**What is an Autoencoder?**

An autoencoder is a type of artificial [neural network](https://deepai.org/machine-learning-glossary-and-terms/neural-network) used to learn efficient codings of unlabeled data, typically for the purpose of dimensionality reduction. An autoencoder employs [unsupervised learning](https://deepai.org/machine-learning-glossary-and-terms/unsupervised-learning) to learn a representation (encoding) for a set of data, typically for the purpose of reducing the dimensionality of the data. The network is trained to compress the input into a lower-dimensional code and then reconstruct the output from this representation to match the original input as closely as possible, hence the name 'autoencoder'.

**Autoencoder Architecture**

The architecture of an autoencoder is designed to be symmetrical with a bottleneck in the middle. The network consists of two main parts:

* **Encoder:** This is the part of the network that compresses the input into a latent-space representation. It encodes the input data as a compressed representation in a reduced dimension. The encoder layer maps the input data to the hidden layer.
* **Decoder:** This part aims to reconstruct the input data from the latent space representation. The decoder layer maps the hidden layer to the reconstruction of the input data.

The bottleneck, which is the layer that contains the code, represents the compressed knowledge of the input data. The key idea is that the autoencoder learns to ignore the noise and capture the most salient features of the data.

**Training an Autoencoder**

Autoencoders are trained to minimize reconstruction errors, often using a [loss function](https://deepai.org/machine-learning-glossary-and-terms/loss-function) such as mean squared error (MSE) between the input and its reconstruction. During training, the weights of the neural network are adjusted to minimize the loss function, and the process is typically performed using [backpropagation](https://deepai.org/machine-learning-glossary-and-terms/backpropagation).

Training involves presenting the network with unlabeled input data, passing it through the encoder to get the encoded representation, then passing the encoded representation through the decoder to get the reconstructed input, and finally comparing the reconstructed input to the original input.

**Variants of Autoencoders**

There are several types of autoencoders, each with its own specific use cases:

* [**Denoising Autoencoder**](https://deepai.org/machine-learning-glossary-and-terms/denoising-autoencoder)**:** This variant is trained to use a corrupted version of the input data and learn to recover the original, uncorrupted data.
* **Sparse Autoencoder:** This type adds a sparsity constraint on the hidden units during training, forcing the model to respond to the unique statistical features of the input data.
* **Variational Autoencoder (VAE):**

Unlike a traditional autoencoder, a VAE learns to generate new data that could have been part of the original dataset. It is a generative model that defines a [probability distribution](https://deepai.org/machine-learning-glossary-and-terms/probability-distribution) for the input data.

* **Convolutional Autoencoder:** Designed for input data that has a grid-like topology, such as images, this variant uses convolutional layers instead of fully connected layers in the encoder and decoder networks.

**Applications of Autoencoders**

Autoencoders have a wide range of applications, including:

* **Dimensionality Reduction:**

Similar to PCA, autoencoders can be used for reducing the dimensionality of data for visualization or preprocessing before applying other [machine learning](https://deepai.org/machine-learning-glossary-and-terms/machine-learning) techniques.

* **Feature Learning:** Autoencoders can learn useful features automatically from the input data, which can then be used for tasks such as classification or prediction.
* [**Anomaly Detection**](https://deepai.org/machine-learning-glossary-and-terms/anomaly-detection)**:** By learning to reproduce the normal input data, autoencoders can be used to detect anomalies by measuring the reconstruction error of new data.
* **Data Denoising:** Denoising autoencoders can be used to remove noise from data, which is particularly useful in image processing.
* **Generative Models:** Variational autoencoders can generate new data instances that are similar to the input data, which can be used for tasks such as data augmentation.

**Challenges with Autoencoders**

Despite their usefulness, autoencoders do have limitations and challenges. They may sometimes learn trivial solutions, such as the identity function, which do not provide any useful representation of the data. Additionally, the choice of the size of the hidden layer, the regularization method, and the loss function can significantly affect the performance and the type of features learned by the autoencoder.

Furthermore, autoencoders are unsupervised learning models, and as such, they do not use any label information. This means that the features learned may not be optimal for any supervised task that the encoded representation is later used for.

**Conclusion**

Autoencoders are powerful neural network models for unsupervised learning of data codings. They are useful for a variety of tasks, including dimensionality reduction, feature learning, and anomaly detection. While they are not without challenges, autoencoders remain an important tool in the machine learning practitioner's toolkit, especially for tasks involving complex data structures such as images and text.

**Introduction to Autoencoders: From The Basics to Advanced Applications in PyTorch**

A walkthrough of Autoencoders, their variations, and potential applications in the real world.

Dec 2023 · 10 min read

Traditional feedforward [**neural networks**](https://www.datacamp.com/blog/what-are-neural-networks) can be great at performing tasks such as classification and regression, but what if we would like to implement solutions such as signal denoising or anomaly detection? One way to do this is by using Autoencoders.

This tutorial provides a practical introduction to Autoencoders, including a hands-on example in PyTorch and some potential use cases.

You can follow along in the [**Datacamp Code Workspace with code from the tutorial**](https://app.datacamp.com/workspace/w/838aa776-2259-464f-a673-399857850b8a/edit).

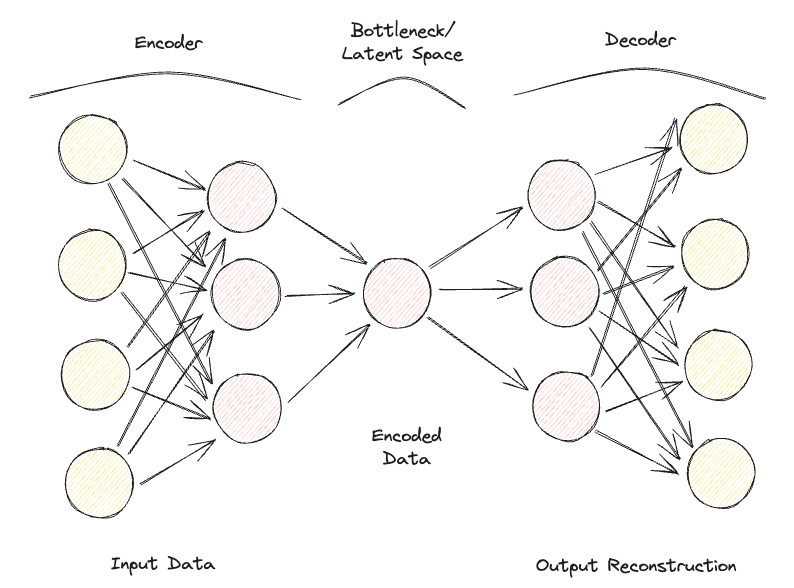
**What are Autoencoders?**

Autoencoders are a special type of unsupervised feedforward neural network (no labels needed!). The main application of Autoencoders is to accurately capture the key aspects of the provided data to provide a compressed version of the input data, generate realistic synthetic data, or flag anomalies.

Autoencoders are composed of 2 key fully connected feedforward neural networks (Figure 1):

* **Encoder**: compresses the input data to remove any form of noise and generates a latent space/bottleneck. Therefore, the output neural network dimensions are smaller than the input and can be adjusted as a hyperparameter in order to decide how much lossy our compression should be.
* **Decoder**: making use of only the compressed data representation from the latent space, tries to reconstruct with as much fidelity as possible the original input data (the architecture of this neural network is, therefore, generally a mirror image of the encoder). The “goodness” of the prediction can then be measured by calculating the reconstruction error between the input and output data using a loss function.

Repeating iteratively this process of passing data through the encoder and decoder and measuring the error to tune the parameters through backpropagation, the Autoencoder can, with time, correctly work with extremely difficult forms of data.



*Figure 1: Autoencoder Architecture (Image by Author).*

If an Autoencoder is provided with a set of input features completely independent of each other, then it would be really difficult for the model to find a good lower-dimensional representation without losing a great deal of information (lossy compression).

Autoencoders can, therefore, also be considered a dimensionality reduction technique, which compared to traditional techniques such as [**Principal Component Analysis**](https://www.datacamp.com/tutorial/principal-component-analysis-in-python) (PCA), can make use of non-linear transformations to project data in a lower dimensional space. If you are interested in learning more about other Feature Extraction techniques, additional information is available in [**this feature extraction tutorial.**](https://ppiconsulting.dev/blog/blog29/).

Additionally, compared to standard data compression algorithms like [**gzpi**](https://10web.io/site-speed-glossary/gzip-compression/), Autoencoders can not be used as general-purpose compression algorithms but are handcrafted to work best just on similar data on which they have been trained on.

