Autoencoders for Anomaly Detection

Autoencoders

An autoencoder is a type of neural network

- The network is designed to reconstruct its input vector
- The input is a tensor x and the output should be similar to the same tensor x

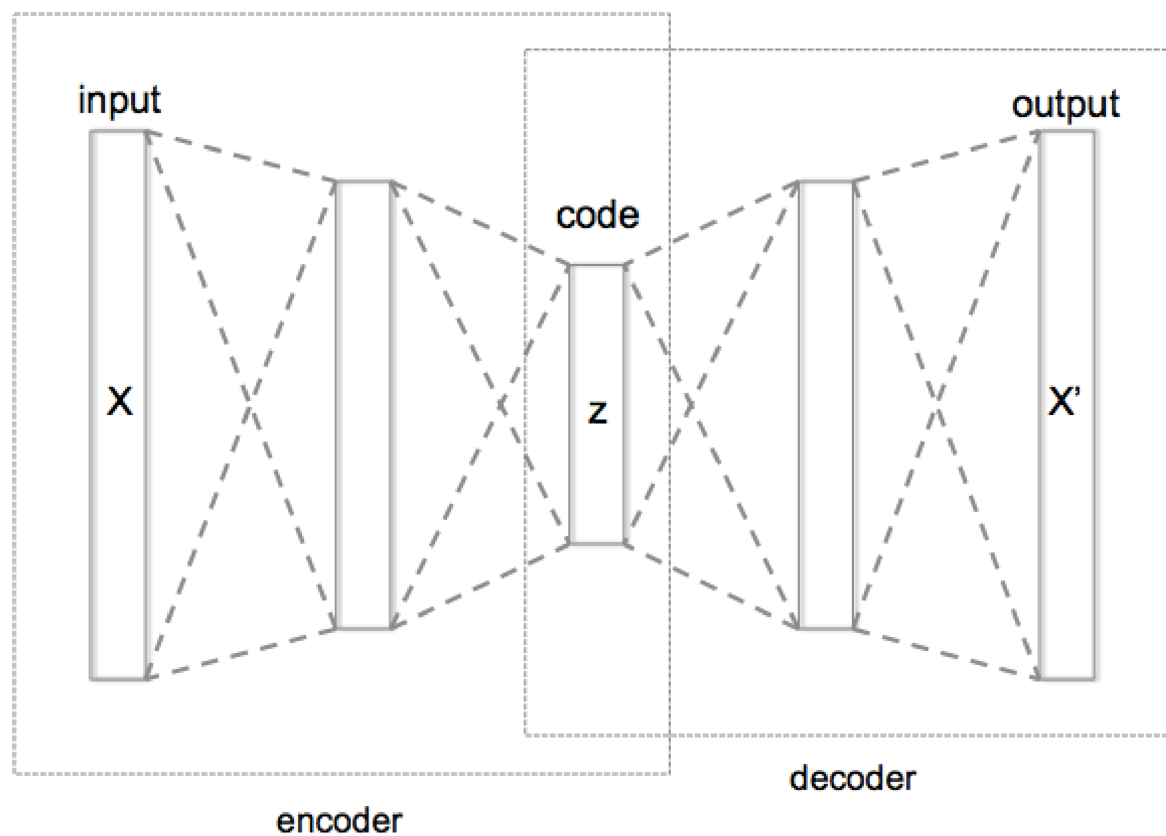


Figure from Wikipedia

Autoencoders

Autoencoders can be broken down in two halves

- An encoder, i.e. $e(x, \theta_e)$, mapping x into a vector of latent variables z
- A decoder, i.e. $d(z, \theta_d)$, mapping z into reconstructed input tensor

