Anomaly Detection with Robust Deep Autoencoders

Alessandro Trenta

Introduction

- Anomaly detection is a fundamental problem when dealing with large amounts of data.
- A single outlier or anomaly in the pipeline of a product could lead to bad quality or even in security hazards.
- e.g. we want to know if a battery is in good state and to predict whether there is a concrete risk of damage/explosion.
- The main challenge is related to the fact that anomalies are sparse points hidden in a large amount of normal data.

Introduction

- In this presentation we are going to have a look at the Robust Deep Autoencoder models [ZP17] and their applications.
- These models could be used to remove noise from data and, most importantly, detect anomalies.
- First, we are going to have a look at the main ideas and components of the model.

Deep Autoencoders

- A Deep Autoencoder (DAE) has two main components: an encoder E and a Decoder D. It produces a low dimensional representation of data $Z = E(X)$.
- The DAE learns the identity map so that the reconstruction $\bar{X} = D(E(X))$ is as close as possible to the original input X.
- E, D can be any mapping between the data space and the coded space. Usually we use FCNN or more complex models (e.g. LSTM or GRU).
- The loss function: it is the minimum reconstruction error w.r.t. some parametrized encoding and decoding functions and a distance (in this case the L_2 norm)

$$\min_{\theta,\phi} \|X - D_\theta(E_\phi(X))\|_2 \tag{1}$$

Principal Component Analysis (PCA)

- Assume following data shape: N samples of d dimensional data $X \in \mathbb{R}^{N \times d}$ and assume each feature has 0 mean.
- Mathematically, PCA is an orthogonal linear transformation U s.t. in the new coordinate system the i-th component has the i-th greatest data variance.
- PCA is often used for dimensionality reduction or encoding: simply project the data on the first $k < d$ principal components and use them as new features.

Principal Component Analysis

Define:

$$w_1 = \underset{\|w\|_2=1}{\arg\max} \|Xw\|_2^2 = \underset{w}{\arg\max} \frac{w^T X^T X w}{w^T w} \tag{2}$$

for the first component. Then for the k-th component we first subtract the first $k-1$ principal component from X

$$\hat{X}_k = X - \sum_{i=1}^{k-1} X w_i w_i^T \tag{3}$$

and finally solving again the similar problem:

$$w_k = \underset{\|w\|_2=1}{\arg\max} \|\hat{X}_k w\|_2^2 = \underset{w}{\arg\max} \frac{w^T \hat{X}_k^T \hat{X}_k w}{w^T w} \tag{4}$$

Robust Principal Component Analysis

- Robust Principal Component Analysis (RPCA) is a generalization of PCA that aims to reduce the sensitivity of PCA to outliers and noise.
- Idea: separate noise and outliers, then learn a low dimensional representation of cleaned data.
- Assume that data X can be represented as $X = L + S$: L has low rank and is the low-dimensional representation of X while S is a sparse matrix containing outliers and anomalous data.

Robust Principal Component Analysis

- The problem can be addressed as:

$$\min_{L,S} \rho(L) + \lambda \|S\|_0 \tag{5}$$

$$\text{s. t. } \|X - L - S\|_F^2 = 0 \tag{6}$$

where $\rho(\cdot)$ is the rank of a matrix and we used the zero norm.

- This optimization problem is NP-hard and tractable only for small metrices.
- We use this objective is instead:

$$\min_{L,S} \|L\|_* + \lambda \|S\|_1 \tag{7}$$

$$\text{s. t. } \|X - L - S\|_F^2 = 0 \tag{8}$$

where $\|\cdot\|_*$ is the nuclear norm i. e. the sum of singular values of a matrix.

Robust Deep Autoencoders

- Robust Deep Autoencoders (RDAE) combine the representation learning of DAEs and the anomaly detection capability of RPCA.
- Noise and outliers are incompressible in the lower dimensional space we want to represent our data in.
- We want to exclude anomalies and learn a low dimensional representation of data.
- There are two kinds of RDAE, one for l_1 regularization and one for $l_{2,1}$.

RDAE with l_1 regularization

- We try to decompose data as $X = L_D + S$ as in RPCA.
- We then combine the two losses in the following minimization problem

$$\min_{\theta} \|L_D - D_\theta(E_\theta(L_D))\|_2 + \lambda \|S\|_0 \tag{9}$$

$$\text{s.t. } X - L_D - S = 0 \tag{10}$$

- The parameter λ controls the sparsity of S. A smaller λ means that the norm of S is less important w.r.t. DAE loss (better reconstruction, less outliers) and viceversa.
- Finding the best value for λ is the main challenge.

The true objective

- The previous loss is higly non tractable. We focus on the following problem:

$$\min_{\theta} \|L_D - D_\theta(E_\theta(L_D))\|_2 + \lambda\|S\|_1 \tag{11}$$

$$\text{s.t. } X - L_D - S = 0 \tag{12}$$

- The autoencoder is trained with L_D, the supposedly cleaned part.

Regularization

- The RDAE with l_1 penalization assumes that outliers and noise are not structured. The l_1 penalty just induces sparsity. We could have different kind of anomalies:
- Feature (column) wise: a feature is corrupted in many samples e.g. a broken pixel in a sensor.
- Data (row) wise: a particular sample is anomalous.

The $l_{2,1}$ norm

- The $l_{2,1}$ norm is defined as ($X \in \mathbb{R}^{N \times d}$):

$$\|X\|_{2,1} = \sum_{j=1}^{n} \|X_j\|_2 = \sum_{j=1}^{n} \left(\sum_{i=1}^{N} |X_{ij}|^2 \right)^{\frac{1}{2}} \qquad (13)$$

- The $l_{2,1}$ norm can be seen as introducting a l_2 norm regularization over data for each feature and then adding a l_1 regularization accross features.
- We can also do the other way around: to recognize data anomalies (by row) just apply the $l_{2,1}$ norm to X^T.

- The final optimization problem for the RDAE with $l_{2,1}$ regularization for data anomalies is then

$$\min_{\theta} \|L_D - D_\theta(E_\theta(L_D))\|_2 + \lambda \|S^T\|_{2,1} \tag{14}$$
$$\text{s.t. } X - L_D - S = 0 \tag{15}$$

- For detecting feature anomalies we just need to change the objective to

$$\min_{\theta} \|L_D - D_\theta(E_\theta(L_D))\|_2 + \lambda \|S\|_{2,1} \tag{16}$$
$$\text{s.t. } X - L_D - S = 0 \tag{17}$$

The proximal operator

- To see in detail the training procedure for the RDAE we first need to consider the proximal operator.
- General framework: find the solution to $\min f(x) + \lambda g(x)$ where g is convex. Consider

$$\text{prox}_{\lambda,g}(x) = \arg\min_{y} g(y) + \frac{1}{2\lambda}\|x - y\|_2^2 \tag{18}$$

- In the case of proximal gradient optimization the iterative step is defined as:

$$x^{k+1} = \text{prox}_{\lambda,g}(x^k - \alpha \nabla f(x^k)) \tag{19}$$

- In this case we then want to obtain a solution of the problems

$$\mathrm{prox}_{\lambda,l_1}(x) = \arg\min_y l_1(y) + \frac{1}{2\lambda}\|x-y\|_2^2 \qquad (20)$$

$$\mathrm{prox}_{\lambda,l_{2,1}}(x) = \arg\min_y l_{2,1}(y) + \frac{1}{2\lambda}\|x-y\|_2^2 \qquad (21)$$

- As we will see shortly we have explicit formulas for this operators and they can get computed really fast.