Some of the most common hyperparameters that can be tuned when optimizing your Autoencoder are:

* The number of layers for the Encoder and Decoder neural networks
* The number of nodes for each of these layers
* The loss function to use for the optimization process (e.g., binary cross-entropy or mean squared error)
* The size of the latent space (the smaller, the higher the compression, acting, therefore as a regularization mechanism)

Finally, Autoencoders can be designed to work with different types of data, such as tabular, time-series, or image data, and can, therefore, be designed to use a variety of layers, such as [**convolutional layers**](https://www.datacamp.com/tutorial/convolutional-neural-networks-python), for image analysis.

Ideally, a well-trained Autoencoder should be responsive enough to adapt to the input data in order to provide a tailor-made response but not so much as to just mimic the input data and not be able to generalize with unseen data (therefore overfitting).

**Types of Autoencoders**

Over the years, different types of Autoencoders have been developed:

* Undercomplete Autoencoder
* Sparse Autoencoder
* Contractive Autoencoder
* Denoising Autoencoder
* Convolutional Autoencoder
* Variational Autoencoder

Let’s explore each in more detail.

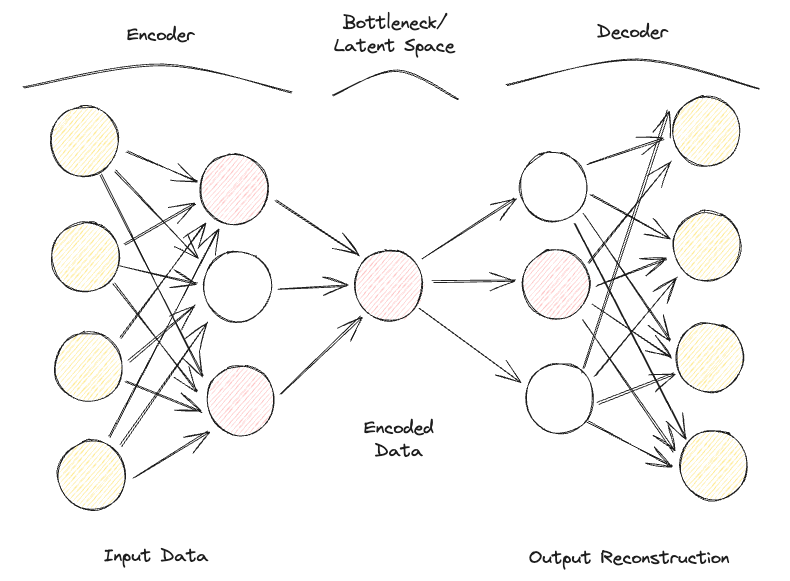
**Undercomplete Autoencoder**

This is the simplest version of an autoencoder. In this case, we don’t have an explicit regularization mechanism, but we ensure that the size of the bottleneck is always lower than the original input size to avoid overfitting. This type of configuration is typically used as a dimensionality reduction technique (more powerful than PCA since its also able to capture non-linearities in the data).

**Sparse Autoencoder**

A Sparse Autoencoder is quite similar to an Undercomplete Autoencoder, but their main difference lies in how regularization is applied. In fact, with Sparse Autoencoders, we don’t necessarily have to reduce the dimensions of the bottleneck, but we use a loss function that tries to penalize the model from using all its neurons in the different hidden layers (Figure 2).

This penalty is commonly referred to as a **sparsity function**, and it's quite different from traditional regularization techniques since it doesn’t focus on penalizing the size of the weights but the number of nodes activated.



In this way, different nodes could specialize for different input types and be activated/deactivated depending on the specifics of the input data. This sparsity constraint can be induced by using L1 Regularization and KL divergence, effectively preventing the model from overfitting.

**Contractive Autoencoder**

The main idea behind Contractive Autoencoders is that given some similar inputs, their compressed representation should be quite similar (neighborhoods of inputs should be contracted in small neighborhood of outputs). In mathematical terms, this can be enforced by keeping input hidden layer activations derivatives small when fed similar inputs.

**Denoising Autoencoder**

With Denoising Autoencoders, the input and output of the model are no longer the same. For example, the model could be fed some low-resolution corrupted images and work for the output to improve the quality of the images. In order to assess the performance of the model and improve it over time, we would then need to have some form of labeled clean image to compare with the model prediction.

**Convolutional Autoencoder**

To work with image data, Convolutional Autoencoders replace traditional feedforward neural networks with [**Convolutional Neural Networks**](https://www.datacamp.com/tutorial/cnn-tensorflow-python) for both the encoder and decoder steps. Updating type of loss function, etc., this type of Autoencoder can also be made, for example, Sparse or Denoising, depending on your use case requirements.

**Variational Autoencoder**

In every type of Autoencoder considered so far, the encoder outputs a single value for each dimension involved. With Variational Autoencoders (VAE), we make this process instead probabilistic, creating a probability distribution for each dimension. The decoder can then sample a value from each distribution describing the different dimensions and construct the input vector, which it can then be used to reconstruct the original input data.

One of the main applications of Variational Autoencoders is for generative tasks. In fact, sampling the latent model from distributions can enable the decoder to create new forms of outputs that were previously not possible using a deterministic approach.

If you are interested in testing an online a Variational Autoencoder trained on the MNIST dataset, you can find [**a live example**](https://ppiconsulting.dev/Projects/ONNX/home.html).

**Autoencoder Hands-On Example**

We are now ready to go through a practical demonstration of how Autoencoders can be used for dimensionality reduction. Our Deep Learning framework of choice for this exercise is going to be PyTorch.

The[**Kaggle Rain in Australia dataset**](https://www.kaggle.com/datasets/jsphyg/weather-dataset-rattle-package) is going to be used for this demonstration. All the code used as part of this article is available at [**this DataCamp Workspace**](https://app.datacamp.com/workspace/w/838aa776-2259-464f-a673-399857850b8a/edit).

First of all, we import all the necessary libraries and remove any missing values and non-numeric columns (Figure 3).

import numpy as np

import pandas as pd

import torch

import torch.nn as nn

import torch.optim as optim

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler, LabelEncoder

from mpl\_toolkits.mplot3d import Axes3D

df = pd.read\_csv("../input/weather-dataset-rattle-package/weatherAUS.csv")

df = df.drop(['Date', 'Location', 'WindDir9am',

'WindGustDir', 'WindDir3pm'], axis=1)

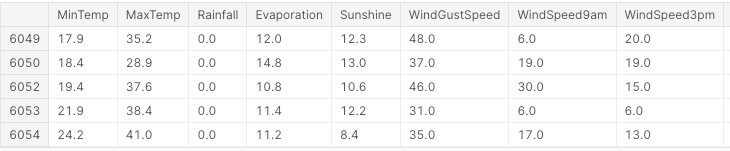
df = df.dropna(how = 'any')

df.loc[df['RainToday'] == 'No', 'RainToday'] = 0

df.loc[df['RainToday'] == 'Yes', 'RainToday'] = 1

df.head()

OpenAI



*Figure 3: Sample of Dataset Columns (Image by Author).*

At this point, we are ready to divide the data into features and labels, normalize the features, and convert the labels into a numeric format.

In this case, we have a starting feature set composed of 17 columns. The overall objective of the analysis would then be to correctly predict if it's going to rain the next day or not.

X, Y = df.drop('RainTomorrow', axis=1, inplace=False), df[['RainTomorrow']]

# Normalizing Data

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

# Converting to PyTorch tensor

X\_tensor = torch.FloatTensor(X\_scaled)

# Converting string labels to numerical labels

label\_encoder = LabelEncoder()

Y\_numerical = label\_encoder.fit\_transform(Y.values.ravel())

OpenAI

In PyTorch, we can now define the Autoencoder model as a class and specify the encoder and decoder models with two linear layers.

# Defining Autoencoder model

class Autoencoder(nn.Module):

def \_\_init\_\_(self, input\_size, encoding\_dim):

super(Autoencoder, self).\_\_init\_\_()

self.encoder = nn.Sequential(

nn.Linear(input\_size, 16),

nn.ReLU(),

nn.Linear(16, encoding\_dim),

nn.ReLU()

)

self.decoder = nn.Sequential(

nn.Linear(encoding\_dim, 16),

nn.ReLU(),

nn.Linear(16, input\_size),

nn.Sigmoid()

)

def forward(self, x):

x = self.encoder(x)

x = self.decoder(x)

return x

OpenAI

Now that the model is set up, we can proceed to specify our encoding dimensions to be equal to 3 (to make it easy to plot them later on) and run through the training process.

