Autoencoders for Dimensionality Reduction

Air-Quality Exploration with Autoencoders

This mini-project investigates hourly air-pollution data collected at several RAMA stations during 2023.
After merging the individual Excel logs we performed the following workflow:

1. **Cleaning & harmonisation**

    - Converted separate `FECHA` + `HORA` columns to a single timestamp.
    - Replaced sentinel values (–99, –999) by `NaN`, interpolated gaps, and winsorised the extreme 0.2 % / 99.8 % tails.
    - Retained only station–pollutant columns with ≥ 50 % valid data and noticeable variance.

2. **Dimensionality reduction with an autoencoder**

    - Standardised each feature using *median* and *IQR* (`RobustScaler`) to tame residual outliers.
    - Trained a fully-connected **autoencoder** with a 2-neuron bottleneck (`latent = 2`).
      The network learns a non-linear mapping
      [\mathbf{x};\mapsto;f_{\theta}(\mathbf{x})\in\mathbb{R}^2] that best reconstructs the original 30-dimensional pollution vector.
    - Extracted the 2-D latent coordinates for every timestamp and visualised them, colouring by hour of day and by month.

Autoencoders serve here as a **non-linear alternative to PCA**:
they can compress highly correlated pollutants while preserving complex inter-station relationships that linear projections might miss.

In [ ]:
```
import glob, os, re
import pandas as pd
import numpy as np
from functools import reduce
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import RobustScaler
from tensorflow.keras import layers, callbacks, Model, Input
import tensorflow as tf, matplotlib.pyplot as plt
```

In [ ]:
```
folder = "23RAMA"
paths  = glob.glob(os.path.join(folder, "*.xls"))

dfs = {}
for p in paths:
    varname = re.sub(r"\.xls$", "", os.path.basename(p))
```

```
    df_raw  = pd.read_excel(p)

    df_raw["FECHA"] = pd.to_datetime(df_raw["FECHA"], dayfirst=True)
    df_raw["datetime"] = df_raw["FECHA"] + pd.to_timedelta(df_raw["HORA"]
    df_raw = df_raw.drop(columns=["FECHA", "HORA"])

    tidy = df_raw.melt(
        id_vars="datetime",
        var_name="station",
        value_name=varname
    )
    dfs[varname] = tidy
```

In [ ]:

```
long_list = list(dfs.values())
merged = reduce(
    lambda a, b: pd.merge(a, b, on=["datetime", "station"], how="inner"),
    long_list
)
merged = merged.dropna().reset_index(drop=True)
print(merged.head())
```

```
             datetime station  2023CO  2023NOX  2023NO  2023PM25  2023PM
10  \
0 2023-01-01 01:00:00     AJM    0.45        9       1       -99       -
99
1 2023-01-01 02:00:00     AJM    0.43        8       0       -99       -
99
2 2023-01-01 03:00:00     AJM    0.42        8       0       -99       -
99
3 2023-01-01 04:00:00     AJM    0.48       12       1       -99       -
99
4 2023-01-01 05:00:00     AJM    0.37       11       1       -99       -
99

   2023SO2  2023O3  2023NO2  2023PMCO
0        1      30        8       -99
1        1      30        7       -99
2        1      31        7       -99
3        1      26       11       -99
4        1      25       11       -99
```

In [ ]:

```
feature_cols = [c for c in merged.columns if c not in ["datetime", "stati
wide = (merged
        .set_index(["datetime", "station"])
        .unstack("station")[feature_cols])
wide.columns = [f"{v}_{s}" for v, s in wide.columns]
wide = wide.sort_index()
```