- For the l_1 norm, the solution to the proximal problem is

$$\text{prox}_{\lambda,l_1}(x) = \begin{cases} x_i - \lambda, & x_i > \lambda \\ x_i + \lambda, & x_i < -\lambda \\ 0, & x_i \in [-\lambda, \lambda] \end{cases} \quad (22)$$

  for $S \in \mathbb{R}^{N \times d}$ it gets applied element by element.
- $l_{2,1}$ norm: for feature anomalies we obtain (letting $S_{\cdot j}$ be the column vector $S_{ij}, j = 1, \ldots, N$)

$$(\text{prox}_{\lambda,l_{2,1}}(S))_{ij} = \begin{cases} S_{ij} - \lambda \frac{S_{ij}}{\|S_{\cdot j}\|_2}, & \|S_{\cdot j}\|_2 > \lambda \\ 0, & \|S_{\cdot j}\|_2 \leq \lambda \end{cases} \quad (23)$$

- Substitute S with S^T for data anomalies.

The main algorithm

- The proposed method to train the RDAE is the Alternating Direction Method of Multipliers (ADMM).
- It is a two-step iterative process to solve the problem

$$\min_{\theta} \|L_D - D_\theta(E_\theta(L_D))\|_2 + \lambda \|S^T\|_{2,1} \tag{24}$$

$$\text{s.t. } X - L_D - S = 0 \tag{25}$$

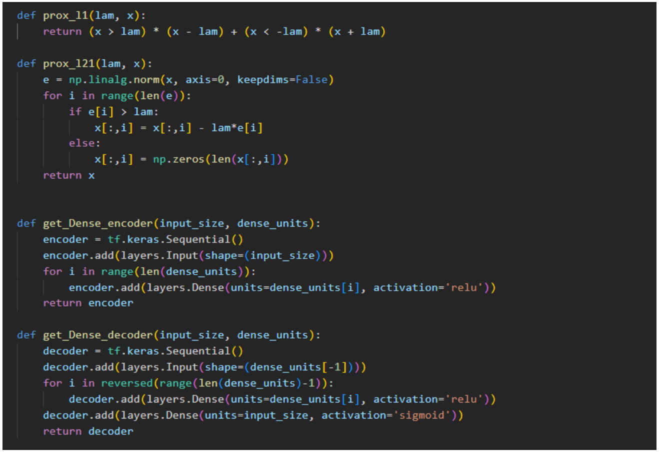
- First, we fix S and optimize the DAE loss $\|L_D - D_\theta(E_\theta(L_D))\|_2$ with backpropagation.
- Then, we fix L_D and optimize the regularization term with the proximal method.

The full procedure is the following: given input $X \in \mathbb{R}^{N \times n}$, initialize $L_D \in \mathbb{R}^{N \times n}$, $S \in \mathbb{R}^{N \times n}$ as zero matrices, $L_S = X$ and initialize the DAE randomly. For each iteration do:

- $L_D = X - S$
- Minimize $\|L_D - D_\theta(E_\theta(L_D))\|_2$ with backpropagation.
- Set $L_D = D(E(L_D))$ as the reconstruction.
- Set $S = X - L_D$.
- Optimize S using a $\text{prox}_{\lambda, l.}$ function of choice.
- If $c_1 = \frac{\|X - L_D - S\|_2}{\|X\|_2} < \epsilon$ or $c_2 = \frac{\|L_S - L_D - S\|_2}{\|X\|_2} < \epsilon$ we have early convergence.
- Set $L_S = L_D + S$.

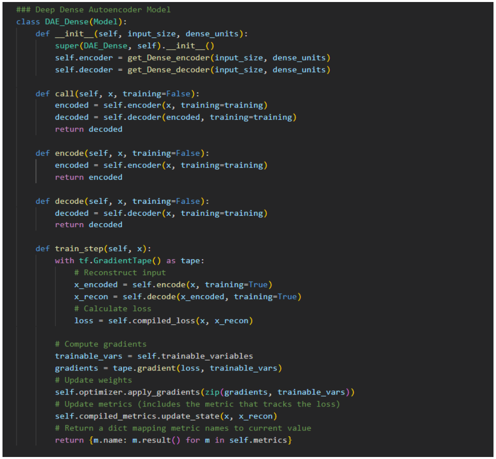
Return L_D and S.

Implementation



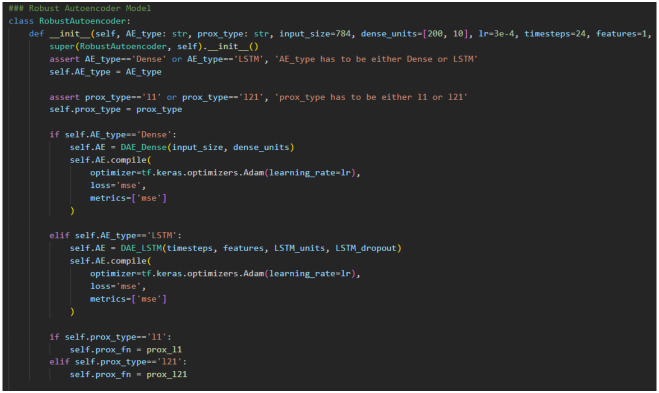
```python
def prox_l1(lam, x):
    return (x > lam) * (x - lam) + (x < -lam) * (x + lam)

def prox_l21(lam, x):
    e = np.linalg.norm(x, axis=0, keepdims=False)
    for i in range(len(e)):
        if e[i] > lam:
            x[:,i] = x[:,i] - lam*e[i]
        else:
            x[:,i] = np.zeros(len(x[:,i]))
    return x


def get_Dense_encoder(input_size, dense_units):
    encoder = tf.keras.Sequential()
    encoder.add(layers.Input(shape=(input_size)))
    for i in range(len(dense_units)):
        encoder.add(layers.Dense(units=dense_units[i], activation='relu'))
    return encoder

def get_Dense_decoder(input_size, dense_units):
    decoder = tf.keras.Sequential()
    decoder.add(layers.Input(shape=(dense_units[-1])))
    for i in reversed(range(len(dense_units)-1)):
        decoder.add(layers.Dense(units=dense_units[i], activation='relu'))
    decoder.add(layers.Dense(units=input_size, activation='sigmoid'))
    return decoder
```