# Setting random seed for reproducibility

torch.manual\_seed(42)

input\_size = X.shape[1] # Number of input features

encoding\_dim = 3 # Desired number of output dimensions

model = Autoencoder(input\_size, encoding\_dim)

# Loss function and optimizer

criterion = nn.MSELoss()

optimizer = optim.Adam(model.parameters(), lr=0.003)

# Training the autoencoder

num\_epochs = 20

for epoch in range(num\_epochs):

# Forward pass

outputs = model(X\_tensor)

loss = criterion(outputs, X\_tensor)

# Backward pass and optimization

optimizer.zero\_grad()

loss.backward()

optimizer.step()

# Loss for each epoch

print(f'Epoch [{epoch + 1}/{num\_epochs}], Loss: {loss.item():.4f}')

# Encoding the data using the trained autoencoder

encoded\_data = model.encoder(X\_tensor).detach().numpy()

OpenAI

Epoch [1/20], Loss: 1.2443

Epoch [2/20], Loss: 1.2410

Epoch [3/20], Loss: 1.2376

Epoch [4/20], Loss: 1.2342

Epoch [5/20], Loss: 1.2307

Epoch [6/20], Loss: 1.2271

Epoch [7/20], Loss: 1.2234

Epoch [8/20], Loss: 1.2196

Epoch [9/20], Loss: 1.2156

Epoch [10/20], Loss: 1.2114

Epoch [11/20], Loss: 1.2070

Epoch [12/20], Loss: 1.2023

Epoch [13/20], Loss: 1.1974

Epoch [14/20], Loss: 1.1923

Epoch [15/20], Loss: 1.1868

Epoch [16/20], Loss: 1.1811

Epoch [17/20], Loss: 1.1751

Epoch [18/20], Loss: 1.1688

Epoch [19/20], Loss: 1.1622

Epoch [20/20], Loss: 1.1554

OpenAI

Finally, we can now plot the resulting embedding dimensions (Figure 4). As it can be seen in the image below, we successfully managed to reduce the dimensionality of our feature set from 17 dimensions to just 3, while still being able, to a good extent, to correctly separate in our 3-dimensional space samples between the different classes.

# Plotting the encoded data in 3D space

fig = plt.figure(figsize=(8, 8))

ax = fig.add\_subplot(111, projection='3d')

scatter = ax.scatter(encoded\_data[:, 0], encoded\_data[:, 1],

encoded\_data[:, 2], c=Y\_numerical, cmap='viridis')

# Mapping numerical labels back to original string labels for the legend

labels = label\_encoder.inverse\_transform(np.unique(Y\_numerical))

legend\_labels = {num: label for num, label in zip(np.unique(Y\_numerical),

labels)}

# Creating a custom legend with original string labels

handles = [plt.Line2D([0], [0], marker='o', color='w',

markerfacecolor=scatter.to\_rgba(num),

markersize=10,

label=legend\_labels[num]) for num in np.unique(Y\_numerical)]

ax.legend(handles=handles, title="Raining Tomorrow?")

# Adjusting the layout to provide more space for labels

ax.xaxis.labelpad = 20

ax.yaxis.labelpad = 20

ax.set\_xticks([])

ax.set\_yticks([])

ax.set\_zticks([])

# Manually adding z-axis label for better visibility

ax.text2D(0.05, 0.95, 'Encoded Dimension 3', transform=ax.transAxes,

fontsize=12, color='black')

ax.set\_xlabel('Encoded Dimension 1')

ax.set\_ylabel('Encoded Dimension 2')

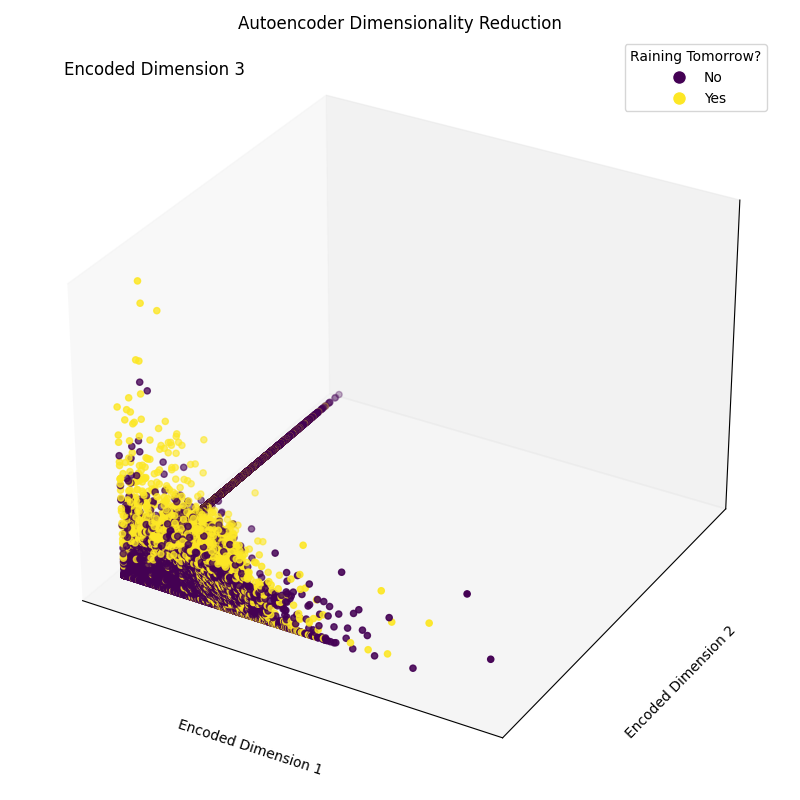
ax.set\_title('Autoencoder Dimensionality Reduction')

plt.tight\_layout()

plt.savefig('Rain\_Prediction\_Autoencoder.png')

plt.show()

OpenAI



*Figure 4: Resulting Encoded Dimensions (Image by Author).*

**Real-life Applications of Autoencoders**

**Image Compression/Denoising**

One of the main applications of Autoencoders is to compress images to reduce their overall file size while trying to keep as much of the valuable information as possible or restore images that have been degraded over time.

**Anomaly Detection**

Since Autoencoders are good at distinguishing essential characteristics of data from noise, they can be used in order to detect anomalies (e.g., if an image has been photoshopped, if there are unusual activities in a network, etc.)

**Data Generation**

Variational Autoencoders and Generative Adversarial Networks (GAN) are frequently used in order to generate synthetic data (e.g., realistic images of people).

**Conclusion**

In conclusion, Autoencoders can be a really flexible tool to power different forms of use cases. In particular, Variational Autoencoders and the creation of GANs opened the door to the development of Generative AI, offering us the first glimpses of how AI can be used to generate new forms of content never seen before.

This tutorial was an introduction to the field of Autoencoders, although there is still a lot to learn! DataCamp has a wide variety of resources about this topic, such as [**how to implement Autoencoders in Keras**](https://www.datacamp.com/tutorial/autoencoder-keras-tutorial) or use them as a [**classifier**](https://www.datacamp.com/tutorial/autoencoder-classifier-python)!

[Building Autoencoders in Keras](https://blog.keras.io/building-autoencoders-in-keras.html)

Sat 14 May 2016

*By*[*Francois Chollet*](https://twitter.com/fchollet)

In [Tutorials](https://blog.keras.io/category/tutorials.html).

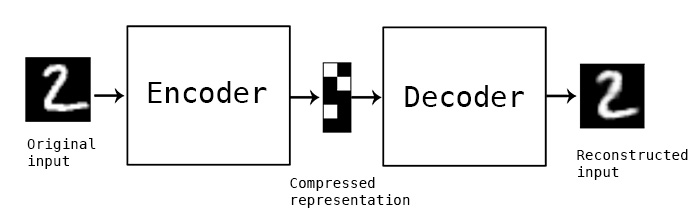
This post was written in early 2016. It is therefore badly outdated.

In this tutorial, we will answer some common questions about autoencoders, and we will cover code examples of the following models:

* a simple autoencoder based on a fully-connected layer
* a sparse autoencoder
* a deep fully-connected autoencoder
* a deep convolutional autoencoder
* an image denoising model
* a sequence-to-sequence autoencoder
* a variational autoencoder

**Note: all code examples have been updated to the Keras 2.0 API on March 14, 2017. You will need Keras version 2.0.0 or higher to run them.**

What are autoencoders?



"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 will only be able to compress data similar to what they have been 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 would do a rather poor job of compressing pictures of trees, because the features it would learn would be 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 learned automatically from data examples, which is a useful property: it means that it is easy to train specialized instances of the algorithm that will perform well on a specific type of input. It doesn't require any new engineering, just appropriate training data.

To build an autoencoder, you need three things: an encoding function, a decoding function, and a distance function between the amount of information loss between the compressed representation of your data and the decompressed representation (i.e. a "loss" function). The encoder and decoder will be chosen to be parametric functions (typically neural networks), and to be differentiable with respect to the distance function, so the parameters of the encoding/decoding functions can be optimize to minimize the reconstruction loss, using Stochastic Gradient Descent. It's simple! And you don't even need to understand any of these words to start using autoencoders in practice.

Are they good at data compression?

Usually, not really. In picture compression for instance, it is pretty difficult to train an autoencoder that does a better job than a basic algorithm like JPEG, and typically the only way it can be achieved is by restricting yourself to a very specific type of picture (e.g. one for which JPEG does not do a good job). The fact that autoencoders are data-specific makes them generally impractical for real-world data compression problems: you can only use them on data that is similar to what they were trained on, and making them more general thus requires *lots* of training data. But future advances might change this, who knows.