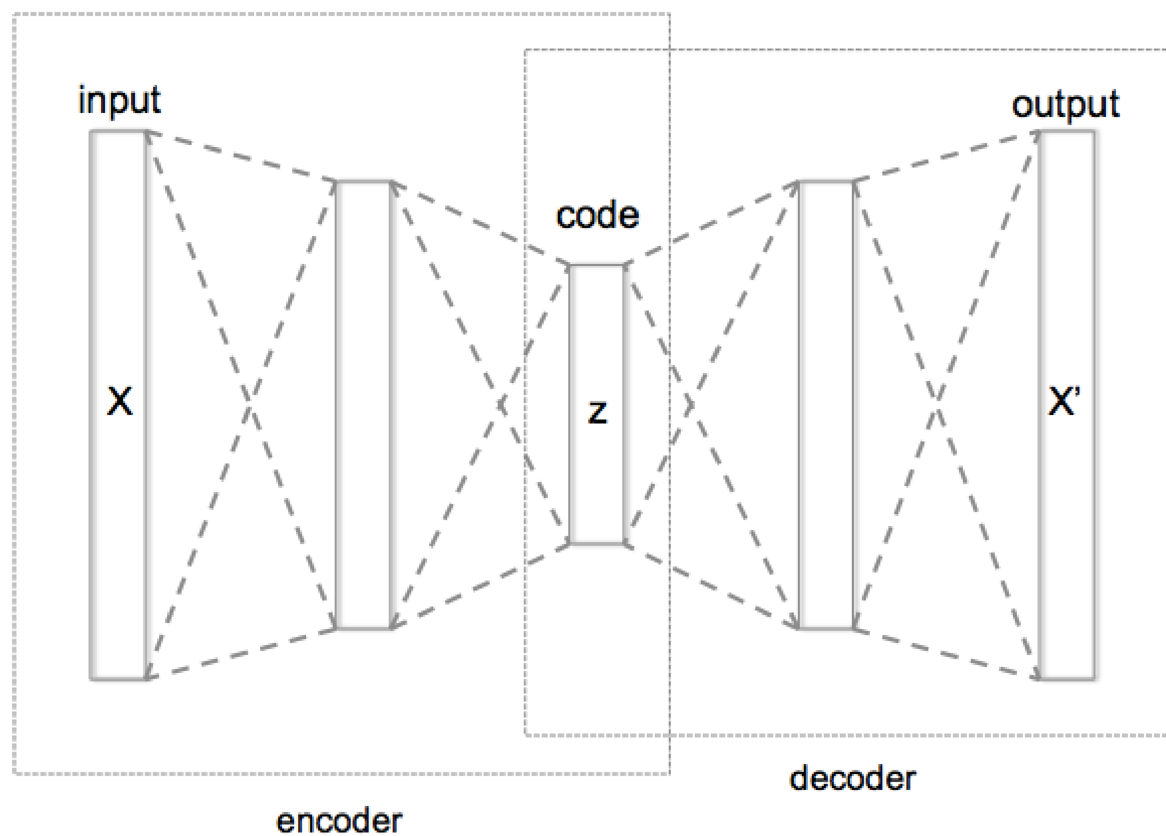


Figure from Wikipedia

Training an Autoencoder

Autoencoders are typically trained for minimum MSE:

$$\arg\min_{\theta_e,\theta_d} \|d(e(\hat{x}_i, \theta_e), \theta_d) - \hat{x}_i\|_2^2$$

- I.e. d, when applied to the output of e
- ...Should approximately return the input vector itself

A nice tutorial about autoencoders can be found on the Keras blog

There is a risk that an autoencoder learns a trivial transformation ($x' = x$)

This is obviously undesired, and it can be avoided by:

- Choosing a small-dimensional latent space (compressing autoencoder)
- By encouraging sparse encodings with an L1 regularizer (sparse autoencoder)

Autoencoders for Anomaly Detection

Autoencoders can be used for anomaly detection

...By using the reconstruction error as an anomaly signal, e.g.:

$$\|x - d(e(x, \theta_e), \theta_d)\|_2^2 \geq \theta$$

This approach has some PROs and CONs compared to KDE

- The size of a Neural Network does not depend on the size of the training set
- Neural Networks have good support for high dimensional data
- ...Plus limited overfitting and fast prediction/detection time
- However, error reconstruction can be harder than density estimation

Let's prepare the data to test the approach

When working with NNs, it's important to always standardize/normalize the data

But why?

