Published in Towards Data Science



Artem Oppermann Follow
Jan 14, 2020 · 15 min read · Member-only · Listen

# Anomaly Detection with Autoencoders in TensorFlow 2.0

A Guide on how to implement Neural Networks in TensorFlow 2.0 to detect anomalies.

**In this detailed guide, I will explain how Deep Learning can be used in the field of Anomaly Detection. Furthermore, I will explain how to implement a Deep Neural Network Model for Anomaly Detection in TensorFlow 2.0. All source code and the corresponding dataset is, of course, available for you to download- nice ;)**

## Table of Content

## 1. Introduction

An anomaly refers to a data instance that is significantly different from other instances in the dataset. Often times they are harmless. These can only be statistical outliers or errors in the data. But sometimes an anomaly in the data may indicate some potentially harmful events that have occurred previously.

**Such an event can be financial fraud.**

In this article, I will show you how to use the power of deep neural networks to identify fraudulent financial credit card transactions that can be identified as anomalies in a dataset.

## 2. Anomaly Detection

Unfortunately, regular feedforward neural networks are not well-suited for detecting anomalies. Because of this, I will introduce a new neural network architecture called Autoencoder.

But first, I would like to explain the term anomaly in more detail and show why anomaly detection systems play such an important role in the field of predictive analytics.

A common observation that can be made when analyzing real-world data is that some instances or observations in the dataset can make one suspicious because they differ significantly from most of the data in the dataset. These particular data instances are different in that they do not match other expected patterns or behaviors in the dataset. Such cases are commonly known as anomalies.

Please consider the following dataset that consists of 4 features.

| x1 | x2 | x3 | x4 |
|---|---|---|---|
| 15240 | 253 | 682 | 0.012 |
| 102 | -0.20 | 10 | 23 |
| 10102 | 201 | 452 | 0.21 |
| 12425 | 120 | 682 | 0.201 |
| 354 | 15 | 4523 | -0.230 |
| 3501 | 174 | 754 | 0.001 |

A closer look reveals some irregularities in this dataset. The values in the second and fifth data instances are very different from those in other instances. The feature values of these two instances differ a lot from corresponding feature values of other instances.

Clearly we are observing two anomalies here. Anomalies in the data can occur for several reasons. Sometimes anomalies are errors in the data that occurred during data collection or preprocessing.

Of course, there is also a good chance that anomalies may belong to the actual data. In this case, they are nothing more than just some statistical outliers. On the other hand, anomalies can sometimes indicate a new, previously unknown underlying event that caused these anomalies in the first place.

In this case, the anomalies in the data may indicate events such as fraud, abuse or service disruption, all of which pose a danger to a business or organization. The question you may be asking yourself now is why we can see these events as anomalies in the data.

The simple answer is that fraud, abuse or service disruption are of course events that do not conform to the usual, expected behavior or processes in a business or organization. In fact, these are quite rare events.

## 3. Uses Cases for Anomaly Detection Systems

Let's take a look at some particular business field where anomalies in the data may suggest a potential threat or problem.


**Banking:** In the field of banking, anomalies may be related to unusually high purchases/deposits or cyber-intrusions


**Healthcare:** In health care fraud in claims and payments may stand out as anomalies. But more importantly, abnormalities in patient's health data can suggest a disease or an aggravation of the health situation


**Manufacturing:** In manufacturing abnormal machine behavior can be registered as an anomaly in the data that is produced by the machine


**Finance:** As already mentioned in the example before, In the field of finance (that also can belong to banking and vice versa) fraudulent financial transactions can be registered as anomalous data instances


**Smart home:** In the case of smart homes energy leakage can cause some unexpected observations in the data

Telecom

**Telecom**: In telecommunication Roaming abuse, revenue fraud, service disruptions may be recognized as abnormal instances in the data.

As you can see from these examples, anomaly detection is an important part of a variety of business areas. Anomaly detection models can protect companies and institutions from financial and personal damage. And even be a potential lifesaver in healthcare.

## 4. Anomaly Use Case: Financial Fraud

Take fraud as an example. Imagine that you are withdrawing money from your bank account. You are doing it once per week, every time from an ATM in your hometime. And you are withdrawing every time the amount of money in the range lets say between 100–250 $. Of course, every time you withdraw money, your bank collects the data associated with the withdraw, such as time, place, amount of money, etc.