What are autoencoders good for?

They are rarely used in practical applications. In 2012 they briefly found an application in greedy layer-wise pretraining for deep convolutional neural networks [1], but this quickly fell out of fashion as we started realizing that better random weight initialization schemes were sufficient for training deep networks from scratch. In 2014, batch normalization [2] started allowing for even deeper networks, and from late 2015 we could train arbitrarily deep networks from scratch using residual learning [3].

Today two interesting practical applications of autoencoders are **data denoising** (which we feature later in this post), and **dimensionality reduction for data visualization**. With appropriate dimensionality and sparsity constraints, autoencoders can learn data projections that are more interesting than PCA or other basic techniques.

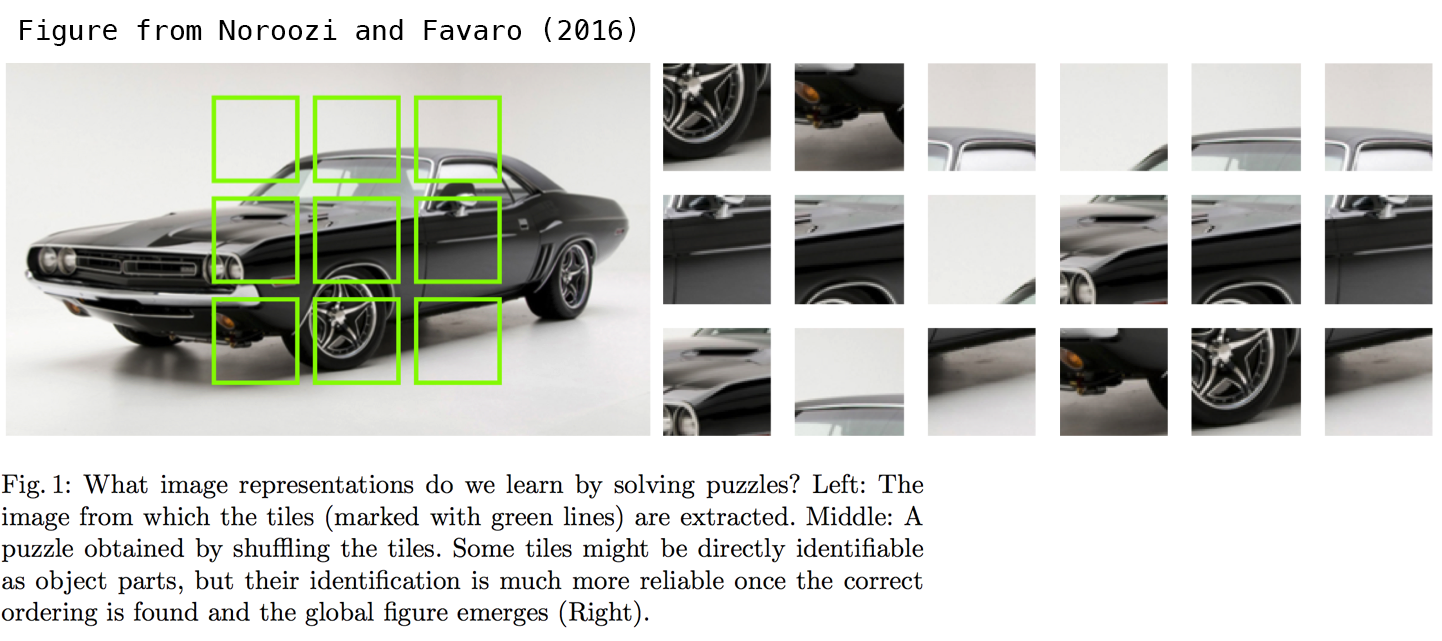
For 2D visualization specifically, [t-SNE](https://en.wikipedia.org/wiki/T-distributed_stochastic_neighbor_embedding) (pronounced "tee-snee") is probably the best algorithm around, but it typically requires relatively low-dimensional data. So a good strategy for visualizing similarity relationships in 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. Note that a nice parametric implementation of t-SNE in Keras was developed by Kyle McDonald and [is available on Github](https://github.com/kylemcdonald/Parametric-t-SNE/blob/master/Parametric%20t-SNE%20(Keras).ipynb). Otherwise [scikit-learn](http://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html) also has a simple and practical implementation.

So what's the big deal with autoencoders?

Their main claim to fame comes from being featured in many introductory machine learning classes available online. As a result, a lot of newcomers to the field absolutely love autoencoders and can't get enough of them. This is the reason why this tutorial exists!

Otherwise, one reason why they 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. the learning of useful representations without the need for labels. Then again, autoencoders are not a true unsupervised learning technique (which would imply a different learning process altogether), they are a *self-supervised* technique, a specific instance of *supervised learning* where the targets are generated from the input data. In order to get self-supervised models to learn interesting features, you have 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 here. At this point there is 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 label-supervized learning induces (where targets are fairly abstract concepts "invented" by humans such as "dog", "car"...). In fact, one may argue that the best features in this regard are those that are the *worst* at exact input reconstruction while achieving high performance on the main task that you are interested in (classification, localization, etc).

In self-supervized learning applied to vision, a potentially fruitful alternative to autoencoder-style input reconstruction is the use of toy tasks such as jigsaw puzzle solving, or detail-context matching (being able to match high-resolution but small patches of pictures with low-resolution versions of the pictures they are extracted from). The following paper investigates jigsaw puzzle solving and makes for a very interesting read: Noroozi and Favaro (2016) [Unsupervised Learning of Visual Representations by Solving Jigsaw Puzzles](http://arxiv.org/abs/1603.09246). Such tasks are providing the model with built-in assumptions about the input data which are missing in traditional autoencoders, such as *"visual macro-structure matters more than pixel-level details"*.



Let's build the simplest possible autoencoder

We'll start simple, with a single fully-connected neural layer as encoder and as decoder:

import keras

from keras import layers

# 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 image

input\_img = keras.Input(shape=(784,))

# "encoded" is the encoded representation of the input

encoded = layers.Dense(encoding\_dim, activation='relu')(input\_img)

# "decoded" is the lossy reconstruction of the input

decoded = layers.Dense(784, activation='sigmoid')(encoded)

# This model maps an input to its reconstruction

autoencoder = keras.Model(input\_img, decoded)

Let's also create a separate encoder model:

# This model maps an input to its encoded representation

encoder = keras.Model(input\_img, encoded)

As well as the decoder model:

# This is our encoded (32-dimensional) input

encoded\_input = keras.Input(shape=(encoding\_dim,))

# Retrieve the last layer of the autoencoder model

decoder\_layer = autoencoder.layers[-1]

# Create the decoder model

decoder = keras.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 Adam optimizer:

autoencoder.compile(optimizer='adam', loss='binary\_crossentropy')

Let's prepare our input data. 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()

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)

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))

After 50 epochs, the autoencoder seems to reach a stable train/validation loss value of about 0.09. We can try to visualize the reconstructed inputs and the encoded representations. We will use Matplotlib.

# Encode and decode some digits

# Note that we take them from the \*test\* set

encoded\_imgs = encoder.predict(x\_test)

decoded\_imgs = decoder.predict(encoded\_imgs)

# Use Matplotlib (don't ask)

import matplotlib.pyplot as plt

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()

Here's what we get. The top row is the original digits, and the bottom row is the reconstructed digits. We are losing quite a bit of detail with this basic approach.



Adding a sparsity constraint on the encoded representations

In the previous example, the representations were only constrained by the size of the hidden layer (32). In such a situation, what typically happens is that the hidden layer is learning an approximation of [PCA (principal component analysis)](https://en.wikipedia.org/wiki/Principal_component_analysis). But another way to constrain the representations to be compact is to add a sparsity contraint on the activity of the hidden representations, so fewer units would "fire" at a given time. In Keras, this can be done by adding an activity\_regularizer to our Dense layer:

from keras import regularizers

encoding\_dim = 32

input\_img = keras.Input(shape=(784,))

# Add a Dense layer with a L1 activity regularizer

encoded = layers.Dense(encoding\_dim, activation='relu',

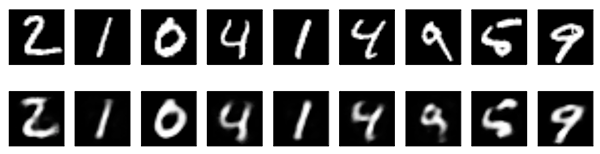
activity\_regularizer=regularizers.l1(10e-5))(input\_img)

decoded = layers.Dense(784, activation='sigmoid')(encoded)

autoencoder = keras.Model(input\_img, decoded)

Let's train this model for 100 epochs (with the added regularization the model is less likely to overfit and can be trained longer). The models ends with a train loss of 0.11 and test loss of 0.10. The difference between the two is mostly due to the regularization term being added to the loss during training (worth about 0.01).

