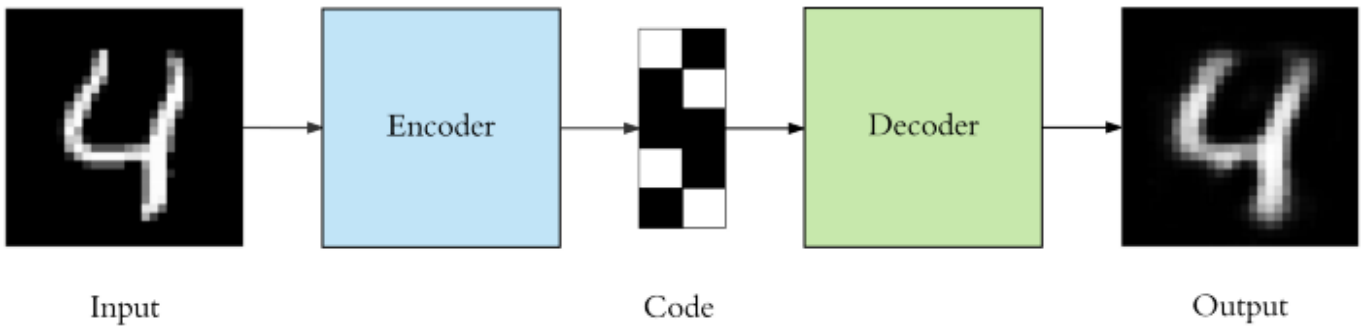
Autoencoders.

The code for this article is available here as a Jupyter notebook, feel free to download and try it out yourself.

1. Introduction

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.



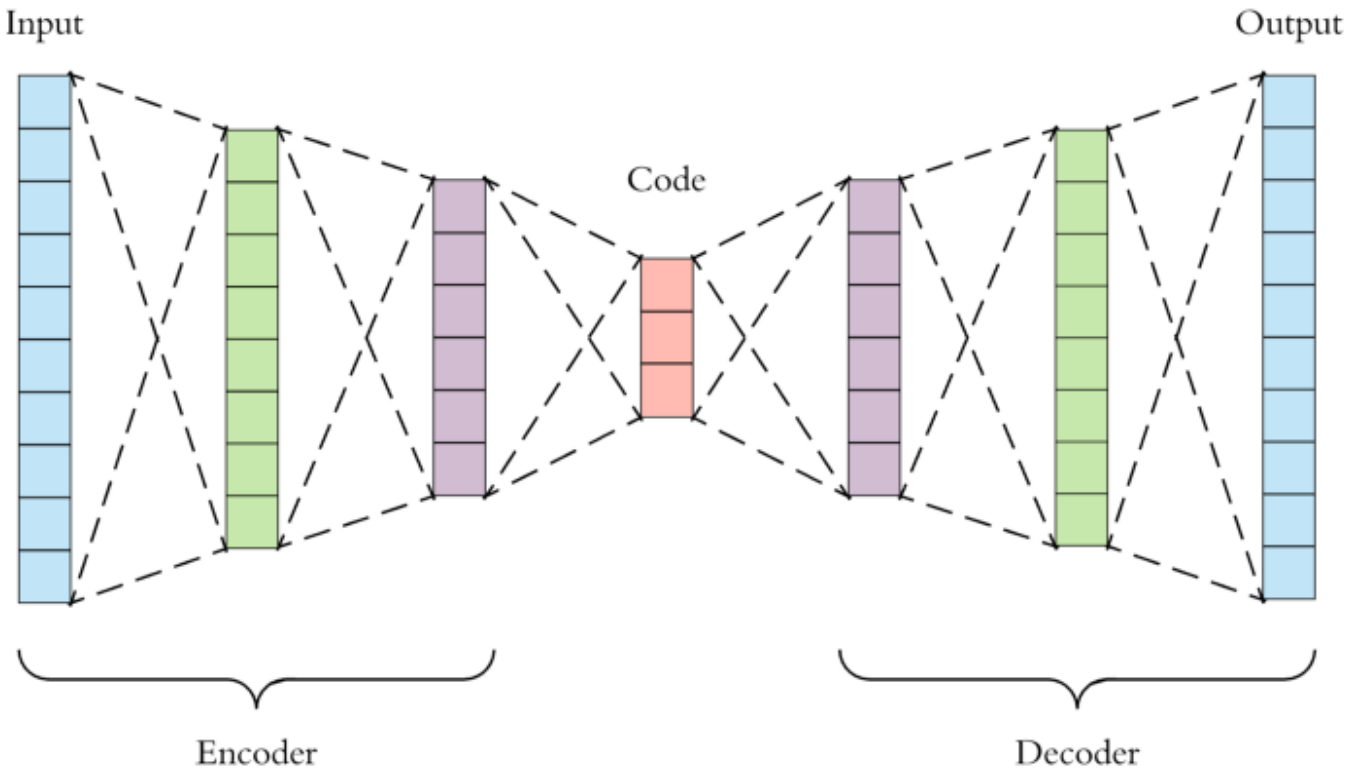
To build an autoencoder we need 3 things: an encoding method, decoding method, and a loss function to compare the output with the target. We will explore these in the next section.