As long as you stick to your usual pattern, the collected data instances look pretty much the same. Now imagine the case where your card was stolen and the thieves found out your ATM personal identification number. The thieves use the opportunity and withdraw a large amount of money that is far above your usual amount from an ATM that is not in your hometown.

As you can imagine, the data instance that contains the related information for this specific cash withdrawal differs significantly from the previous

instances in terms of the amount of money and the place of withdrawal.

In this case, the data instance would attract attention as an anomaly.

Of course, the anomaly and the kind of threat it may suggest depends on the industry and the associated type of data. In any case, the goal of anomaly detection models is to detect abnormal data so that steps can be taken to further investigate the detected anomalies and to avoid possible threats or problems for the company or its customers.

## 5. How does an Autoencoder work?

For the rest of the article, I will solely focus on financial fraud as a case study for the detection of anomalies.

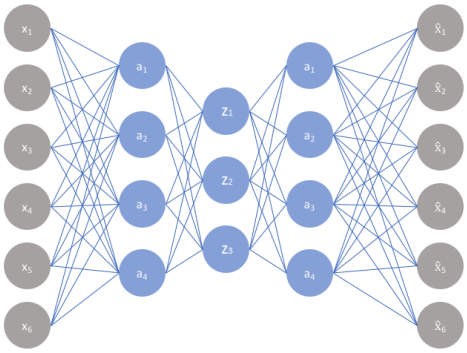
This is the time when we can use deep neural networks to our advantage. Neural networks are universal pattern recognition systems that can identify patterns and associate them with fraudulent behavior that we humans would never consider.

For example, patterns such as time spent hovering over a particular button or area of the screen while a transaction is being prepared.

A person may never find out if this pattern is related to a kind of fraudulent behavior or not. For neural networks, however, this is not a problem at all. A neural network architecture that is well-suited for identifying such patterns is called an Autoencoder.

It would certainly go beyond the scope of this article to explain to you in detail this neural network architecture. Instead, I would like to give a short overview of this network. For a more detailed explanation of Autoencoder, please feel free to read the article " Deep Autoencoder in TensorFlow 2.0 ".

The simplest form of Autoencoder is a feedforward neural network that you are already familiar with. As with feedforward neural networks, an Autoencoder has an input layer, an output layer, and one or more hidden layers. The following image shows the architecture of an AutoEncoder. As we can see the input layer and the output layer of an AutoEncoder have the same number of neurons.



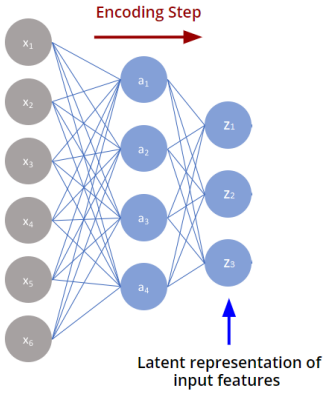
An Autoencoder can be divided into two parts:

- The Encoder
- The Decoder

### The Encoder

The encoder refers to the first half of the autoencoder, where the number of hidden neurons decreases as the network gets deeper.

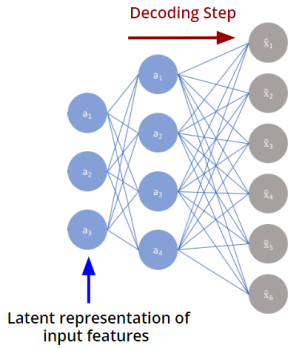
The decreasing number of neurons forces an “encoding” or compression of the input features $x$ into a shorter representation that can be found in the middle hidden layer. Let us call this representation of $x$ as hidden vector $z$.




### The Decoder

The decoder represents the part of the autoencoder where the number of neurons in the hidden layers increases again.

The shorter, latent representation of input features that are encoded into the middle hidden layer, are used by the decoder to reconstruct the original input features $x$. Let us call the reconstructed input as $x\_hat$.




If the decoder is able to reconstruct the input features ***x*** from this much shorter vector ***z*** than this means that ***x*** contains a lot for information that is not relevant and can be discarded.