Here's a visualization of our new results:



They look pretty similar to the previous model, the only significant difference being the sparsity of the encoded representations. encoded\_imgs.mean() yields a value 3.33 (over our 10,000 test images), whereas with the previous model the same quantity was 7.30. So our new model yields encoded representations that are twice sparser.

Deep autoencoder

We do not have to limit ourselves to a single layer as encoder or decoder, we could instead use a stack of layers, such as:

input\_img = keras.Input(shape=(784,))

encoded = layers.Dense(128, activation='relu')(input\_img)

encoded = layers.Dense(64, activation='relu')(encoded)

encoded = layers.Dense(32, activation='relu')(encoded)

decoded = layers.Dense(64, activation='relu')(encoded)

decoded = layers.Dense(128, activation='relu')(decoded)

decoded = layers.Dense(784, activation='sigmoid')(decoded)

Let's try this:

autoencoder = keras.Model(input\_img, decoded)

autoencoder.compile(optimizer='adam', loss='binary\_crossentropy')

autoencoder.fit(x\_train, x\_train,

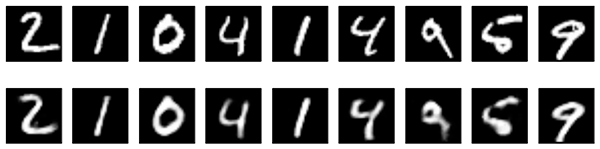
epochs=100,

batch\_size=256,

shuffle=True,

validation\_data=(x\_test, x\_test))

After 100 epochs, it reaches a train and validation loss of ~0.08, a bit better than our previous models. Our reconstructed digits look a bit better too:



Convolutional autoencoder

Since our inputs are images, it makes sense to use convolutional neural networks (convnets) as encoders and decoders. In practical settings, autoencoders applied to images are always convolutional autoencoders --they simply perform much better.

Let's implement one. The encoder will consist in a stack of Conv2D and MaxPooling2D layers (max pooling being used for spatial down-sampling), while the decoder will consist in a stack of Conv2D and UpSampling2D layers.

import keras

from keras import layers

input\_img = keras.Input(shape=(28, 28, 1))

x = layers.Conv2D(16, (3, 3), activation='relu', padding='same')(input\_img)

x = layers.MaxPooling2D((2, 2), padding='same')(x)

x = layers.Conv2D(8, (3, 3), activation='relu', padding='same')(x)

x = layers.MaxPooling2D((2, 2), padding='same')(x)

x = layers.Conv2D(8, (3, 3), activation='relu', padding='same')(x)

encoded = layers.MaxPooling2D((2, 2), padding='same')(x)

# at this point the representation is (4, 4, 8) i.e. 128-dimensional

x = layers.Conv2D(8, (3, 3), activation='relu', padding='same')(encoded)

x = layers.UpSampling2D((2, 2))(x)

x = layers.Conv2D(8, (3, 3), activation='relu', padding='same')(x)

x = layers.UpSampling2D((2, 2))(x)

x = layers.Conv2D(16, (3, 3), activation='relu')(x)

x = layers.UpSampling2D((2, 2))(x)

decoded = layers.Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = keras.Model(input\_img, decoded)

autoencoder.compile(optimizer='adam', loss='binary\_crossentropy')

To train it, we will use the original MNIST digits with shape (samples, 3, 28, 28), and we will just normalize pixel values between 0 and 1.

from keras.datasets import mnist

import numpy as np

(x\_train, \_), (x\_test, \_) = mnist.load\_data()

x\_train = x\_train.astype('float32') / 255.

x\_test = x\_test.astype('float32') / 255.

x\_train = np.reshape(x\_train, (len(x\_train), 28, 28, 1))

x\_test = np.reshape(x\_test, (len(x\_test), 28, 28, 1))

Let's train this model for 50 epochs. For the sake of demonstrating how to visualize the results of a model during training, we will be using [the TensorFlow backend](https://blog.keras.io/building-autoencoders-in-keras.html?authuser=0) and the TensorBoard callback.

First, let's open up a terminal and start a TensorBoard server that will read logs stored at /tmp/autoencoder.

tensorboard --logdir=/tmp/autoencoder

Then let's train our model. In the callbacks list we pass an instance of the TensorBoard callback. After every epoch, this callback will write logs to /tmp/autoencoder, which can be read by our TensorBoard server.

from keras.callbacks import TensorBoard

autoencoder.fit(x\_train, x\_train,

epochs=50,

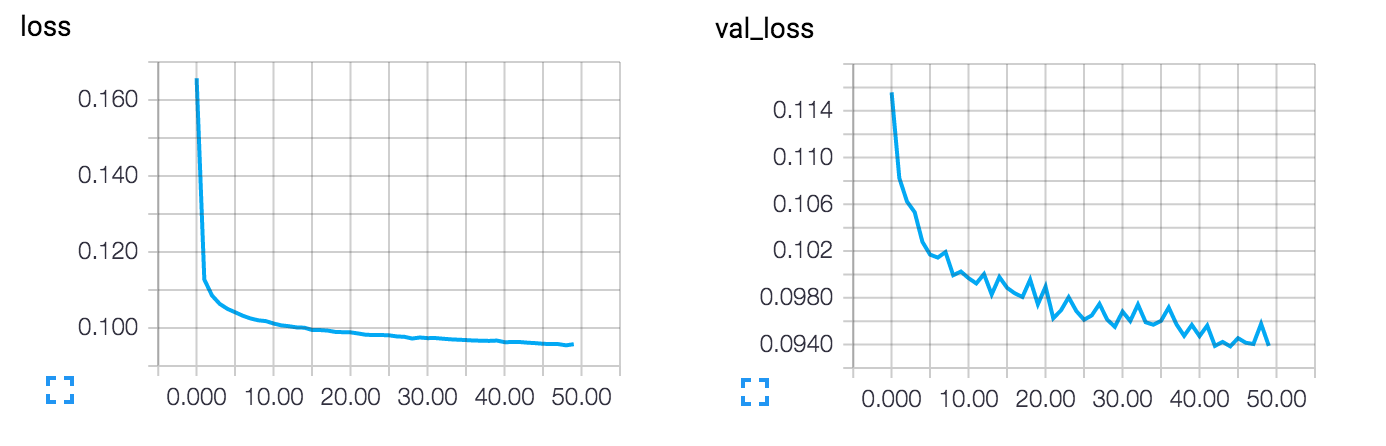
batch\_size=128,

shuffle=True,

validation\_data=(x\_test, x\_test),

callbacks=[TensorBoard(log\_dir='/tmp/autoencoder')])

This allows us to monitor training in the TensorBoard web interface (by navighating to http://0.0.0.0:6006):



The model converges to a loss of 0.094, significantly better than our previous models (this is in large part due to the higher entropic capacity of the encoded representation, 128 dimensions vs. 32 previously). Let's take a look at the reconstructed digits:

decoded\_imgs = autoencoder.predict(x\_test)

n = 10

plt.figure(figsize=(20, 4))

for i in range(1, n + 1):

# Display original

ax = plt.subplot(2, n, i)

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 + n)

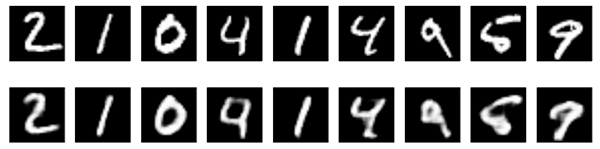
plt.imshow(decoded\_imgs[i].reshape(28, 28))

plt.gray()

ax.get\_xaxis().set\_visible(False)

ax.get\_yaxis().set\_visible(False)

plt.show()



We can also have a look at the 128-dimensional encoded representations. These representations are 8x4x4, so we reshape them to 4x32 in order to be able to display them as grayscale images.

encoder = keras.Model(input\_img, encoded)

encoded\_imgs = encoder.predict(x\_test)

n = 10

plt.figure(figsize=(20, 8))

for i in range(1, n + 1):

ax = plt.subplot(1, n, i)

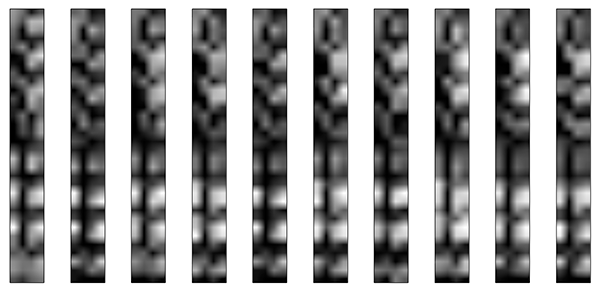
plt.imshow(encoded\_imgs[i].reshape((4, 4 \* 8)).T)

plt.gray()

ax.get\_xaxis().set\_visible(False)

ax.get\_yaxis().set\_visible(False)

plt.show()



Application to image denoising

Let's put our convolutional autoencoder to work on an image denoising problem. It's simple: we will train the autoencoder to map noisy digits images to clean digits images.