NNs and Standardization

The main reason is due to (stochastic) gradient descent

The performance of SGD depends a lot on its starting point

- DL libraries all come with robust weight initialization procedures
  - ...And robust default parameters for the gradient descent algorithms
- ...But those are designed for data that is:
  - Reasonably close to zero
  - Mostly contained in a $[-1, 1]^n$ box

You can use NNs with non standardize data

...But expect far less reliable results

- In addition, vector output should always be standardized/normalized
- We'll see why in a short while

Data Preparation

We'll prepare our data as we did for KDE

First we apply a standardization step:

```
In [2]: tr_end, val_end = 3000, 4500
        hpcs = hpc.copy()
        tmp = hpcs.iloc[:tr_end]
        hpcs[inputs] = (hpcs[inputs] - tmp[inputs].mean()) / tmp[inputs].std()
```

The we separate a training, validation, and test set

```
In [3]: trdata = hpcs.iloc[:tr_end]
        valdata = hpcs.iloc[tr_end:val_end]
        tsdata = hpcs.iloc[val_end:]
```

Building an Autoencoder

The we can build an autoencoder (we'll use tensorflow 2.0 and keras)

First, we build the model using (e.g.) the functional API

In [4]:
```
from tensorflow import keras
from tensorflow.keras import layers, callbacks

input_shape = (len(inputs), )
ae_x = keras.Input(shape=input_shape, dtype='float32')
ae_z = layers.Dense(64, activation='relu')(ae_x)
ae_y = layers.Dense(len(inputs), activation='linear')(ae_z)
ae = keras.Model(ae_x, ae_y)
```

```
2022-10-06 11:45:46.136120: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFl
ow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following C
PU instructions in performance-critical operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
```

Autoencoders in Keras

Then we can prepare our model for training

In keras terms, we compile it

```
In [5]: ae.compile(optimizer='RMSProp', loss='mse')
```

- We are using the `RMSProp` optimizer (a variant of Stochastic Gradient Descent)

Then we can start training:

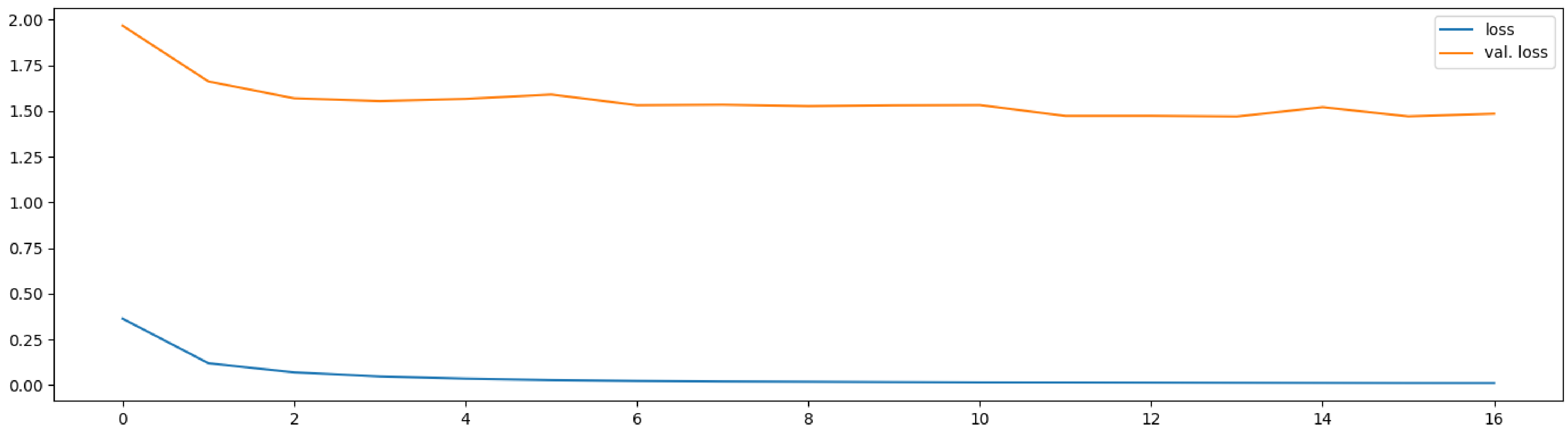
```
In [6]: cb = [callbacks.EarlyStopping(patience=3, restore_best_weights=True)]
        history = ae.fit(trdata[inputs], trdata[inputs], validation_split=0.1,
                         callbacks=cb, batch_size=32, epochs=30, verbose=0)
```

- We are using a callback to stop training early
- ...If no improvement on the validation set is observed for 3 epochs

