1. What are the main tasks that autoencoders are used for?

An autoencoder is an unsupervised learning technique for neural networks that learns efficient data representations (encoding) by training the network to ignore signal “noise.” Autoencoders can be used for image denoising, image compression, and, in some cases, even generation of image data.

Auto-Encoder: What Is It? And What Is It Used For? (Part 1)

A Gentle Introduction to Auto-Encoder and Some Of Its Common Use Cases With Python Code

Background:

Autoencoder is an unsupervised artificial neural network that learns how to efficiently compress and encode data then learns how to reconstruct the data back from the reduced encoded representation to a representation that is as close to the original input as possible.

Autoencoder, by design, reduces data dimensions by learning how to ignore the noise in the data.

Here is an example of the input/output image from the [MNIST](http://yann.lecun.com/exdb/mnist/index.html) dataset to an autoencoder.

Autoencoder for MNIST

Autoencoder Components:

Autoencoders consists of 4 main parts:

1- Encoder: In which the model learns how to reduce the input dimensions and compress the input data into an encoded representation.

2- Bottleneck: which is the layer that contains the compressed representation of the input data. This is the lowest possible dimensions of the input data.

3- Decoder: In which the model learns how to reconstruct the data from the encoded representation to be as close to the original input as possible.

4- Reconstruction Loss: This is the method that measures measure how well the decoder is performing and how close the output is to the original input.

The training then involves using back propagation in order to minimize the network’s reconstruction loss.

You must be wondering why would I train a neural network just to output an image or data that is exactly the same as the input! This article will cover the most common use cases for Autoencoder. Let’s get started:

Autoencoder Architecture:

The network architecture for autoencoders can vary between a simple FeedForward network, LSTM network or Convolutional Neural Network depending on the use case. We will explore some of those architectures in the new next few lines.

1- Autoencoder for Anomaly Detection:

There are many ways and techniques to detect anomalies and outliers. I have covered this topic in a different post below:

[5 Ways to Detect Outliers That Every Data Scientist Should Know (Python Code)](https://towardsdatascience.com/5-ways-to-detect-outliers-that-every-data-scientist-should-know-python-code-70a54335a623" \t "_blank)

[Detecting Anomalies is critical to any business either by identifying faults or being proactive. This article discusses…](https://towardsdatascience.com/5-ways-to-detect-outliers-that-every-data-scientist-should-know-python-code-70a54335a623" \t "_blank)

[towardsdatascience.com](https://towardsdatascience.com/5-ways-to-detect-outliers-that-every-data-scientist-should-know-python-code-70a54335a623" \t "_blank)

However, if you have correlated input data, the autoencoder method will work very well because the encoding operation relies on the correlated features to compress the data.

Let’s say that we have trained an autoencoder on the MNIST dataset. Using a simple FeedForward neural network, we can achieve this by building a simple 6 layers network as below:

As you can see in the output, the last reconstruction loss/error for the validation set is 0.0193 which is great. Now, if I pass any normal image from the MNIST dataset, the reconstruction loss will be very low (< 0.02) BUT if I tried to pass any other different image (outlier or anomaly), we will get a high reconstruction loss value because the network failed to reconstruct the image/input that is considered an anomaly.

Notice in the code above, you can use only the encoder part to compress some data or images and you can also only use the decoder part to decompress the data by loading the decoder layers.

Now, let’s do some anomaly detection. The code below uses two different images to predict the anomaly score (reconstruction error) using the autoencoder network we trained above. the first image is from the MNIST and the result is 5.43209. This means that the image is not an anomaly. The second image I used, is a completely random image that doesn’t belong to the training dataset and the results were: 6789.4907. This high error means that the image is an anomaly. The same concept applies to any type of dataset.

2- Image Denoising:

Image denoising

Denoising or noise reduction is the process of removing noise from a signal. This can be an image, audio or a document. You can train an Autoencoder network to learn how to remove noise from pictures. In order to try out this use case, let’s re-use the famous MNIST dataset and let’s create some synthetic noise in the dataset. The code below will simply add some noise to the dataset then plot a few pictures to make sure that we’ve successfully created.

First, we define the input layer and the dimensions of the input data. MNIST dataset has images that are reshaped to be 28 X 28 in dimensions. Since the images are greyscaled, the colour channel of the image will be 1 so the shape is (28, 28, 1).

The second layer is the convolution layer, this layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs. 32 is the number of output filters in the convolution and (3, 3) is the kernel size.

After each convolution layer, we use MaxPooling function to reduce the dimensions. The (28, 28, 32) is reduced by a factor of two so it will be (14, 14, 32) after the first MaxPooling then (7, 7, 32) after the second MaxPooling. This is the encoded representation of the image.

If you scale the ConvNet above, you can use it to denoise any type of images, audio or scanned documents.

In this part of the article, I covered two important use cases for autoencoders and I build two different neural network architectures — CNN and FeedForward. In part 2, I will cover another 2 important use cases for Autoencoders. The first one will be how to use autoencoder with a sequence of data by building an LSTM network and the second use case is a called Variational Autoencoder (VAE) which is mainly used in Generative Models and generating data or images. Stay tuned!

2.Suppose you want to train a classifier, and you have plenty of unlabeled training data but only a few thousand labeled instances. How can autoencoders help? How would you proceed?

Autoencoders are a specific type of feedforward neural networks where the input is the same as the output. They compress the input into a lower-dimensional code and then reconstruct the output from this representation. The code is a compact “summary” or “compression” of the input, also called the latent-space representation.

An autoencoder consists of 3 components: encoder, code and decoder. The encoder compresses the input and produces the code, the decoder then reconstructs the input only using this code.

To build an autoencoder we need 3 things: an encoding method, decoding method, and a loss function to compare the output with the target. We will explore these in the next section.

Autoencoders are mainly a dimensionality reduction (or compression) algorithm with a couple of important properties:

Data-specific: Autoencoders are only able to meaningfully compress data similar to what they have been trained on. Since they learn features specific for the given training data, they are different than a standard data compression algorithm like gzip. So we can’t expect an autoencoder trained on handwritten digits to compress landscape photos.

Lossy: The output of the autoencoder will not be exactly the same as the input, it will be a close but degraded representation. If you want lossless compression they are not the way to go.