In [ ]:

```
wide = wide.replace([-99, -999], np.nan)
print("After sentinel→NaN:", wide.shape)

valid_frac = (wide.notna().mean())
wide = wide.loc[:, valid_frac >= 0.50]

print("After column filter (≥50 % data):", wide.shape)
wide = wide.interpolate(limit_direction='both')
before = wide.shape[0]
wide   = wide.dropna(how='any')
print(f"Rows dropped for residual NaNs: {before - wide.shape[0]}")
print("After row filter:", wide.shape)
```

```
wide = wide.loc[:, wide.std() > 1e-3]
print("After flat-column filter:", wide.shape)
```

```
After sentinel→NaN: (5088, 144)
After column filter (≥50 % data): (5088, 103)
Rows dropped for residual NaNs: 0
After row filter: (5088, 103)
After flat-column filter: (5088, 103)
```

In [ ]:

```
X_train, X_val = train_test_split(
    wide.values,
    test_size=0.2,
    shuffle=False
)

scaler = RobustScaler()
X_train = scaler.fit_transform(X_train)
X_val   = scaler.transform(X_val)

n_feat = X_train.shape[1]

inp = Input((n_feat,))
z   = layers.Dense(2, name='z')(inp)
out = layers.Dense(n_feat)(z)

ae = Model(inp, out)
ae.compile('adam', 'mse')

stop = callbacks.EarlyStopping(patience=10, restore_best_weights=True)

ae.fit(X_train, X_train,
       validation_data=(X_val, X_val),
       epochs=100,
       batch_size=128,
       callbacks=[stop],
       verbose=2)

encoder = Model(ae.input, ae.get_layer('z').output)
```

```
Epoch 1/100
32/32 - 1s - 27ms/step - loss: 2.1673 - val_loss: 1.2858
Epoch 2/100
32/32 - 0s - 9ms/step - loss: 2.0055 - val_loss: 1.2243
Epoch 3/100
32/32 - 0s - 9ms/step - loss: 1.8041 - val_loss: 1.1655
Epoch 4/100
32/32 - 0s - 9ms/step - loss: 1.6357 - val_loss: 1.1235
Epoch 5/100
32/32 - 0s - 9ms/step - loss: 1.5298 - val_loss: 1.0936
Epoch 6/100
32/32 - 0s - 9ms/step - loss: 1.4571 - val_loss: 1.0689
Epoch 7/100
32/32 - 0s - 9ms/step - loss: 1.4000 - val_loss: 1.0477
Epoch 8/100
32/32 - 0s - 9ms/step - loss: 1.3481 - val_loss: 1.0245
Epoch 9/100
32/32 - 0s - 9ms/step - loss: 1.2972 - val_loss: 0.9972
Epoch 10/100
32/32 - 0s - 9ms/step - loss: 1.2438 - val_loss: 0.9654
Epoch 11/100
32/32 - 0s - 9ms/step - loss: 1.1890 - val_loss: 0.9341
Epoch 12/100
32/32 - 0s - 9ms/step - loss: 1.1381 - val_loss: 0.8989
Epoch 13/100
32/32 - 0s - 9ms/step - loss: 1.0961 - val_loss: 0.8707
Epoch 14/100
32/32 - 0s - 9ms/step - loss: 1.0656 - val_loss: 0.8473
Epoch 15/100
32/32 - 0s - 9ms/step - loss: 1.0464 - val_loss: 0.8302
Epoch 16/100
32/32 - 0s - 9ms/step - loss: 1.0342 - val_loss: 0.8208
Epoch 17/100
32/32 - 0s - 9ms/step - loss: 1.0272 - val_loss: 0.8162
Epoch 18/100
32/32 - 0s - 9ms/step - loss: 1.0228 - val_loss: 0.8113
Epoch 19/100
32/32 - 0s - 9ms/step - loss: 1.0189 - val_loss: 0.8083
Epoch 20/100
32/32 - 0s - 9ms/step - loss: 1.0166 - val_loss: 0.8051
Epoch 21/100
32/32 - 0s - 9ms/step - loss: 1.0148 - val_loss: 0.8034
Epoch 22/100
32/32 - 0s - 9ms/step - loss: 1.0134 - val_loss: 0.8021
Epoch 23/100
32/32 - 0s - 9ms/step - loss: 1.0123 - val_loss: 0.8026
Epoch 24/100
32/32 - 0s - 9ms/step - loss: 1.0113 - val_loss: 0.8011
Epoch 25/100
32/32 - 0s - 9ms/step - loss: 1.0106 - val_loss: 0.8001
Epoch 26/100
32/32 - 0s - 9ms/step - loss: 1.0096 - val_loss: 0.8004
Epoch 27/100
32/32 - 0s - 9ms/step - loss: 1.0089 - val_loss: 0.7992
Epoch 28/100
32/32 - 0s - 9ms/step - loss: 1.0085 - val_loss: 0.7997
Epoch 29/100
32/32 - 0s - 9ms/step - loss: 1.0076 - val_loss: 0.7975
Epoch 30/100
32/32 - 0s - 9ms/step - loss: 1.0075 - val_loss: 0.7991
```