Autoencoders in Keras

Let's have a look at the loss evolution over different epochs

In [7]:
```
util.plot_training_history(history, figsize=figsize)
```



Autoencoders in Keras

Finally, we can obtain the predictions

In [8]:
```
preds = pd.DataFrame(index=hpcs.index, columns=inputs, data=ae.predict(hpcs[inputs], verbose=0))
preds.head()
```

Out[8]:

	ambient_temp	cmbw_p0_0	cmbw_p0_1	cmbw_p0_10	cmbw_p0_11	cmbw_p0_12	cmbw_p0_13	cmbw_p0_14	cmbw_p0_2	cmbw_p0_:
0	-1.082064	-0.884785	-0.945090	2.303042	2.474257	2.314500	2.277722	2.201775	-2.035323	1.269203
1	-1.040860	-0.554659	-0.094386	2.217015	2.273416	2.229946	2.261420	2.227186	0.391483	-0.516403
2	-0.956512	-0.886489	-0.326548	2.231952	2.311509	2.232465	2.262239	2.456877	0.370581	0.492824
3	-0.985948	-0.649309	-0.478655	2.185980	2.330112	2.171434	2.168338	2.178050	0.676569	0.932359
4	-1.107542	-0.615712	-0.510661	2.213217	2.309212	2.294946	2.314302	2.311393	0.744839	0.962220

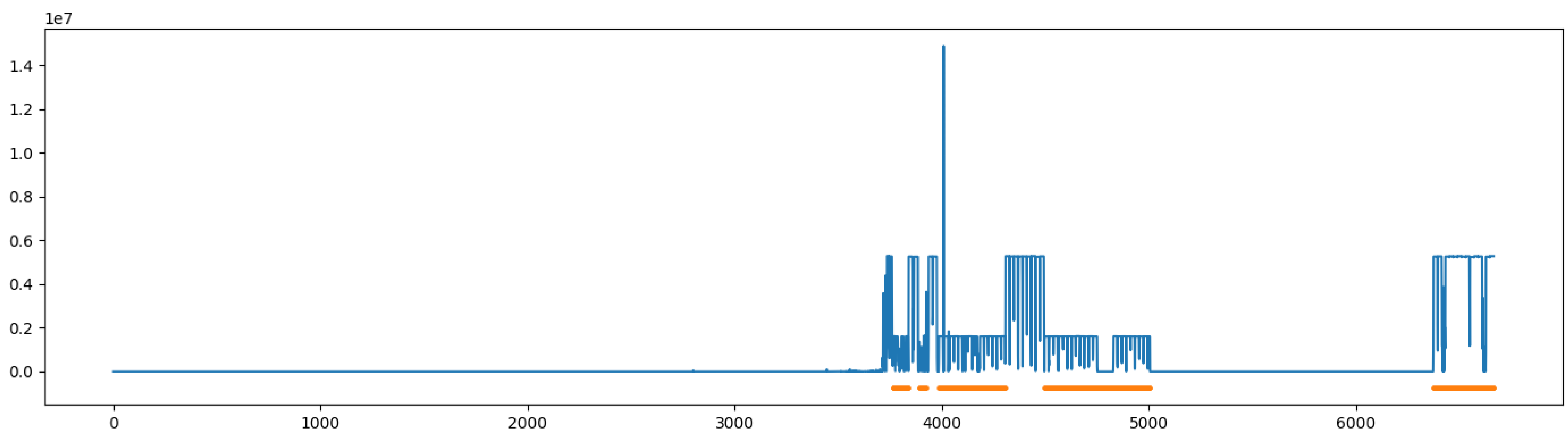
5 rows × 159 columns

- These are the reconstructed values for all the input features

Alarm Signal

We can finally obtain our alarm signal, i.e. the sum of squared errors

In [9]:
```
labels = pd.Series(index=hpcs.index, data=(hpcs['anomaly'] != 0), dtype=int)
sse = np.sum(np.square(preds - hpcs[inputs]), axis=1)
signal_ae = pd.Series(index=hpcs.index, data=sse)
util.plot_signal(signal_ae, labels, figsize=figsize)
```



This is very similar to the KDE signal: why?

