Classical or Variational?

A Comparison of Two Autoencoders

Fabrizio Patuzzo

April 2020

The aim of this short paper is to compare the performance of two autoencoders, one classical and one variational, in an anomaly detection contest. Moreover, we would like to take a look inside, to better understand how they work - just like a mechanic might look inside a car's engine to debug problems or enhance its performance.

1 One dataset, two autoencoders

Our dataset is composed of 1000 black and white images, 32x32 pixels wide, each containing a straight line that separates the image into a white and a black half. The line can have different slopes, but always passes through the center of the image. We would like to train an autoencoder to reproduce the input images. Figure 1 shows a few images from the training dataset, shown using the colormap 'viridis' (viridis maps white to yellow, black to violet and gray to green).

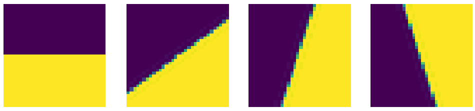


Figure 1: The initial training set

We compare the results obtained using two autoencoders: A1 and A2.

A1 is an MLP autoencoder with the following architecture:

LAYER	type	neurons	activation
0	Input	32x32	
1	Dense	2	relu
2 (bottleneck)	Dense	1	
3	Dense	2	relu
4	Dense	32x32	sigmoid

A2 is an autoencoder with the same architecture, but of a variational type (see Section 3).

2 Results using A1

After training A1, the autoencoder managed to reconstruct the input images, but with some blur along the dividing line, as shown in Figure 2.

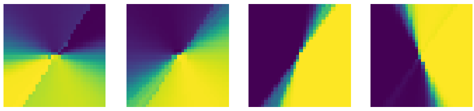


Figure 2: Results using A1

To try to pinpoint the cause of the blurry output, let us apply A1 to reconstruct an even simpler dataset, composed of four black and white images 2x2 pixels wide, representing lines with a slope of any multiple of 90 degrees. The dataset is shown in Figure 3.

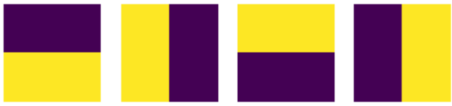


Figure 3: Simplified training set

Can A1 learn to reconstruct the inputs, or does the blur occur in this case, too? After training the autoencoder a dozen times (more on this later), it managed to reconstruct the inputs perfectly.

Let us see the elegant solution the network found, and in particular how the four output neurons vary depending on the bottleneck value.

To succeed, for some bottleneck value the four outputs should be 0011 (the first image), 1010 (the second), 1100 and 0101.

1. Figure 4 shows the weights the network found leading from the bottleneck to output neuron 1.

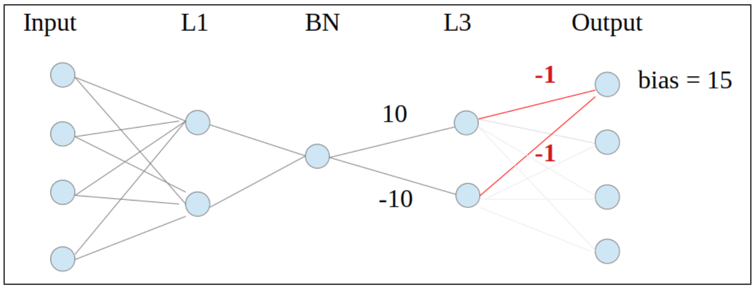


Figure 4: weights leading to output neuron 1

If BN=-2 the output (before the sigmoid) would be -5 (notice there is a relu after L3). If BN=0, the output would be 15. And if BN=2, it would be -5 again.

