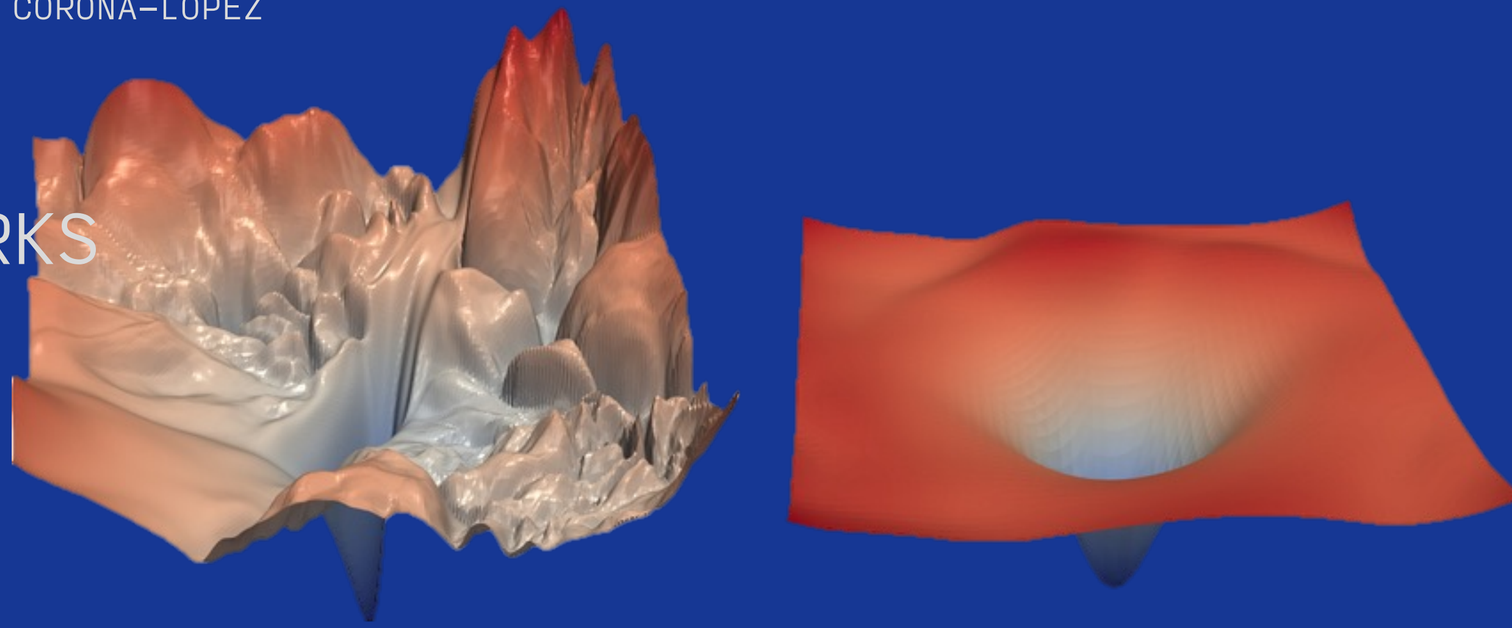
SESSION 03

WORKSHOP FEBRUARY 2026 BY DR CORONA-LOPEZ

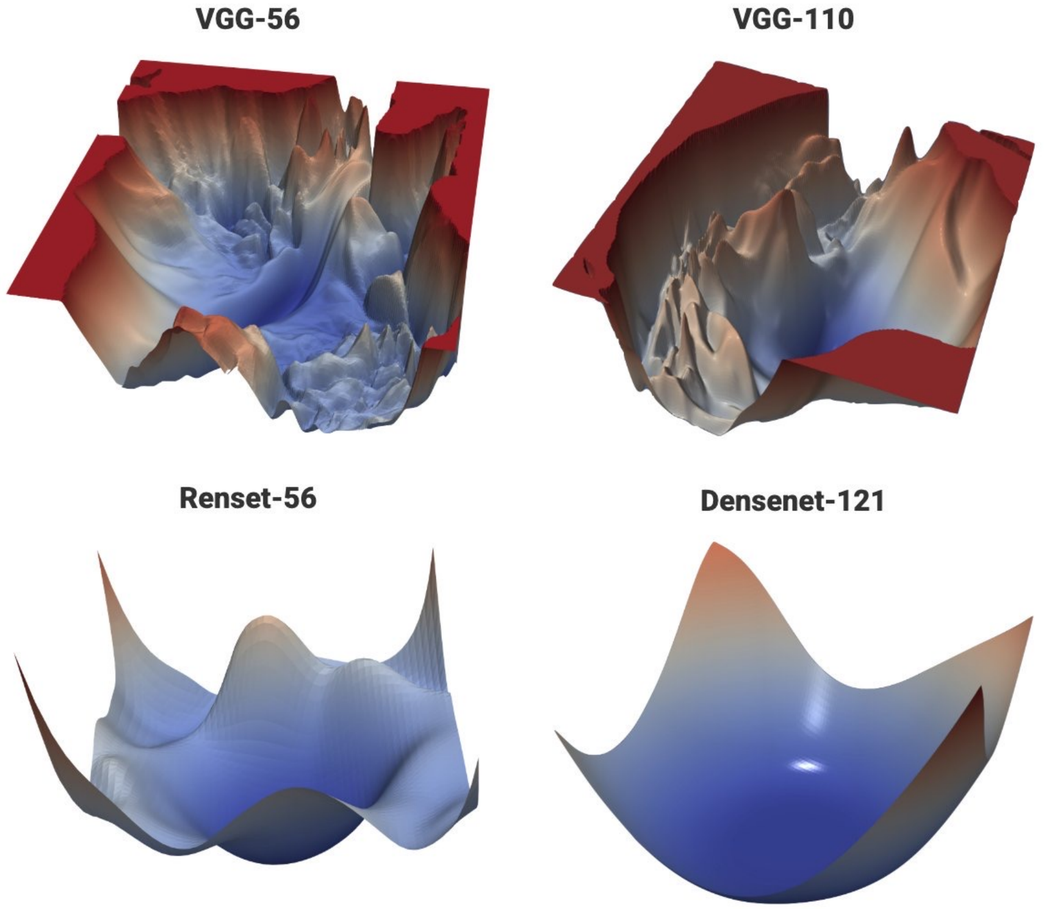
TRAINING NEURAL NETWORKS



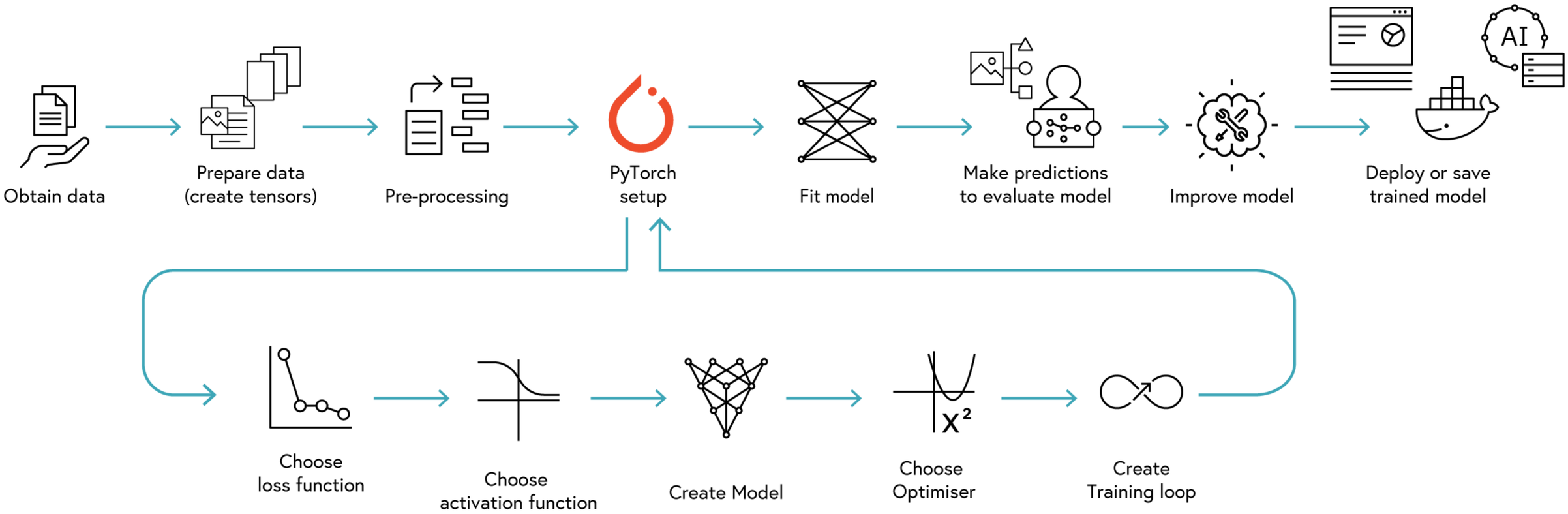
FACULTY OF SCIENCE AND ENGINEERING UNIVERSITY OF MANCHESTER

AGENDA

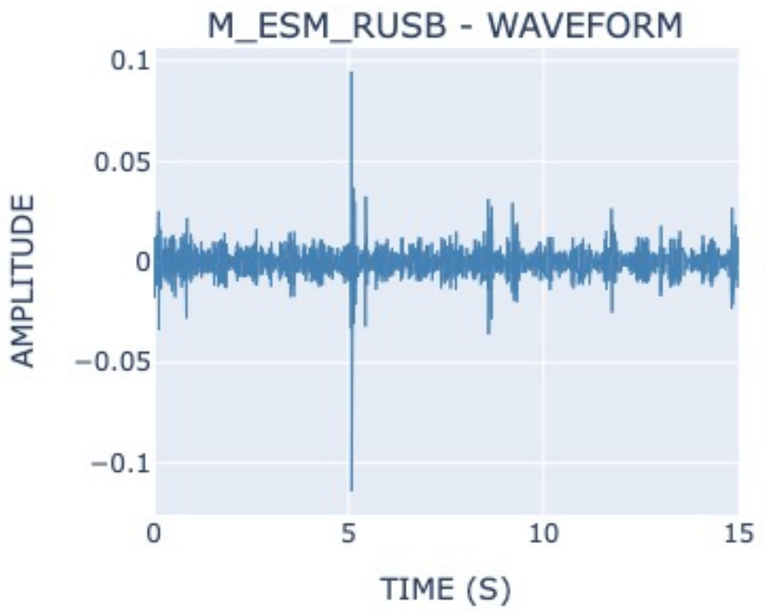
- **the pytorch workflow:** adapting the standard loop for unsupervised learning.
- **case study:** heart and lung sounds (hls-cmds) dataset.
- **audio preprocessing:** dimensionality reduction.
- **autoencoder architecture:** encoders, decoders.
- **training & validation:** overfitting/underfitting.
- **anomaly detection:** reconstruction error as a diagnostic tool.



WORKFLOW


Obtain data
Prepare data (create tensors)
Pre-processing
PyTorch setup
Fit model
Make predictions to evaluate model
Improve model
AI
Deploy or save trained model
Choose loss function
Choose activation function
Create Model
X²
Choose Optimiser
Create Training loop

CASE STUDY



the dataset: heart and lung sounds from a clinical manikin digital stethoscope (HLS-CMDS).

- total of 535 audio recordings of patient cardiac and respiratory cycles.

the data classes:

- **normal:** healthy heart and lung sounds.
- **abnormal:** pathological sounds including murmurs, crackles, wheezing, and atrial fibrillation.

the real-world challenge: in a clinical setting, we have access to massive amounts of "healthy" baseline data.

however, abnormal data is rare, highly varied, and hard to collect for every possible disease.

