaxis().set_visible(False)
        plt.gca().get_yaxis().set_visible(False)
    else:
        plt.axis("off")
    plt.imshow(image, cmap="binary")
save_fig("semantic_interpolation_plot", tight_layout=False)
```

Semantic interpolation between these images



Another Keras Implementation of VAE
Representing `codings` as images

Another implementation in Keras

- Keras' API makes it convenient to implement autoencoders

```
from tensorflow import keras
from tensoflow.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

# this is our input placeholder
input_img = Input(shape=(784,))
# "encoded" is the encoded representation of the input
encoded = Dense(encoding_dim, activation='relu')(input_img)
# "decoded" is the lossy reconstruction of the input
decoded = Dense(784, activation='sigmoid')(encoded)

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

Encoder, Decoder

- We will create a separate encoder model:

```
# this model maps an input to its encoded representation
encoder = Model(input_img, encoded)
```

- As well as the decoder model:

```
# create a placeholder for an encoded (32-dimensional) input
encoded_input = Input(shape=(encoding_dim,))
# retrieve the last layer of the autoencoder model
decoder_layer = autoencoder.layers[-1]
# create the decoder model
decoder = Model(encoded_input, decoder_layer(encoded_input))
```

- Now let's train our autoencoder to reconstruct MNIST digits.
- First, we'll configure our model to use a per-pixel binary crossentropy loss, and the Adadelta optimizer:

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

- We're using MNIST digits, and we're discarding the labels (since we're only interested in encoding/decoding the input images).

```
from keras.datasets import mnist
import numpy as np
(x_train, _), (x_test, _) = mnist.load_data()
```

Normalize and Train

- We will normalize all values between 0 and 1 and we will flatten the 28x28 images into vectors of size 784.

```
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
print (x_train.shape)
print (x_test.shape)
(60000, 784)
(10000, 784)
```

- Now let's train our autoencoder for 50 epochs:

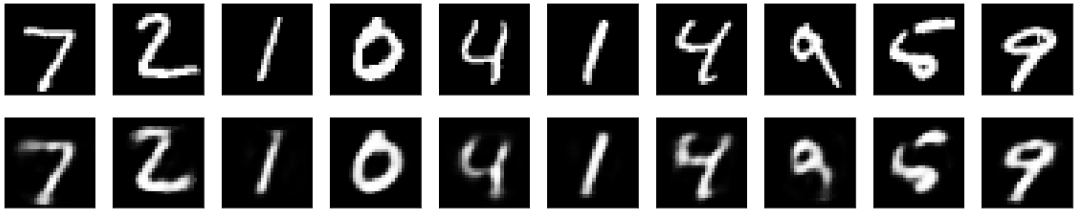
```
autoencoder.fit(x_train, x_train,
                epochs=50,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))
Train on 60000 samples, validate on 10000 samples
Epoch 1/50
60000/60000 [==============================] - 3s 57us/step - loss: 0.3685 -
val_loss: 0.2712
Epoch 2/50
60000/60000 [==============================] - 3s 48us/step - loss: 0.2636 -
val_loss: 0.2524
Epoch 3/50
60000/60000 [==============================] - 3s 48us/step - loss: 0.2421 -
val_loss: 0.2296
```

- After 50 epochs, the autoencoder seems to reach a stable train/test loss value of about 0.11.

Visualize Reconstructed Digits

- We can try to visualize the reconstructed inputs and the encoded representations.

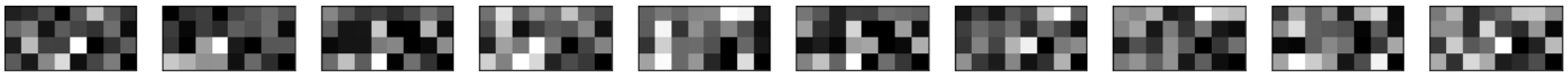
```
# encode and decode some digits from the *test* set
encoded_imgs = encoder.predict(x_test)
decoded_imgs = decoder.predict(encoded_imgs)
# use Matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
n = 10  # how many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
    # display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28,28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    # display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



Visualize `Codings`

- We are curious what the codings look like. That is why we created encoder model and asked it to predict `encoded_imgs`.

```
import matplotlib.pyplot as plt
%matplotlib inline
n = 10  # how many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):

    # display codings as 4x8 array
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(encoded_imgs[i].reshape(4, 8))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



- Hard to interpret. That is what it is. Note that you can easily extract these `codings`, if you define encoder as a separate model.