For simplicity, let us suppose that the sigmoid turns negative numbers to '0's and positive numbers to '1's (like a Heaviside function). In this case, output 1 as a function of the bottleneck would be:

BN	-∞		-1.5		1.5		+∞
output 1		0		1		0	

2. Figure 5 illustrates the weights leading to output neuron 2.

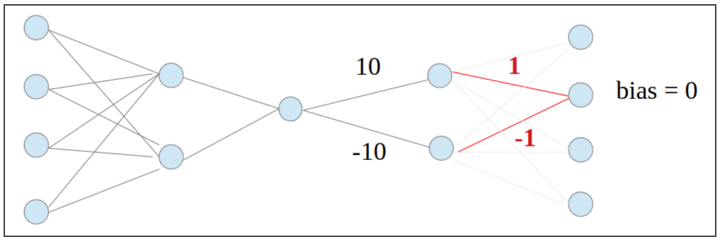


Figure 5: weights leading to output neuron 2

As a consequence, output 2 as a function of the bottleneck would be:

BN	-∞		0		+∞
output 2		0		1	

3. In Figure 6, one can see the connections leading to output neuron 3.

Output 3 as a function of the bottleneck is:

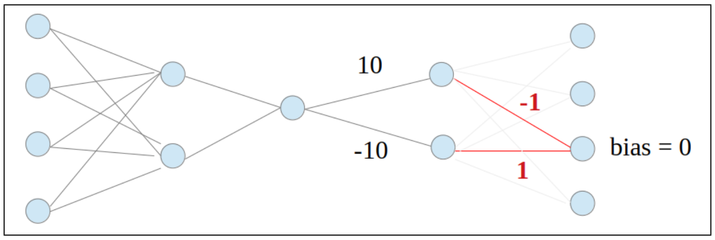


Figure 6: weights leading to output neuron 3

BN	-∞		0		+∞
output 3		1		0	

4. Finally, Figure 7 shows the weights that lead to neuron 4.

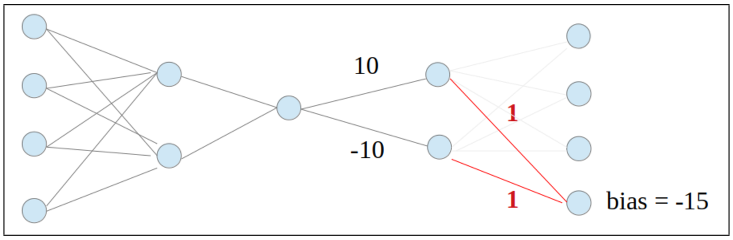


Figure 7: connections leading to output neuron 4

Output 4, for different bottleneck values would be:

BN	-∞		-1.5		1.5		+∞
output 4		1		0		1	

With a bottleneck value between $-\infty$ and -1.5, one obtains the first image: 0011. With a bottleneck value between -1.5 and 0, one obtains the second: 1010. With a value between 0 and $+1.5$, the third (1100), and with a value between $+1.5$ and $+\infty$, the fourth: 0101.

This, however, does not explain why the blur occurred. But let us remember that we have replaced the sigmoid with a Heaviside function. Does it make any difference? If we use a sigmoid, when the bottleneck value approaches one of the thresholds (-1.5, 0 or 1.5), the value of the output neurons concerned will assume an in-between gray value, as one can see in Figure 8.

So, when the bottleneck value is close to a threshold, the output becomes blurry. One way to remove the blur would be to use a Heaviside instead of a sigmoid after the output. But since we needed the sigmoid to train the network, we can

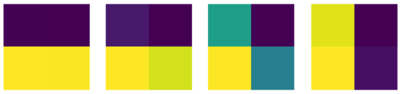


Figure 8: outputs obtained tuning the bottleneck value

first compute inverse sigmoid of the output, and then the Heaviside function of the result.

This operation worked, and Figure 9 shows the new output, after this postprocessing step: the blur disappeared.

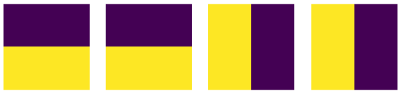


Figure 9: the blur disappears

Is this by any chance the same reason why the outputs were blurry in our initial dataset? And can we apply the same trick to remove it? Yes! By applying the inverse sigmoid, and then the Heaviside function of the result, the blur disappears also in our initial dataset, leading to quite decent results, as one can see in Figure 10, a convincing indication that this was the source of the problem.

