Autoencoders for Defect Detection in Images

An example of using autoencoders to detect flaws in manufacturing.

Ryan Revilla
Follow

5 min read

.

Sep 5, 2025

Listen
Share
More

If you are not a Medium Member, you can read this story for free here.

Detecting defective products is a primary goal in any manufacturing line. Today, computer vision for quality control is readily available. However, the basic computer vision methods require that a defect be well-quantified. This makes the case for introducing machine learning to help detect less quantifiable defects.

That being said, images collected on a manufacturing line can be skewed towards non-defective products, where images of defective products are rare. This makes a simple classifier difficult to train. Here, I'll show one method involving an autoencoder to get around this problem.

This is part of some ongoing work I am doing. So, future posts may show improvements in methodology.

The method

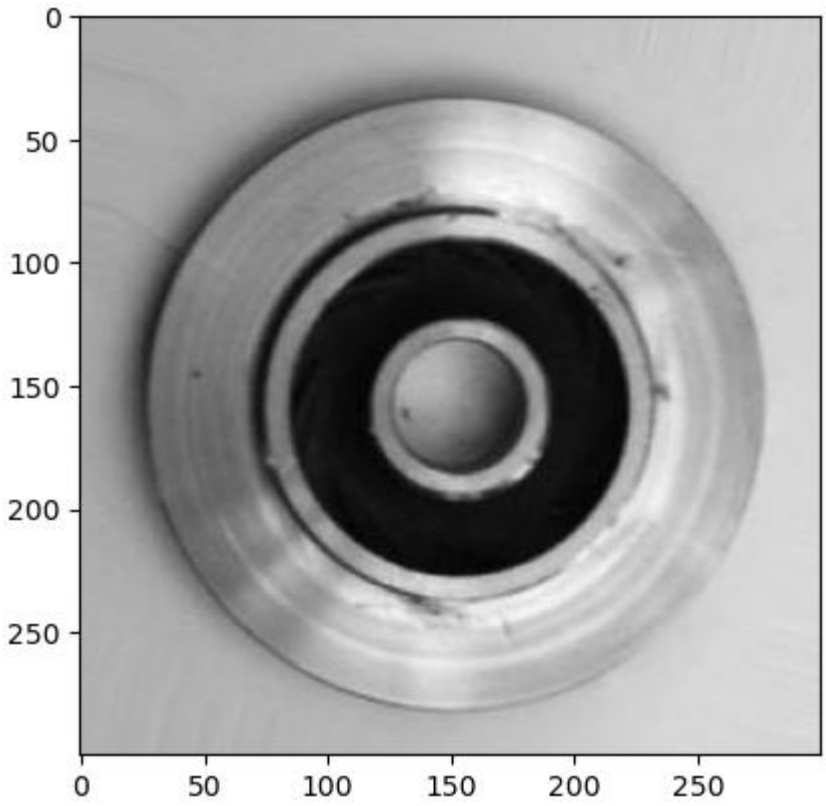
For those who are unfamiliar with an autoencoder, it is a machine learning model for reconstructing input data.

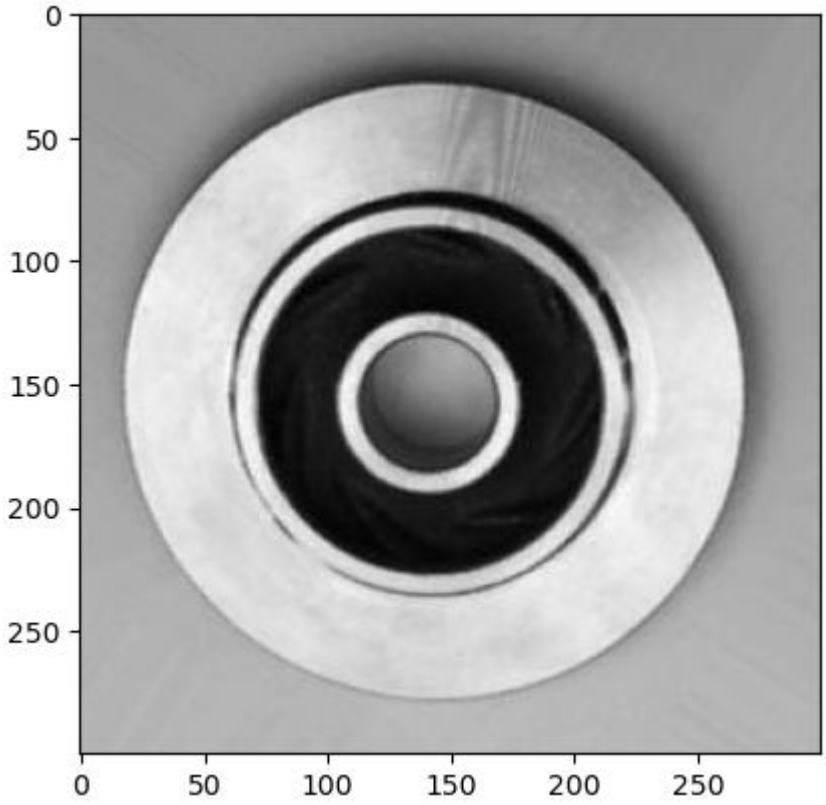
This sounds somewhat a trivial task. But, for this task we can get clever. If we train an autoencoder only on non-defective parts, it should reliably reproduce these parts. When fed an image of a defective part, there should be some error in the reproduction.

