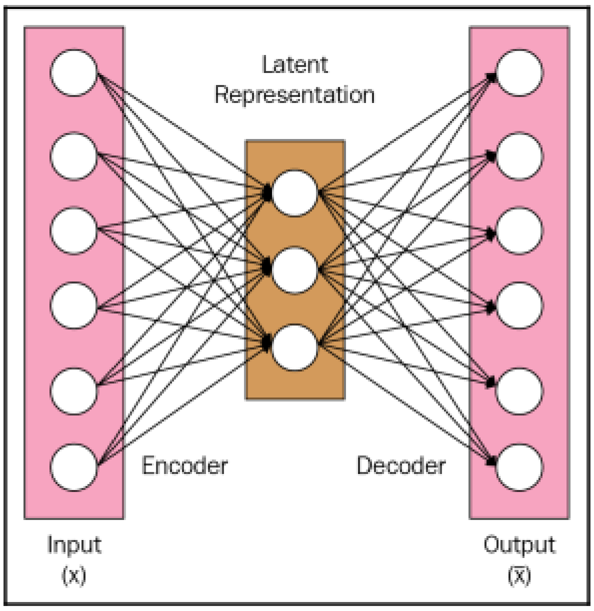
Remove Noisy From Images Using Autoencoders

Autoencoders represent a paradigm shift from the conventional neural networks. The goal of autoencoders is to learn a **Latent Representation** of the input. This representation is usually a compressed representation of the original input.

All autoencoder have an **Encoder** and a **Decoder**. The role of encoder is to encode the input to a learned, compressed representation, and the role of the decoder is to reconstruct the original input using the compressed representation.

The following diagram illustrates the architecture of a typical autoencoder:



Notice that, in the preceding diagram, we do not require a label *y*. This distinction means that autoencoders are a form of unsupervised learning.

Latent representation

The purpose of autoencoders is to compress the learned representation of the input. By forcing the learned representation to be compressed (that is, having smaller dimensions compared to the input), we essentially force the neural network to learn the most remarkable representation of the input. This ensures that the learned representation only captures the most relevant characteristics of the input.

With this latent representation learned by the autoencoder, we can then do the following:

- Reduce the dimensionality of the input data. The latent representation is a natural reduced representation of the input data.
- Remove any noise from the input data (known as denoising). Noise is not a remarkable characteristic and therefore should be easily identifiable by using the latent representation.

Autoencoders for data compression

So far we seen how autoencoders are able to learn a reduced representation of the input data and it is natural to think that they can do a good job at generalized data compression. However, they are poor at generalized data compression, such as image and audio compression,because the learned latent representation only represents the data on which it was trained. So, autoencoders only work well for images similar to those on which it was trained.

Furthermore, autoencoders are a "lossy" form of data compression, which means that the output from autoencoders will have less information when compared to the original input. These characteristics mean that autoencoders are poor at being generalized data compression techniques.

MNIST handwritten digits dataset

One of the datasets that we will use for this chapter is the MNIST handwritten digits dataset. The MNIST dataset contains 70000 samples of handwritten digits, each of 28 x 28 pixels. Each sample contains only one digit within the image, and all samples are labeled.

All MNIST digits are showed below:

We can see that the digits are definitely handwritten, and each 28 x 28 image captures only one digit. The autoencoder should be able to learn the compressed representation of these digits, that should be smaller than 28 x 28, and to reproduce the images using this compressed representation.

In order to build an autoencoder, we need to find the ideal size of the hidden layer (**Latent Representation**). Ideally, the size of the hidden layer should balance between being:

- Sufficiently *small* enough to represent a compressed representation of the input features
- Sufficiently *large* enough for the decoder to reconstruct the original input without too much loss

In other words, the size of the hidden layer is a hyperparameter that we need to select carefully to obtain the best results.