Here's how we will generate synthetic noisy digits: we just apply a gaussian noise matrix and clip the images between 0 and 1.

from keras.datasets import mnist

import numpy as np

(x\_train, \_), (x\_test, \_) = mnist.load\_data()

x\_train = x\_train.astype('float32') / 255.

x\_test = x\_test.astype('float32') / 255.

x\_train = np.reshape(x\_train, (len(x\_train), 28, 28, 1))

x\_test = np.reshape(x\_test, (len(x\_test), 28, 28, 1))

noise\_factor = 0.5

x\_train\_noisy = x\_train + noise\_factor \* np.random.normal(loc=0.0, scale=1.0, size=x\_train.shape)

x\_test\_noisy = x\_test + noise\_factor \* np.random.normal(loc=0.0, scale=1.0, size=x\_test.shape)

x\_train\_noisy = np.clip(x\_train\_noisy, 0., 1.)

x\_test\_noisy = np.clip(x\_test\_noisy, 0., 1.)

Here's what the noisy digits look like:

n = 10

plt.figure(figsize=(20, 2))

for i in range(1, n + 1):

ax = plt.subplot(1, n, i)

plt.imshow(x\_test\_noisy[i].reshape(28, 28))

plt.gray()

ax.get\_xaxis().set\_visible(False)

ax.get\_yaxis().set\_visible(False)

plt.show()

noisy digits

If you squint you can still recognize them, but barely. Can our autoencoder learn to recover the original digits? Let's find out.

Compared to the previous convolutional autoencoder, in order to improve the quality of the reconstructed, we'll use a slightly different model with more filters per layer:

input\_img = keras.Input(shape=(28, 28, 1))

x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(input\_img)

x = layers.MaxPooling2D((2, 2), padding='same')(x)

x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(x)

encoded = layers.MaxPooling2D((2, 2), padding='same')(x)

# At this point the representation is (7, 7, 32)

x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(encoded)

x = layers.UpSampling2D((2, 2))(x)

x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(x)

x = layers.UpSampling2D((2, 2))(x)

decoded = layers.Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = keras.Model(input\_img, decoded)

autoencoder.compile(optimizer='adam', loss='binary\_crossentropy')

Let's train it for 100 epochs:

autoencoder.fit(x\_train\_noisy, x\_train,

epochs=100,

batch\_size=128,

shuffle=True,

validation\_data=(x\_test\_noisy, x\_test),

callbacks=[TensorBoard(log\_dir='/tmp/tb', histogram\_freq=0, write\_graph=False)])

Now let's take a look at the results. Top, the noisy digits fed to the network, and bottom, the digits are reconstructed by the network.



It seems to work pretty well. If you scale this process to a bigger convnet, you can start building document denoising or audio denoising models. [Kaggle has an interesting dataset to get you started](https://www.kaggle.com/c/denoising-dirty-documents).

Sequence-to-sequence autoencoder

If you inputs are sequences, rather than vectors or 2D images, then you may want to use as encoder and decoder a type of model that can capture temporal structure, such as a LSTM. To build a LSTM-based autoencoder, first use a LSTM encoder to turn your input sequences into a single vector that contains information about the entire sequence, then repeat this vector n times (where n is the number of timesteps in the output sequence), and run a LSTM decoder to turn this constant sequence into the target sequence.

We won't be demonstrating that one on any specific dataset. We will just put a code example here for future reference for the reader!

timesteps = ... # Length of your sequences

input\_dim = ...

latent\_dim = ...

inputs = keras.Input(shape=(timesteps, input\_dim))

encoded = layers.LSTM(latent\_dim)(inputs)

decoded = layers.RepeatVector(timesteps)(encoded)

decoded = layers.LSTM(input\_dim, return\_sequences=True)(decoded)

sequence\_autoencoder = keras.Model(inputs, decoded)

encoder = keras.Model(inputs, encoded)

Variational autoencoder (VAE)

Variational autoencoders are a slightly more modern and interesting take on autoencoding.

What is a variational autoencoder, you ask? It's a type of autoencoder with added constraints on the encoded representations being learned. More precisely, it is an autoencoder that learns a [latent variable model](https://en.wikipedia.org/wiki/Latent_variable_model) for its input data. So instead of letting your neural network learn an arbitrary function, you are learning the parameters of a probability distribution modeling your data. If you sample points from this distribution, you can generate new input data samples: a VAE is a "generative model".

How does a variational autoencoder work?

First, an encoder network turns the input samples x into two parameters in a latent space, which we will note z\_mean and z\_log\_sigma. Then, we randomly sample similar points z from the latent normal distribution that is assumed to generate the data, via z = z\_mean + exp(z\_log\_sigma) \* epsilon, where epsilon is a random normal tensor. Finally, a decoder network maps these latent space points back to the original input data.

The parameters of the model are trained via two loss functions: a reconstruction loss forcing the decoded samples to match the initial inputs (just like in our previous autoencoders), and the KL divergence between the learned latent distribution and the prior distribution, acting as a regularization term. You could actually get rid of this latter term entirely, although it does help in learning well-formed latent spaces and reducing overfitting to the training data.

Because a VAE is a more complex example, we have made the code available on Github as [a standalone script](https://github.com/fchollet/keras/blob/master/examples/variational_autoencoder.py). Here we will review step by step how the model is created.

First, here's our encoder network, mapping inputs to our latent distribution parameters:

original\_dim = 28 \* 28

intermediate\_dim = 64

latent\_dim = 2

inputs = keras.Input(shape=(original\_dim,))

h = layers.Dense(intermediate\_dim, activation='relu')(inputs)

z\_mean = layers.Dense(latent\_dim)(h)

z\_log\_sigma = layers.Dense(latent\_dim)(h)

We can use these parameters to sample new similar points from the latent space:

from keras import backend as K

def sampling(args):

z\_mean, z\_log\_sigma = args

epsilon = K.random\_normal(shape=(K.shape(z\_mean)[0], latent\_dim),

mean=0., stddev=0.1)

return z\_mean + K.exp(z\_log\_sigma) \* epsilon

z = layers.Lambda(sampling)([z\_mean, z\_log\_sigma])

Finally, we can map these sampled latent points back to reconstructed inputs:

# Create encoder

encoder = keras.Model(inputs, [z\_mean, z\_log\_sigma, z], name='encoder')

# Create decoder

latent\_inputs = keras.Input(shape=(latent\_dim,), name='z\_sampling')

x = layers.Dense(intermediate\_dim, activation='relu')(latent\_inputs)

outputs = layers.Dense(original\_dim, activation='sigmoid')(x)

decoder = keras.Model(latent\_inputs, outputs, name='decoder')

# instantiate VAE model

outputs = decoder(encoder(inputs)[2])

vae = keras.Model(inputs, outputs, name='vae\_mlp')

What we've done so far allows us to instantiate 3 models:

* an end-to-end autoencoder mapping inputs to reconstructions
* an encoder mapping inputs to the latent space
* a generator that can take points on the latent space and will output the corresponding reconstructed samples.

We train the model using the end-to-end model, with a custom loss function: the sum of a reconstruction term, and the KL divergence regularization term.

reconstruction\_loss = keras.losses.binary\_crossentropy(inputs, outputs)

reconstruction\_loss \*= original\_dim

kl\_loss = 1 + z\_log\_sigma - K.square(z\_mean) - K.exp(z\_log\_sigma)

kl\_loss = K.sum(kl\_loss, axis=-1)

kl\_loss \*= -0.5

vae\_loss = K.mean(reconstruction\_loss + kl\_loss)

vae.add\_loss(vae\_loss)

vae.compile(optimizer='adam')

We train our VAE on MNIST digits:

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

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:])))

vae.fit(x\_train, x\_train,

epochs=100,

batch\_size=32,

validation\_data=(x\_test, x\_test))

Because our latent space is two-dimensional, there are a few cool visualizations that can be done at this point. One is to look at the neighborhoods of different classes on the latent 2D plane:

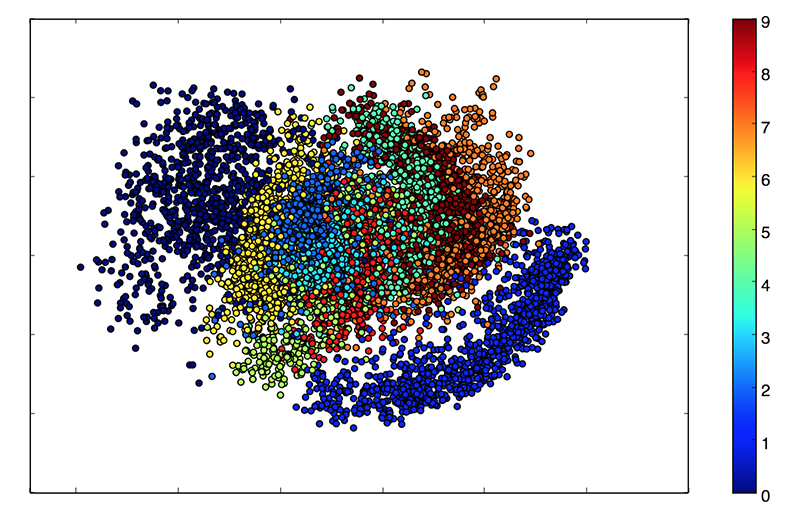
x\_test\_encoded = encoder.predict(x\_test, batch\_size=batch\_size)

plt.figure(figsize=(6, 6))

plt.scatter(x\_test\_encoded[:, 0], x\_test\_encoded[:, 1], c=y\_test)

plt.colorbar()

plt.show()



Each of these colored clusters is a type of digit. Close clusters are digits that are structurally similar (i.e. digits that share information in the latent space).