Implementation



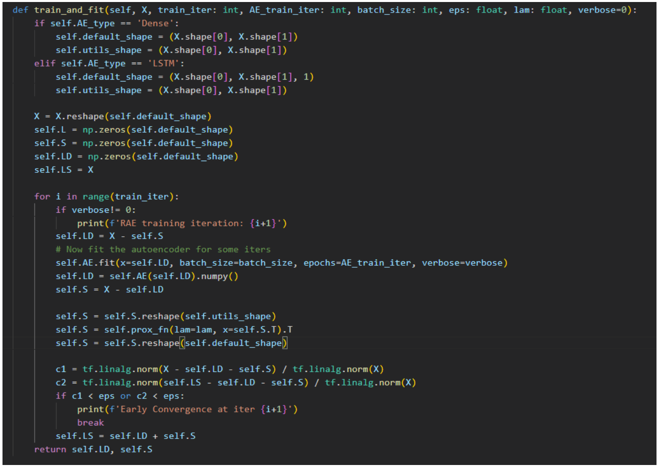
```python
### Deep Dense Autoencoder Model
class DAE_Dense(Model):
    def __init__(self, input_size, dense_units):
        super(DAE_Dense, self).__init__()
        self.encoder = get_Dense_encoder(input_size, dense_units)
        self.decoder = get_Dense_decoder(input_size, dense_units)

    def call(self, x, training=False):
        encoded = self.encoder(x, training=training)
        decoded = self.decoder(encoded, training=training)
        return decoded

    def encode(self, x, training=False):
        encoded = self.encoder(x, training=training)
        return encoded

    def decode(self, x, training=False):
        decoded = self.decoder(x, training=training)
        return decoded

    def train_step(self, x):
        with tf.GradientTape() as tape:
            # Reconstruct input
            x_encoded = self.encode(x, training=True)
            x_recon = self.decode(x_encoded, training=True)
            # Calculate loss
            loss = self.compiled_loss(x, x_recon)

        # Compute gradients
        trainable_vars = self.trainable_variables
        gradients = tape.gradient(loss, trainable_vars)
        # Update weights
        self.optimizer.apply_gradients(zip(gradients, trainable_vars))
        # Update metrics (includes the metric that tracks the loss)
        self.compiled_metrics.update_state(x, x_recon)
        # Return a dict mapping metric names to current value
        return {m.name: m.result() for m in self.metrics}
```

Implementation



```
### Robust Autoencoder Model
class RobustAutoencoder:
    def __init__(self, AE_type: str, prox_type: str, input_size=784, dense_units=[200, 10], lr=3e-4, timesteps=24, features=1,
        super(RobustAutoencoder, self).__init__()
        assert AE_type=='Dense' or AE_type=='LSTM', 'AE_type has to be either Dense or LSTM'
        self.AE_type = AE_type

        assert prox_type=='l1' or prox_type=='l21', 'prox_type has to be either l1 or l21'
        self.prox_type = prox_type

        if self.AE_type=='Dense':
            self.AE = DAE_Dense(input_size, dense_units)
            self.AE.compile(
                optimizer=tf.keras.optimizers.Adam(learning_rate=lr),
                loss='mse',
                metrics=['mse']
            )

        elif self.AE_type=='LSTM':
            self.AE = DAE_LSTM(timesteps, features, LSTM_units, LSTM_dropout)
            self.AE.compile(
                optimizer=tf.keras.optimizers.Adam(learning_rate=lr),
                loss='mse',
                metrics=['mse']
            )

        if self.prox_type=='l1':
            self.prox_fn = prox_l1
        elif self.prox_type=='l21':
            self.prox_fn = prox_l21
```

Implementation



```python
def train_and_fit(self, X, train_iter: int, AE_train_iter: int, batch_size: int, eps: float, lam: float, verbose=0):
    if self.AE_type == 'Dense':
        self.default_shape = (X.shape[0], X.shape[1])
        self.utils_shape = (X.shape[0], X.shape[1])
    elif self.AE_type == 'LSTM':
        self.default_shape = (X.shape[0], X.shape[1], 1)
        self.utils_shape = (X.shape[0], X.shape[1])

    X = X.reshape(self.default_shape)
    self.L = np.zeros(self.default_shape)
    self.S = np.zeros(self.default_shape)
    self.LD = np.zeros(self.default_shape)
    self.LS = X

    for i in range(train_iter):
        if verbose!= 0:
            print(f'RAE training iteration: {i+1}')
        self.LD = X - self.S
        # Now fit the autoencoder for some iters
        self.AE.fit(x=self.LD, batch_size=batch_size, epochs=AE_train_iter, verbose=verbose)
        self.LD = self.AE(self.LD).numpy()
        self.S = X - self.LD

        self.S = self.S.reshape(self.utils_shape)
        self.S = self.prox_fn(lam=lam, x=self.S.T).T
        self.S = self.S.reshape(self.default_shape)

        c1 = tf.linalg.norm(X - self.LD - self.S) / tf.linalg.norm(X)
        c2 = tf.linalg.norm(self.LS - self.LD - self.S) / tf.linalg.norm(X)
        if c1 < eps or c2 < eps:
            print(f'Early Convergence at iter {i+1}')
            break
        self.LS = self.LD + self.S
    return self.LD, self.S
```

Results

- We now have a look at how the model performs on some tasks.
- We will initially use the MNIST digit dataset, using the 50000 train images available.
- Data was flattened from images of shape $(28, 28, 1)$ into vectors of length 784. Train data is then a matrix in $\mathbb{R}^{50000\times784}$.
- Pixel walues are converted from integers between 0 and 255 to floats between 0 and 1.

l_1 Robust Deep Autoencoder

- We take a RDAE with a FCNN architecture with layers of size 784 (input), 200 and 10 (the bottleneck and hidden feature layer). 10 outer iterations and 100 inner iterations.
- The training images get corrupted with a percentage of pixel (from 5% to 50%) changed to a random value between 0 and 1. These are used to train the RDAE.
- From the RDAE obtain the two matrices L_D, the cleaned data, and S, the sparse and anomalous part.

