Icon

Description automatically generatedAutoencoders

In this chapter we will look at autoencoders (including the sparse variant). This is a theory chapter, so it will cover the mathematics and the fundamentals of autoencoders. We will discuss what they are, what are the limitations, the typical use cases and we will look at some examples. We will start with a general introduction to autoencoders and we will discuss the role of the activation function in the output layer. We will then discuss the loss functions that are typically used and what is the reconstruction error. We will then look at typical applications as dimensionality reduction, classification and anomaly detection.

# What is an autoencoder

As we have seen in many previous chapters, neural networks are typically used in a supervised setting. Meaning that for each training observations we will have a label or expected value . The neural network model will then try to learn the relationship between the input data and the expected values. Now suppose we have only unlabeled observations, meaning we only have our training dataset , made of the observations with

Where in general, as noted previously in the book, with . In general, we can give an intuitive definition of an autoencoder as

Definition: an autoencoder is a type of neural network that learn to reconstruct the input observations with the lowest error possible*[[1]](#footnote-1)*.

In other words, you are training a neural network to learn the identity function. At this point you may wonder for which purposes learning the identity function would be useful. This will become clear immediately in the next sections.

# Feed Forward Autoencoders

A Feed Forward Autoencoder (FFA) is a neural network made by dense layers. In Figure (25.1) you can see an example of an FFA.



Figure 25.1: A typical architecture of a Feed Forward Autoencoder. The number of neurons in the layers at first goes down as we move through the network until it reaches the middle, and then starts to grow again until the last layer has the same number of neurons as the input dimensions.

A typical architecture (although it is no strict requirements) has an odd number of layers and is symmetrical with respect to the middle layer. Typically, the first layer has a number of neurons (the size of the input observation . As we move toward the center of the network, the number of neurons in each layer drops in some measure. The middle layer (remember we have an odd number of layers) is normally the one with the smallest number of neurons of all. In almost all practical applications the layers after the middle one, are a mirrored version of the layers before the middle one. For example an autoencoder with 3 layers could have the following numbers of neurons: , and then (supposing we are working on a problem where the input dimension is ). All the layers up to and including the middle one make what is called the **encoder**, and all the layers from and including the middle one (up to the output) make what is called the **decoder**, as you can see depicted in Figure (25.1).

The **encoder** can be written generally as a function

Where is the output of the middle layer in Figure (25.1) when we do a forward pass through the network up to the middle layer for input observation . Note that with . Also, is relevant to point out that could even be one or two orders of magnitude smaller than . The **decoder (and the output of the network** ) **can be written as a generic function**

If the FFA training is successful the output will be a good approximation of the input, in other words . What is important to notice, is that the decoder is then able to reconstruct the input by using only much smaller number () of features than the input observations originally have (). The output of the middle layer are called a *learned representation* of the input observation .

**Note** The **encoder** is able to reduce the number of dimensions of the input observation () and to create a learned representation (of the input that has a much smaller dimension (. This learned representation is enough for the **decoder** to reconstruct the input accurately (if the autoencoder training was successful as intended).

## Activation Function of the Output Layer

In autoencoders the activation function of the output layer plays a particularly important role. The most used possibilities are two: ReLU and sigmoid. Let’s have a look at both and give some tips on when to use which and why you should choose one instead of the other.

### ReLU

The ReLU activation function can assume all values in the range . Note that it cannot assume negative values. If your input has also negative values, the ReLU is of course a very bad choice, and the identity function would be a much better choice.

**Note** The ReLU activation function for the output layer is well suited for cases when your input observations assume positive real values.

### Sigmoid

The sigmoid function can assume all values in the range . This activation function can only be used if your input observations are all in the range or if you have normalized them to be in that range. Consider as an example the MNIST dataset. Each value of an image is the gray values of a pixel that can assume any value from to. Normalizing the data by dividing the pixel values by would make each observation (each image) have only pixel values between and . In this case the sigmoid would be a good choice for the activation function of the output layer.

**Note** The sigmoid activation function for the output layer is a good choice in all cases where the input observations assume only values between and or if you have normalized them to assume values in the range .

## Loss Function

As with any neural network, we need a loss function to minimize. This loss function should measure how big is the difference between the input and output . Remember that an autoencoder is trying to learn the identity function, therefore you want to find the weights in the network that gives you the smallest difference according to some metric between and . This metric will be the loss function that we will use to train our autoencoder with backpropagation. There are two loss functions that are widely used for autoencoders: the Mean Squared Error (MSE) and the Binary Cross Entropy (BCE). Let’s have a deeper look at both since they can only be used when specific requirements are met.