Nature of Machine Learning
The Manifold Hypothesis

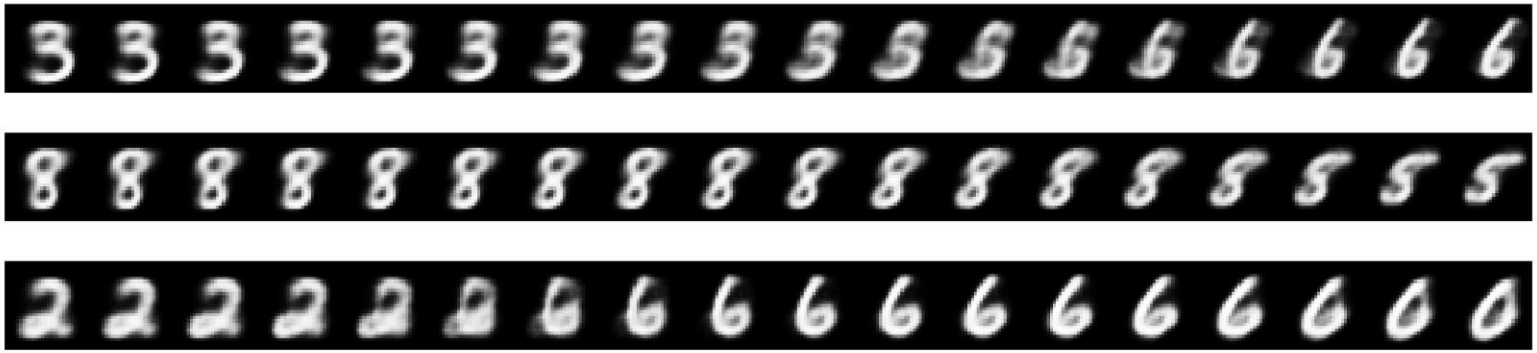
The Manifold Hypothesis

- This material is presented in Chapter 5, Chollet's Book, 2nd Edition and
- The article: "Neural Networks, Manifolds, and Topology"

  https://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Space of events, Continuous Transformations

- So far, in couple of examples, we looked at MNIST images, each an 28x28 array of integers between 0 and 255. The total number of possible images of this size is 256 to the power of 784. The number is greater than the number of atoms in the universe.
- Very few points in the space of dimension 256**(28x28) look like valid MNIST samples.
- Handwritten digits only occupy a tiny subspace of the parent space of all possible 28x28 int8 arrays. This subspace is not just a set of points sprinkled at random in the parent space: it is highly structured.
- The subspace of valid handwritten digits is continuous. If you take a sample and modify it a little, it is still recognizable as the same handwritten digit. All samples in the valid subspace are connected by smooth paths that run through the subspace.
- This means that if you take two random MNIST digits A and B, there exists a sequence of "intermediate" images that morph A into B, such that two consecutive digits are very close to each other. Three examples of possible continuous transformations from 3 to 6, 8 to 5 and 2 to 0 are presented below.



The subspace of Handwritten Digits is a Manifold

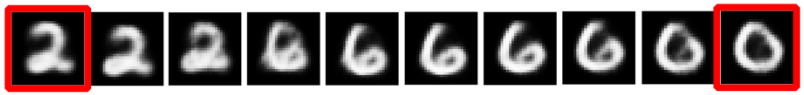
- In technical terms, we say that handwritten digits form a ***manifold in*** the space of possible 28x28 int8 arrays.
- A "manifold“ is a lower-dimensional subspace of some parent space, that is locally similar to a linear (Euclidian) space. For instance, a smooth curve in the plane is a 1D manifold within a 2D space. A smooth surface in a 3D space is a 2D manifold.
- *We assume that all natural data lies on a low-dimensional manifold within the high-dimensional space where it is encoded.*
- That assumption is called “the Manifold Hypothesis”. It appears to be true for MNIST digits, but it could also be demonstrated to be true for human faces, tree morphologies, the sounds of the human voice, and even natural languages.