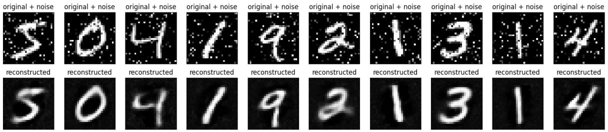


Figure: RAE cleaned data, corruption 10%

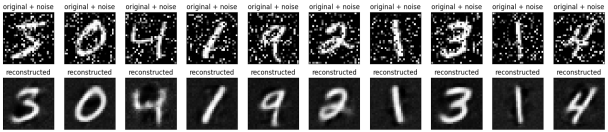


Figure: RAE cleaned data, corruption 20%

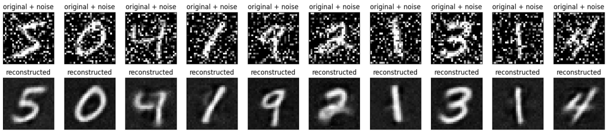


Figure: RAE cleaned data, corruption 30%

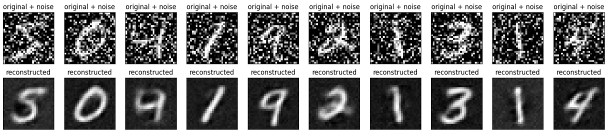


Figure: RAE cleaned data, corruption 40%

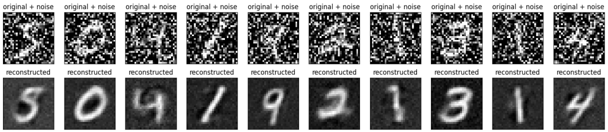


Figure: RAE cleaned data, corruption 50%

$l_{2,1}$ Robust Deep Autoencoder

- To assess the performance of RDAE in anomaly detection we start by using a synthetic labeled dataset.
- All the 4 digit images in the training set are collected in our dataset.
- Then, some images are chosen at random from all the other digits until they reach 5% of total images in the dataset.
- These will be considered as the outliers of our data.

- The $l_{2,1}$ RDAE is trained on this dataset without any side information. It has to recognize outliers completely on its own. Same architecture as before.
- The only parameter that requires tuning is λ. We assess performance with the accuracy, precision, recall and $F1 = 2\frac{PR}{P+R}$ metrics.
- We consider an instance anomalous whenever the S matrix has non-zero entries on its row.

Anomaly detection performance

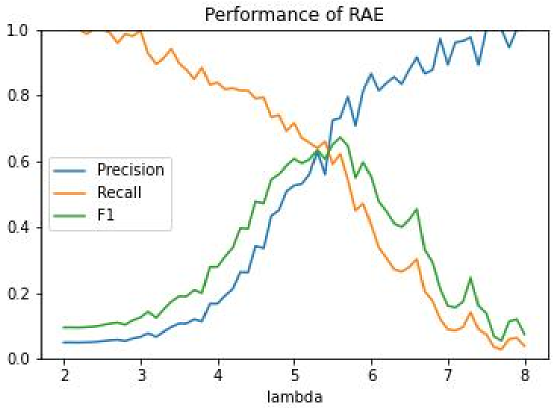


Figure: $L_{2,1}$ RDAE anomaly detection performance. λ from 2 to 8

Anomaly detection performance

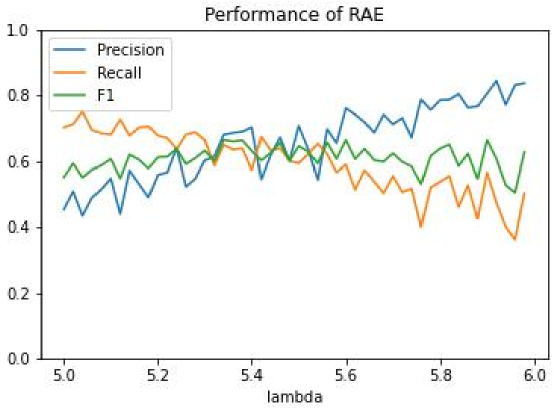


Figure: $L_{2,1}$ RDAE anomaly detection performance. λ from 5 to 6

Anomaly detection performance

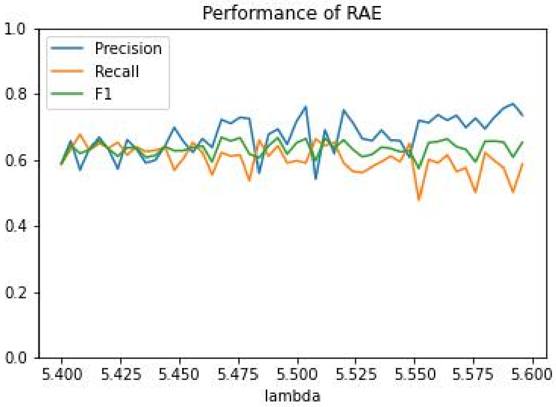


Figure: $L_{2,1}$ RDAE anomaly detection performance. λ from 5.4 to 5.6

- The maximum performance is obtained with $\lambda = 5.468$ with an F_1 score of 0.668.
- Focusing on all values from $\lambda = 5$ to $\lambda = 6$ the RDAE has an accuracy of over 95% in recognizing anomalies. The F_1 score in this range is almost everytime above 0.55.
- The F_1 score is almost everytime above 0.6 for λ in the $[5.4, 5.6]$ range.

For each value of λ we analize 3 images:

- The reconstruction of the original images from the DAE in the RDAE $\bar{X} = D(E(X))$.
- The final L_D image (the "clean" version, in this case it should only contain 4s)
- S image, which should be non empty only for outliers.

We look 3 different values for λ: the best one identified above, 8.0 which adds too much penalization with few outliers identified and 4.0 which is a low value and a lot of 4s are considered outliers.

Original Images data

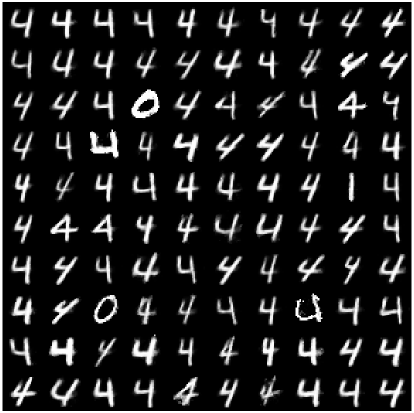
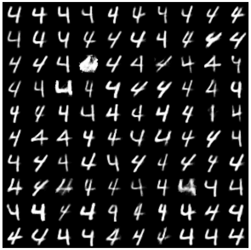
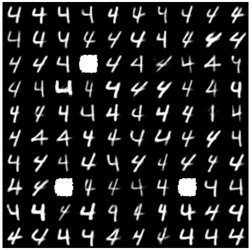


Figure: Original images for the $L_{2,1}$ RDAE

$\lambda = 5.468$


(a) $\bar{X}$, reconstruction


(b) L_D, cleaned data

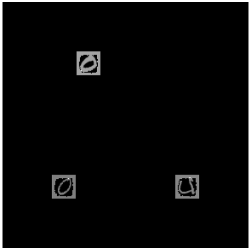
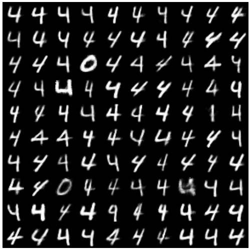
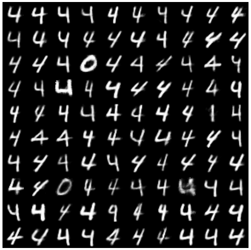