Unsupervised: To train an autoencoder we don’t need to do anything fancy, just throw the raw input data at it. Autoencoders are considered an unsupervised learning technique since they don’t need explicit labels to train on. But to be more precise they are self-supervised because they generate their own labels from the training data.

2. Architecture

Let’s explore the details of the encoder, code and decoder. Both the encoder and decoder are fully-connected feedforward neural networks, essentially the ANNs we covered in [Part 1](https://medium.com/towards-data-science/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6). Code is a single layer of an ANN with the dimensionality of our choice. The number of nodes in the code layer (code size) is a hyperparameter that we set before training the autoencoder.

This is a more detailed visualization of an autoencoder. First the input passes through the encoder, which is a fully-connected ANN, to produce the code. The decoder, which has the similar ANN structure, then produces the output only using the code. The goal is to get an output identical with the input. Note that the decoder architecture is the mirror image of the encoder. This is not a requirement but it’s typically the case. The only requirement is the dimensionality of the input and output needs to be the same. Anything in the middle can be played with.

There are 4 hyperparameters that we need to set before training an autoencoder:

Code size: number of nodes in the middle layer. Smaller size results in more compression.

Number of layers: the autoencoder can be as deep as we like. In the figure above we have 2 layers in both the encoder and decoder, without considering the input and output.

Number of nodes per layer: the autoencoder architecture we’re working on is called a stacked autoencoder since the layers are stacked one after another. Usually stacked autoencoders look like a “sandwitch”. The number of nodes per layer decreases with each subsequent layer of the encoder, and increases back in the decoder. Also the decoder is symmetric to the encoder in terms of layer structure. As noted above this is not necessary and we have total control over these parameters.

Loss function: we either use mean squared error (mse) or binary crossentropy. If the input values are in the range [0, 1] then we typically use crossentropy, otherwise we use the mean squared error. For more details check out this [video](https://www.youtube.com/watch?v=xTU79Zs4XKY).

Autoencoders are trained the same way as ANNs via backpropagation. Check out the [introduction](https://medium.com/towards-data-science/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6#04e7) of Part 1 for more details on how neural networks are trained, it directly applies to the autoencoders.

3. Implementation

Now let’s implement an autoencoder for the following architecture, 1 hidden layer in the encoder and decoder.

We will use the extremely popular MNIST [dataset](https://www.tensorflow.org/get_started/mnist/beginners) as input. It contains black-and-white images of handwritten digits.

They’re of size 28x28 and we use them as a vector of 784 numbers between [0, 1]. Check the [jupyter notebook](https://github.com/ardendertat/Applied-Deep-Learning-with-Keras/blob/master/notebooks/Part%203%20-%20Autoencoders.ipynb" \t "_blank) for the details.

We will now implement the autoencoder with Keras. The hyperparameters are: 128 nodes in the hidden layer, code size is 32, and binary crossentropy is the loss function.

This is very similar to the ANNs we worked on, but now we’re using the Keras functional API. Refer to [this guide](https://keras.io/getting-started/functional-api-guide/) for details, but here’s a quick comparison. Before we used to add layers using the sequential API as follows:

model.add(Dense(16, activation='relu'))  
model.add(Dense(8, activation='relu'))

With the functional API we do this:

layer\_1 = Dense(16, activation='relu')(input)  
layer\_2 = Dense(8, activation='relu')(layer\_1)

It’s more verbose but a more flexible way to define complex models. We can easily grab parts of our model, for example only the decoder, and work with that. The output of Dense method is a callable layer, using the functional API we provide it with the input and store the output. The output of a layer becomes the input of the next layer. With the sequential API the add method implicitly handled this for us.

Note that all the layers use the relu activation function, as it’s the standard with deep neural networks. The last layer uses the sigmoid activation because we need the outputs to be between [0, 1]. The input is also in the same range.

Also note the call to fit function, before with ANNs we used to do:

model.fit(x\_train, y\_train)

But now we do:

model.fit(x\_train, x\_train)

Remember that the targets of the autoencoder are the same as the input. That’s why we supply the training data as the target.

Visualization

Now let’s visualize how well our autoencoder reconstructs its input.

We run the autoencoder on the test set simply by using the predict function of Keras. For every image in the test set, we get the output of the autoencoder. We expect the output to be very similar to the input.

They are indeed pretty similar, but not exactly the same. We can notice it more clearly in the last digit “4”. Since this was a simple task our autoencoder performed pretty well.

Advice

We have total control over the architecture of the autoencoder. We can make it very powerful by increasing the number of layers, nodes per layer and most importantly the code size. Increasing these hyperparameters will let the autoencoder to learn more complex codings. But we should be careful to not make it too powerful. Otherwise the autoencoder will simply learn to copy its inputs to the output, without learning any meaningful representation. It will just mimic the identity function. The autoencoder will reconstruct the training data perfectly, but it will be overfitting without being able to generalize to new instances, which is not what we want.

This is why we prefer a “sandwitch” architecture, and deliberately keep the code size small. Since the coding layer has a lower dimensionality than the input data, the autoencoder is said to be undercomplete. It won’t be able to directly copy its inputs to the output, and will be forced to learn intelligent features. If the input data has a pattern, for example the digit “1” usually contains a somewhat straight line and the digit “0” is circular, it will learn this fact and encode it in a more compact form. If the input data was completely random without any internal correlation or dependency, then an undercomplete autoencoder won’t be able to recover it perfectly. But luckily in the real-world there is a lot of dependency.

4. Denoising Autoencoders

Keeping the code layer small forced our autoencoder to learn an intelligent representation of the data. There is another way to force the autoencoder to learn useful features, which is adding random noise to its inputs and making it recover the original noise-free data. This way the autoencoder can’t simply copy the input to its output because the input also contains random noise. We are asking it to subtract the noise and produce the underlying meaningful data. This is called a denoising autoencoder.

The top row contains the original images. We add random Gaussian noise to them and the noisy data becomes the input to the autoencoder. The autoencoder doesn’t see the original image at all. But then we expect the autoencoder to regenerate the noise-free original image.

There is only one small difference between the implementation of denoising autoencoder and the regular one. The architecture doesn’t change at all, only the fit function. We trained the regular autoencoder as follows:

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

Denoising autoencoder is trained as:

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

Simple as that, everything else is exactly the same. The input to the autoencoder is the noisy image, and the expected target is the original noise-free one.