Building Autoencoders For MNIST Dataset

Before building our model, we have to preprocess our data. There are two preprocessing steps required:

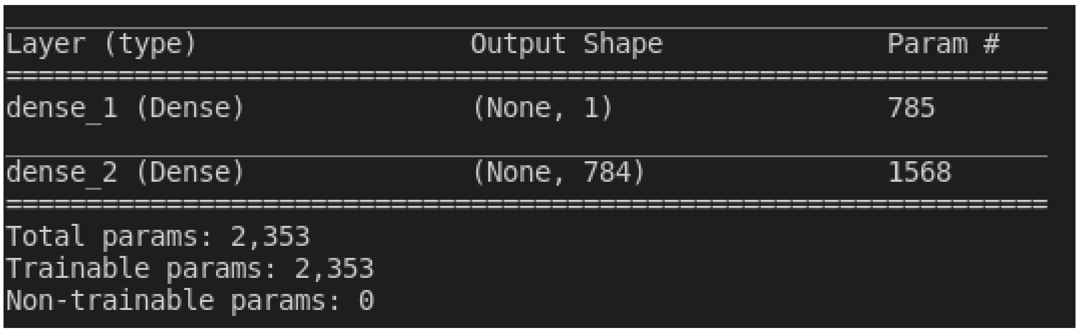
- Reshape the images from a 28 x 28 vector to (28*28) x 1 vector.

- Normalize the values of the vector between 0 and 1 from the current 0 to 255. This smaller range values makes it easier to train our neural network using the data.

After we done that step, we will start creating a basic autoencoder. From autoencoder diagram, we will clearly see that the hidden layer is a fully connected layer. The number of units we will use for now are just 1. The `input_shape` of hidden layer will be 28*28 and the `activation` function is the `relu` activation function.

The output layer is also a fully connected layer and the size of the output layer should be 28*28, since we are trying to output the original 28 x 28 image. We will use a `sigmoid` activation function for the output to constrain the values per pixel between 0 and 1.

The total parameters of model are:

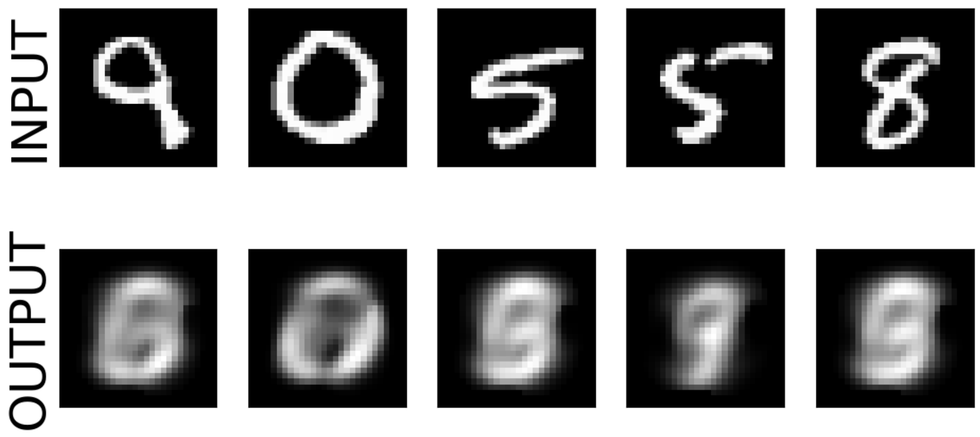


```
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 1)                 785
_________________________________________________________________
dense_2 (Dense)              (None, 784)               1568
=================================================================
Total params: 2,353
Trainable params: 2,353
Non-trainable params: 0
```

We will compile the model using the `adam` optimizer and `mean_square_error` as the **loss** function. The `mean_square_error` is useful in this case because we need a loss function that quantifies the pixel-wise contrast between the input and the output.

We will train the model for 10 epochs and we will use the train dataset as both the input(x) and output(y) since we are trying to train the auto encoder to produce output that is identical to the input.