The equations that describe the entire process of encoding and decoding of ***x*** loos as follows:

$$\vec{a_1} = \sigma(W_1 \cdot \vec{x})$$
$$\vec{z} = \sigma(W_2 \cdot \vec{a_1})$$
$$\vec{a_2} = \sigma(W_3 \cdot \vec{z})$$
$$\hat{\vec{x}} = \sigma(W_4 \cdot \vec{a_2})$$

Here **W1**, **W2**, **W3**, and **W4** represent the weight matrices that connect the layers of the autoencoder. sigma represents an arbitrary non-linear activation function.

This latent representation ***z*** is very important because it can be used for many different purposes, such as detecting fraud in the financial transaction, as we will see in a few minutes. First, we will discuss how to train an autoencoder so that we get an accurate latent representation of the inputs.

## Training of an Autoencoder

The autoencoder is trained the same way as a feedforward neural network. We must just minimize the distance between input features and their reconstructed counterpart ***x_hat***.

The distance between ***x*** and ***x_hat*** can be properly measured by the mean squared error loss function. In order to minimize the distance or the value of

the loss function we must use regular gradient descent approach:

Minimize MSE loss with Gradient Descent

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=0}^{N}(\vec{x_i} - \hat{\vec{x_i}})^2$$

By minimizing the MSE loss function we reduce the difference between ***x*** and ***x_hat***. These automatically leads to a better latent representation ***z*** of the input features ***x***.

## 6. Anomaly Detection with AutoEncoder

Now that we have learned what an AutoEncoder is and how it can be trained, let's discuss how we can use this neural network architecture to detect anomalies in the data.

We have to be clear about the fact that anomalies in a dataset are very rare events.

This means that the datasets that should be examined for anomalies of any kind are very imbalanced. The vast majority of the instances in the dataset will be absolutely normal — in contrast to the only very few anomalous data instances.

To put this in perspective, in the upcoming anomaly detection example, the dataset that we are going to use will contain more than 280,000 instances of credit card transaction data, of which only 492 are fraudulent. *That's just 0.17% of the data instances that can be considered as anomalies.* And this imbalance is a serious problem.

The mentioned dataset is the famous " Credit Card Fraud Dataset ". It contains transactions made by credit cards in September 2013 by European cardholders. This dataset presents transactions that occurred in two days, where we have 492 frauds out of 284,807 transactions.

The dataset looks as follows:

| | Time | V1 | V2 | V3 | V4 | V5 | ... | V28 | Amount | Class |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0.0 | -1.359807 | -0.072781 | 2.536347 | 1.378155 | -0.338321 | ... | -0.021053 | 149.62 | 0 |
| **1** | 0.0 | 1.191857 | 0.266151 | 0.166480 | 0.448154 | 0.060018 | ... | 0.014724 | 2.69 | 0 |
| **2** | 1.0 | -1.358354 | -1.340163 | 1.773209 | 0.379780 | -0.503198 | ... | -0.059752 | 378.66 | 0 |
| **3** | 1.0 | -0.966272 | -0.185226 | 1.792993 | -0.863291 | -0.010309 | ... | 0.061458 | 123.50 | 1 |
| **4** | 2.0 | -1.158233 | 0.877737 | 1.548718 | 0.403034 | -0.407193 | ... | 0.215153 | 69.99 | 0 |

- Features **V1**, ... , **V28** are principal components obtained with PCA (principal component analysis)
- "**Time**": Seconds elapsed between each transaction and the very first transaction
- "**Amount**": The transaction amount
- "**Class**": Is our label. 1 represents a fraudulent transaction, 0 otherwise

## Train Autoencoder for Anomaly Detection

We will have difficulties performing the feature label-based supervised training of neural networks as before. Because this time we just do not have enough data instances that describe the anomalies that we want to detect — in our case fraudulent transactions.

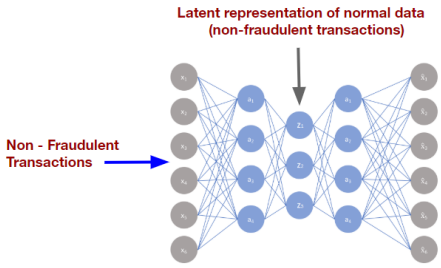
Therefore, it might be very difficult for our model to learn the general concept of an anomaly or a fraudulent transaction, simply because the model almost never sees these kinds of instances. Add to that the fact that we need to divide the dataset into a training, testing and validation set, which would further reduce the number of anomalies in the corresponding datasets.