(c) S, outliers

Figure: Accuracy: 0.970, precision: 0.722, recall: 0.621, F1 score: 0.668

$\lambda = 8.0$


(a) $\bar{X}$, reconstruction


(b) L_D, cleaned data

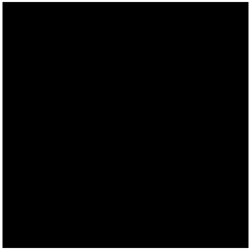
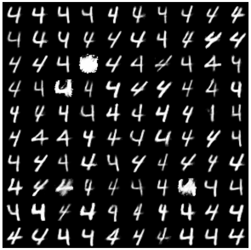
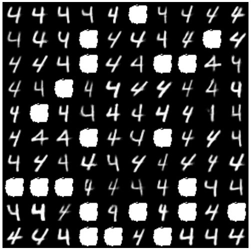

(c) S, outliers

Figure: Accuracy: 0.953, precision: 1.00, recall: 0.0386, F1 score: 0.0743

$\lambda = 4.0$


(a) $\bar{X}$, reconstruction


(b) L_D, cleaned data

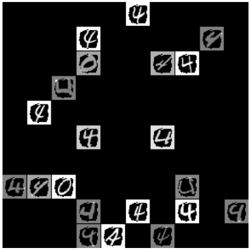

(c) S, outliers

Figure: Accuracy: 0.788, precision: 0.167, recall: 0.839, F1 score: 0.278

- The performance of the RDAE as outlied detector is compared with the one obtained using the isolation forest method.
- The isolation forest method was a SOTA method for outlier detection. It is based on the idea that outliers are few, different and separated from the rest.
- These outliers gets recognized using isolation trees which try to separate points from others.
- The only parameter to be optimized is the outlier fraction (from 0 to 0.5).

Isolation forest performance

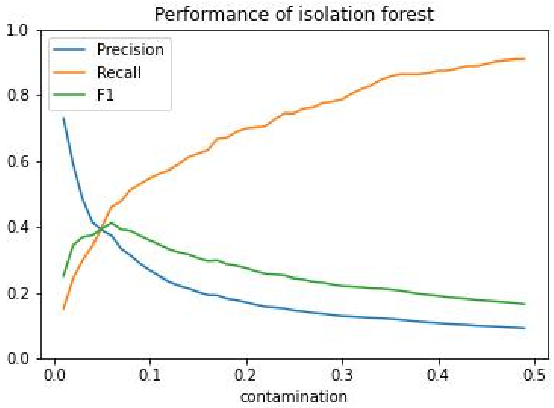


Figure: Isolation forest performance

- The best results is 0.41 with the outlier fraction set to the value of 0.06 (which is really close to the outlier true fraction of 5%).
- In each case the performance by isolation forest is far worse than the RDAE.

Real World Experiment

- Now we have assessed the performance on a synthetic dataset we want to see how to apply this model to real world challenges.
- Data is in general UNLABELED. Specially with loads of data, we can't do hand labeling of anomalies.
- We want to see if the model still works for time series.
- This could have a true application in Electra Vehicles products: given historical battery data if we are able to detect anomalies we can better spot dangers and errors.

Time series experiment

- In this case we are going to use a dataset similar to our interest
- It is taken from the Numenta Anomaly Benchmark (NAB). The database is called machine temperature system failure.
- It is the sensor data of an internal component of a large, industrial mahcine. It should have 3 kinds of anomalies: the first anomaly is a planned shutdown of the machine. The second anomaly should be difficult to detect and directly led to the third anomaly, a catastrophic failure of the machine.

Data processing pipeline

- Data has 22464 timesteps in total. Data signals are taken every 5 minutes.
- I chose to consider subsequences of length 144: this corresponds of windows of 12 hours. The final dataset has then 22321 training time series.
- Data is normalized all togheter to be in $(0, 1)$.

RDAE architectures

I tried using 3 architectures for the autoencoder part in the RDAE.

- The first one is a Dense Neural Network with hidden layers of 60 and 20. It is trained for 20 outer iterations and 50 inner iterations for the autoencoder, batch size 256 , $\epsilon = 10^{-8}$.
- The second and the third one are a LSTM and a GRU with two layers of 32 and 16 units, 10 outer iterations and 25 inner iterations with same batch size as before.

Analysis

- Since data is unlabeled we don't have a clear benchmark for finding the correct value for λ.
- I tried different values for λ to see how the number of outliers scales.
- For each architecture I picked some random anomalies and non-anomalies, to show how the RDAE is acting on time series and to have a look at what kind of anomalies it detectes.

Anomalies found

λ	0.1	0.5	0.7	1.0	2.0	3.0	3.2	3.3	4
Dense	All	751	250	14	0	0	0	0	0
LSTM	All	7525	4208	2068	306	109	74	9	0
GRU	All	7277	4505	2454	331	139	103	80	0