Because the VAE is a generative model, we can also use it to generate new digits! Here we will scan the latent plane, sampling latent points at regular intervals, and generating the corresponding digit for each of these points. This gives us a visualization of the latent manifold that "generates" the MNIST digits.

# Display a 2D manifold of the digits

n = 15 # figure with 15x15 digits

digit\_size = 28

figure = np.zeros((digit\_size \* n, digit\_size \* n))

# We will sample n points within [-15, 15] standard deviations

grid\_x = np.linspace(-15, 15, n)

grid\_y = np.linspace(-15, 15, n)

for i, yi in enumerate(grid\_x):

for j, xi in enumerate(grid\_y):

z\_sample = np.array([[xi, yi]])

x\_decoded = decoder.predict(z\_sample)

digit = x\_decoded[0].reshape(digit\_size, digit\_size)

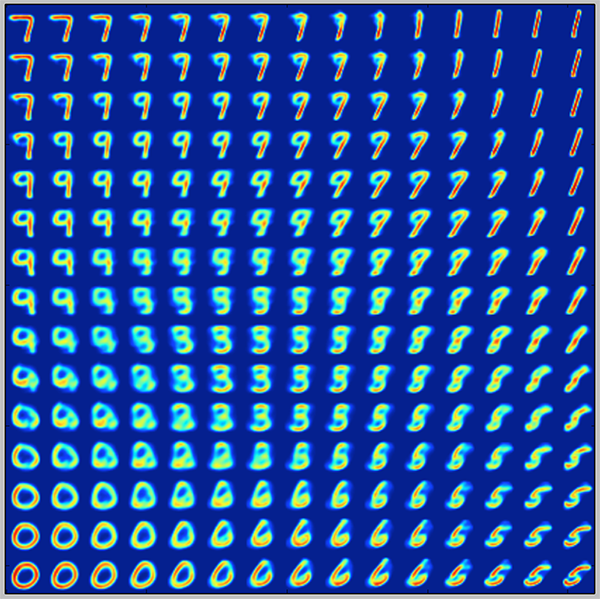
figure[i \* digit\_size: (i + 1) \* digit\_size,

j \* digit\_size: (j + 1) \* digit\_size] = digit

plt.figure(figsize=(10, 10))

plt.imshow(figure)

plt.show()



**1. Introduction**

Information theory is a scientific field of great interest as it is expected to provide answers for today’s data storage and data analysis problems. It is this field that provides the theoretical base for data compression, after all. There are purely statistical approaches but**machine learning offers flexible neural network structures that can compress data for a variety of applications.**

In this tutorial, we’ll discuss the function, structure, hyper-parameters, training, and applications of different common autoencoder types.

**2. Function**

**There are many ways to compress data and purely statistical methods such as**[**principal component analysis (PCA)**](https://www.baeldung.com/cs/principal-component-analysis)can help identify the key features accountable for the variability in the data and use these to represent the information using fewer bits. This type of compression can be called “Dimensionality Reduction”. However, this PCA solution can only offer an encoding with linearly uncorrelated features.

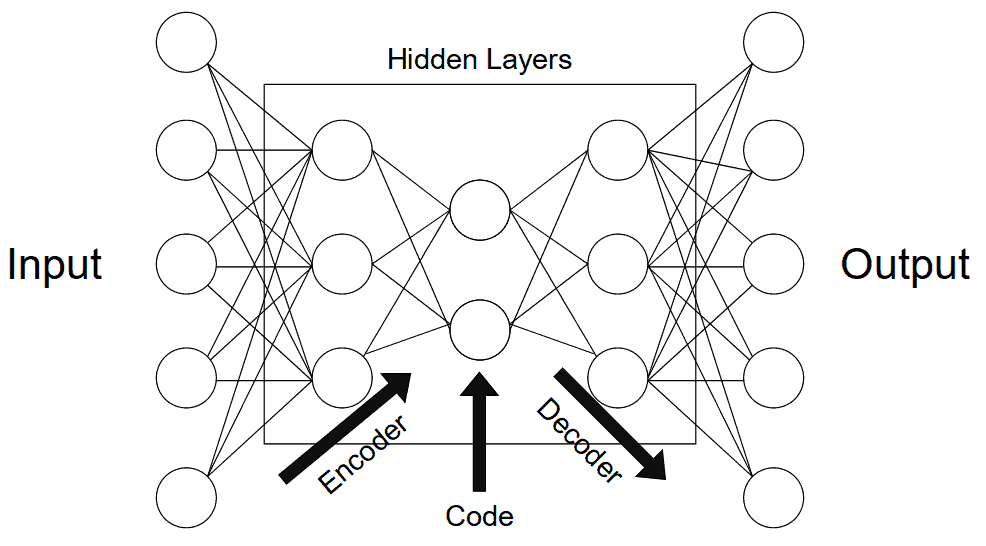
The machine learning alternative solution proposes neural networks that are structured as autoencoder models.**Autoencoders can learn richer, non-linear encoding features.** These features can correlate with each other and are therefore not necessarily orthogonal to one another. Using these functions we can represent complex data in a latent space.

For example, when discussing Convolutional Neural Networks (CNNs) for facial recognition, these can be used as autoencoders for image data. They can allow a user to store photos using less space by encoding the image by lowering the quality just a little bit. **The difference or loss in quality between the output and input images is called reconstruction loss.**

The main goal for autoencoders is to represent complex data using as little code as possible with little to no reconstruction or “compression” loss. To do so, the autoencoder has to look at the data and construct a function that can transform a particular instance of data into a meaningful code. We can think of this as a remapping of the original data using fewer dimensions. We can also keep in mind that this code has to be interpreted later on by a decoder to access the data.

**3. General Structure**

To better understand the functioning of these models, we’ll decompose them into individual components. **We’ll describe the structure in the same order that the data travels our neural net, discussing encoders first, then the bottleneck or “code” and then decoders.** These segments of autoencoders can be seen as different layers of nodes in a model. The image below helps interpret these three different components. The arrow indicates the flow of the data in our model:



**3.1. Encoder**

The encoder’s role is to compress the data into a code. In neural networks, we can implement this phenomenon by connecting a series of pooling layers, each one reducing the number of dimensions that are present in the data. In doing this, we can de-noise data by keeping only the parts of it that are relevant enough to encode.

The layer of the neural network that has the fewest dimensions, generally in the middle of all the layers, is called the bottleneck. If we’re interested in keeping more information about a specific instance, we’d need a larger code size. This “code size” hyper-parameter is important as it defines how much data gets to be encoded and simultaneously regulates our model. A very big code will represent more noise but too little code may not accurately represent the data at hand.

**3.2. Decoder**

The decoder component of the network acts as an interpreter for the code. In the case of convolutional neural nets, it can reconstruct an image based on a particular code. We can think of this component as a value extraction, interpretation, or decompression tool. Image segmentation is usually the decoders’ job as well. If we keep our last example of classifying dog pictures, we could continue decoding while adding a copy of previous compressed layers to each layer of decoder nodes. In doing so, we should obtain a photograph with a subset of pixels that identify the dog in the picture and segment it from the background and other objects.

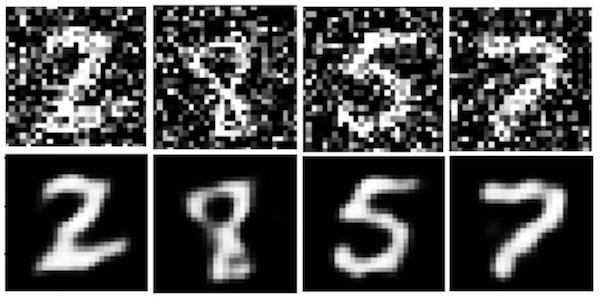
**Generative models such as Variational Auto-Encoders (VAEs) can even use the decoder to render data that doesn’t exist.**This can be useful for data augmentation purposes. In having diverse data sets from generative models, other models can learn more thoroughly from data.