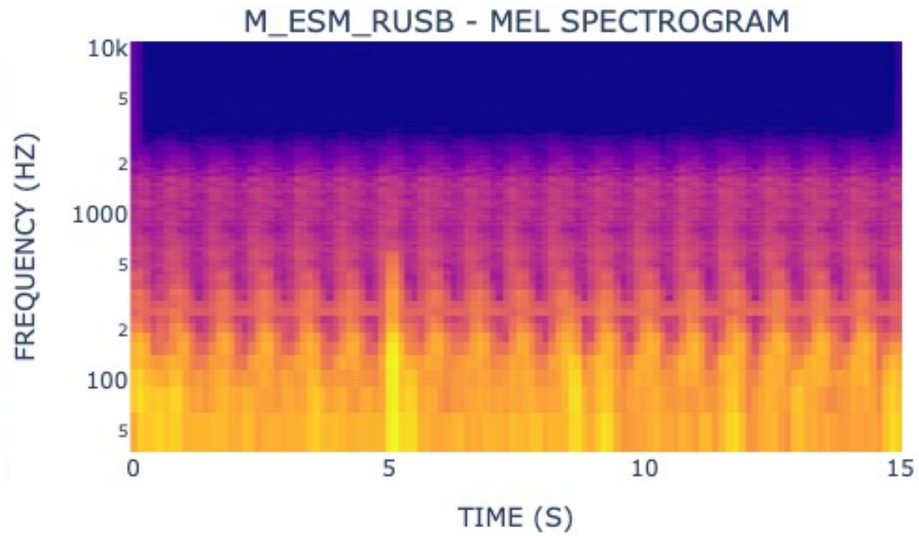
AUDIO PREPROCESSING

the problem with raw audio:

- **high dimensionality:** a mere 15 seconds of audio at 22khz equals over 330,000 data points.
- **temporal redundancy:** adjacent samples are highly correlated, leading to inefficient learning.
- **noise sensitivity:** background clinical noise affects every single sample.

the solution: MFCCs (mel frequency cepstral coefficients).

- **MFCCs** compress the audio into a highly meaningful, compact 2d feature matrix (a spectrogram).
- **the mel scale:** the human ear does not perceive sound linearly; we are much more sensitive to changes in lower frequencies than higher ones. MFFCs mathematically warp the data to mimic human hearing.



DATA SPLIT

```python
from sklearn.model_selection import train_test_split
import numpy as np

# Assume X contains our audio features (MFCCs)
# Assume y contains our labels (0: Normal, 1: Abnormal)

# Step 1: Isolate Normal and Abnormal data
X_normal = X[y == 0]
X_abnormal = X[y == 1]

# Step 2: Split the NORMAL data (70% Train, 15% Val, 15% Test)
X_train, X_temp = train_test_split(X_normal,
                                   test_size=0.30,
                                   random_state=42)
X_val, X_test_normal = train_test_split(X_temp,
                                        test_size=0.50,
                                        random_state=42)

# Step 3: Create the final Test set (Mix of Normal + ALL
Abnormal)
X_test = np.vstack((X_test_normal, X_abnormal))

# (Optional: Generate the matching y_test labels for final
evaluation)
y_test = np.concatenate((np.zeros(len(X_test_normal)),
                         np.ones(len(X_abnormal))))

print(f"Training shapes (100% Normal): {X_train.shape}")
print(f"Validation shapes (100% Normal): {X_val.shape}")
print(f"Test shapes (Mixed): {X_test.shape}")
```

approach to splitting: *in standard supervised learning, our train and test sets contain all classes. for unsupervised anomaly detection, our splitting strategy must reflect our goal of establishing a "healthy baseline."*

1. training set: 100% normal sounds.

- used exclusively to teach the autoencoder how to compress and reconstruct a healthy heartbeat.

2. **validation set:** 100% normal sounds.

- used to monitor the model during training and prevent it from overfitting.

3. **test set:** mixed normal & abnormal sounds.

- a completely independent set of recordings containing real-world diseases. used at the very end to evaluate if our anomaly threshold **actually works.**

NORMALISATION

why normalise?