Visualization

Now let’s visualize whether we are able to recover the noise-free images.

Looks pretty good. The bottom row is the autoencoder output. We can do better by using more complex autoencoder architecture, such as convolutional autoencoders. We will cover convolutions in the upcoming article.

5. Sparse Autoencoders

We introduced two ways to force the autoencoder to learn useful features: keeping the code size small and denoising autoencoders. The third method is using regularization. We can regularize the autoencoder by using a sparsity constraint such that only a fraction of the nodes would have nonzero values, called active nodes.

In particular, we add a penalty term to the loss function such that only a fraction of the nodes become active. This forces the autoencoder to represent each input as a combination of small number of nodes, and demands it to discover interesting structure in the data. This method works even if the code size is large, since only a small subset of the nodes will be active at any time.

It’s pretty easy to do this in Keras with just one parameter. As a reminder, previously we created the code layer as follows:

code = Dense(code\_size, activation='relu')(input\_img)

We now add another parameter called activity\_regularizer by specifying the regularization strength. This is typically a value in the range [0.001, 0.000001]. Here we chose 10e-6.

code = Dense(code\_size, activation='relu', activity\_regularizer=l1(10e-6))(input\_img)

The final loss of the sparse model is 0.01 higher than the standard one, due to the added regularization term.

Let’s demonstrate the encodings generated by the regularized model are indeed sparse. If we look at the histogram of code values for the images in the test set, the distribution is as follows:

The mean for the standard model is 6.6 but for the regularized model it’s 0.8, a pretty big reduction. We can see that a large chunk of code values in the regularized model are indeed 0, which is what we wanted. The variance of the regularized model is also fairly low.

6. Use Cases

Now we might ask the following questions. How good are autoencoders at compressing the input? And are they a commonly used deep learning technique?

Unfortunately autoencoders are not widely used in real-world applications. As a compression method, they don’t perform better than its alternatives, for example jpeg does photo compression better than an autoencoder. And the fact that autoencoders are data-specific makes them impractical as a general technique. They have 3 common use cases though:

Data denoising: we have seen an example of this on images.

Dimensionality reduction: visualizing high-dimensional data is challenging. t-SNE is the most commonly used method but struggles with large number of dimensions (typically above 32). So autoencoders are used as a preprocessing step to reduce the dimensionality, and this compressed representation is used by t-SNE to visualize the data in 2D space. For great articles on t-SNE refer [here](https://distill.pub/2016/misread-tsne/) and [here](http://colah.github.io/posts/2014-10-Visualizing-MNIST/).

Variational Autoencoders (VAE): this is a more modern and complex use-case of autoencoders and we will cover them in another article. But as a quick summary, VAE learns the parameters of the probability distribution modeling the input data, instead of learning an arbitrary function in the case of vanilla autoencoders. By sampling points from this distribution we can also use the VAE as a generative model.

3.If an autoencoder perfectly reconstructs the inputs, is it necessarily a good autoencoder? How can you evaluate the performance of an autoencoder?

Let’s start with a quick overview of autoencoders’ architecture.

Autoencoders consist of 3 parts:

1. Encoder: A module that compresses [the train-validate-test set input data](https://www.v7labs.com/blog/train-validation-test-set) into an encoded representation that is typically several orders of magnitude smaller than the input data.

2. Bottleneck: A module that contains the compressed knowledge representations and is therefore the most important part of the network.

3. Decoder: A module that helps the network“decompress” the knowledge representations and reconstructs the data back from its encoded form. The output is then compared with a ground truth.

Ready to explore this topic more in-depth?

Let’s break it down.

The relationship between the Encoder, Bottleneck, and Decoder

Encoder

The encoder is a set of convolutional blocks followed by pooling modules that compress the input to the model into a compact section called the bottleneck.

The bottleneck is followed by the decoder that consists of a series of upsampling modules to bring the compressed feature back into the form of an image. In case of simple autoencoders, the output is expected to be the same as the input with reduced noise.

However, for variational autoencoders it is a completely new image, formed with information the model has been provided as input.

Bottleneck

The most important part of the neural network, and ironically the smallest one, is the bottleneck. The bottleneck exists to restrict the flow of information to the decoder from the encoder, thus,allowing only the most vital information to pass through.

Since the bottleneck is designed in such a way that the maximum information possessed by an image is captured in it, we can say that the bottleneck helps us form a knowledge-representation of the input.

Thus, the encoder-decoder structure helps us extract the most from an image in the form of data and establish useful correlations between various inputs within the network.

A bottleneck as a compressed representation of the input further prevents the neural network from memorising the input and overfitting on the data.

As a rule of thumb, remember this: The smaller the bottleneck, the lower the risk of overfitting.

However—

Very small bottlenecks would restrict the amount of information storable, which increases the chances of important information slipping out through the pooling layers of the encoder.

Decoder

Finally, the decoder is a set of upsampling and convolutional blocks that reconstructs the bottleneck's output.

Since the input to the decoder is a compressed knowledge representation, the decoder serves as a “decompressor” and builds back the image from its latent attributes.

How to train autoencoders?

You need to set 4 hyperparameters before training an autoencoder:

Code size: The code size or the size of the bottleneck is the most important hyperparameter used to tune the autoencoder. The bottleneck size decides how much the data has to be compressed. This can also act as a regularisation term.

Number of layers: Like all neural networks, an important hyperparameter to tune autoencoders is the depth of the encoder and the decoder. While a higher depth increases model complexity, a lower depth is faster to process.

Number of nodes per layer: The number of nodes per layer defines the weights we use per layer. Typically, the number of nodes decreases with each subsequent layer in the autoencoder as the input to each of these layers becomes smaller across the layers.

Reconstruction Loss: The [loss function](https://www.v7labs.com/blog/pytorch-loss-functions) we use to train the autoencoder is highly dependent on the type of input and output we want the autoencoder to adapt to. If we are working with image data, the most popular loss functions for reconstruction are MSE Loss and L1 Loss. In case the inputs and outputs  are within the range [0,1], as in MNIST, we can also make use of Binary Cross Entropy as the reconstruction loss.