The Manifold Hypothesis

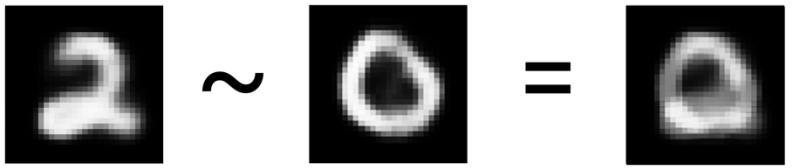
- The manifold hypothesis implies that:
  - Machine learning models only fit relatively simple, low-dimensional, highly-structured subspaces (latent manifold) within their potential input space
  - Within one of these manifolds, it is always possible to interpolate between two inputs, that is to say, morph one into another via a continuous path along which all points fall on the manifold.
- The ability to interpolate between samples is the key to understanding generalization in deep learning.

Interpolation as a Source of Generalization

- If you work with data points that can be interpolated, you can start making sense of points you've never seen before, by relating them to other points that lie close on the manifold. In other words, you can make sense of the totality of the space using only a sample of the space. You can use interpolation to fill in the blanks.
- Manifold interpolation is different from the linear interpolation



Manifold interpolation
(intermediate point
on the latent manifold)



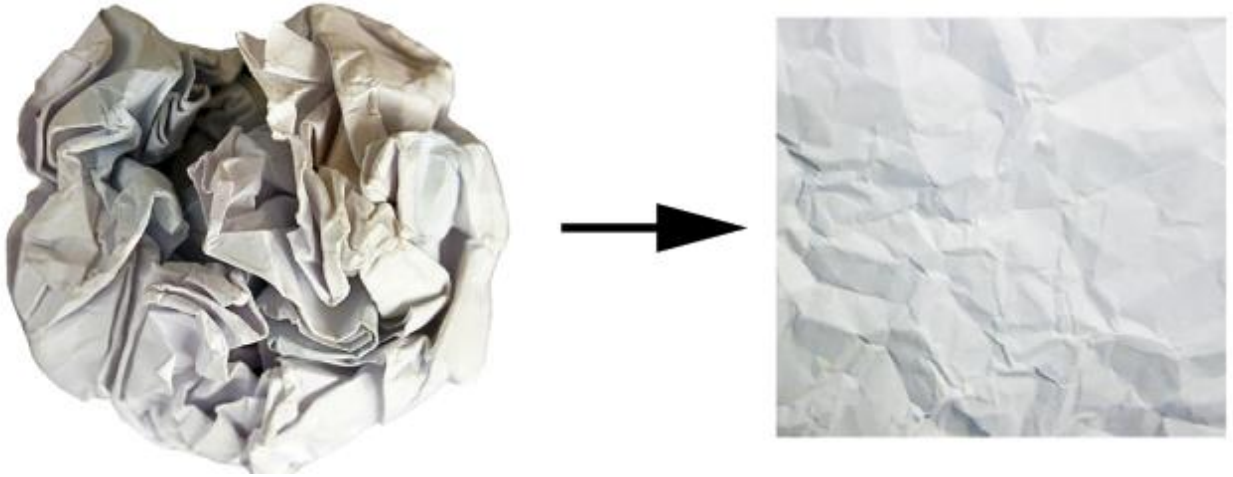
Linear interpolation
(average in the encoding space)

Generalization is Interpolation on Manifolds

- Deep learning achieves generalization via interpolation on a learned approximation of the data manifold.
- The interpolation is not identical to the generalization.
- Interpolation can only help you make sense of things that are very close to what you've seen before: it enables local generalization.
- Humans deal with extreme novelty all the time. Humans are capable of extreme generalization, which is enabled by cognitive mechanisms other than interpolation—abstraction, symbolic models of the world, reasoning, logic, common sense, innate priors about the world. What we generally call reason, as opposed to intuition and pattern recognition. The latter are largely interpolative in nature, but the former isn't. Both are essential to intelligence.