Table: Anomalies found by the two architecures w.r.t. λ

Dense RDAE

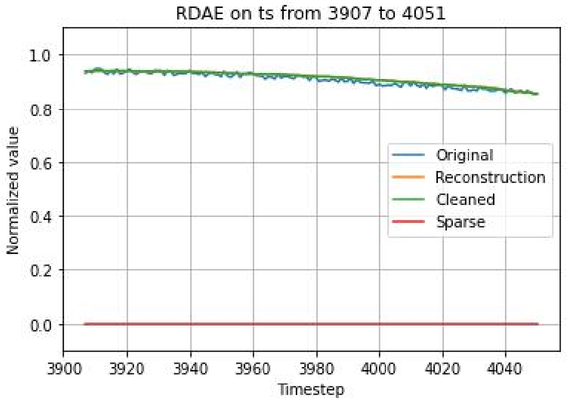


Figure: Example of a non anomaly subsequence for $\lambda = 1.0$

Dense RDAE

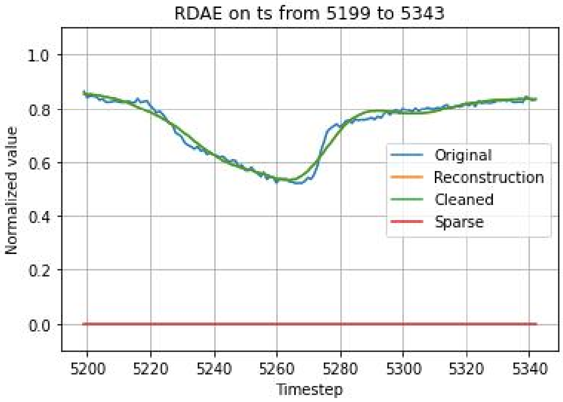


Figure: Example of a non anomaly subsequence for $\lambda = 1.0$

Dense RDAE

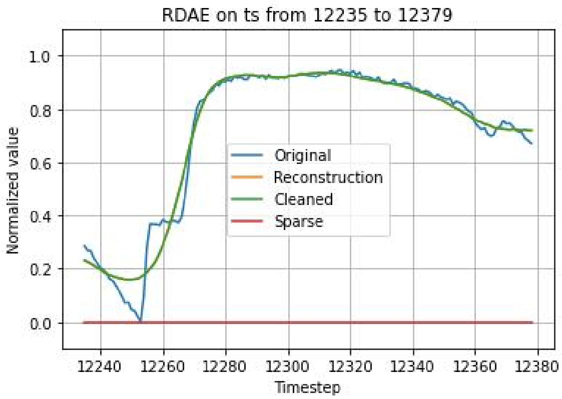


Figure: Example of a non anomaly subsequence for $\lambda = 1.0$

Dense RDAE

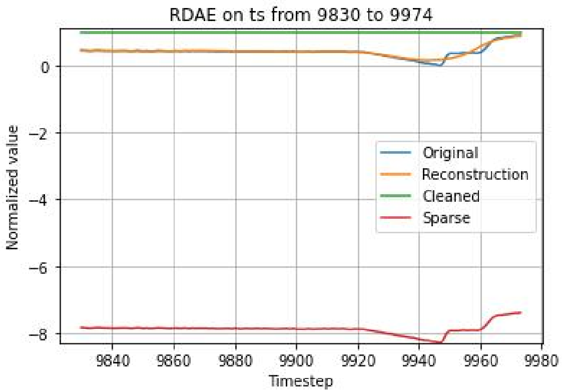


Figure: Example of a anomaly subsequence for $\lambda = 1.0$

Dense RDAE

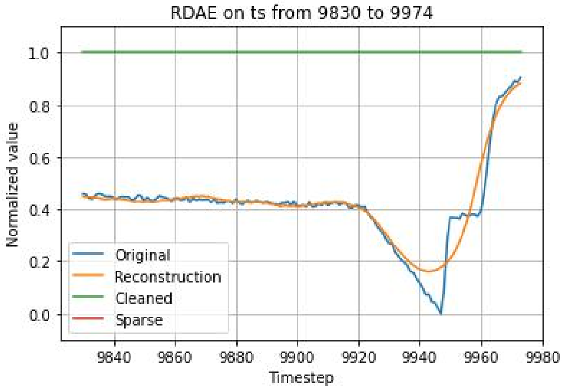


Figure: Example of a anomaly subsequence for $\lambda = 1.0$

Dense RDAE

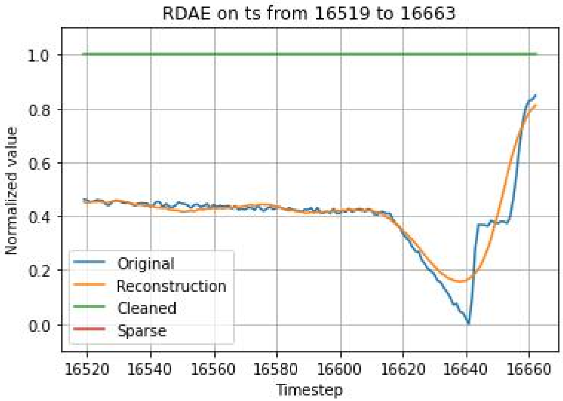


Figure: Example of a anomaly subsequence for $\lambda = 1.0$

Dense RDAE

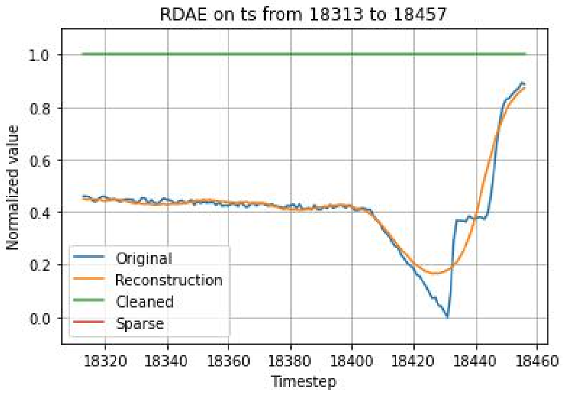


Figure: Example of a anomaly subsequence for $\lambda = 1.0$

Dense RDAE

- All of the anomalies found reach the 0 value (min temperature of all time series).
- Note that the different anomalies found DO NOT overlap. So each of the failures is only recognized once.
- This may also create problems, since as you can see one failure is not recognized as anomaly.
- In general, the reconstruction is a non-noisy version of the signal.