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. 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.

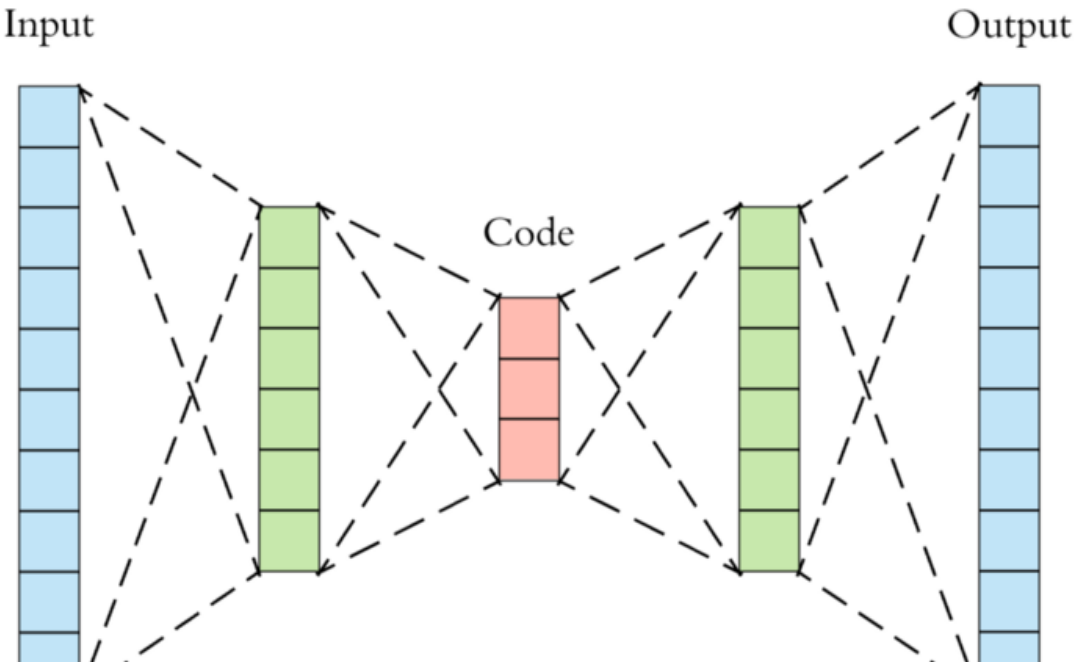
There are 4 hyperparameters that we need to set before training an autoencoder:

- 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.

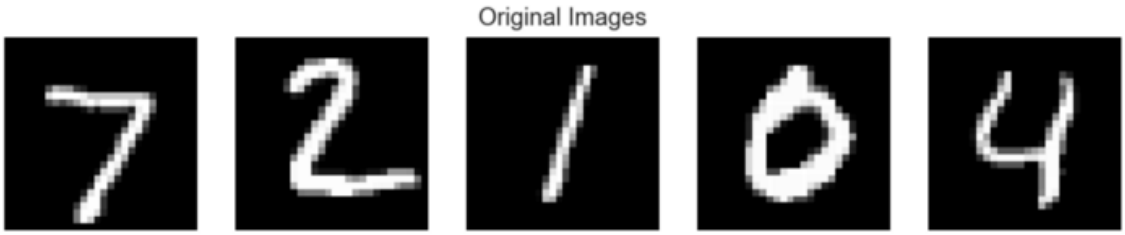
Autoencoders are trained the same way as ANNs via backpropagation. Check out the introduction of Part 1 for more details on how neural networks are trained, it directly applies to the autoencoders.

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 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 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 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:

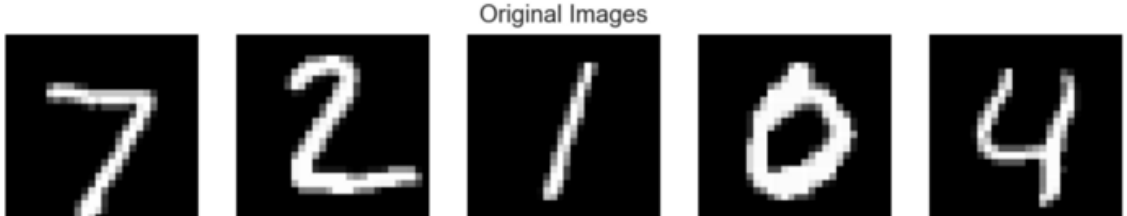
```
model.fit(x_train, x_train)
```

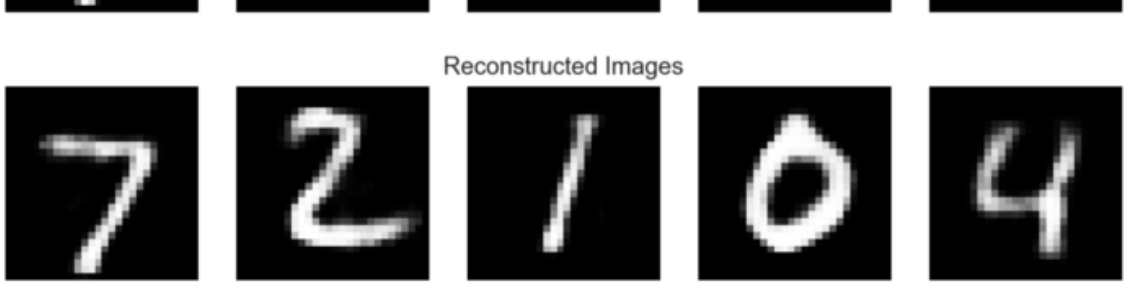
Remember that the targets of the autoencoder are the same as the input. That's why we supply the training data as the target.

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.

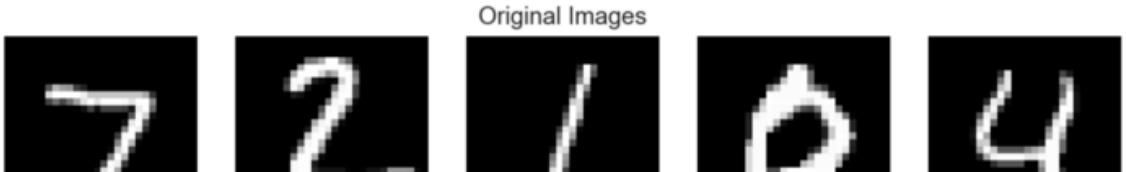
Advice

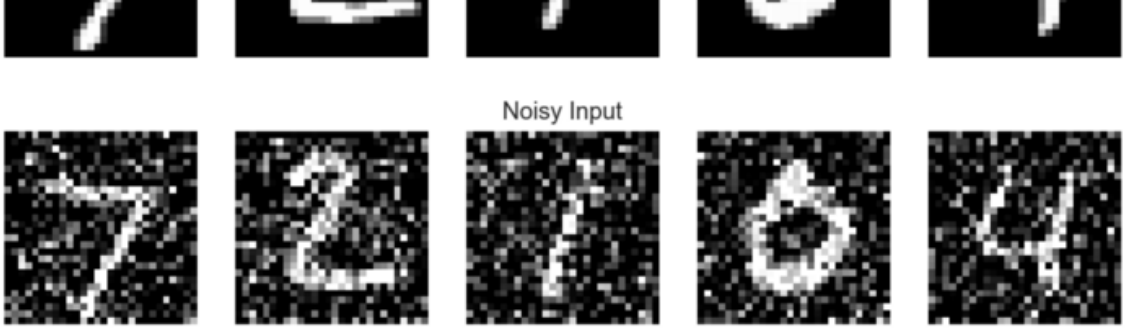
We have total control over the architecture of the autoencoder. We can make it very powerful by increasing the number of layers, nodes per layer and most importantly the code size. Increasing these hyperparameters will let the autoencoder to learn more complex codings. But we should be careful to not 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.