The general consequence of a very limited number of anomalous data instances is that our model would be very inefficient in classifying these events, as in the vast majority of cases the model only learns from normal data instances. Fortunately, this is the time when we can use an Autoencoder to help us out there. We can use the unique properties of this neural network to solve the problem of having a very imbalanced dataset.

This means that instead of providing labels that classify the input features being either fraudulent or not, we compare the prediction of the Autoencoder with the initial input features.

Additionally, in the case of fraud detection, we must train the Autoencoder only on non-fraudulent data instances. During training, the Autoencoder will

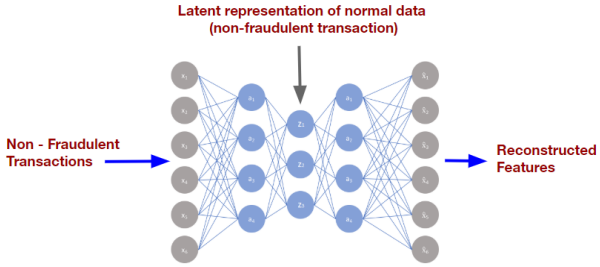
see millions of only non-fraudulent credit card transactions.




In this way, we use the Autoencoder to encode the underlying or most relevant information of the input features into a shorter, latent representation. Intuitively, we can say that this way the Autoencoder only learns the concept of an absolutely normal credit card transaction.

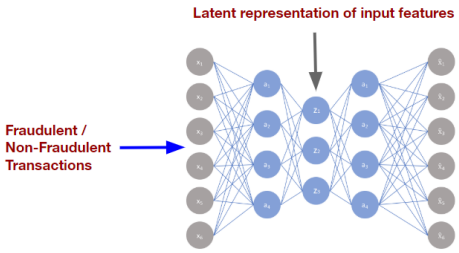
And this learned concept of a normal credit card transaction can be found as the latent representation in the middle layer, which is used to recreate the original features used as input data for the Autoencoder.

## Use trained Autoencoder to detect Anomalies




After training with the normal data instances, we can finally use the neural network to detect the anomalies.

This time, we can show the network data of both types — anomalies as well as normal data. As before, the input features are encoded by the autoencoder into a latent representation that is used to reconstruct the input.




As already mentioned, an AutoEncoder uses the mean squared error function as the loss function to measure the difference or the error between the reconstructed and the original input.

**Please pay attention to the following:** If the autoencoder was trained properly we expect the error between the output and the input to be quite small for normal data.

***However, this is not the case for the anomalies or in our case fraudulent data transactions.***

Remember that during the training the autoencoder learned only the concepts and characteristics of normal data instance. This means that the weights and biases of the autoencoder have only been adjusted to encode and reconstruct normal data — in our case non-fraudulent transactions.

If we now try to encode a fraudulent transaction, the latent representation of this transaction would be significantly different from the latent representation of a normal transaction.

**As a direct result, the reconstructed input would differ even more from the original input, resulting in a much larger error than in the case of normal transactions.**

The knowledge of the fact that fraudulent input data results in higher loss values of the mean squared error loss function can be used to our advantage. All we need to do is to find a **loss threshold** that distinguishes the normal from fraudulent data.

**In practice, this would mean that data instances for which we get a loss value that is above this threshold classify this instance as an anomaly or in our case fraudulent transaction.**

On the other hand, data instances with loss values below this threshold can be

considered as normal data or non-fraudulent transactions.

- **Case 1:** The MSE loss value for an input feature is **above the loss threshold** → the input feature is an anomaly(here: fraudulent transaction)
- **Case 2:** The MSE loss value for an input feature is **below the threshold** → the input feature is normal (here: non-fraudulent transaction)

### Recap: Anomaly Detection with Autoencoder

Let us summarize what we have learned about how to detect anomalies in a dataset.

1. First, if your dataset that should be examined for anomalies is imbalanced (which is almost always the case) you must use an autoencoder
2. Train the autoencoder using the mean squared error loss function only on normal data, don't use any anomalous data instances during training
3. After the training, the loss value for anomalies should be much higher than for normal data instances
4. Now, you have to find out a loss value threshold that best distinguishes anomalies from normal data.

### Finding a proper Loss-Value-Threshold

In order to find this threshold value that best distinguishes anomalies from normal data you simply have to try out many different threshold values.