Semantic of Neural Regressors

Let's try to understand what we have just done

When we train an autoencoder (renamed here as $\boldsymbol{h}$), we solve:

$$\arg\min_{\theta} \|h(\hat{x}_i, \theta) - \hat{x}_i\|_2^2$$

By expanding the L2 norm, we get:

$$\arg\min_{\theta} \sum_{i=1}^{m} \sum_{j=1}^{n} \left(h_j(\hat{x}_i, \theta) - \hat{x}_{i,j}\right)^2$$

By introducing a $\log$ and $\exp$ transformation we obtain:

$$\arg\min_{\theta} \log \exp\left(\sum_{i=1}^{m} \sum_{j=1}^{n} \left(h_j(\hat{x}_i, \theta) - \hat{x}_{i,j}\right)^2\right)$$

Semantic of Neural Regressors

Then, from the last step:

$$\arg\min_{\theta} \log \exp\left(\sum_{i=1}^{m} \sum_{j=1}^{n} \left(h_j(\hat{x}_i, \theta) - \hat{x}_{i,j}\right)^2 \right)$$

We rewriting the outer sum using properties of exponentials:

$$\arg\min_{\theta} \log \prod_{i=1}^{m} \exp\left(\sum_{j=1}^{n} \left(h_j(\hat{x}_i, \theta) - \hat{x}_{i,j}\right)^2 \right)$$

Then we rewrite the inner sum in matrix form:

$$\arg\min_{\theta} \log \prod_{i=1}^{m} \exp\left(\left(h(\hat{x}_i, \theta) - \hat{x}_{i,j}\right)^T I \left(h(\hat{x}_i, \theta) - \hat{x}_{i,j}\right) \right)$$

Semantic of Neural Regressors

Starting from the last step:

$$\arg\min_{\theta} \log \prod_{i=1}^{m} \exp\left(\left(h(\hat{x}_i, \theta) - \hat{x}_{i,j}\right)^T I \left(h(\hat{x}_i, \theta) - \hat{x}_{i,j}\right)\right)$$

We make a few adjustment that do not change the optimal solution:

- We negate the argument of $\exp$ and swap the $\arg\min$ for a $\arg\max$
- We multiply exponential argument by $1/2$
- We multiply the exponential by $1/\sqrt{2\pi}$

$$\arg\max_{\theta} \log \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(h(\hat{x}_i, \theta) - \hat{x}_{i,j}\right)^T I \left(h(\hat{x}_i, \theta) - \hat{x}_{i,j}\right)\right)$$

Semantic of Neural Regressors

Let's look at our last formulation:

$$\arg\max_{\theta} \log \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(h(\hat{x}_i, \theta) - \hat{x}_{i,j}\right)^T I \left(h(\hat{x}_i, \theta) - \hat{x}_{i,j}\right)\right)$$

The term inside the product is the PDF of a multivariate normal distribution

$$\arg\min_{\theta} \log \prod_{i=1}^{m} f\left(\hat{x}_i, h(\hat{x}_i), I\right)$$

- In particular a distribution centered on $h(\hat{x}_i)$
- ...With independent Normal components
- ...All having unit variance

Semantic of Neural Regressors

Let's discuss some implications

When we use a MSE loss, we are training for maximum likelihood

- ...Just like density estimators
- This is actually true for many ML approaches

Semantic of Neural Regressors

Let's discuss some implications

When we use a MSE loss, we are training for maximum likelihood

- ...Just like density estimators
- This is actually true for many ML approaches

The output of a (MSE trained) regressor has a probabilistic interpretation

- Specifically, the output is the mean of a conditional distribution
- The distribution represents the variability of the target
- ...Once the effect of the input is taken into account
- Another way to think of it: noise around the prediction

Semantic of Neural Regressors

Let's discuss some implications

We are implicitly assuming that the noise is normally distributed

- This true in many cases, but not always
- E.g., sometimes large values are under-represented
- ....Leading to log-normal distributions
- In this cases, applying a log transformation to the output can be very effective

Semantic of Neural Regressors

Let's discuss some implications

We are implicitly assuming that the noise is normally distributed

- This true in many cases, but not always
- E.g., sometimes large values are under-represented
- ....Leading to log-normal distributions
- In this cases, applying a log transformation to the output can be very effective

We are also assuming that the all output components have the same variance

- Non necessarily unit variance (the equivalence holds for any constant σ)
- This is another (very) good reason to standardize the output