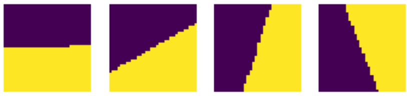


Figure 10: the blur disappears in the initial dataset

3 Results using A2

Let us compare these results with those obtained with a variational autoencoder.

Figure 11 shows the famous variational architecture.

In a variational autoencoder, an input produces a bottleneck value picked at random from a normal distribution, where a is the mean of the distribution, and b the logarithm of its standard deviation.

But to build a fully working variational autoencoder, it is not sufficient to adapt its architecture: one also needs to add a new term to the loss:

$$loss = mse{-}0.5 \cdot mean(1 + b - a^2 - e^b) \tag{1}$$

The change in the architecture and in the loss produce two differences in a variational autoencoder:

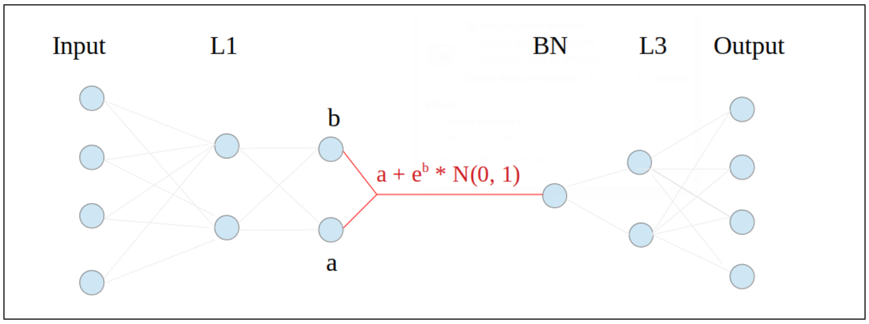


Figure 11: The variational architecture

1. Because of the new architecture, a specific input image i does not produce a single bottleneck value, but a bottleneck value picked at random from a normal distribution with $\mu_i = a$ and $\sigma_i = e^b$.

2. The new loss encourages the bottleneck values to follow a normal distribution with μ =0 and $\sigma = 1$. Hence, the bottleneck values will tend to remain within the range [-3, 3] with a probability of 0.97.

Notice that, from the bottleneck on, the network is identical to A1. Hence, our first guess was that A2 might find a similar solution to A1, and possibly the facts that all bottleneck values are obliged to be close together and that there is some randomness in producing bottleneck values might only make things worse.

Let us train the variational autoencoder using our initial training set. Figure 12 shows the distribution of the mean values (a).

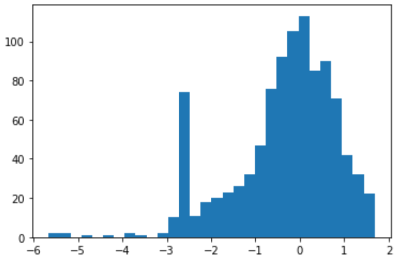


Figure 12: The mean values of the bottleneck

Figure 13 illustrates the distribution of the standard deviations (e^b).

The standard deviations are all around 0.01, which means that a given input might produce bottleneck values of, for instance, 1.29, 1.3, or 1.31 at different runs.

Now, let us see how the network learned to reconstruct the input images. Fig-

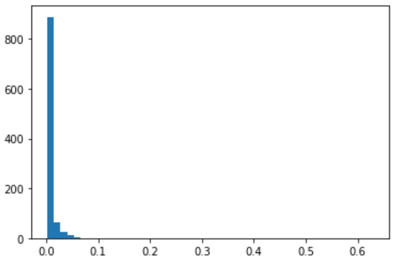


Figure 13: standard deviations

ure 14 shows the results. They are nearly perfect. And this time, it was not necessary to do the postprocessing step.