For this procedure, I recommend taking a few thousand data instances, where approximately 5–10% of these instances are anomalies. Then you simply compute for each instance the loss value, compare it with the threshold and classify it as either an anomaly or not.

In the end, you have to check how good these classifications were, using the actual labels. To quantify the classification result I recommend to use the evaluation metrics that we have covered in the previous article " Evaluation Metrics in Data Science and Machine Learning ".

Having the results of the evaluation metrics, you can decide whether this threshold was good and adjust the threshold if necessary.

## 7. Fraud Detection in TensorFlow 2.0

As you might have already guessed the anomaly detection model will be an Autoencoder that will identify fraudulent financial transactions in the previously introduced dataset.

All source code and used datasets can be accessed in my GitHub repository of this project. Feel free do download the code and try it out for yourself.

Unfortunately, I can not go into detail on all used functions and classes, because it would go beyond the scope of the course. Rather I would like to focus solely on the implementation of the neural network model. All other used classes and methods (in particular data-preprocessing can be viewed in the GitHub repository). All source code is well documented so that you will not have any difficulties in understanding the code.

On that note, let us begin...

First I like to define a **BaseModel** class that contains the methods for initialization of weights and biases and computation of the forward pass.

These methods will be later inherited to the class that will define the actual anomaly detection model. This way we can later define several Autoencoder models (with different hyperparameters for example) that will inherit methods from the **BaseModel** class. By doing this we can save quite a lot of code since the different versions of the model share the same functionalities.

In **init_variables(self)** we define the weight and biases. As you can see the autoencoder will have three hidden layers in total. The number of neurons in these layers is [20, 8, 20].

```
import tensorflow as tf
from tensorflow.keras import Model

class BaseModel(Model):
    '''This is the base model class that inherits some sharable functions to the the mo
    This class also inherits from the tensorflow.keras.Model.

    '''
    def __init__(self):

        super(BaseModel, self).__init__()

        self.weight_initializer=tf.random_normal_initializer(mean=0.0, stddev=0.25)
        self.bias_initializer=tf.zeros_initializer()

    def init_variables(self):
        '''Initialize the parameters of the neural network. '''

```

```
        self.W1=tf.compat.v1.get_variable('W1',shape=[29,20], initializer=self.weight_i
        self.W2=tf.compat.v1.get_variable('W2',shape=[20,8], initializer=self.weight_in
        self.W3=tf.compat.v1.get_variable('W3',shape=[8,20], initializer=self.weight_in
        self.W4=tf.compat.v1.get_variable('W3',shape=[20,29], initializer=self.weight_i

        self.b1=tf.compat.v1.get_variable('b1',shape=[20], initializer=self.bias_initia
        self.b2=tf.compat.v1.get_variable('b2',shape=[8], initializer=self.bias_initial
        self.b3=tf.compat.v1.get_variable('b3',shape=[20], initializer=self.bias_initia


    def forward_propagation(self, x):
        '''Compute the forward pass given the input features x.

        @param x: input features x

        @return prediction: the reconstructed input features x
        '''

        with tf.name_scope('feed_forward'):

            # First hidden layer
            z1=tf.linalg.matmul(x, self.W1)+self.b1
            a1=tf.nn.relu(z1)

            # Second hidden layer
            z2=tf.linalg.matmul(a1,self.W2)+self.b2
            a2=tf.nn.relu(z2)

            # Third hidden layer
            z3=tf.linalg.matmul(a2,self.W3)+self.b3
            a3=tf.nn.relu(z3)

            prediction=tf.linalg.matmul(a3,self.W4)

        return prediction
```

base_model.py hosted with ♥ by GitHub view raw

**forward_propagation(self, x)** just computes the output of the autoencoder, which is the reconstructed input features

In the following, we can define the actual class where the autoencoder will be trained:

```
class AnomalyDetector(BaseModel):
    '''This class represents the class for training of the neural network for anomaly d
    In particular this class is used for training only. The learned weights and biases
    by the inference model to make the actual anomaly detection in production environme
    '''

    def __init__(self):

        super(AnomalyDetector, self).__init__()
        self.init_variables()

```

```
        self.optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)


    def compute_loss(self, x_train):
        '''Compute MSE loss function.

        @param x_train: input features
        '''

        return loss


    def train(self, x_train):
         '''Train the autoencoder.

         @parameter x_train: training input features
         '''

         # Compute the gradients and apply the gradient descent step
         with tf.GradientTape() as tape:
             gradients = tape.gradient(self.compute_loss(x_train), self.trainable_varia
             gradient_variables = zip(gradients, self.trainable_variables)
             self.optimizer.apply_gradients(gradient_variables)
```

anomaly_detection.py hosted with ❤ by GitHub view raw



This class contains the methods for the computation of the Mean Squared Error loss value and the gradient descent step.