LSTM RDAE

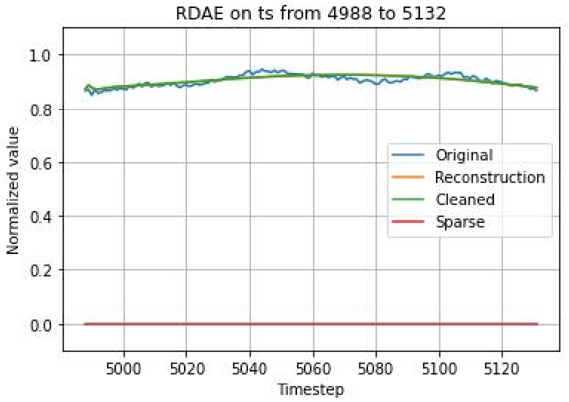


Figure: Example of a non anomaly subsequence for $\lambda = 3.3$

LSTM RDAE

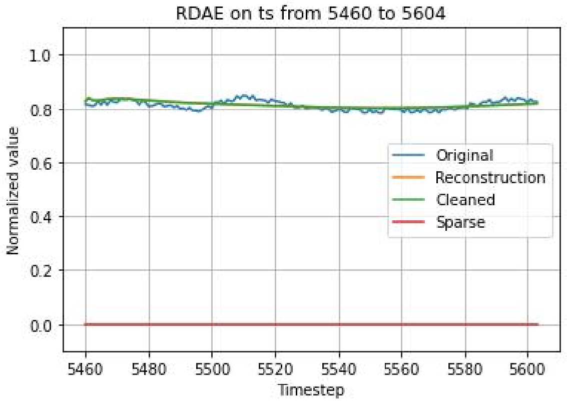


Figure: Example of a non anomaly subsequence for $\lambda = 3.3$

LSTM RDAE

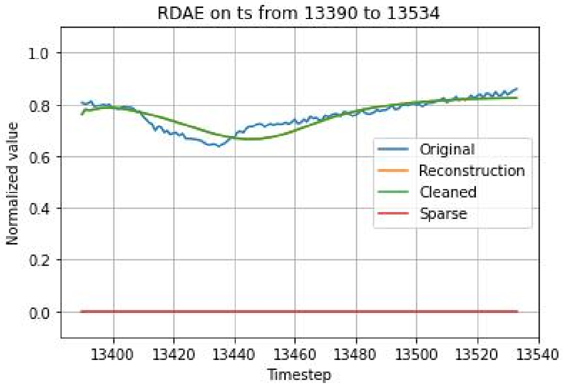


Figure: Example of a non anomaly subsequence for $\lambda = 3.3$

LSTM RDAE

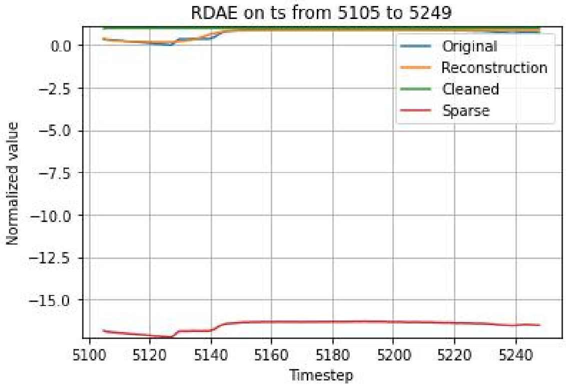


Figure: Example of a anomaly subsequence for $\lambda = 3.3$

LSTM RDAE

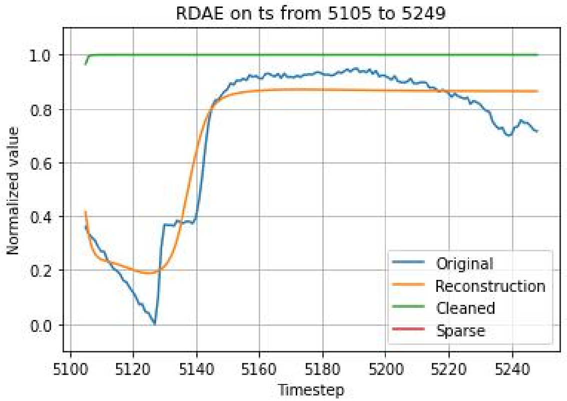


Figure: Example of a anomaly subsequence for $\lambda = 3.3$

LSTM RDAE

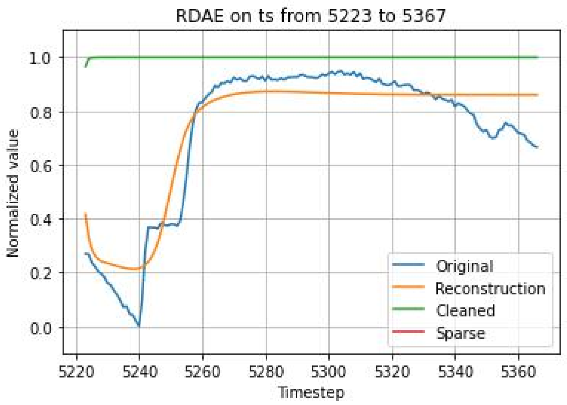


Figure: Example of a anomaly subsequence for $\lambda = 3.3$

LSTM RDAE

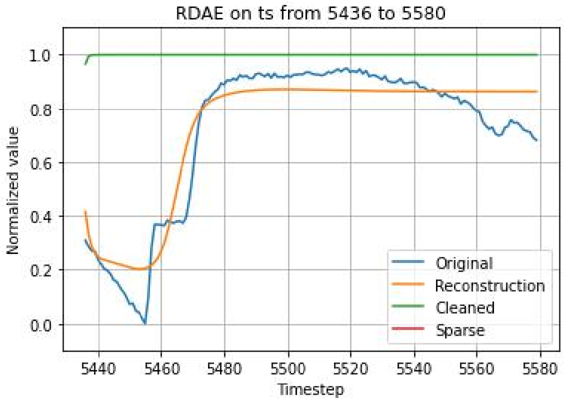


Figure: Example of a anomaly subsequence for $\lambda = 3.3$

LSTM RDAE

- All of the anomalies found reach the 0 value (min temperature of all time series).
- Also in this case different anomalies found DO NOT overlap. So each of the failures is only recognized once. In this case this happens at the beginning of the subsequence
- The reconstruction here is far worse than in the dense case.
- Performance could be improved using more parameters in the LSTM case. Note that computation time is much higher (~ 4 minutes for dense, ~ 20 minutes for LSTM, with the help of a RTX3070 laptop).