💡 Pro tip: Looking for quality training data? Check out [65+ Best Free Datasets for Machine Learning](https://www.v7labs.com/blog/best-free-datasets-for-machine-learning) to find the right dataset for your needs.

Finally, let’s explore different types of autoencoders that you might encounter.

5 types of autoencoders

The idea of autoencoders for neural networks isn't new.

In fact—

The first applications date to the 1980s. Initially used for dimensionality reduction and feature learning, an autoencoder concept has evolved over the years and is now widely used for learning generative models of data.

Here are five popular autoencoders that we will discuss:

Undercomplete autoencoders

Sparse autoencoders

Contractive autoencoders

Denoising autoencoders

Variational Autoencoders (for generative modelling)

1. Undercomplete Autoencoders

An undercomplete autoencoder is one of the simplest types of autoencoders.

The way it works is very straightforward—

Undercomplete autoencoder takes in an image and tries to predict the same image as output, thus reconstructing the image from the compressed bottleneck region.

Undercomplete autoencoders are truly unsupervised as they do not take any form of label, the target being the same as the input.

The primary use of autoencoders like such is the generation of the latent space or the bottleneck, which forms a compressed substitute of the input data and can be easily decompressed back with the help of the network when needed.

This form of compression in the data can be modeled as a form of dimensionality reduction.

When we think of dimensionality reduction, we tend to think of methods like PCA (Principal Component Analysis) that form a lower-dimensional hyperplane to represent data in a higher-dimensional form without losing information.

However—

PCA can only build linear relationships. As a result, it is put at a disadvantage compared with methods like undercomplete autoencoders that can learn non-linear relationships and, therefore, perform better in dimensionality reduction.

This form of nonlinear dimensionality reduction where the autoencoder learns a non-linear manifold is also termed as manifold learning.

Effectively, if we remove all non-linear activations from an undercomplete autoencoder and use only linear layers, we reduce the undercomplete autoencoder into something that works at an equal footing with PCA.

The loss function used to train an undercomplete autoencoder is called reconstruction loss, as it is a check of how well the image has been reconstructed from the input.

Although the reconstruction loss can be anything depending on the input and output, we will use an L1 loss to depict the term (also called the norm loss) represented by:

As the loss function has no explicit regularisation term, the only method to ensure that the model is not memorising the input data is by regulating the size of the bottleneck and the number of hidden layers within this part of the network—the architecture.

2. Sparse Autoencoders

Sparse autoencoders are similar to the undercomplete autoencoders in that they use the same image as input and ground truth. However—

The means via which encoding of information is regulated is significantly different.

While undercomplete autoencoders are regulated and fine-tuned by regulating the size of the bottleneck, the sparse autoencoder is regulated by changing the number of nodes at each hidden layer.

Since it is not possible to design a neural network that has a flexible number of nodes at its hidden layers, sparse autoencoders work by penalizing the [activation of some neurons](https://www.v7labs.com/blog/neural-networks-activation-functions) in hidden layers.

In other words, the loss function has a term that calculates the number of neurons that have been activated and provides a penalty that is directly proportional to that.

This penalty, called the sparsity function, prevents the neural network from activating more neurons and serves as a regularizer.

While typical regularizers work by creating a penalty on the size of the weights at the nodes, sparsity regularizer works by creating a penalty on the number of nodes activated.

This form of regularization allows the network to have nodes in hidden layers dedicated to find specific features in images during training and treating the regularization problem as a problem separate from the latent space problem.

We can thus set latent space dimensionality at the bottleneck without worrying about regularization.

There are two primary ways in which the sparsity regularizer term can be incorporated into the loss function.

L1 Loss: In here, we add the magnitude of the sparsity regularizer as we do for general regularizers:

KL-Divergence: In this case, we consider the activations over a collection of samples at once rather than summing them as in the L1 Loss method. We constrain the average activation of each neuron over this collection.

Considering the ideal distribution as a Bernoulli distribution, we include KL divergence within the loss to reduce the difference between the current distribution of the activations and the ideal (Bernoulli) distribution.  
3. Contractive Autoencoders

Similar to other autoencoders, contractive autoencoders perform task of learning a representation of the image while passing it through a bottleneck and reconstructing it in the decoder.

The contractive autoencoder also has a regularization term to prevent the network from learning the identity function and mapping input into the output.

Contractive autoencoders work on the basis that similar inputs should have similar encodings and a similar latent space representation. It means that the latent space should not vary by a huge amount for minor variations in the input.

To train a model that works along with this constraint, we have to ensure that the derivatives of the hidden layer activations are small with respect to the input.

should be as small as possible.

An important thing to note in the loss function (formed from the norm of the derivatives and the reconstruction loss) is that the two terms contradict each other.

While the reconstruction loss wants the model to tell differences between two inputs and observe variations in the data, the frobenius norm of the derivatives says that the model should be able to ignore variations in the input data.

Putting these two contradictory conditions into one loss function enables us to train a network where the hidden layers now capture only the most essential information. This information is necessary to separate images and ignore information that is non-discriminatory in nature, and therefore, not important.

The gradient is summed over all training samples, and a frobenius norm of the same is taken.

4. Denoising Autoencoders

Denoising autoencoders, as the name suggests, are autoencoders that remove noise from an image.

As opposed to autoencoders we’ve already covered, this is the first of its kind that does not have the input image as its ground truth.

In denoising autoencoders, we feed a noisy version of the image, where noise has been added via digital alterations. The noisy image is fed to the encoder-decoder architecture, and the output is compared with the ground truth image.

The denoising autoencoder gets rid of noise by learning a representation of the input where the noise can be filtered out easily.

While removing noise directly from the image seems difficult, the autoencoder performs this by mapping the input data into a lower-dimensional manifold (like in undercomplete autoencoders), where filtering of noise becomes much easier.

Essentially, denoising autoencoders work with the help of non-linear dimensionality reduction. The loss function generally used in these types of networks is L2 or L1 loss.

5. Variational Autoencoders

Standard and variational autoencoders learn to represent the input just in a compressed form called the latent space or the bottleneck.

Therefore, the latent space formed after training the model is not necessarily continuous and, in effect, might not be easy to interpolate.