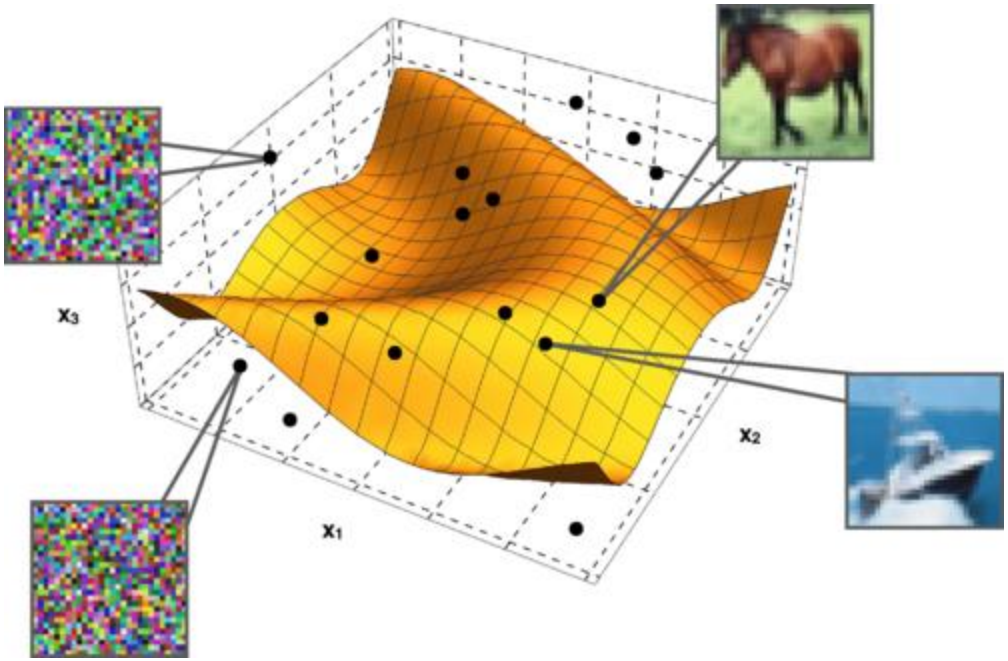
Crumped Paper Metaphor

- A sheet of paper represents a 2D manifold within 3D space. A deep learning model is a tool for uncrumpling paper balls, that is, for disentangling latent manifolds.



- A deep-learning model is basically a very high-dimensional curve or surface.
- That curve is smooth and continuous (with additional constraints on its structure, originating from model architecture priors), since it needs to be differentiable.
- That curve is fitted to data points via gradient descent—smoothly and incrementally.
- Deep learning is by construction about taking a big, complex curve—a manifold—and incrementally adjusting its parameters until it fits some training data points.

Low dimensionality of manifold of CIFAR10 images



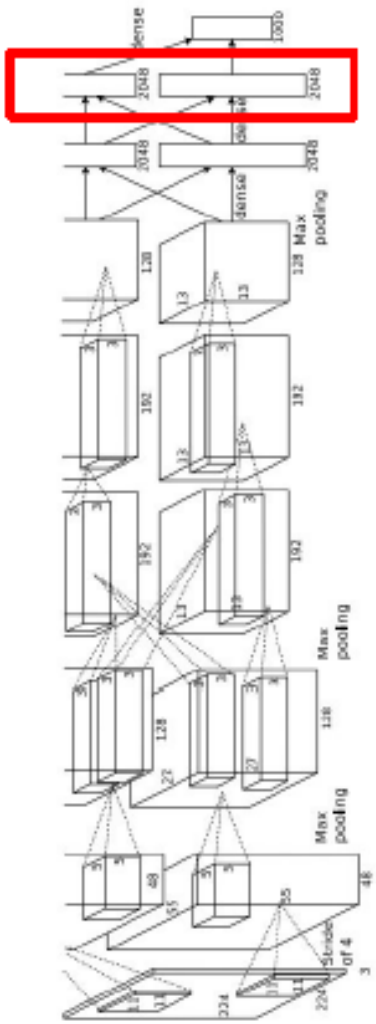
- Each black point indicates a possible mapping from in a high-dimensional input space R^N ~ 3*256^(32*32) = 3x2 ^{8192} ~ 3x 10 ^{2400}
- Most points in this space cannot be interpreted as images at all; however, those points that can be interpreted as real images tend to concentrate on a lower-dimensional manifold, here sketched as a two-dimensional curved surface in a three-dimensional space.
- The intrinsic dimension D of these lower-dimensional manifolds can be measured numerically. Image and text from https://journals.aps.org/prx/abstract/10.1103/PhysRevX.10.041044#