Presumably, if the autoencoder learns to reproduce good part images well, this error should be large enough between defective parts and good parts that one could form a decision boundary.

The dataset: images of brake caliper defects

For public demonstration of these methods, I'll be using a publicly available dataset. This dataset contains images of both good and defective castings on brake calipers, it can be found here on Kaggle for free.





(Left) An example of a

defective product from the dataset. (Right) Example of a good product from the dataset. Images made by author in matplotlib. Source data.

A simple autoencoder in Keras

For initial testing, I went with a simple autoencoder made with fully-connected, dense layers. The implementation can be seen here:

```
stacked_encoder = Sequential([
    layers.Input(img.shape), # img.shape is 300 x 300 pixels
    layers.Flatten(),
    layers.Dense(100, activation="relu"),
    layers.Dense(30, activation="relu"),
])
stacked_decoder = Sequential([
    layers.Dense(100, activation="relu"),
    layers.Dense(img.shape[0] * img.shape[1] * img.shape[2],
activation="sigmoid"),
    layers.Reshape([img.shape[0], img.shape[1],img.shape[2]]),
])
stacked_ae = Sequential([stacked_encoder, stacked_decoder])
stacked_ae.compile(loss="mse", optimizer="nadam")
stacked_ae.summary()
```

Here, we separated the autoencoder into two parts: *encoder* and *decoder*. For this specific case, we won't use this separation. In similar cases, the encoder is used on its own to perform the same task we have here, which is considered a method of anomaly detection. For an example of that, you can see this YouTube video.