For example—

This is what a variational autoencoder would learn from the input:

While these attributes explain the image and can be used in reconstructing the image from the compressed latent space, they do not allow the latent attributes to be expressed in a probabilistic fashion.

Variational autoencoders deal with this specific topic and express their latent attributes as a probability distribution, leading to the formation of a continuous latent space that can be easily sampled and interpolated.

When fed the same input, a variational autoencoder would construct latent attributes in the following manner:

The latent attributes are then sampled from the latent distribution formed and fed to the decoder, reconstructing the input.

The motivation behind expressing the latent attributes as a probability distribution can be very easily understood via statistical expressions.

Here’s how this works—

We aim at identifying the characteristics of the latent vector z that reconstructs the output given a particular input. Effectively, we want to study the characteristics of the latent vector given a certain output x[p(z|x)].

While estimating the distribution becomes impossible mathematically, a much simpler and easier option is to build a parameterized model that can estimate the distribution for us. It does it by minimizing the KL divergence between the original distribution and our parameterized one.

Expressing the parameterized distribution as q, we can infer the possible latent attributes used in the image reconstruction.

Assuming the prior z to be a multivariate Gaussian model, we can build a parameterized distribution as one containing two parameters, the mean and the variance. The corresponding distribution is then sampled and fed to the decoder, which then proceeds to reconstruct the input from the sample points.

But—

While this seems easy in theory, it becomes impossible to implement because backpropagation cannot be defined for a random sampling process performed before feeding the data to the decoder.

To get by this hurdle, we use the reparameterization trick—a cleverly defined way to bypass the sampling process from the neural network.

What is it all about?

In the reparameterization trick, we randomly sample a valueε from a unit Gaussian and then scale this by the latent distribution varianceσ and shift it by the mean μ of the same.

Now, we have left behind the sampling process as something done outside what the backpropagation pipeline handles, and the sampled value ε acts just like another input to the model, that is fed at the bottleneck.

A diagrammatic view of what we attain can be expressed as:

The variational autoencoder thus allows us to learn smooth latent state representations of the input data.

To train a VAE, we use two loss functions: the reconstruction loss and the other being the KL divergence.

While reconstruction loss enables the distribution to correctly describe the input, by focusing only on minimizing the reconstruction loss, the network learns very narrow distributions—akin to discrete latent attributes.

The KL divergence loss prevents the network from learning narrow distributions and tries to bring the distribution closer to a unit normal distribution.

The summarised loss function can be expressed as:

The primary use of variational autoencoders can be seen in generative modeling.

Sampling from the latent distribution trained and feeding the result to the decoder can lead to data being generated in the autoencoder.

A sample of MNIST digits generated by training a variational autoencoder is shown below:

Applications of autoencoders

Now that you understand various types of autoencoders, let’s summarize some of their most common use cases.

1. Dimensionality reduction

Undercomplete autoencoders are those that are used for dimensionality reduction.

These can be used as a pre-processing step for dimensionality reduction as they can perform fast and accurate dimensionality reductions without losing much information.

Furthermore, while dimensionality reduction procedures like PCA can only perform linear dimensionality reductions, undercomplete autoencoders can perform large-scale non-linear dimensionality reductions.

2. Image denoising

Autoencoders like the denoising autoencoder can be used for performing efficient and highly accurate image denoising.

Unlike traditional methods of denoising, autoencoders do not search for noise, they extract the image from the noisy data that has been fed to them via learning a representation of it. The representation is then decompressed to form a noise-free image.

Denoising autoencoders thus can denoise complex images that cannot be denoised via traditional methods.

3. Generation of image and time series data

Variational Autoencoders can be used to generate both image and time series data.

The parameterized distribution at the bottleneck of the autoencoder can be randomly sampled to generate discrete values for latent attributes, which can then be forwarded to the decoder,leading to generation of image data. VAEs can also be used to model time series data like music.

💡 Pro tip: Check out a [recent application of VAEs](https://magenta.tensorflow.org/music-vae) in the domain of musical tone generation.

4. Anomaly Detection

Undercomplete autoencoders can also be used for anomaly detection.

For example—consider an autoencoder that has been trained on a specific dataset P. For any image sampled for the [training dataset](https://www.v7labs.com/blog/quality-training-data-for-machine-learning-guide), the autoencoder is bound to give a low reconstruction loss and is supposed to reconstruct the image as is.

For any image which is not present in the training dataset, however, the autoencoder cannot perform the reconstruction, as the latent attributes are not adapted for the specific image that has never been seen by the network.

As a result, the outlier image gives off a very high reconstruction loss and can easily be identified as an anomaly with the help of a proper threshold.

Autoencoders in a nutshell: Key Takeaways

Well, that was a lot to take in. Let’s do a quick recap of everything you've learned in this guide:

An autoencoder is an unsupervised learning technique for neural networks that learns efficient data representations (encoding) by training the network to ignore signal “noise.”

Autoencoders can be used for image denoising, image compression, and, in some cases, even generation of image data.

While autoencoders might seem easy at the first glance (as they have a very simple theoretical background), making them learn a representation of the input that is meaningful is quite difficult.

Autoencoders like the undercomplete autoencoder and the sparse autoencoder do not have large scale applications in [computer vision](https://www.v7labs.com/blog/what-is-computer-vision) compared to VAEs and DAEs which are still used in works since being proposed in 2013 (by Kingmaet al).

4.What are undercomplete and overcomplete autoencoders? What is the main risk of an excessively undercomplete autoencoder? What about the main risk of an overcomplete autoencoder?

Autoencoders. We will be going over the following parts.

Generalities about Autoencoders

Denoising Autoencoders (DAE)

Implementation and training of a DAE with Keras

Generalities about Autoencoders

If we had to summarize them in one sentence, it would probably sound like:

Autoencoders are neural network trained in an unsupervised way to attempt to copy inputs to outputs.

Yes I know, it may seem quite easy and useless. However, we will see that this is neither trivial nor pointless. In fact, Autoencoders are deep models capable of learning dense representations of the input. These representations are called latent representations or codings.

An Autoencoder has two distinct components :

An encoder: This part of the model takes in parameter the input data and compresses it. E(x) = c where x is the input data, c the latent representation and E our encoding function.