Semantic of Neural Regressors

Let's discuss some implications

We are also assuming that the noise on all output components is independent

- This might be true even if the output components themselves are correlated
- ...But still it is not true in all cases

Semantic of Neural Regressors

Let's discuss some implications

We are also assuming that the noise on all output components is independent

- This might be true even if the output components themselves are correlated
- ...But still it is not true in all cases

All these implicit assumption can make the problem harder

- This is why we mentioned that error reconstruction can be harder than density estimation

Semantic of Neural Regressors

Let's discuss some implications

We are also assuming that the noise on all output components is independent

- This might be true even if the output components themselves are correlated
- ...But still it is not true in all cases

All these implicit assumption can make the problem harder

- This is why we mentioned that error reconstruction can be harder than density estimation

Finally, our alarm signal can be interpreted as a density:

- To see why, just apply the transformation to $\|x - d(e(x, \theta_e), \theta_d)\|_2^2$
- This fact explains why the signal is similar to the KDE one

Threshold Optimization

The threshold can be optimized as usual

```
In [11]: c_alarm, c_missed, tolerance = 1, 5, 12
         cmodel =  util.HPCMetrics(c_alarm, c_missed, tolerance)
         th_range = np.linspace(1e4, 2e5, 200)

         th_ae, val_cost_ae = util.opt_threshold(signal_ae[tr_end:val_end], hpcs['anomaly'][tr_end:val_er
         print(f'Best threshold: {th_ae:.3f}')
         tr_cost_ae = cmodel.cost(signal_ae[:tr_end], hpcs['anomaly'][:tr_end], th_ae)
         print(f'Cost on the training set: {tr_cost_ae}')
         print(f'Cost on the validation set: {val_cost_ae}')
         ts_cost_ae = cmodel.cost(signal_ae[val_end:], hpcs['anomaly'][val_end:], th_ae)
         print(f'Cost on the test set: {ts_cost_ae}')
```

```
Best threshold: 113115.578
Cost on the training set: 0
Cost on the validation set: 263
Cost on the test set: 265
```

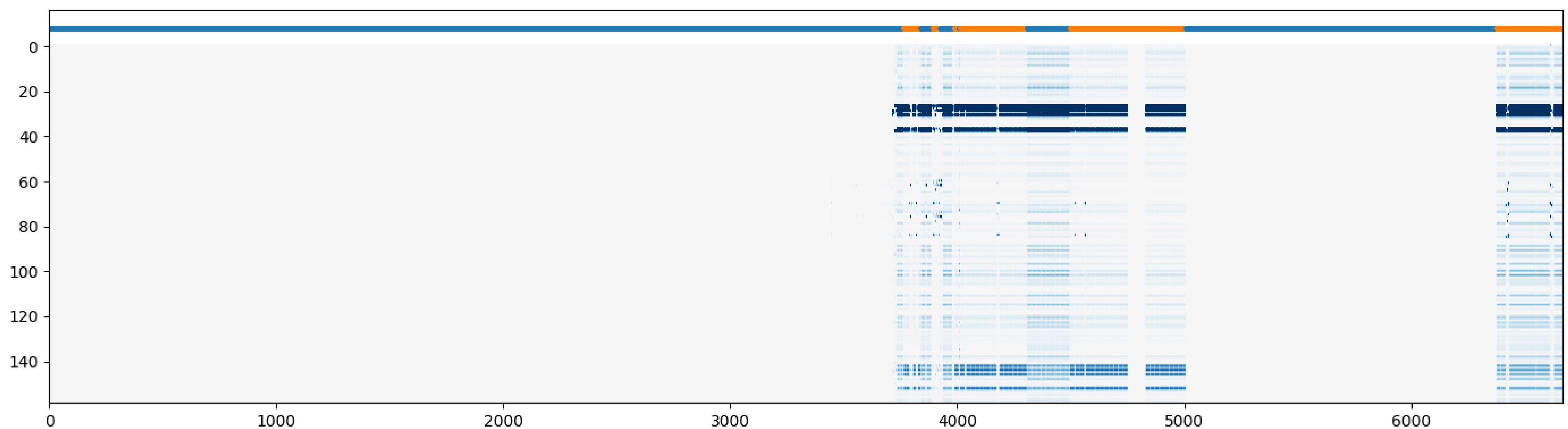
- The performance is similar to KDE (not surprisingly)

Mutiple Signal Analysis

But autoencoders do more than just anomaly detection!