### Mean Square Error

Since an autoencoder is trying to solve a regression problem, the most common choice as a loss function is the Mean Square Error (MSE):

Where the symbol indicates the norm of a vector[[2]](#footnote-2). This loss function is particularly well suited when the inputs are real values that can assume any value and it can be used in almost all cases, independently of how you choose your output layer activation function or of how you normalize your input data.

It is easy to show that the minimum of is found for . To prove it, let’s calculate the derivative of with respect to a specific observation Remember that the minimum is found when the condition

Is met for all . To simplify the calculations let’s assume that the inputs are one dimensional[[3]](#footnote-3) and let’s indicate them with . We can write

Equation (25.5) is satisfied when as can be easily seen from Equation (25.6), as we wanted to prove. To be really precise we need to show that

As you should know from calculus. This is easily proved as we have

That is clearly greater than zero, therefore confirming our assumption that for we indeed have a minimum.

### Binary Cross Entropy

If the activation function of the output layer of the FFA is a sigmoid function, thus limiting neuron outputs to be between 0 and 1, and the input features are normalized to be between 0 and 1 we can use as loss function the binary cross entropy, indicated here with . The formula for it is

Where the sum is over the entire set of observations. Can we prove that minimizing this loss function is equivalent to reconstructing the input as well as possible? Let’s calculate where has a minimum with respect to . In other words, we need to find out what values should assume to minimize . As we have done for the MSE, to make the calculations easier, let’s consider the simplified case where and are one-dimensional and let’s indicate them with and .

To find the minimum of a function, as you should know from calculus, we need the first derivative of . In particular we need to solve the set of equations

In this case it is easy to show that the binary cross entropy is minimized when for .

To find when the is minimized we can derive with respect to a specific input

Now remember that we need to satisfy the condition

That can clearly happen only if as can be seen from Equation (25.11). To make sure that this is a minimum we need to evaluate the second derivative, since as you should know from calculus the point for which the first derivative is zero is a minimum only if

We can calculate the second derivative at the minimum point easily

Now remember that . We can immediately see that the denominator of the previous formula is greater than zero. The nominator is also clearly greater than zero since . Dividing two positive numbers gives a positive number, thus we have just proved that

The minimum of the cost function is reached when the output is exactly equal to the inputs, as we wanted to prove.

**Note** an important prerequisite for using the binary cross entropy loss function is that the inputs **must** be normalized between 0 and 1 and that the activation function for the last layer must be a *sigmoid* or *softmax* function.

## Reconstruction Error

The reconstruction error (RE) is a metric that gives you an indication of how good (or bad) the autoencoder was able to reconstruct the input observation . The most typical RE used is the MSE

That can be easily calculated. The RE is used often when doing anomaly detection with autoencoders, as we will explain later. There is an easy intuitive explanation of the reconstruction error. When the RE is big the autoencoder was not able to reconstruct the input well, while when it is small, the reconstruction was successful. In Figure (25.2) you can see an example of big and small reconstruction error when an autoencoder tries to reconstruct an image.

Diagram

Description automatically generated

Figure 25.2: An example of big and small reconstruction error when an autoencoder tries to reconstruct an image.

## Example: reconstruction of hand-written digits

Let’s now see how an autoencoder performs with a real example, using the MNIST dataset. This dataset[[4]](#footnote-4) contains hand-written digits from 0 to 9. Each image is pixels with only gray values, that means that we have 784 features (the pixel gray values) as inputs. Let’s start with an autoencoder with 3 layers with the numbers of neurons in each layer equal to . Note that the first and last layers **must** have a dimension equal to the input dimensions. For this example, we used the Adam optimizer[[5]](#footnote-5), as loss function the cross-entropy and we trained the model for epochs with a batch size of . In Figure (25.3) you can see two lines of images of digits. The line at the top contains ten random images from the original dataset, while the one at the bottom are the reconstructed images with the autoencoder we just described.

A picture containing sign

Description automatically generated

Figure 25.3: In the top line you can see the original digits from the MNIST dataset. While the line below are the digits reconstructed by the autoencoder with number of neurons equal to (784, 16, 784).

It is impressive that to reconstruct an image with 784 pixels only 16 features are needed to have a result that, although not perfect, allows us to understand almost perfectly what digits was used as input. Increasing the size of the middle layer to (and leaving all other parameters the same) gets a much better result as you can see in Figure (25.4).

A picture containing text

Description automatically generated

Figure 25.4: In the top line you can see the original digits from the MNIST dataset. While the line below are the digits reconstructed by the autoencoder with number of neurons equal to (784, 64, 784).

This tells us that the information of the images is really contained in a much lower number of features than 784.

**Note** An autoencoder with a middle layer smaller than the input dimensions can be used to extract the important features of an input dataset creating a learned representation of the inputs given by the function . Effectively an FFA can be used to perform dimensionality reduction.