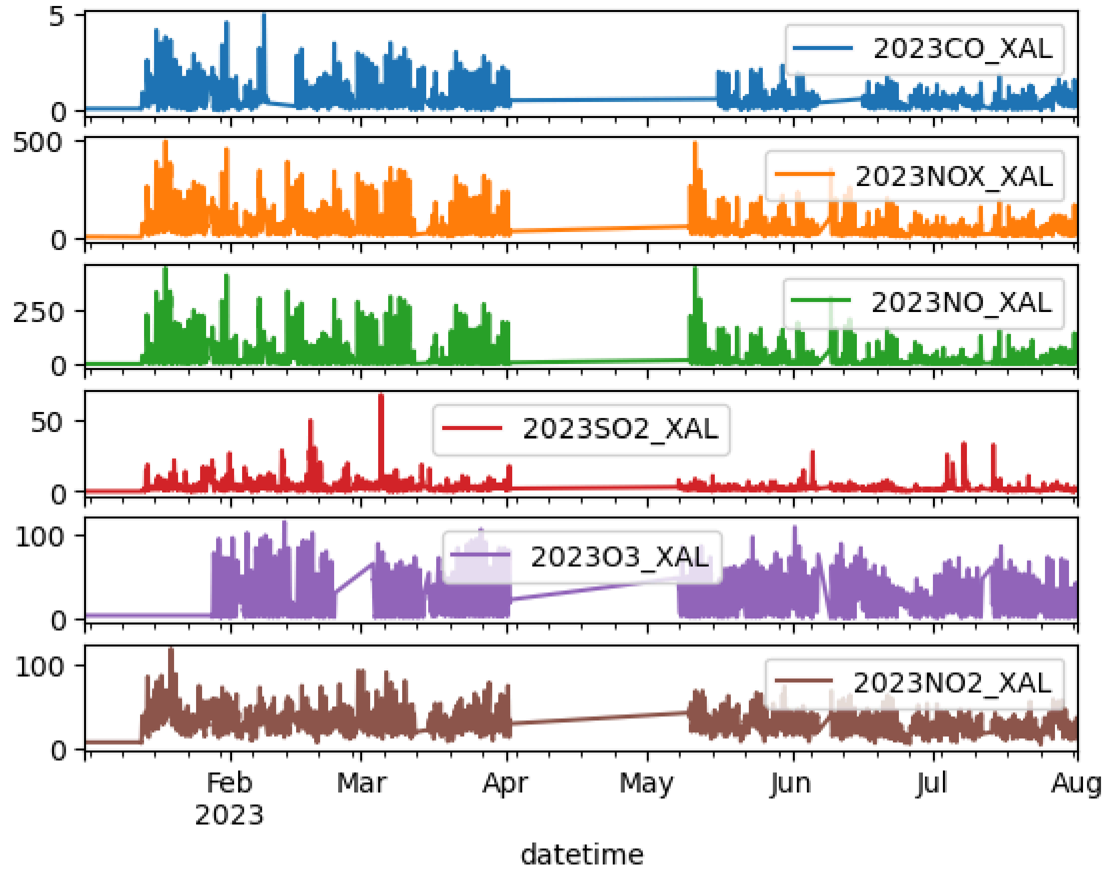
```
Epoch 31/100
32/32 - 0s - 9ms/step - loss: 1.0070 - val_loss: 0.7975
Epoch 32/100
32/32 - 0s - 9ms/step - loss: 1.0065 - val_loss: 0.7979
Epoch 33/100
32/32 - 0s - 9ms/step - loss: 1.0063 - val_loss: 0.7973
Epoch 34/100
32/32 - 0s - 9ms/step - loss: 1.0060 - val_loss: 0.7974
Epoch 35/100
32/32 - 0s - 8ms/step - loss: 1.0054 - val_loss: 0.7967
Epoch 36/100
32/32 - 0s - 9ms/step - loss: 1.0053 - val_loss: 0.7984
Epoch 37/100
32/32 - 0s - 11ms/step - loss: 1.0051 - val_loss: 0.7955
Epoch 38/100
32/32 - 0s - 8ms/step - loss: 1.0048 - val_loss: 0.7964
Epoch 39/100
32/32 - 0s - 9ms/step - loss: 1.0044 - val_loss: 0.7972
Epoch 40/100
32/32 - 0s - 9ms/step - loss: 1.0047 - val_loss: 0.7974
Epoch 41/100
32/32 - 0s - 9ms/step - loss: 1.0047 - val_loss: 0.7982
Epoch 42/100
32/32 - 0s - 9ms/step - loss: 1.0042 - val_loss: 0.7959
Epoch 43/100
32/32 - 0s - 9ms/step - loss: 1.0041 - val_loss: 0.7948
Epoch 44/100
32/32 - 0s - 9ms/step - loss: 1.0038 - val_loss: 0.7964
Epoch 45/100
32/32 - 0s - 8ms/step - loss: 1.0041 - val_loss: 0.7951
Epoch 46/100
32/32 - 0s - 8ms/step - loss: 1.0036 - val_loss: 0.7963
Epoch 47/100
32/32 - 0s - 9ms/step - loss: 1.0035 - val_loss: 0.7958
Epoch 48/100
32/32 - 0s - 9ms/step - loss: 1.0035 - val_loss: 0.7971
Epoch 49/100
32/32 - 0s - 9ms/step - loss: 1.0032 - val_loss: 0.7967
Epoch 50/100
32/32 - 0s - 8ms/step - loss: 1.0033 - val_loss: 0.7965
Epoch 51/100
32/32 - 0s - 8ms/step - loss: 1.0034 - val_loss: 0.7969
Epoch 52/100
32/32 - 0s - 9ms/step - loss: 1.0030 - val_loss: 0.7966
Epoch 53/100
32/32 - 0s - 9ms/step - loss: 1.0029 - val_loss: 0.7964
```