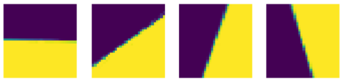


Figure 14: Outputs using A2 on the initial dataset

How is it possible that A2 produced better results? Because it found some weights in L3 and L4 high enough to output clear white or black (and not gray) pixels.

Let us have a look. The weights in L3 are approximately 10 and -10, and the weights in L4 are also all around either 10 or -10. Suppose we squeeze 100 bottleneck values between -3 and 3. They will have a gap of 0.06 between them, and this gap will be multiplied by 100 in the output neurons, becoming gaps of 6 units, sufficient for the sigmoid to output clear black or white pixels.

Why did A1 not find them? Because, for some reason, it converged to a suboptimal solution. It was unable to find high weights in L4.

While we were mainly interested in applying A2 to our initial dataset, we are still curious to see how it would perform on our simplified dataset of 2x2 images. The answer was slightly surprising: its performance is catastrophic!

Figure 15 shows the reconstructions of the input images of the simplified dataset by A2.

Not only A2 failed to properly reconstruct the inputs, but the outputs would be different at every run of the autoencoder.

The reason is that while using the initial dataset (which contained 1000 images) the standard deviations equalled roughly 0.01, using the reduced dataset of four images, the standard deviations increased to values of roughly 0.5 - leading to largely unpredictable outputs.

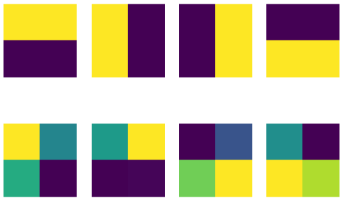


Figure 15: Results using A2 on the simplified dataset

This suggests that a variational autoencoder, while it might perform better than a classical one with a large training set, it will not perform well with a small training set.

4 Trying to avoid suboptimal solutions

Why does a network converge to a suboptimal solution?

We mentioned that, even with the simplified dataset of 2x2 pixel images, the training occasionally converged to a suboptimal solution. This actually happened quite often, and one might need to train the network a dozen times before the solution is found. Not to mention that one might give up training it, jumping to the conclusion that the architecture is not suitable to solve the problem.

Let us survey some of the problems that might occur during training.

1. With an observed frequency of about 1/8, A1 found the optimal solution shown in Figure 16, and the loss converged to 0.

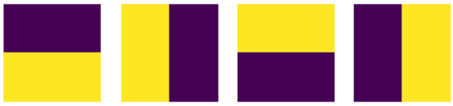


Figure 16: Optimal solution

2. About 1/3 of the times, the network produced two gray pixels in two output images, as shown in Figure 17, and the loss approached 1/16.

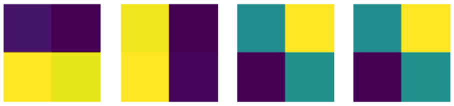


Figure 17: suboptimal solution 1

This happens when two input images (here images 3 and 4) produce exactly the same value in the bottleneck.

The main reason is that, after the weight initialization, both images produced negative values in all neurons in L1, which the relu converted into zeros. For example, input images 3 and 4 will produce the same value in a neuron in L1 if both $w1+w2$ and $w2+w4$ happen to be negative, which happens with a frequency of 1/4.

Since there are 6 possible pairs of input images, the probability that this happens to at least one of the six pairs in both neurons in L1 is $1-(15/16)^6 \approx 0.32$.

3. Occasionally, this same problem can occur with two pairs of images, leading to a loss of 1/8, as exemplified in Figure 18.

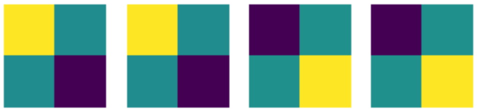


Figure 18: suboptimal solution 2

4. About 1/3 times, all the outputs were gray, and the loss remained at 1/4. This unfortunate outcome is shown in Figure 19.

Figure 19: suboptimal solution 3

This can happen for various reasons. One of them is that all weights in L1 are initialized with a negative value, which produces all bottlenecks equal to zero.