Looking at some reconstructions

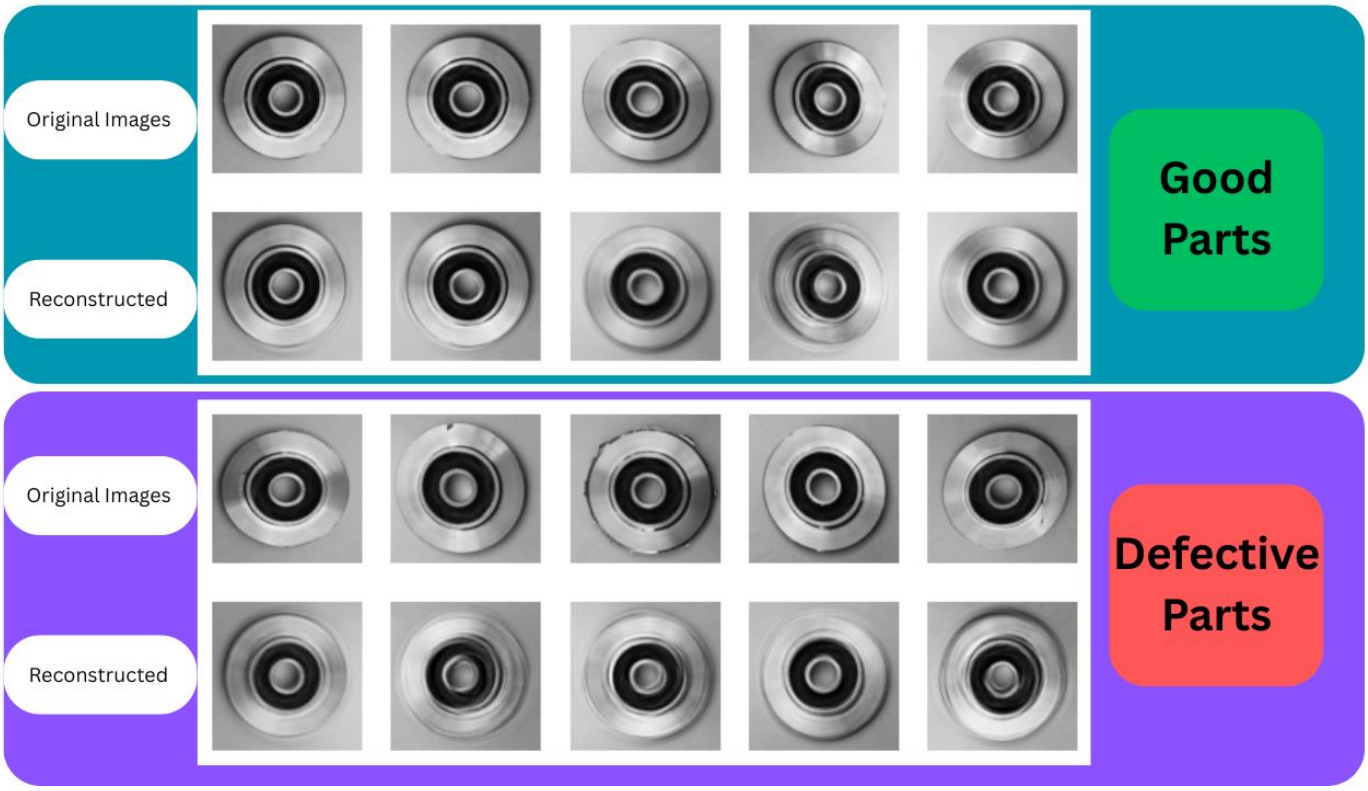
For training, I used this line of code:

```
history = stacked_ae.fit(X_good,X_good,epochs=200)
```

Here, you'll notice that the `X_good` is used for both the input and the training labels. This variable is from a set of non-defective parts of the dataset. I was sure to split these into training and validation splits beforehand, but that is outside the scope of this article.

Plotting some of the reconstructions of the non-defective (good) and defective parts, we can see there's a clear difference. Albeit, for some good parts there is room for improvement.