Embedding the codes with t-SNE

- CNNs can be interpreted as gradually transforming the images into a representation in which the classes are separable by a linear classifier.
- We can get a rough idea about the topology of this space by embedding images into two dimensions so that their low-dimensional representation has approximately similar distances as their high-dimensional representation.
- There are many embedding methods that have been developed with the intention of projecting high-dimensional vectors to a low-dimensional space while preserving the pairwise distances of the points.
- Among those, t-SNE is one of the best-known methods that consistently produces visually-pleasing results.
- **t-distributed stochastic neighbor embedding (t-SNE)** is a machine learning algorithm for dimensionality reduction developed by Geoffrey Hinton and Laurens van der Maaten. *van der Maaten, L.J.P.; Hinton, G.E. (Nov 2008). "Visualizing Data Using t-SNE"(PDF). Journal of Machine Learning Research. **9**: 2579–2605.*
- t-SNE is a nonlinear dimensionality reduction technique that is particularly well-suited for (embedding) projecting high-dimensional data into a space of two or three dimensions, which could then be visualized in a scatter plot.
- Specifically, t-SNE represent each high-dimensional point (object) by a two- or three-dimensional point in such a way that similar objects are represented by nearby points and dissimilar objects are represented by distant points and distances between similar points are roughly maintained.

t-SNE on Alex Layer

- The last layer in AlexNet produces 4096 dimensional vector.
- t-SNE use distances between those vectors as the measure of “closeness”.
- To produce an embedding, we can take a set of images and use AlexNet (or some other CNN) to extract the 4096-dimensional vector right before the classifier.
- We can then plug these 4096-dimensional vectors into t-SNE and get 2-dimensional vector for **e**ach image. To obtain the image below, on top of every such 2-d t-SNE vector we place the corresponding image



Continuity over the Manifold

- You see that similar images create clusters. Sailboats, watches, airplanes, houses, etc.
- There is a continuum of image transformations as we slide along low dimensional manifold. Even cars are close to trucks. Boats are closer to trucks than horses. Moose is closer to horses than trucks, etc.

Appendix: Kullback-Leibler Divergence

Kullback-Leibler Divergence

- In mathematical statistics, the *Kullback–Leibler divergence* (also called **relative entropy**) is a measure of how one probability distribution, q, is different from a second, reference probability distribution, p.
- In the simple case, a Kullback–Leibler divergence of 0 indicates that the two distributions are identical. In simplified terms, KL divergence is a measure of surprise.
- We can rewrite the formula from the previous slides as:

$$D_{KL}(p||q) = p\log(p/q) + (1-p)\log((1-p)/(1-q))$$
$$D_{KL}(p||q) = p\log(p) - p\log(q) + (1-p)\log(1-p) - (1-p)\log(1-q)$$
$$= p\log(p) + (1-p)\log(1-p) - p\log(q) - (1-p)\log(1-q)$$

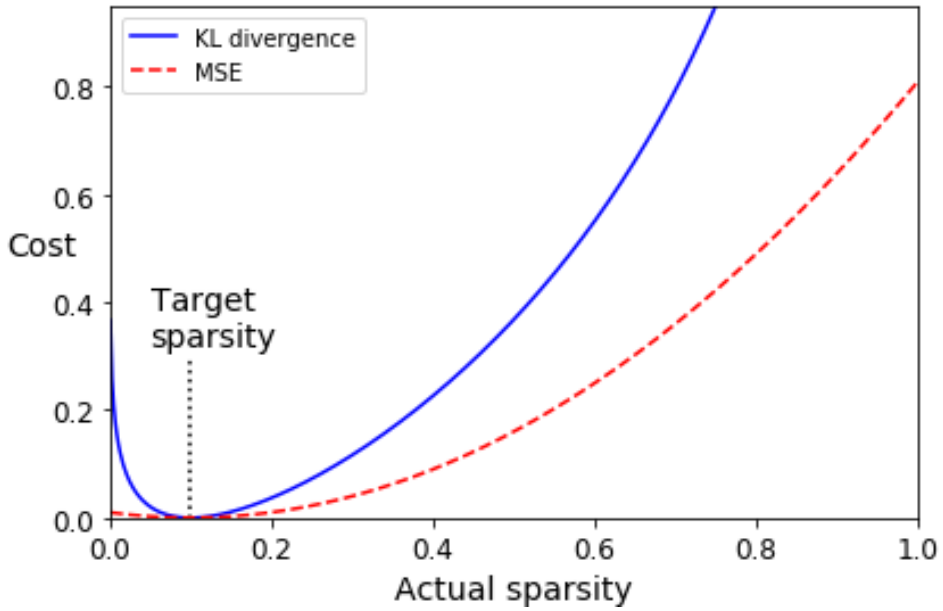
- And see that Kullback–Leibler divergence is actually equal to the difference between the cross entropy and the entropy:

$$D_{KL}(p||q) = -H(p) + H(p,q)$$

- Entropy measures the average length of information you get from a sample drawn from a given probability distribution $(p, 1-p)$.
- Cross entropy is a measure of the length of information when the actual probability q is different from the desired probability p.