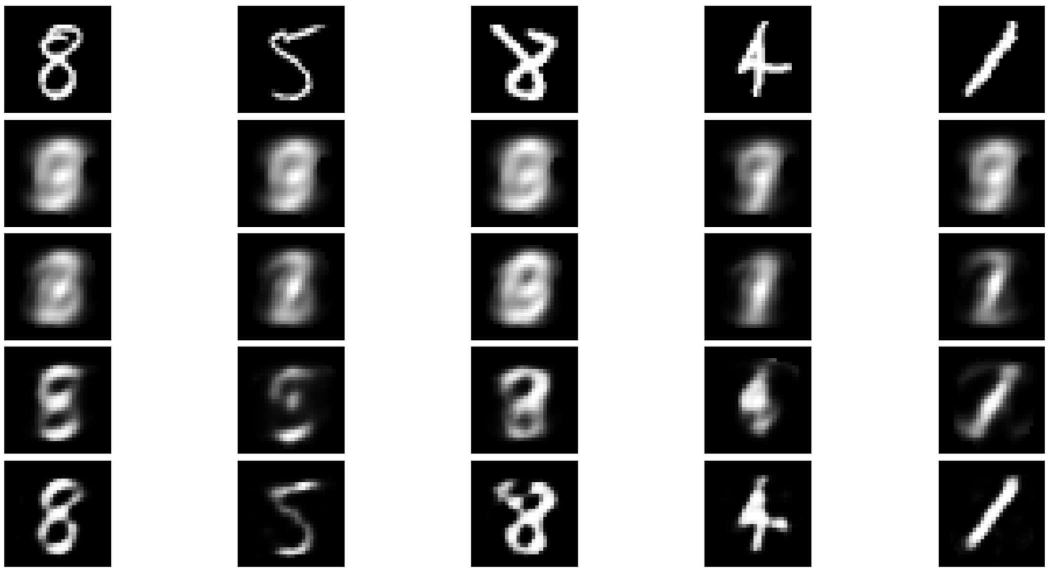
By selecting five random images from testing set, we plot the autoenconder output images as follow:



For 1 unit of hidden layer, we receive terrible results. The images are blurry and they don't resemble our original input images.

We will try to train more autoencoders with different hidden layer sizes to see how they perform. We will create and train another 3 basic autoencoder models with 2,8 and 32 nodes in the hidden layer.

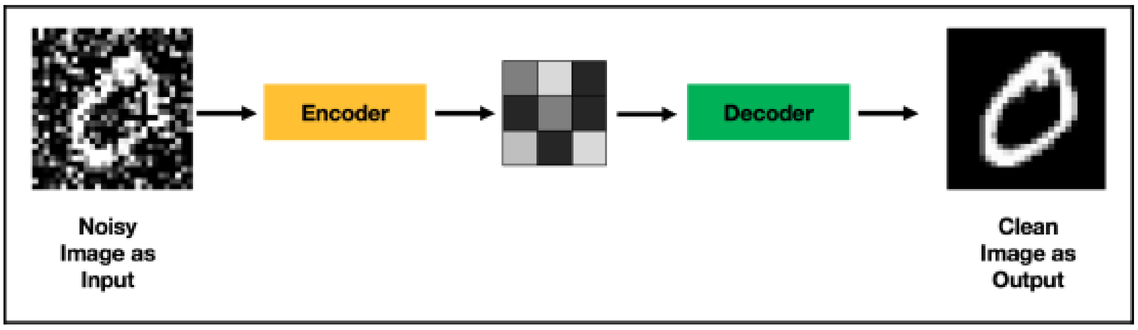
After fitting, we choose 5 random outputs for each model and we get the following output:



We can clearly see a transition as we augment the number of nodes in the hidden layer. Gradually, we see that the output images become clearer and closer to the original input as we increase the number of nodes in the hidden layer.

Denoising autoencoders

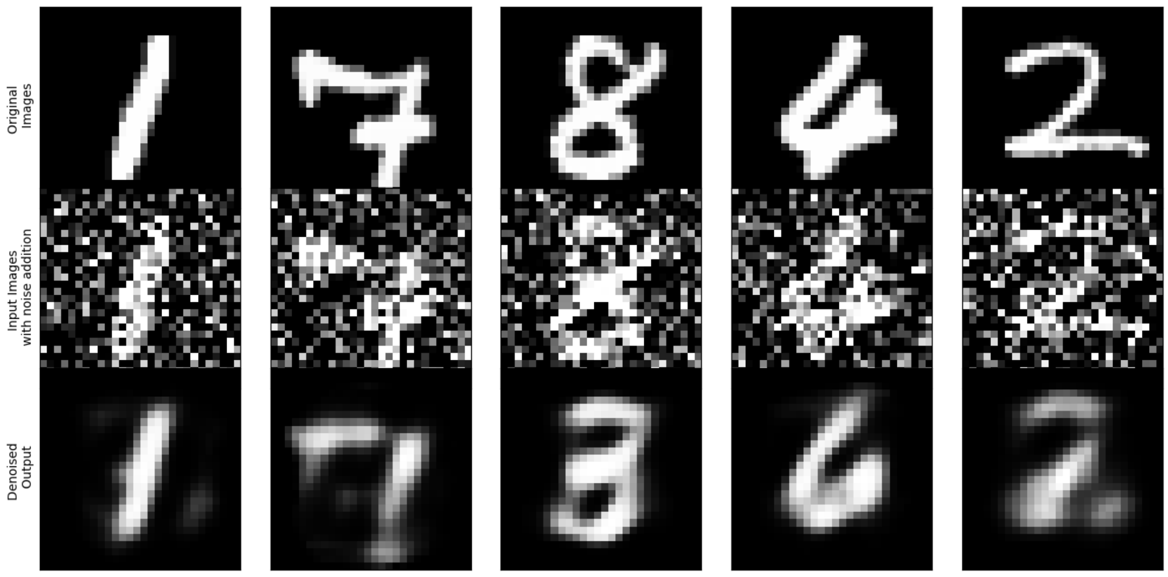
Another interesting application of auto-encoders is image denoising. Image noise is defined as a random variations of brightness in an image that may originate from the sensors of digital camera. We can train auto-encoders for image denoising. Instead of using the same input and output when training conventional auto encoders, we use a noisy image as the input and a clean reference image for the auto-encoder to compare its output against.