**4. Different Types**

Autoencoders are applied in many different fields across machine learning and computer science. Here are the main types that we can encounter and their respective common applications.

**4.1. Denoising Autoencoder**

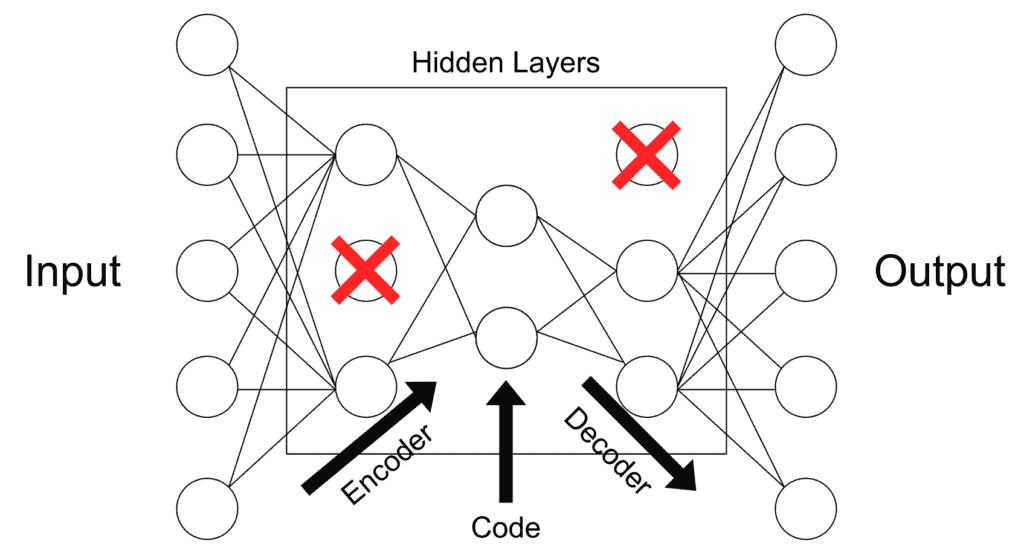
These types of autoencoders are meant to encode noisy data efficiently to leave random noise out of the code. In doing so, the output of the autoencoder is meant to be de-noised and therefore different than the input. We can see what an implementation of this would look like using the popular MNIST dataset, as presented in the image below:



These types of autoencoders can be used for feature extraction and data de-noising.

**4.2. Sparse Autoencoder**

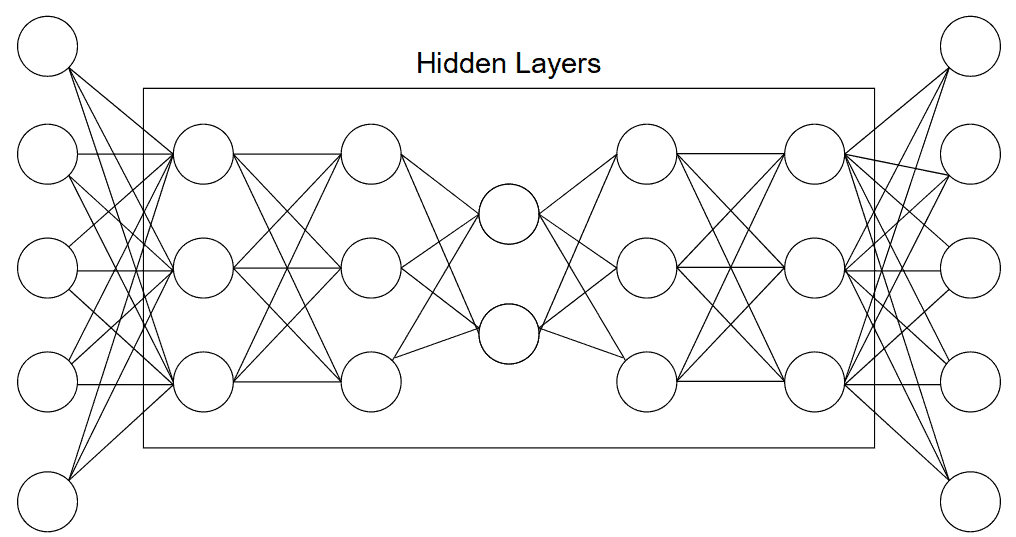
This type of autoencoder explicitly penalizes the use of hidden node connections. This regularizes the model, keeping it from overfitting the data. This “sparsity penalty” is added to the reconstruction loss to obtain a global loss function. Alternatively, one could just delete a set number of connections in the hidden layer:



This is more of a regularization method that can be used with a broad array of encoder types. Applications may vary.

**4.3. Deep Autoencoder**

Deep autoencoders are composed of two symmetrical deep-belief networks. This structure is similar to the “general structure” representation of an autoencoder using nodes and connections found above. These mirrored components can be described as two restricted Boltzmann machines acting as encoders and decoders:



Autoencoders of this type are used for a wide array of purposes such as feature extraction, dimensionality reduction, and data compression.

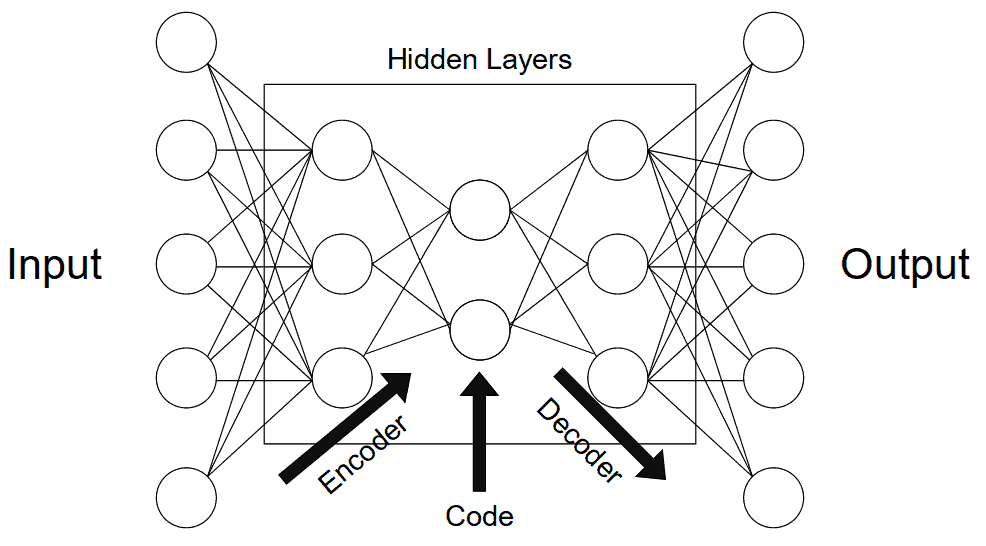
**4.4. Contractive Autoencoder**

**Contractive autoencoders penalize big variations in code when small changes in the input occur.** This means that similar inputs should have similar codes. This usually means that the information captured by the autoencoder is meaningful and represents a large variance in the data. To implement this, we can add a penalty loss to the global loss function of the autoencoder.

Data augmentation is a great type of application for these types of autoencoders.

**4.5. Undercomplete Autoencoder**

These autoencoders have smaller hidden dimensions in comparison to input. This means that they excel at capturing only the most important features present in the data. These types of autoencoders usually do not need regularization as they do not aim to reproduce outputs similar to inputs but rather rely on the compression stage to capture meaningful features in the data:



Feature extraction is the main type of application for this type of autoencoders.

**4.6. Convolutional Autoencoder**

Convolutional autoencoders can use a sum of different signals to encode and decode. The most common version of this is probably a U-Net convolutional model. This model developed for biological imaging applications will interpret the output of different filters across an image to classify and ultimately segment the image data. Similar convolutional models are used today to segment images.

Applications are found for these autoencoders in computer vision and pattern recognition.

**4.7. Variational Autoencoder**

Variational autoencoders or VAEs assume encode data as distributions as opposed to single points in space. This is a way of having a more regular latent space that we can better use to generate new data with. Hence, VAEs are referred to as [generative models](https://www.baeldung.com/cs/applications-of-generative-models). The original [“Auto-Encoding Variational Bayes” paper published by Diederik P Kingma and Max Welling](https://arxiv.org/abs/1312.6114) details the complete functioning of these particular models.

The process to achieve this latent space distribution makes the training process a little different. First, instead of mapping an instance as a point in space, it is mapped as the center of a normal distribution. Next, we take a point from that distribution to decode and compute the reconstruction error that will be backpropagated across the network.