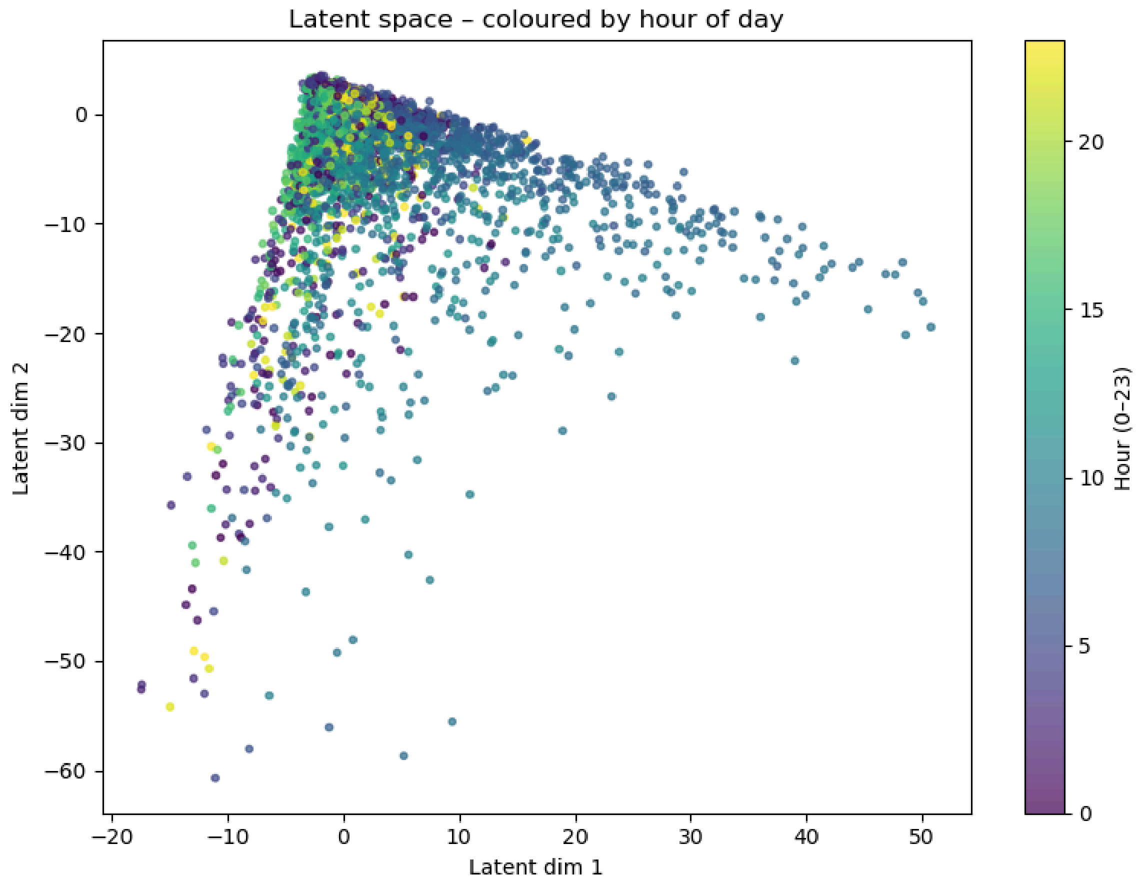
In [ ]:

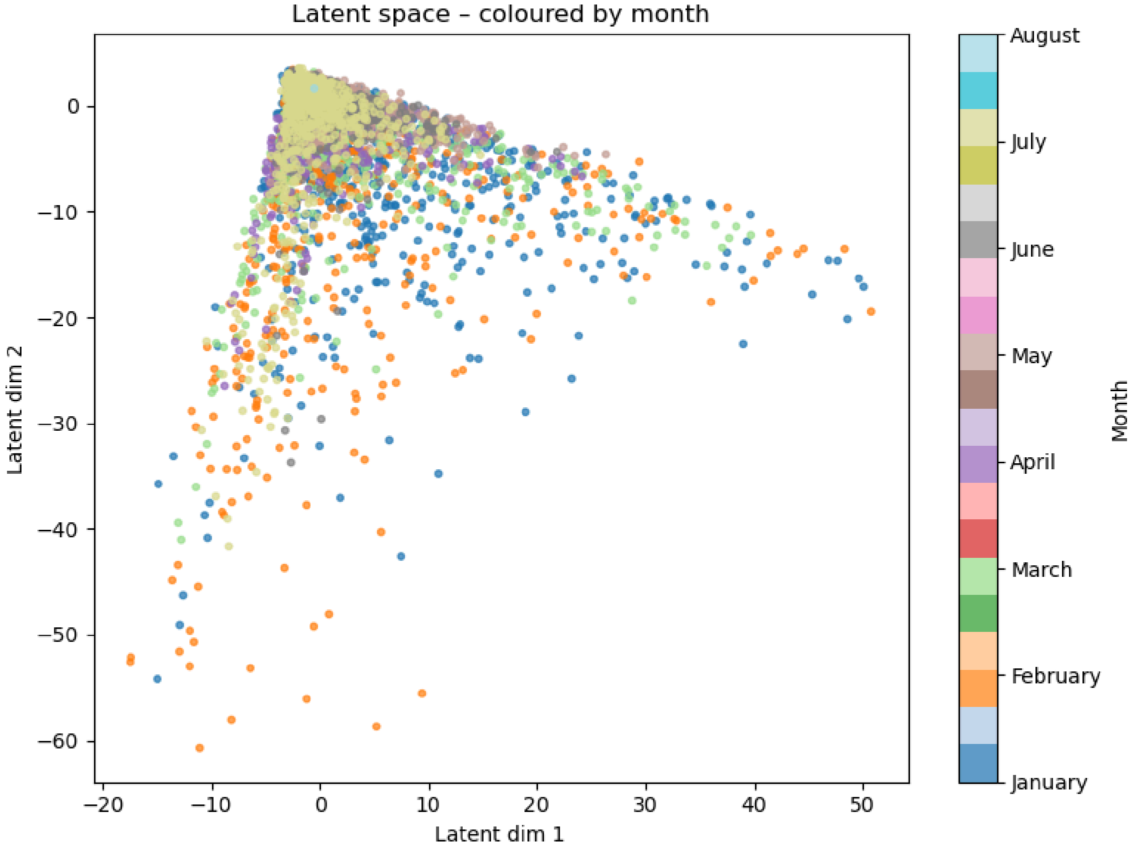
```
wide.index = pd.to_datetime(wide.index)
wide.filter(like='_XAL').plot(subplots=True)



X_full     = wide.values
X_full_std = scaler.transform(X_full)

Z_full     = encoder.predict(X_full_std)

emb_df = pd.DataFrame({
    "z1": Z_full[:, 0],
    "z2": Z_full[:, 1],
    "datetime": wide.index
```

```python
})
emb_df["hour"]  = emb_df["datetime"].dt.hour
emb_df["month"] = emb_df["datetime"].dt.month_name()

# scatter plot hours
plt.figure(figsize=(8, 6))
plt.scatter(emb_df["z1"], emb_df["z2"],
            c=emb_df["hour"], cmap="viridis", s=10, alpha=0.7)
plt.xlabel("Latent dim 1"); plt.ylabel("Latent dim 2")
plt.title("Latent space – coloured by hour of day")
plt.colorbar(label="Hour (0–23)")
plt.tight_layout(); plt.show()



#Scatter by month
month_to_int = {m: i for i, m in enumerate(emb_df["month"].unique())}
month_ints   = emb_df["month"].map(month_to_int)

plt.figure(figsize=(8, 6))
plt.scatter(emb_df["z1"], emb_df["z2"],
            c=month_ints, cmap="tab20", s=10, alpha=0.7)
plt.xlabel("Latent dim 1"); plt.ylabel("Latent dim 2")
plt.title("Latent space – coloured by month")
cbar = plt.colorbar(ticks=list(month_to_int.values()))
cbar.ax.set_yticklabels(list(month_to_int.keys()))
cbar.set_label("Month")
plt.tight_layout(); plt.show()

# hist
X_recon = ae.predict(X_full_std)
mse     = np.mean(np.square(X_recon - X_full_std), axis=1)

plt.figure(figsize=(7, 4))
plt.hist(mse, bins=50, edgecolor="black")
plt.xlabel("Per-sample MSE (standardised scale)")
plt.ylabel("Frequency")
plt.title("Distribution of reconstruction error")
plt.tight_layout(); plt.show()
```