Press enter or click to view image in full size

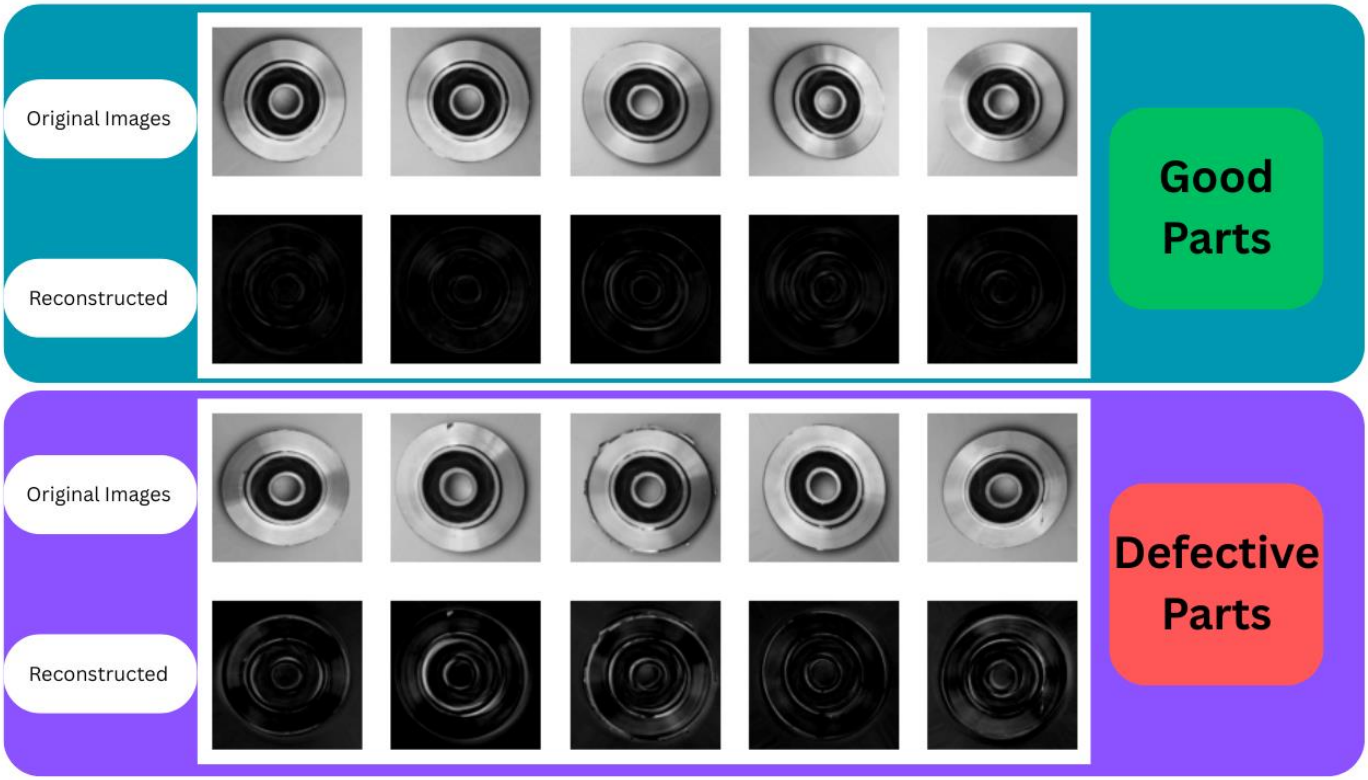


Original images compared against the reconstructions from the autoencoder. These are shown for both good parts and defective parts. Image by Author.

To get a starker change in the results, we'll take the difference between the reconstruction image and the original image. This is accomplished by

merely subtracting the two. We'll also constrain them to values between 0–1 by using `np.clip()`.

Press enter or click to view image in full size



Input images are subtracted from the reconstructed images. These are clipped to be within 0–1 range. Original images are plotted against "reconstructed" images, where here the reconstruction is really the difference between the original and the output of the autoencoder. (Top) Good parts. (Bottom) Defective parts.

The result shows a more exaggerated difference between good and defective parts. It's almost as if the defects are highlighted now. This should be good enough to run a basic classifier.

The function I used to plot the images above can be seen below. This function allows for both plotting the image reconstructions as well as the error between the two by changing the `plot_diff` *boolean.*

```
def plot_reconstructions(model, images=X_good,
n_images=5,plot_diff=False):
    reconstructions = np.clip(model.predict(images[:n_images]), 0, 1)

    if plot_diff:
        reconstructions = reconstructions - images
        reconstructions = np.clip(reconstructions,0,1)

    fig = plt.figure(figsize=(n_images * 1.5, 3))
    for image_index in range(n_images):
        plt.subplot(2, n_images, 1 + image_index)
        plt.imshow(images[image_index], cmap="binary")
        plt.axis("off")
        plt.subplot(2, n_images, 1 + n_images + image_index)
        plt.imshow(reconstructions[image_index], cmap="binary")
        plt.axis("off")
```

Laying the groundwork for a classifier

To use a classifier to determine if a part is good or defective, one needs to make sure the results are separable in some way. For me, I wanted each image to come down to a single number. To do so, one can use a number of methods. For example:

- Sum over all of the image
- Take the mean of the image
- Take the standard deviation over the image