The training process should be self-explanatory:

```
### Define some hyperparameters ###

# Number of training samples
n_training_samples=250000
# Batch size
batch_size=64
# Learning rate
learning_rate=0.001
# Number of test data samples
n_test_samples=34806
# Number of epochs
num_epoch=50
# number of batches
n_batches=int(n_training_samples/batch_size)
# Evaluate model after number of steps
eval_after=1000
# Path of the TF Records datasets for the training
train_path=os.path.abspath(os.path.join(os.path.dirname("__file__"), '..', 'data/tf_rec
# Path of the TF Records datasets for the testing
test_path=os.path.abspath(os.path.join(os.path.dirname("__file__"), '..', 'data/tf_reco

# Loss value threshold
```

```
THRESHOLD=10

# Initialize the instance of the class for anomaly detection
model=AnomalyDetector()

# Initialihe the instance of the class for the perfromance measurements
performance=Performance(THRESHOLD)

# Get the training and test datasets
training_dataset=get_training_data(train_path)
test_dataset=get_test_data(test_path)

#Iterate over the epochs
for epoch in range(num_epoch):

    temp_loss=0

    #Iterate over the batches
    for step, x_train in enumerate(training_dataset):

        # Extract features and labels
        features, labels=x_train

        # Train the model
        model.train(features)

        # Compute the loss
        loss_values=model.compute_loss(features)
        temp_loss+=loss_values

        # Evalute the model on the test set
        if step>0 and step%eval_after==0:

            # Iteratre over the test dataset
            for step_test, x_test in enumerate(test_dataset):

                # Extract the features and labels
                features, label=x_test

                label=label.numpy()

                # Compute the loss
                test_loss=model.compute_loss(features)

                # Evaluate the test data sample
                performance.eval_prediciton(test_loss, label)


            print('epoch_nr: %d, batch: %d/%d, mse_loss: %.3f'%(epoch, step, n_batches,

            # Plot ROC curce and show other evaluation metrics
            performance.evaluate_model()
            performance.reset()
```

training.py hosted with ❤ by GitHub view raw

First, we divine several hyperparameters like the batch-size, learning rate, etc.

A very important hyperparameter is the introduced loss-value threshold, that I call here as *THRESHOLD=10*. This value of MSE loss function will distinguish during the testing phase anomalies from normal data instances.

Then we define the instance of the classes **AnomalyDetector(),** which is the actual Autoencoder model and **Performance(THRESHOLD)**, where some evaluation metrics (Precision, Recall, F1-Score) will be calculated. If you are not familiar with these metrics, please feel free to check out the article " Evaluation Metrics in Deep Learning."

The training and evaluation datasets were previously converted into the TensorFlow Records data format. In the methods *get_training_data()* and **get_test_data()** I use the **tf.data API** to extract, load, make mini-batches and shuffle the datasets.

In the next step we iterate over the training dataset, compute for each mini-batch the Mean Squared Error loss function and apply gradient descent.

After 1000 mini-batches have been processed, I use the **Performance()** class together with the loss-value-threshold to evaluate the anomaly detector model on approx. 34000 instances in the test dataset. The test dataset contains all 480 fraudulent data instances (anomalies) and the rest are normal data instances.

Just after 5 epochs, we get the following results:

- The anomaly detector has a very good precision value of 93. This means that 93% of the time the model correctly classifies an anomaly.
- The recall value of 43,3 tells us that the autoencoder has identified 43,3% of anomalies in the dataset. Unfortunately more than half of the anomalies remain invisible to the model.

It is interesting to see that for non-fraudulent data instances the mean absolute error loss value is just 0.385. On the other hand, the MSE loss value for fraudulent data instances is with a value of 19.518 much higher. Just as it should be and exactly what we expect.

*Originally published at* https://www.deeplearning-academy.com/p/ai-wiki-anomaly-detection