During the training process, the autoencoder will learn, through the `loss` function, that the noises in the image should be not part of the output, and will learn to output a clean image. In other words, the latent representation should only contain non-noise elements.

First, we will add noise to each pixel in the original image. The noise will have a normal distribution with zero mean value and 0.5 standard deviation. After noise addition, we will chop the input training and test data values between [0,1].

The model prediction of a basic auto-encoder with 16 hidden layer units is showed below:

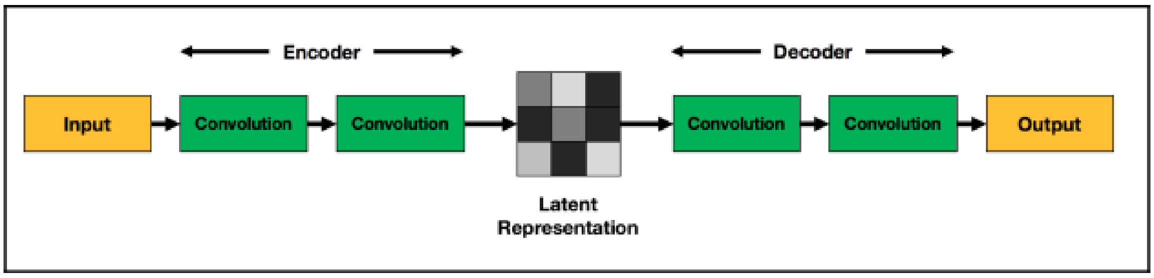


The basic denoising auto-encoder is perfectly capable of removing noise, but it doesn't do a very good job at reconstructing the original image.

Deep convolutional denoising auto-encoder

In order to overcome previous disadvantages, we will use multiple layers and instead of a fully connected dense layer, we use convolutional layers. For denoising auto-encoders, convolutial layers always work better than dense layers for image classification tasks.

The architecture of a deep convolutional autoencoder is illustrated in the following diagram:

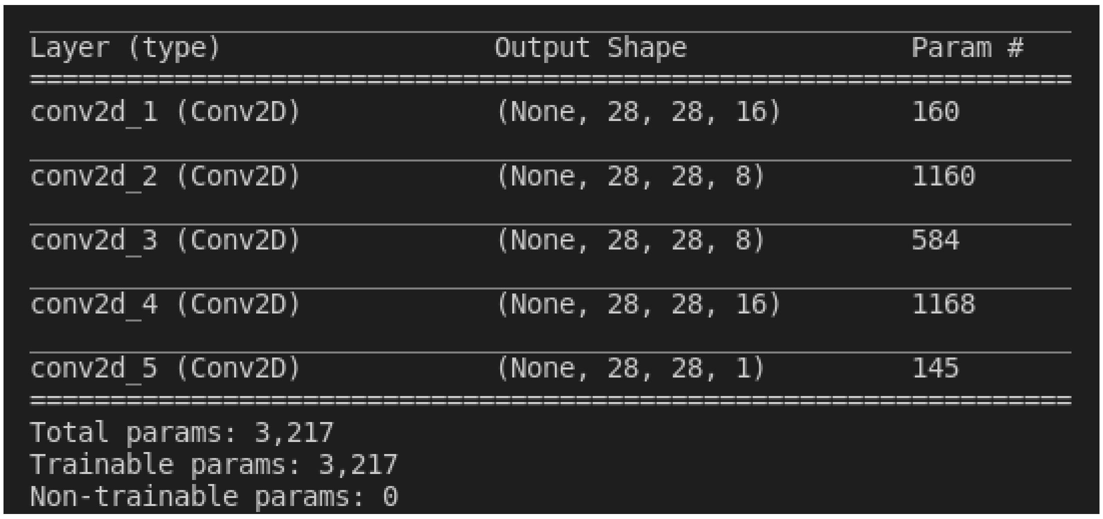


The parameters we need to define about convolutional layers are:

- **Number Of Filters**: We use an increasing number of filters for each layer in the decoder. We use 16 filters for the first convolutional layer and 8 filters for the second convolutional layer in the encoder. Conversely, we use 8 filters for the first convolutional layer and 16 filters for the second layer in the decoder.
- **Filter size**: We use a 3 x 3 typical filter size
- **Padding**: For auto-encoders, we use a same padding. This ensures that the height and width of successive layers remains the same. This is useful because we need to ensure that the dimensions of the final output is the same as the input.

The activation function of encoder and decoder hidden layer nodes is `relu`. As for the output layer, it only has one filter, as we trying to output a 28 x 28 x 1 image and the activation function that we use is `sigmoid`.

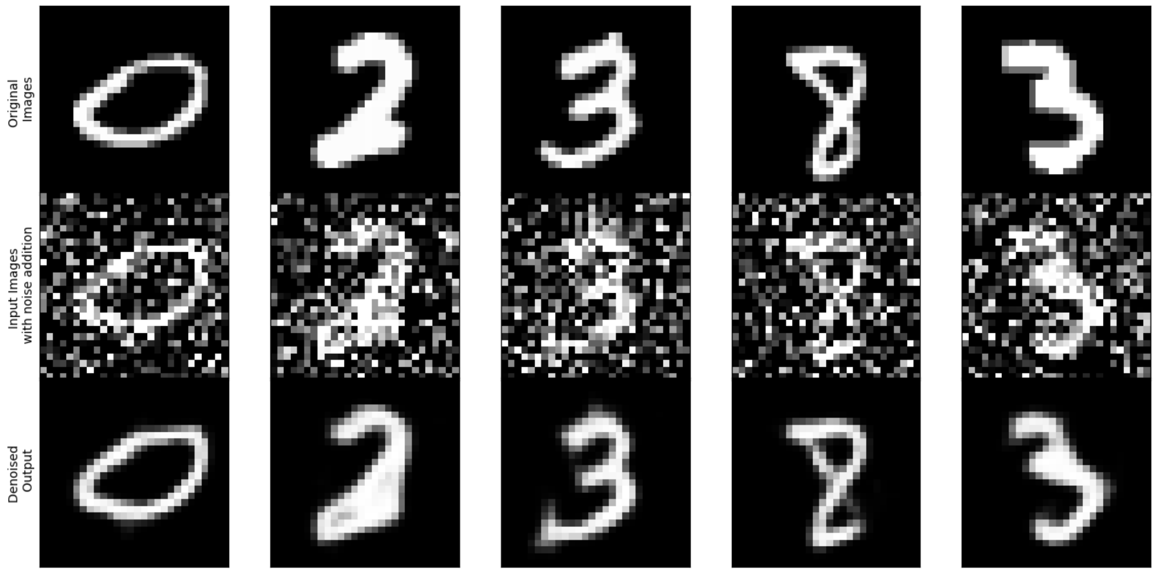
The total parameters of model are:



```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 28, 28, 16)        160
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 28, 28, 8)         1160
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 28, 28, 8)         584
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 28, 28, 16)        1168
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 28, 28, 1)         145
=================================================================
Total params: 3,217
Trainable params: 3,217
Non-trainable params: 0
```

We will compile this model with `adam` optimizer and as loss function, we use `binary_crossentropy` since we need [0,1] pixel values. The model will be trained under 10 epochs.

The output of the network is:



We can observe that the denoised output from deep convolutional auto-encoder is similar to the original images.

We can also improve the complexity of our model by building a deeper network with more layers, and by using more filters in each convolutional layer.