KL Divergence vs. Mean Square Error, Python plot

```
p = 0.1
q = np.linspace(0.001, 0.999, 500)
kl_div = p * np.log(p / q) + (1 - p) * np.log((1 - p) / (1 - q))
mse = (p - q)**2
plt.plot([p, p], [0, 0.3], "k:")
plt.text(0.05, 0.32, "Target\nsparsity", fontsize=14)
plt.plot(q, kl_div, "b-", label="KL divergence")
plt.plot(q, mse, "r--", label="MSE")
plt.legend(loc="upper left ")
plt.xlabel("Actual sparsity")
plt.ylabel("Cost", rotation=0)
plt.axis([0, 1, 0, 0.95])
save_fig("sparsity_loss_plot")
```



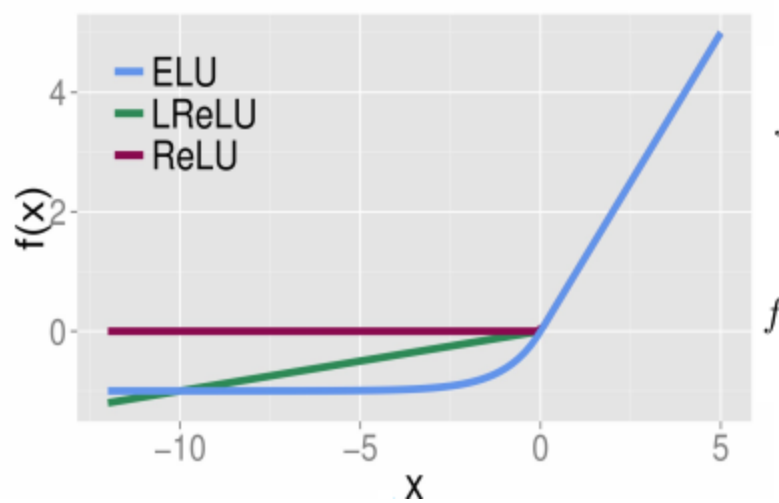
Functions to plot activation histograms

```
def plot_percent_hist(ax, data, bins):
    counts, _ = np.histogram(data, bins=bins)
    widths = bins[1:] - bins[:-1]
    x = bins[:-1] + widths / 2
    ax.bar(x, counts / len(data), width=widths*0.8)
    ax.xaxis.set_ticks(bins)
    ax.yaxis.set_major_formatter(mpl.ticker.FuncFormatter(
        lambda y, position: "{}%".format(int(np.round(100 * y)))))
    ax.grid(True)

def plot_activations_histogram(encoder, height=1, n_bins=10):
    X_valid_codings = encoder(X_valid).numpy()
    activation_means = X_valid_codings.mean(axis=0)
    mean = activation_means.mean()
    bins = np.linspace(0, 1, n_bins + 1)

    fig, [ax1, ax2] = plt.subplots(figsize=(10, 3), nrows=1, ncols=2, sharey=True)
    plot_percent_hist(ax1, X_valid_codings.ravel(), bins)
    ax1.plot([mean, mean], [0, height], "k--", label="Overall Mean =
{:.2f}".format(mean))
    ax1.legend(loc="upper center", fontsize=14)
    ax1.set_xlabel("Activation")
    ax1.set_ylabel("% Activations")
    ax1.axis([0, 1, 0, height])
    plot_percent_hist(ax2, activation_means, bins)
    ax2.plot([mean, mean], [0, height], "k--")
    ax2.set_xlabel("Neuron Mean Activation")
    ax2.set_ylabel("% Neurons")
    ax2.axis([0, 1, 0, height])
```

Appendix
ELU, ReLU, SeLU, He Initiation

ELU

- In autoencoders, we often use Exponential Linear Unit (ELU).
- ELU is defined as a linear function for $x > 0$ and an exponential, asymptotic approach to -α for $x < 0$.
- In the diagram below, we see LReLU (Leaky Rectifying Linear Unit, ReLU and ELU.
- Function $f(x)$ and its derivative $f'(x)$ below describe ELU.