A decoder: This part takes in parameter the latent representation and try to reconstruct the original input. D(c) = x’ where x’ is the output of the decoder and D our decoding function

Undercomplete Autoencoders

Throughout the training phase, the goal is for our network to be able to learn how to reconstruct our input data. The following figure illustrates this idea by showing the Autoencoder model architecture.

However, most of the time, it is not the output of the decoder that interests us but rather the latent space representation. We hope that training the Autoencoder end-to-end will then allow our encoder to find useful features in our data.

The decoder, , is used to train the autoencoder end-to-end, but in practical applications, we often care more about the encoder and the codings.

To highlight important properties, one can, for example, constrain the latent space to be smaller than the dimension of the inputs. In this case, our model is an Undercomplete Autoencoders. In the majority of cases we work with this type of autoencoders since one of the main applications of this architecture is dimensionality reduction.

The learning process is quite regular, it aims at minimizing a loss function.

There are different metrics to quantify this loss function such as the Mean Square Error or the cross-entropy (when the activation function is a sigmoid for instance). This loss must penalize the reconstruction for being dissimilar from x.

Linear Autoencoders & Principal Component Analysis

So one of the main applications of Autoencoders is dimensionality reduction, just like a Principal Component Analysis (PCA). In fact, if the decoder is linear and the cost function is the Mean Square Error, an Autoencoder learns to span the same subspace as the PCA.

Comparing the two methods or explaining them in detail is outside the scope of this article. However, if you are interested, I recommend you to read [this blog post](https://towardsdatascience.com/a-one-stop-shop-for-principal-component-analysis-5582fb7e0a9c) ([this one](https://towardsdatascience.com/autoencoders-vs-pca-when-to-use-which-73de063f5d7) is also very interesting) which gives you a first intuition of PCA as well as [these excellent slides](https://www.cs.toronto.edu/~urtasun/courses/CSC411/14_pca.pdf) which compare the two methods.

But if one were to summarize the two methods…

PCA: In PCA, we also seek to minimize the gap between input data and its reconstruction by measuring and minimizing the distance between the two. (For example the Euclidian distance.)

N.B: Vectors of the decoding matrix must have unit norm and be orthogonal

This optimization problem may be solved using Singular Value Decomposition. Specifically, the optimal P is given by the eigenvectors of the X covariance matrix corresponding to the largest eigenvalues. Again the full demonstration of that property is outside the scope of this tutorial.

Autoencoders: If we choose to train them with the Mean Square Error, then we aim at minimizing

In the case where f and g are linears, the loss function becomes

Both methods have the same objective function, which is convex, but uses two different ways to reach it. Indeed, Autoencoders are feedforward neural networks and are therefore trained as such with, for example, a Stochastic Gradient Descent.

In other words, the Optimal Solution of Linear Autoencoder is the PCA.

Now that the presentations are done, let’s look at how to use an autoencoder to do some dimensionality reduction.

For the sake of simplicity, we will simply project a 3-dimensional dataset into a 2-dimensional space. The first step to do such a task is to generate a 3D dataset. It is possible to do it simply with the following code.

Now we have to create a Linear Autoencoder to perform PCA and this is possible with the following code.

Even if the code, as well as the graph, are quite self-explanatory, let’s take a closer look at them.

First, we define the encoder, a dense layer of 2 neurons that accepts inputs of dimension 3 (according to our dataset). So here we constrained the latent-space representation to be of dimension 2 (the output of the encoder).

Then, we define the decoder, also a dense layer but of 3 neurons this time because we want to reconstruct our 3-dimensional input at the output of the decoder.

The combination of the two forms the Autoencoder.

All that remains is to train it using data as inputs and targets. (We want to reconstruct our inputs remember ?)

And finally, the Autoencoder find the best 2D plane to project the data into while preserving as much variance as possible. Now let’s see the projection of our data!

So, the Autoencoder has reduced the dimensionality of our problem in the same way as PCA would have done.

Regularized Autoencoders

The problem is if we give our network too much capacity with many hidden layers, our model will be able to learn the task of copying data in inputs without extracting important information. In this case, the Autoencoder could not learn anything and would be overfitting. This phenomenon occurs as well with Undercomplete AE as Overcomplete AE (when the codings have higher dimensions than the inputs).  
We want to be able to give our model capacity and not restrict it to just small networks to limit the number of parameters.

To do this, we use Regularized Autoencoders which encourages the model to develop new properties and to generalize better.

There are many different types of Regularized AE, but let’s review some interesting cases.

Sparse Autoencoders: it is simply an AE trained with a sparsity penalty added to his original loss function.

Sparse AEs are widespread for the classification task for instance. This sparsity penalty is simply a regularizer term added to a feedforward network.

Denoising Autoencoders: Adding noise (Gaussian for example) to the inputs forces our model to learn important features from our data.

Right now we are going to dive deeper into the concept of Denoising Autoencoders (DAE)

Denoising Autoencoders (DAE)

This type of Autoencoder is an alternative to the concept of regular Autoencoder we just discussed, which is prone to a high risk of overfitting. In the case of a Denoising Autoencoder, the data is partially corrupted by noises added to the input vector in a stochastic manner. Then, the model is trained to predict the original, uncorrupted data point as it.

Let’s explain the training process at stake here.

An input is sampled from our dataset.

A corrupted version of this input is sampled from a stochastic mapping M( ̃x|x)

(x, ̃x) is used as a training example

Just like regular AE, our DAE is a feedforward network that can be trained with a gradient-based approximate minimization on the negative log-likelihood

What does a Denoising AE learn?

Image by author (inspired by I. Goodfellow, Y. Bengio, A. Courville [Deep Learning](https://www.deeplearningbook.org/))

You can see on the image here some data represented by the blue dots. Our corrupted data will remain in the black circle of equiprobable corruption.

During training, the aim is to minimize the negative log-likelihood cost function.

This optimization leads to minimizing the distance between the corrupted input and the black manifold which characterizes our inputs.

Thus, our model learns a reconstruction vector field D(E(x))-x, some of these vectors are represented by the red arrows.

Again, the goal here is not to make a formal demonstration but rather to give a first understanding of how we can train a Denoising Autoencoder.

So now with the image and the explanations above, we have a better understanding of a DAE’s training.