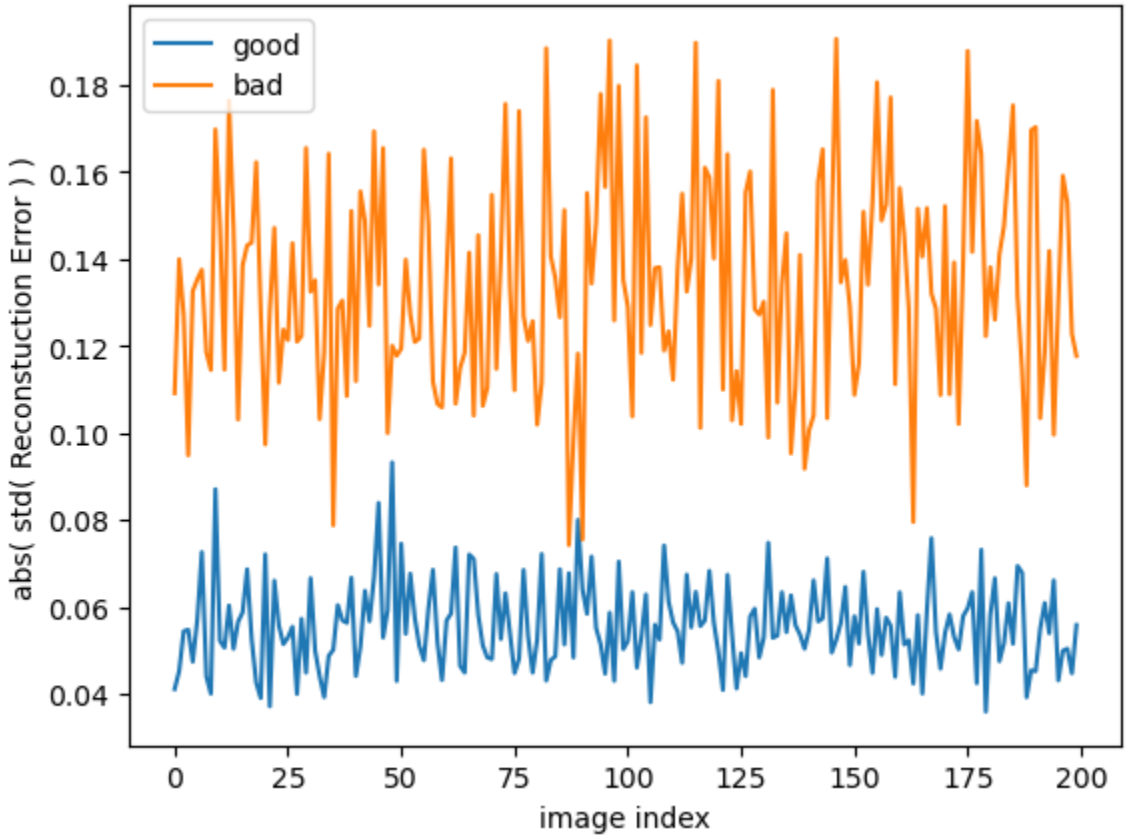
After some testing, these all worked to some degree. However, the standard deviation of the image proved most effective. I also found it useful to take the `np.abs()` of each to make sure the result is positive. Here is the function I used to do so:

```
def get_diff_sums(imgs,autoenc):
    Y = autoenc.predict(imgs) # create reconstructions
    D = Y-imgs # take differences between reconstructions and images
```

```
    sums = []
    for diff in D:
        sums.append(np.abs(np.std(diff))) # take std and abs of each
image
    sums = np.array(sums)
    return sums

good = get_diff_sums(X_good,stacked_ae)
bad = get_diff_sums(X_bad,stacked_ae)
```

Plotting the results, we can see that the good and bad parts are separable. This means a classifier can be applied in the future.



Plot of the results that can be used for classification. Image by Author.

As a note the Reconstruction Error is the difference between the reconstructed image and that of the original image. The reconstruction error is in the same shape as the original image. To make it into a

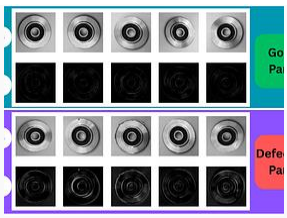
single number for future classification, the standard deviation was taken. Then the absolute value was taken so that values are positive.

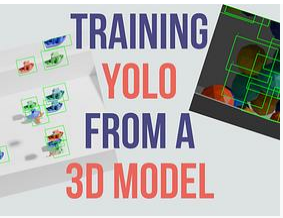
Ryan Revilla

ML / AI in Manufacturing

View list

3 stories





Conclusion

An autoencoder can be a useful tool in detecting defects in images, especially if the dataset mostly contains images of non-defective parts. Here we trained an autoencoder to reproduce an image of a good part. Images of defective parts sent through the autoencoder produced a reconstruction error. The reconstruction image was subtracted from the original image to produce a reconstruction error. Summarizing the

reconstruction error into a single number provided a means of separating good and bad parts, which allows for future classification.