$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha\ (\exp(x) - 1) & \text{if } x < 0 \end{cases}$$

$$f'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ f(x) + \alpha & \text{if } x < 0 \end{cases}$$

- It was shown that use of ELU speeds up the learning process in many image processing deep learning network. Please see:
- "*ELU-Networks: Fast and Accurate CNN Learning on ImageNet"* by Martin Heusel et al. http://image-net.org/challenges/posters/JKU_EN_RGB_Schwarz_poster.pdf

Need for ELU & ReLU Activation Functions

- The need for ELU is due to the problem of vanishing gradients.
- Since we are using Gradient Descent to train neural networks, any zero gradients along a parameter will stop training improvements in that parameter.
- If you get large values out of the neuron then the gradient of `tanh()` function will be near zero and this causes training to get stalled.
- The ReLU function also has similar problems if the value ever goes negative then the gradient is zero and training will stall and get stuck there.
- One solution to this issue is to use the ELU function or a "leaky" ReLU function.
- Leaky ReLU has a low sloped linear function for negative values.
- ELU uses an exponential type function which flattens out to the left of zero, so if values go negative, derivatives and backpropagation can still recover.
- Although, even with ELU, if values become very negative, gradient descent will get stuck again.
- ELU has an advantage that its is fully differentiable even at 0.
- Both ELU and LReLU activation functions demonstrated very good results in training very deep neural networks which would otherwise get stuck in trainings with tanh(), $\sigma()$ or ReLU.

SELU

- SELU, Scaled Exponential Unit, is like ELU but with one additional parameter

$$\text{selu}(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha e^x - \alpha & \text{if } x \leqslant 0 \end{cases}$$

- $\alpha = 1.6733 \; and \; \lambda = 1.0507$

- $\alpha \; and \; \lambda$ are not trainable parameters.
- SELU has self-normalizing properties. Activations that have zero mean and unit variance, propagated through many (`Dense`) network layers, will converge towards zero mean and unit variance.
- This makes the learning highly robust and allows training networks that have many layers.
- Klambauer, Günter, Unterthiner, Thomas, Mayr, Andreas, and Hochreiter, Sepp. *Self-normalizing neural networks*. https://arxiv.org/abs/1706.02515

He_Initializer, Xavier_Initializer

- Some time ago we used random values to set the initial weights away from zero.
- Modern deep networks use so called *He's Initialization or Xavier Initialization (He is the last name of Kaiming He, and not a male version of pronoun She)*
- *"***Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification"** Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun https://arxiv.org/abs/1502.01852v1
- *"***Understanding the difficulty of training deep feedforward neural networks***" by Xavier Glorot and Yoshua Bengio*. http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf
- Keras has several Initializers in the package: `tf.keras.initializers.`

GlorotNormal

- GlorotNormal class
- `tf.keras.initializers.GlorotNormal(seed=None)`
- The Glorot normal initializer, also called Xavier normal initializer.
- Also available via the shortcut function `tf.keras.initializers.glorot_normal`.
- Draws samples from a truncated normal distribution centered on 0 with `stddev = sqrt(2 / (fan_in + fan_out))` where `fan_in` is the number of input units in the weight tensor and `fan_out` is the number of output units in the weight tensor.
- Examples

```
# Standalone usage:
initializer = tf.keras.initializers.GlorotNormal()
values = initializer(shape=(2, 2))
# Usage in a Keras layer:
initializer = tf.keras.initializers.GlorotNormal()
layer = tf.keras.layers.Dense(3, kernel_initializer=initializer)
```

- Arguments
- `seed`: A Python integer. Used to make the behavior of the initializer deterministic. Note that a seeded initializer will not produce the same random values across multiple calls, but multiple initializers will produce the same sequence when constructed with the same seed value

HeNormal class

- `HeNormal` class
- `tf.keras.initializers.HeNormal(seed=None)`
- He normal initializer is also available via the shortcut function `tf.keras.initializers.he_normal`.

- It draws samples from a truncated normal distribution centered on 0 with `stddev = sqrt(2 / fan_in)` where `fan_in` is the number of input units in the weight tensor.

- Examples

```
# Standalone usage:
initializer = tf.keras.initializers.HeNormal()
values = initializer(shape=(2, 2))
# Usage in a Keras layer:
initializer = tf.keras.initializers.HeNormal()
layer = tf.keras.layers.Dense(3, kernel_initializer=initializer)
```

- Arguments
- `seed`: A Python integer. Used to make the behavior of the initializer deterministic. Note that a seeded initializer will not produce the same random values across multiple calls, but multiple initializers will produce the same sequence when constructed with the same seed value.