But there are other reasons. One of them is that both weights in L2 are negative and both weights in L3 are positive (or viceversa). In this case, all inputs will produce negative values out of L3, which the relu converts to zeros, and all outputs will be gray. This scenario will occur with a frequency of $1/8 = 0.125$.

Despite the different scenarios that lead to suboptimal solutions, in all these cases different inputs produce negative values in all neurons after some layer, which the relu converts to zeros - and the output becomes blurry.

One way to reduce the number of times this happens is to slightly increase the number of neurons in the inner layers. We minimized the number of times the network converged to a suboptimal solution using 5 neurons in L1, 2 in L2 and 5 in L3.

But a variational architecture that avoids that different inputs generate exactly the same bottleneck values might also help avoid these kinds of unfortunate problems. Possibly, this characteristic helped A2 find the optimal weights to reconstruct the images in the initial dataset.

5 Conclusions

Can we compare the performance of the two autoencoders on our initial dataset?

- The classical autoencoder produced blurry edges in the output images. We could solve this issue by applying a postprocessing step.
- However, the variational autoencoder performed better than the classical autoencoder to reconstruct the images in the initial dataset.
- We remarked that the variational autoencoder is unlikely to perform well if the training set is too small.
- The classical autoencoder often converged to suboptimal solutions. One way to address this issue was to slightly increase the number of neurons in the inner layers.
- The variational architecture seems to help avoid these kinds of problems.

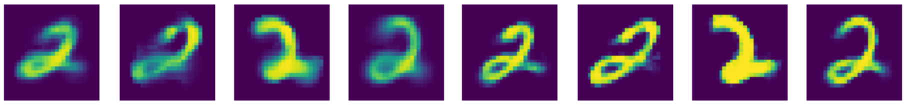
In conclusion, our initial doubts about the ability of a variational autoencoder to perform well in an anomaly detection task were unfounded.

Appendix: A Comparison on the Mnist Dataset

How would A1 and A2 perform in an anomaly detection contest on a different dataset? Let us consider a black and white version of mnist. We train the two autoencoders to reconstruct the digit '2'. The training set is composed of 5958 images. We show a few of them in Figure 20. For these experiments, we use 64 neurons in L1, 8 in L2, and 64 in L3.

Figure 20: The mnist dataset

The results from A1 and A2 are shown in Figure 21a and Figure 21b. A1 managed to reconstruct the '2's, but the images seem slightly blurry, and attained an mse of 0.037. A2, on the other hand, reconstructed more nitid images, reaching an mse of 0.02.

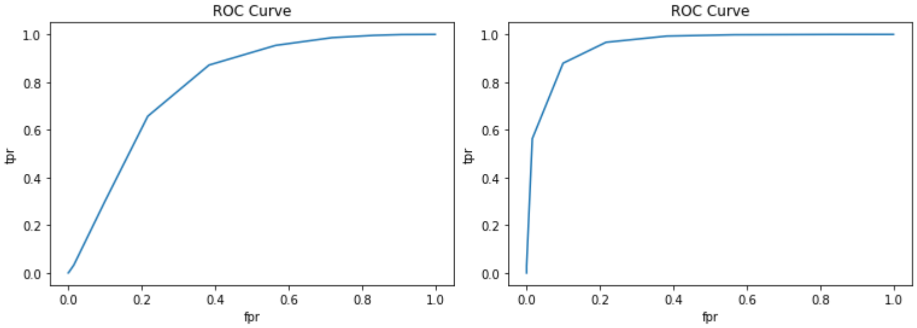


(a) Outputs using A1

(b) Outputs using A2

Figure 21: Reconstructions by the two autoencoders

Figure 22 shows the ROC curve obtained by the two autoencoders in the anomaly detection task.



(a) ROC using A1

(b) ROC using A2

Figure 22: ROC curves using A1 and A2

The AUC using A1 equals 0.79, while the AUC using A2 equals 0.95.