To conclude this theoretical part let us recall the three main advantages of this architecture:

Learns more robust filters

Prevents from learning a simple identify function

Decreases the risk of overfitting that can be problematic with regular AE

Let’s now go into practice and implement a DAE!

Implementation and training of a DAE with Keras

Before starting, it is necessary to introduce a new concept that has not been mentioned so far.

If the problem is to deal with images, Autoencoders with dense layers will not be enough powerful to learn important features on our data.

To solve this problem, we work with Convolutional Autoencoders. In this case, the encoder is a regular CNN network that reduces the spatial dimension of the inputs by increasing the depth. The decoder must therefore do the reverse operation. To do this operation, one solution is to use transpose convolutional layers.

For this implementation, we will work with the Fashion MNIST dataset and some Gaussian Noise as the corruption mapping distribution. I know the exercise is not the most original but now with your theoretical background, you will see it from a brand new light.

First, we load the dataset, create a validation set from the training set and visualize some data. The code is quite straightforward.

Good, no worries here, we have our dataset Fashion MNIST. We can start implementing our encoder without forgetting to add a layer of Gaussian noise. To implement our encoder, here is the code!

Again, the code is quite self-explanatory, we define our model. Then, we reshape our inputs and add some Gaussian Noise and pass it to a regular CNN with Convolutional layers follower by Pooling layers. We also have a graph of this architecture.

Denoising Encoder architecture

Now, the decoder must do the reverse operation and the whole architecture remain symmetrical. Again, the code to implement the decoder is straightforward.Now, as in the first part, all we have to do is assemble the encoder and decoder and train the Autoencoder end-to-end. The optimization will be done on a binary cross-entropy.

Let’s now visualize the reconstructions that our Autoencoder is able to do.

5.How do you tie weights in a stacked autoencoder? What is the point of doing so?

I am currently reading about Autoencoders. From what I understand so far, when we are dealing with a symmetrical autoencoder, a good practice is to tie the weights of the decoder layers to the weights of the encoder layers. With this technique we halve the number of weights in our model, speeding training and limiting the risk of overfitting, since we don't have to learn the weights of the decoder anymore, we just learn the weights of the encoder and set the weights of the decoder accordingly. What I don't understand is the value assigned to the weights of the decoder. Let's say that the autoencoder has a total of NN layers (without counting the input layer), so layer 11 is the first hidden layer, layer N/2N/2 is the codding layer, and layer NN is the output layer. Let's also say that WLWL represents the connection weights of the LthLth layer.

From what I've read so far, if we tie the weights of the decoder layers to the weights of the encoder layer, then the weights of the decoder layers will be:

WN−L+1=W⊺LWN−L+1=WL⊺

where ⊺⊺ denotes the transpose and LL ranges from 1,2,...,N/21,2,...,N/2.

This is my confusion. Why are we using the transpose of the encoder layers weights as the decoder layer weights? Since the encoder has the job of projecting our data into a lower dimension, and then the decoder maps this projection back to the original representation of the data, wouldn't it make more sense to have the weights of the decoder be the inverse of the encoder weights, not the transpose? (Or at least the pseudo-inverse)

If we would use the inverse, that weight matrix would try to project the data back to its original space, it would try to undo the initial projection, and therefore it would try to recreate the initial input, which is what the autoencoder is trying to achieve. But we're not using the inverse. We're using the transpose. And this doesn't seem to make any sense to me.

6.What is a generative model? Can you name a type of generative autoencoder?

we discuss seven types of [generative models](https://analyticsindiamag.com/how-to-secure-deep-learning-models-from-adversarial-attacks/), which are listed below in alphabetical order-

Autoregressive Models

Autoregressive Model or AR model is when a value from a time series is regressed on previous values from that same time series. The order of an autoregression is the number of immediately preceding values in the series that are used to calculate the value at the present time. In simple words, Autoregressive models predict future values based on past values. These models are flexible at handling a wide range of different time-series patterns.

THE BELAMY

Sign up for your weekly dose of what's up in emerging technology.

Top of Form

Email

SIGN UP

Bottom of Form

Know more [here](https://online.stat.psu.edu/stat501/lesson/14/14.1).

Bayesian Network

Bayesian Network or Bayes Network is a [generative probabilistic graphical model](https://analyticsindiamag.com/latest-model-that-might-replace-gans-to-create-deepfakes/)that allows efﬁcient and effective representation of the joint probability distribution over a set of random variables. Bayes Network consists of two main parts, which are structure and parameters. The structure is a directed acyclic graph (DAG), and the parameters consist of conditional probability distributions associated with each node. This network can be used for various applications, such as time series prediction, anomaly detection, reasoning and other such.

Know more [here](https://www.uib.no/en/rg/ml/119695/bayesian-networks).

Generative Adversarial Networks

Generative Adversarial Networks or GANs are popular [generative models](https://analyticsindiamag.com/the-7-key-steps-to-build-your-machine-learning-model/) that include two parts, generators and discriminators. This model works by estimating generative models via an adversarial process. The generative model captures the data distribution, and the discriminative model estimates the probability that a sample came from the training data rather than the generative model. GANs are one one of the trending generative models that have been used to create images of humans that do not exist.

Gaussian Mixture Model

Gaussian Mixture Model is a generative probabilistic model, which assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters. GMMs are commonly used as a [parametric model](http://leap.ee.iisc.ac.in/sriram/teaching/MLSP_16/refs/GMM_Tutorial_Reynolds.pdf) of the probability distribution of features in a biometric system, which includes vocal-tract related spectral features in a speaker recognition system. Thus, GMM parameters are estimated from training data using the iterative Expectation-Maximisation (EM) algorithm or Maximum A Posteriori (MAP) estimation from a well-trained prior model.

Hidden Markov Model

A [Hidden Markov Model](https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2766791/) (HMM) is a statistical model that can be used to describe the evolution of observable events that depend on internal factors, which are not directly observable. The model is popularly known for their effectiveness in modelling the correlations between adjacent symbols, domains, or events, and they have been extensively used in various fields, especially in speech recognition and digital communication. A Hidden Markov Model consists of two stochastic processes, which are an invisible process of hidden states and a visible process of observable symbols.

Know more [here](https://web.stanford.edu/~jurafsky/slp3/A.pdf).