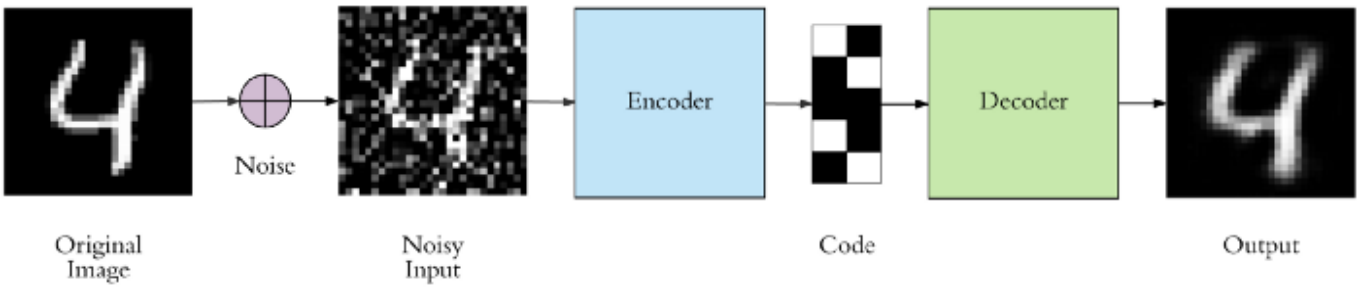
4. Denoising Autoencoders

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)
```

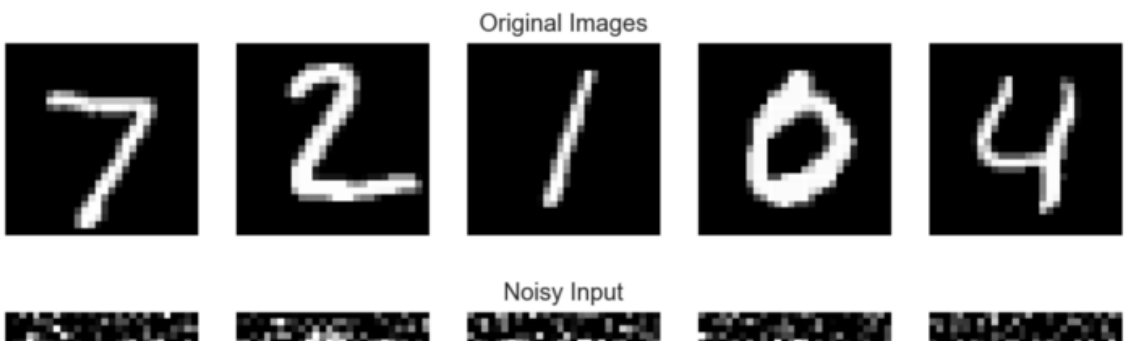
Denoising autoencoder is trained as:

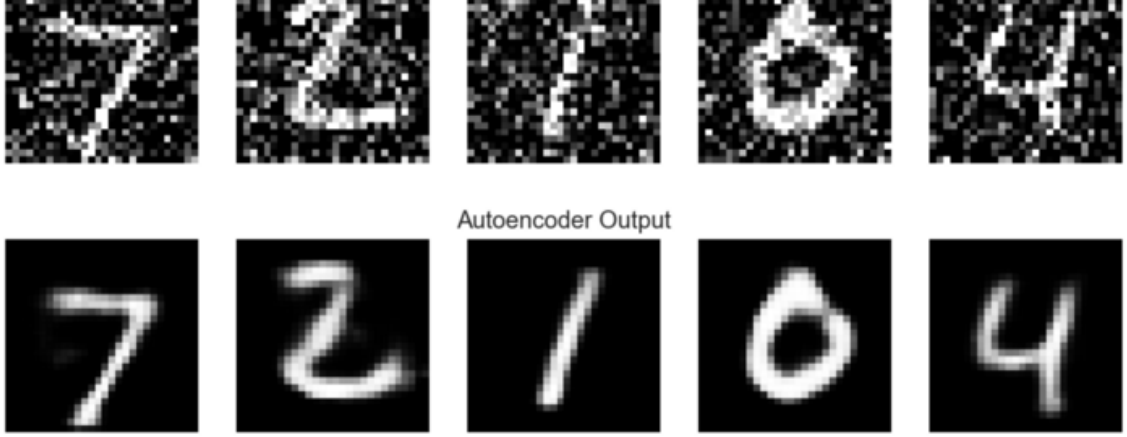
```
autoencoder.fit(x_train_noisy, x_train)
```

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:

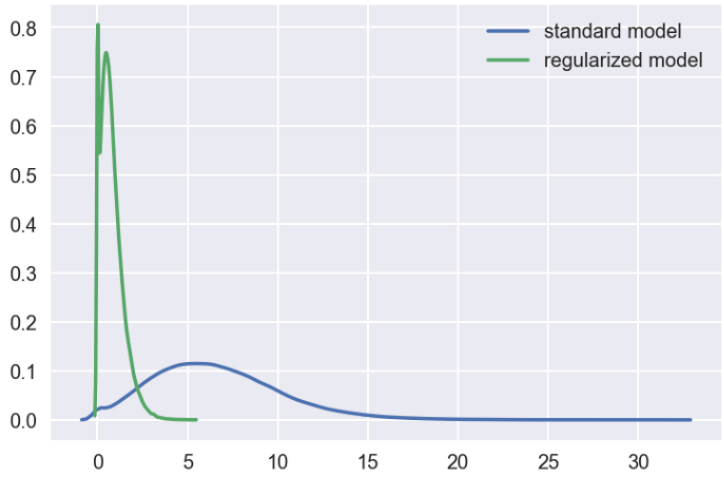
```
code = Dense(code_size, activation='relu')(input_img)
```

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 and here.
- 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. Here is a good reference.

7. Conclusion

Autoencoders are a very useful dimensionality reduction technique. They are very popular as a teaching material in introductory deep learning courses, most likely due to their simplicity. In this article we covered them in detail and I hope you enjoyed it.