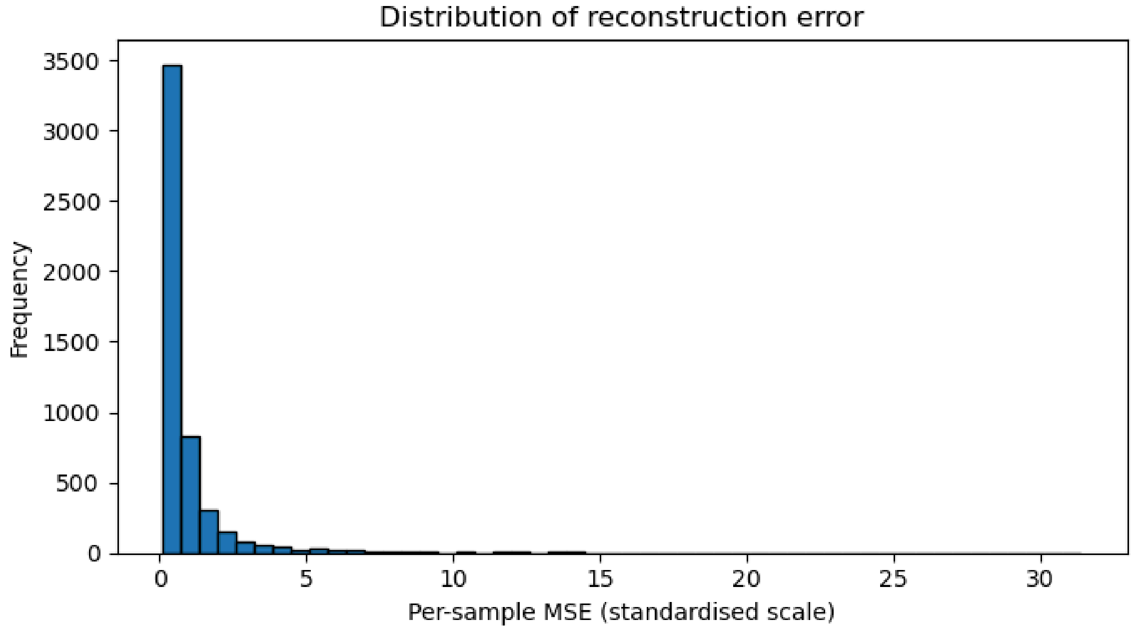
```
159/159 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
```