Latent Dirichlet Allocation (LDA)

Latent Dirichlet Allocation or LDA is a generative probabilistic model with collections of discrete data such as text corpora. LDA is a three-level hierarchical Bayesian model, in which each item of a collection is modelled as a finite mixture over an underlying set of topics. The model has applications to various problems, including collaborative filtering, content-based image retrieval, among others.

Know more [here](http://www.cse.cuhk.edu.hk/irwin.king/_media/presentations/latent_dirichlet_allocation.pdf).

Variational Autoencoders (VAEs)

Variational Autoencoders (VAEs) have been one of the most popular approaches to unsupervised learning of complicated distributions. They are built on top of standard function approximators, which are neural networks and can be trained with stochastic gradient descent. The application of VAEs includes generating various kinds of complicated data, including handwritten digits, faces, CIFAR images, predicting the future from static images and more.

7.What is a GAN? Can you name a few tasks where GANs can shine?

A generative adversarial network (GAN) is a machine learning (ML) model in which two neural networks compete with each other to become more accurate in their predictions. GANs typically run unsupervised and use a cooperative zero-sum game framework to learn.

None of these people are real! These faces were generated by a [computer vision](https://courses.analyticsvidhya.com/courses/computer-vision-using-deep-learning-version2?utm_source=blog&utm_medium=top-5-GANs-applications) technique called GANs, or Generative Adversarial Networks. Full marks to you if you guessed it correctly!

The term ‘GAN’ was introduced by the Ian Goodfellow in 2014 but the concept has been around since as far back as 1990 (pioneered by Jürgen Schmidhuber). But it was only after Goodfellow’s paper on the subject that they gained popularity in the community. And since then, there’s been no looking back for GANs!

In fact, GANs are now ubiquitous. Data scientists and deep learning researchers use this technique to generate photorealistic images, change facial expressions, create computer game scenes, visualize designs, and more recently, even generate awe-inspiring artwork! That’s right – the recent news about AI generated art? It was GANs at work:

Lenovo, TVS, Convergytics, Ripik.AI and many more are hiring | Open to all Data Science Enthusiasts.[Register Now](https://datahack.analyticsvidhya.com/jobathon/?utm_source=blog_india&utm_medium=desktop_banner_between_articles&utm_campaign=27-Oct-2022||&utm_content=registrations)

In this article, we will look at five intriguing applications of GANs that are prevalent in the industry. You might even have come across a few of them without realizing how they worked. I have also provided links to research papers for each GAN application which I encourage you to check out.

To learn more about generative models and how GANs works from scratch, feel free to check out this detailed article:

[What are Generative Models and GANs? The Magic of Computer Vision](https://www.analyticsvidhya.com/blog/2020/01/generative-models-gans-computer-vision/?utm_source=blog&utm_medium=top-5-interesting-applications-gans-deep-learning)

Strap in – this is going to be fun!

Table of Contents

Introductory Explanation of GANs

Applications of GANs

GANs for Image Editing

Using GANs for Security

Generating Data using GANs

GANs for Attention Prediction

GANs for 3D Object Generation

Introductory Explanation of GANs

Right, we have a sense of what GANs can do. But how do they work? What goes on underneath all the wonderful applications this powerful algorithm produces? Let’s understand this using a popular example.

There’s a forger (who creates fake artistry) and an investigator tasked with detecting these fake artworks.

The task of this forger is to create fraudulent imitations of original paintings by famous artists (like Leonardo Da Vinci). If he/she can pass off this work as the original art piece, the forger can potentially net a lot of money.

On the other side of this situation, the art investigator’s task is to catch these forgers. How does he/she do it? The investigator knows what are the properties which set the original artist apart and what kind of painting he/she would have created. The investigator leverages this knowledge against the piece at hand to check if it is real or not.

This contest of forger vs investigator goes on, which ultimately makes world-class investigators (and unfortunately world-class forgers); a battle between good and evil.

Now, consider both forger and investigator as robots. When you train the forger to be a painter and the investigator to tell a  fake painting from the real one – you now have an algorithmic painter at hand! That’s essentially how GANs work on the inside. Awesome, aren’t they?

I haven’t got into the intricate details of GANs here. This is just the tip of the iceberg. If you are interested in learning more about GANs, you should go through this article:

[An introductory guide to Generative Adversarial Networks (GANs) and their promise!](https://www.analyticsvidhya.com/blog/2017/06/introductory-generative-adversarial-networks-gans/?utm_source=blog&utm_medium=top-5-GANs-applications)

Applications of GANs

Now that we have an intuition of how GANs work, let’s put on our exploration hats! It’s time to dive into the interesting applications of GANs that are commonly used in the industry right now.

GANs for Image Editing

Most image editing software these days don’t give us much flexibility to make creative changes in pictures. For example, let’s say you want to change the appearance of a 90-year-old person by changing his/her hairstyle. This can’t be done by the current image editing tools out there. But guess what? Using GANs, we can reconstruct images and attempt to change the appearance drastically.

This [amazing paper](https://arxiv.org/pdf/1611.06355.pdf) demonstrates this very cutting edge application.

Another similar application is image de-raining (or literally removing rainy texture from images). Want an example? Check out the below image taken from [this paper](https://github.com/hezhangsprinter/ID-CGAN).

Using GANs for Security

The rise of artificial intelligence has been wonderful for most industries. But there’s a real concern that has shadowed the entire AI revolution – cyber threats. Even deep neural networks are susceptible to being hacked.

A constant concern of industrial applications is that they should be robust to cyber attacks. There’s a lot of confidential information on the line! GANs are proving to be of immense help here, directly addressing the concern of “adversarial attacks”.

These adversarial attacks use a variety of techniques to fool deep learning architectures. GANs are used to make existing deep learning models more robust to these techniques. How? By creating more such fake examples and training the model to identify them. Pretty clever stuff.

8.What are the main difficulties when training GANs?

However, there exist major challenges in training of GANs, i.e., mode collapse, non-convergence and instability, due to inappropriate design of network architecture, use of objective function and selection of optimization algorithm.

GANs are able to learn distributions on complex high-dimensional data which made it efficient in images and audio processing. Nevertheless, in the training of GANs, some major challenges exist namely mode collapse, non-convergence, and instability.