- Instead of having a single signal we have many
- So we can look at the individual reconstruction errors

In [12]:
```
se = np.square(preds - hpcs[inputs])
signals_ae = pd.DataFrame(index=hpcs.index, columns=inputs, data=se)
util.plot_dataframe(signals_ae, labels, vmin=-5e4, vmax=5e4, figsize=figsize)
```



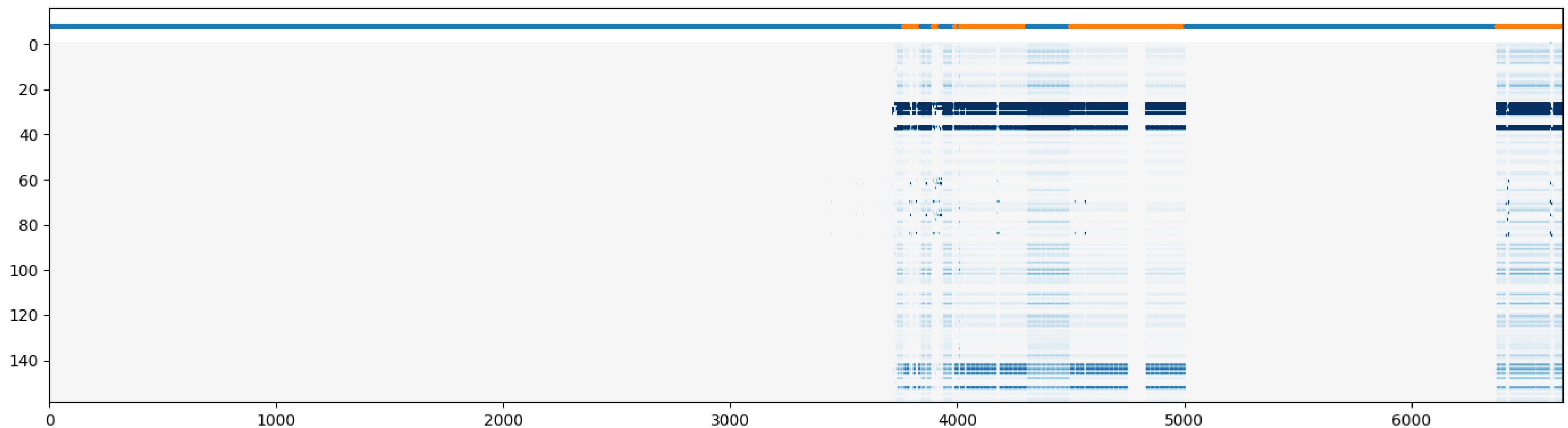
Mutiple Signal Analysis

Reconstruction errors are often concentrated on a few signals

- These correspond to the properties of the input vector that were harder to reconstruct
- ...And often they are useful clues about the nature of the anomaly

In [13]:

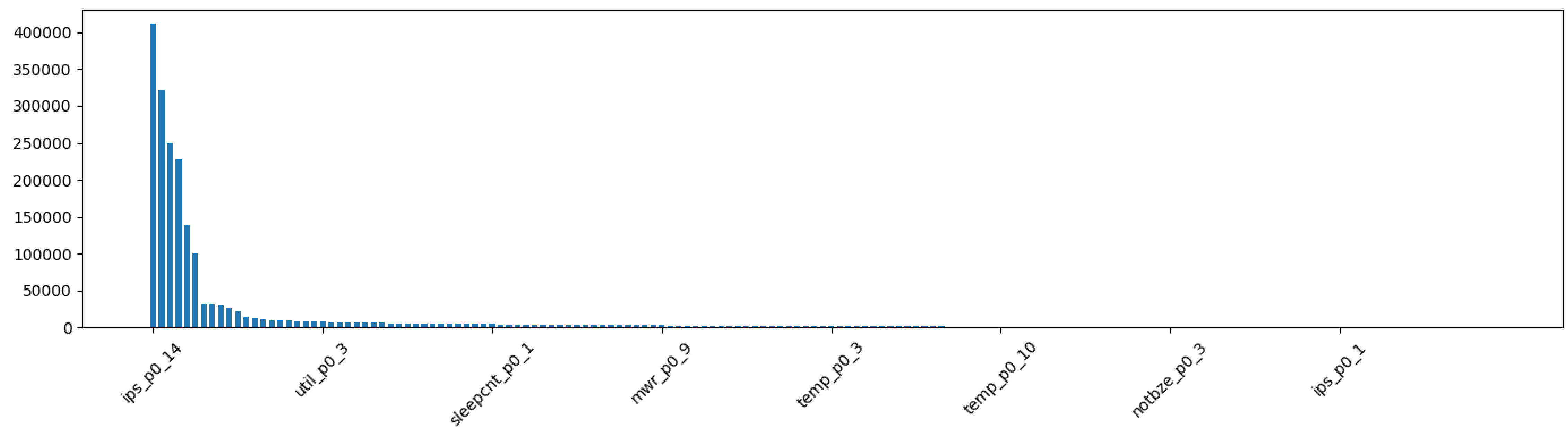
```
se = np.square(preds - hpcs[inputs])
signals_ae = pd.DataFrame(index=hpcs.index, columns=inputs, data=se)
util.plot_dataframe(signals_ae, labels, vmin=-5e4, vmax=5e4, figsize=figsize)
```