5
0
500
0
250
0
50
0
100
0
100
0
2023CO_XAL
2023NOX_XAL
2023NO_XAL
2023SO2_XAL
2023O3_XAL
2023NO2_XAL
Feb
2023
Mar
Apr
May
Jun
Jul
Aug
datetime


Latent space – coloured by hour of day
Latent dim 2
0
−10
−20
−30
−40
−50
−60
−20
−10
0
10
20
30
40
50
Latent dim 1
Hour (0–23)
0
5
10
15
20




```
159/159 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
```

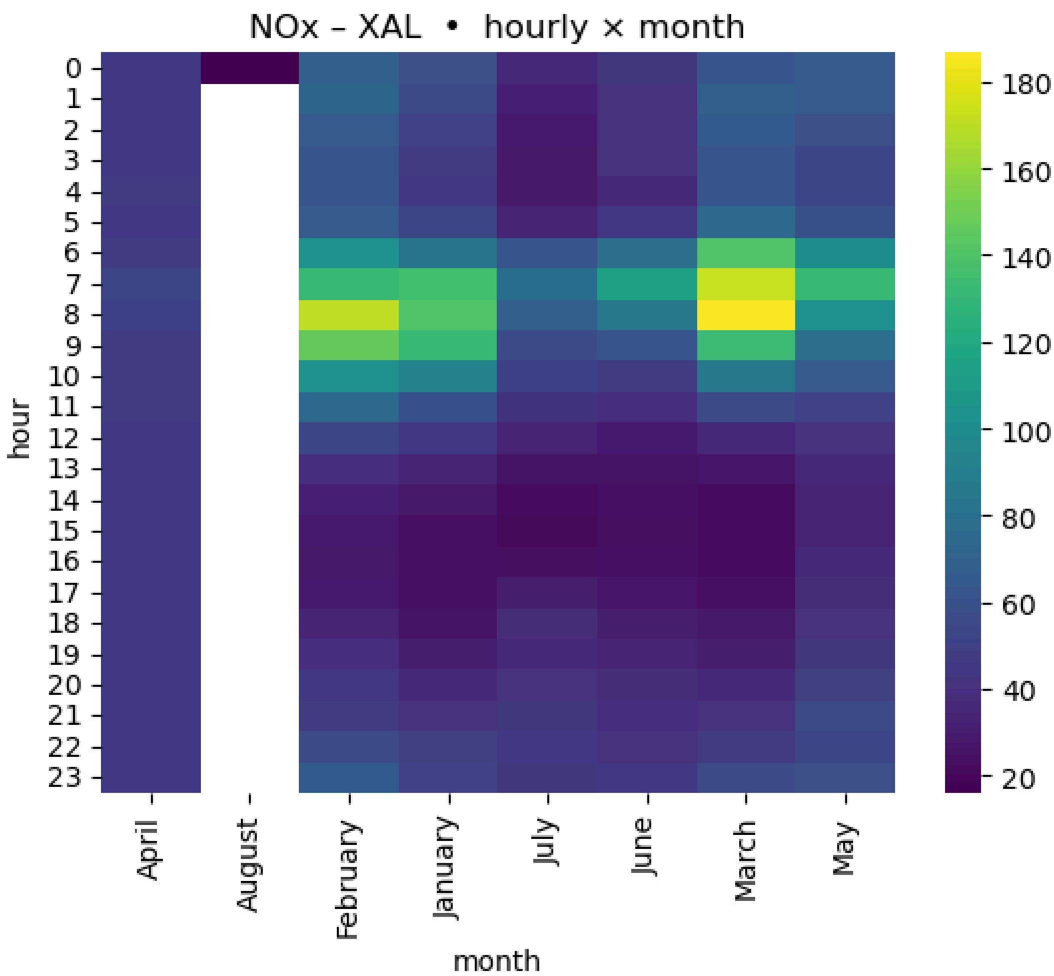



```
In [ ]: station_code = "XAL"

        cols_this_station = [c for c in wide.columns if c.endswith("_" + station_
        wide_station = wide[cols_this_station]
```

```
In [ ]: wide_station["hour"]  = wide_station.index.hour
        wide_station["month"] = wide_station.index.month_name()

        heat = (wide_station
                .groupby(["month","hour"])
                .mean()
                .unstack("month"))
```

```
import seaborn as sns
sns.heatmap(heat[ "2023NOX_XAL" ], cmap="viridis")
plt.title("NOx – XAL  •  hourly × month")
plt.show()
```

```
/var/folders/ml/fyq5x5t55wx4129dn03n32hm0000gn/T/ipykernel_93007/1868932
150.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-d
ocs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  wide_station["hour"]  = wide_station.index.hour
/var/folders/ml/fyq5x5t55wx4129dn03n32hm0000gn/T/ipykernel_93007/1868932
150.py:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-d
ocs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  wide_station["month"] = wide_station.index.month_name()
```



Conclusion

- The 2-D latent space captures **shared variance across pollutants and stations**, organising normal conditions into a dense core and pushing rare combinations into the periphery.

- Colour-by-hour plots show only mild diurnal gradients, suggesting that **station-to-station or weekly factors dominate over hourly cycles** once data are robustly scaled.
- Seasonal separation is weak in two dimensions; a higher-rank latent space (e.g. 4–5 D) or variable-specific models could reveal clearer month-wise arcs.
- Reconstruction-error histograms help **surface potential outliers**: timestamps with MSE far above the median coincide with spikes or periods of missing / imputed data.
- For further insight one could
  1. train **single-pollutant autoencoders** to isolate diurnal rhythms,
  2. aggregate hourly means and cluster stations by their 24-h fingerprints, or
  3. apply sequence models (LSTM-AE) to detect short-lived pollution episodes.

Overall, the project demonstrates how *autoencoders provide an efficient, unsupervised lens* on multivariate environmental data, balancing dimensionality reduction, anomaly detection, and exploratory visualisation in a single framework.

In [ ]:
```

```