GRU RDAE

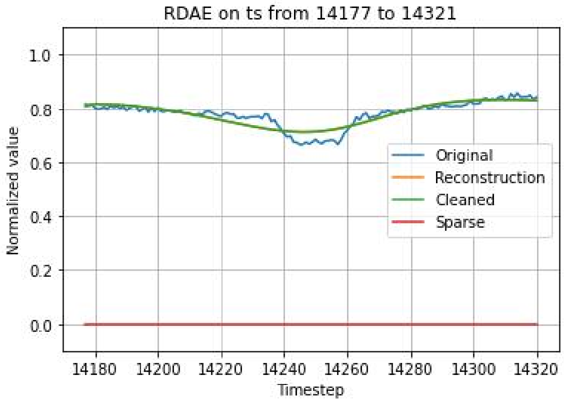


Figure: Example of a non anomaly subsequence for $\lambda = 3.3$

GRU RDAE

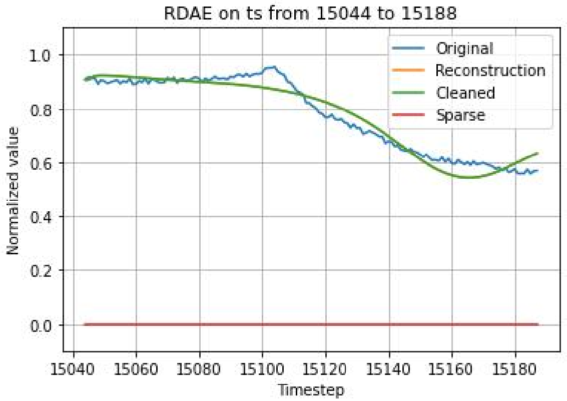


Figure: Example of a non anomaly subsequence for $\lambda = 3.3$

GRU RDAE

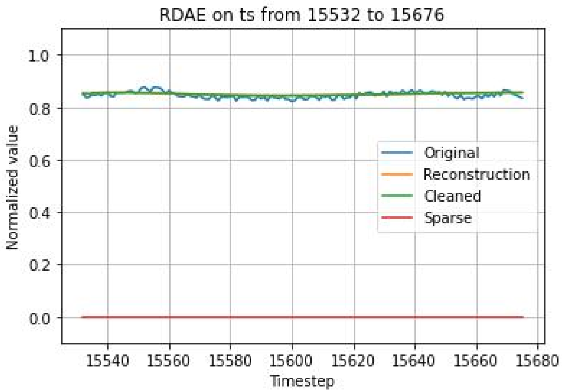


Figure: Example of a non anomaly subsequence for $\lambda = 3.3$

GRU RDAE

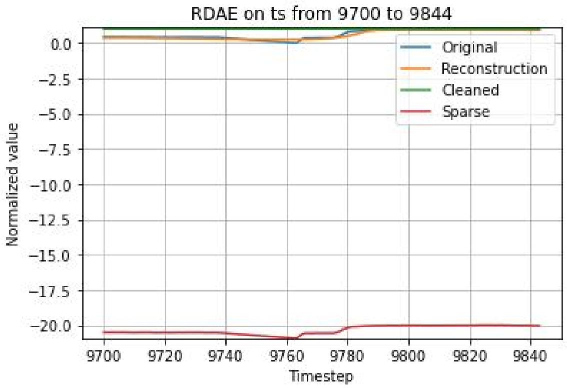


Figure: Example of a anomaly subsequence for $\lambda = 3.3$

GRU RDAE

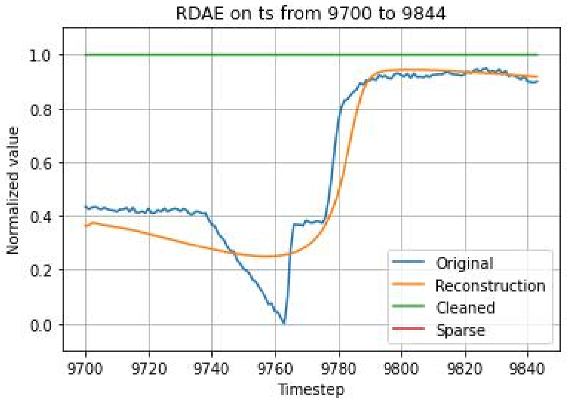


Figure: Example of a anomaly subsequence for $\lambda = 3.3$

GRU RDAE

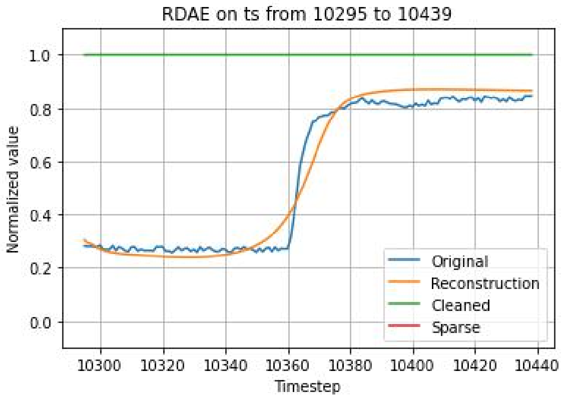


Figure: Example of a anomaly subsequence for $\lambda = 3.3$

GRU RDAE

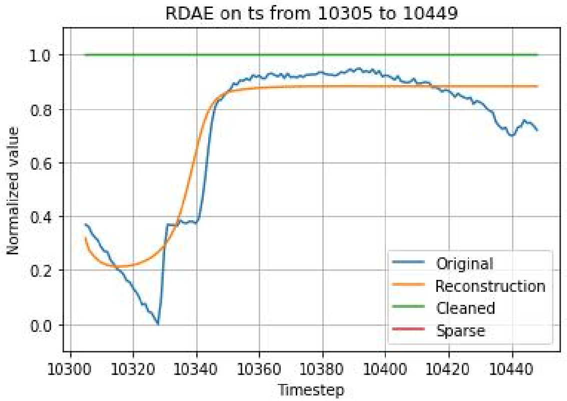


Figure: Example of a anomaly subsequence for $\lambda = 3.3$

GRU RDAE

- In this case, not all of the anomalies found reach the 0 value.
- Also in this case different anomalies found DO NOT overlap. So each of the failures is only recognized once. With GRU this happens in different parts of the subsequence.
- Also here the reconstruction is far worse than in the dense case.
- Again we could increase parameters for better performance. Computation here required $\sim$ 12 minutes with GPU.

Isolation Forest

- I tried to fit an isolation forest on the same data, to see if we get the same kind of outliers.
- With all outlier fraction values tested $(0.0001, 0.0005, 0.001, 0.01)$, non of them showed significant results.
- On the contrary, almost all the anomalies detected I saw were normal time series, almost flat.

Isolation forest anomaly example

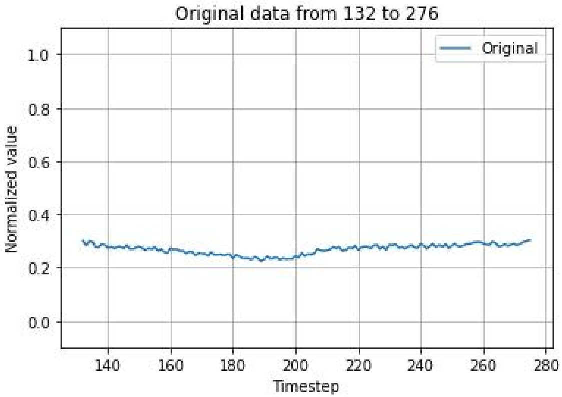


Figure: Example of an anomaly found by the isolation forest with outlier fraction of 0.001

Final comments

- The RDAE is a powerful tool for denoising and anomaly detection.
- Unfortunately the main quest is to find the correct λ value. With unlabeled data this could be very difficult.
- In that case, a possible way out is to know the approximate anomaly rate and to hope the anomalies found match the true ones.

- Note an important thing: after we train the model where is no way to find anomalies on new given data.
- The L_D and S matrices are produced only in the training procedure.
- We can still denoise images with the Autoencoder part.
- It could be tried to add new data after some iteration, without re-initializing. This requires training again the autoencoder, which is the high computational part.

https://github.com/AlexThirty/SaMLMfTSA

Thank you!

Subutai Ahmad, Alexander Lavin, Scott Purdy, and Zuha Agha.
Unsupervised real-time anomaly detection for streaming data.
Neurocomputing, 262:134–147, 2017.
Online Real-Time Learning Strategies for Data Streams.

Emmanuel J. Candes, Xiaodong Li, Yi Ma, and John Wright.
Robust principal component analysis?, 2009.

Neal Parikh.
Proximal algorithms.
Foundations and Trends in Optimization, 1:127–239, 01 2014.

Chong Zhou and Randy C. Paffenroth.
Anomaly detection with robust deep autoencoders.
Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, page 665–674, 2017.