Multiple Signal Analysis

Here are the average errors for all anomalies (sorted by decreasing value)

```
In [14]: mode_1 = hpcs.index[hpcs['anomaly'] != 0]
         tmp =  se.iloc[mode_1].mean().sort_values(ascending=False)
         util.plot_bars(tmp, tick_gap=20, figsize=figsize)
```

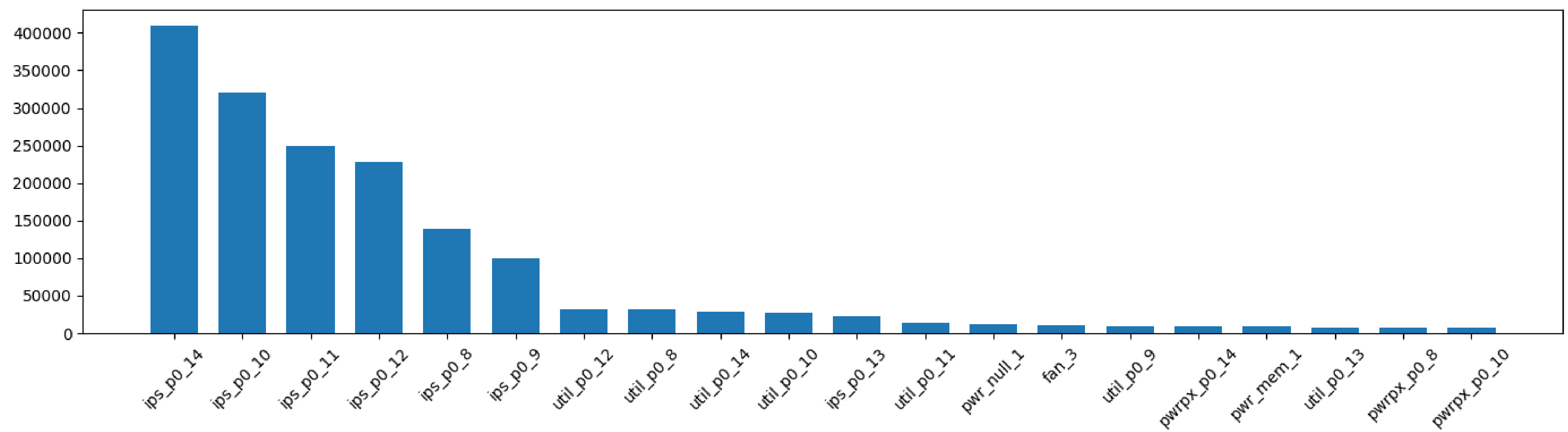


- Errors are concentrated on 10-20 features

Multiple Signal Analysis

These are the 20 largest average errors for all anomalies

```
In [15]: mode_1 = hpcs.index[hpcs['anomaly'] != 0]
         tmp =  se.iloc[mode_1].mean().sort_values(ascending=False)
         util.plot_bars(tmp.iloc[:20], figsize=figsize)
```



- The largest errors are on "ips", then on "util" (utilization)
- This kind of information can be very valuable for a domain expert!