The FFA will not be able to recreate the input digits well, if the number of neurons in the middle layer is reduced too much. In Figure (25.5) you can see the reconstruction of the same digits with an autoencoder with only 8 neurons in the middle layer. With only 8 neurons in the middle layer you can see that some reconstructed digits are wrong. As you can see in Figure (25.5) the 4 is reconstructed as a 9 and a 2 is reconstructed to something that resembles a 3.

A close up of a sign

Description automatically generated

Figure 25.5: In the top line you can see the original digits from the MNIST dataset. While the line below are the digits reconstructed by the autoencoder with number of neuraons equal to (784, 8, 784).

In Figure (25.6) you can see a comparison of the reconstructed digits by all the FFAs we have discussed.



Figure 25.6: In the top line you can see the original digits from the MNIST dataset. The second line of digits are the digits reconsructed by the FFA (784,8,784), the third by the FFA (784,16,784) and the last one by the FFA (784,64,784).

From Figure (25.6) you can clearly see how, increasing the size of the middle layer, the reconstruction gets better and better, as we expected.

For these examples we have used the binary cross entropy as loss function, but the MSE would have worked exactly as well as can be seen in Figure (25.7).

Calendar

Description automatically generated

Figure 25.7: In the top line you can see ten random original digits from the MNIST dataset. The second line of digits are the digits reconstructed with a FFA with 16 neurons in the middle layer and the binary cross entropy as loss function. The last line are images reconstructed with the MSE as loss function.

# Autoencoders Applications

## Dimensionality Reduction

Comparison with PCA, batches, higher dimensions

## Classification

INTRODUCTION TO BE ADDED

### Classification with Latent Features

Let’s now suppose that we want to classify our input images. Of course, we can simply use all the features, in our case the pixel values of the images. We can simply use an algorithm as kNN, for illustrative purposes. Note all the results you see in this section have been run on Google Colab. Doing it with nearest neighbors on the training MNIST dataset (with 60000 images) will take around 16.6 minutes (ca.1000 sec) and gets you an accuracy on the test dataset of on a test dataset of 10000 images. But what happens if we use this algorithm not with the original dataset, but with the latent features ? For example, if we consider an FFA with neurons in the middle layer and we again train a kNN algorithm on the latent features we get an accuracy of 89% in 1.1 sec. We get a gain of a factor of 1000 in running time, for a loss of 7.4% in accuracy[[6]](#footnote-6).

|  |  |  |
| --- | --- | --- |
| Input Data | Accuracy | Running Time |
| Original Data | 96.4% | sec. min. |
| Latent Features | 89% | 1.1 sec. |

Table 25.1: The difference in accuracy and running time when applying the kNN algorithm to the original 784 features or the 8 latent features for the MNIST dataset.

Keep in mind that using features allow us to get a very high accuracy in one second.

We can do the same analysis with the Fashion MNIST dataset. In Table 25.2 you can see the summary of the results.

|  |  |  |
| --- | --- | --- |
| Input Data | Accuracy | Running Time |
| Original Data | 85.4% | sec. min. |
| Latent Features | 79.9% | 1.2 sec. |
| Latent Features | 83.6% | 3.0 sec. |

Table 25.2: The difference in accuracy and running time when applying the kNN algorithm to the original 784 features with a FFA with 8 neurons and with a FFA with 16 neurons for the Fashion MNIST dataset.

It is very interesting to note that with a FFA with 16 neurons in the middle layer we reach an accuracy of 83.6% in just 3 sec. When applying a kNN algorithm to the original features (784) we reach an accuracy just 1.8% higher but with a running time ca. 330 times longer.

**Note** Using autoencoders and doing classification with the latent features is a very viable techniques to reduce the training time by several order of magnitude while incurring in a minor drop in your accuracy.

### Curse of dimensionality in kNN

https://sebastianraschka.com/pdf/lecture-notes/stat479fs18/02\_knn\_notes.pdf

Curse of dimensionality

<https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote02_kNN.html>

## Anomaly Detection

Autoencoders are often used to perform anomaly detection on the most different datasets. Let’s suppose we have a big enough dataset with a few observations that we would classify as outliers. It is important that the number of those outliers is much smaller than the size of the dataset.

1. In this chapter we will discuss at length what we mean with error here. [↑](#footnote-ref-1)
2. The norm of a vector is simply the square root of the sum of the square of the components. [↑](#footnote-ref-2)
3. If we did not have this assumptions, one would have to calculate the gradient of the loss function instead of the simple derivative. [↑](#footnote-ref-3)
4. More information on the dataset can be found here: <http://yann.lecun.com/exdb/mnist/>. [↑](#footnote-ref-4)
5. You can find the entire code at the address <https://adl.toelt.ai>. [↑](#footnote-ref-5)
6. You can run those tests yourself going to Chapter 25 at <https://adl.toelt.ai>. [↑](#footnote-ref-6)