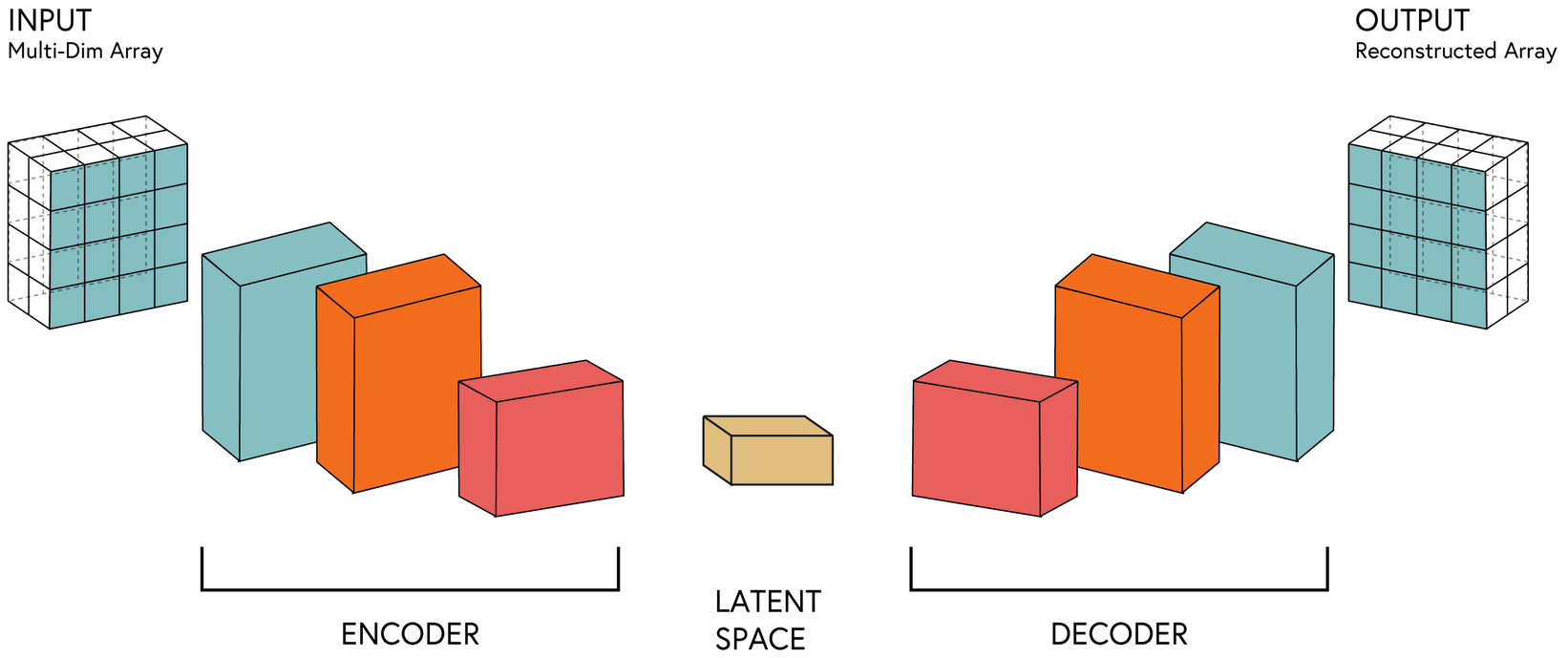
- faster convergence
- numerical stability
- equal feature contribution
- better generalization

min-max scaling:

$$X_{norm} = (X - X_{min})/(X_{max} - X_{min})$$

```python
from sklearn.preprocessing import MinMaxScaler

# Create and apply MinMaxScaler
x_scaler = MinMaxScaler()
y_scaler = MinMaxScaler()

x_scaler.fit(X_train_tensor)
y_scaler.fit(y_train_tensor)

X_train_scaled = torch.tensor(x_scaler.transform(X_train_tensor), dtype=torch.float32)
```

AUTOENCODER



1. **the encoder:** a series of progressively smaller linear layers that squash the high-dimensional input data (our mfccs) down into a smaller representation.
2. **the bottleneck (latent space):** the most restricted, severely compressed point in the network. this forces the model to discard background noise and learn only the absolute most critical "rules" of a healthy heartbeat.
3. **the decoder:** a series of progressively larger layers that take the bottleneck data and attempt to reconstruct the original input exactly as it was.

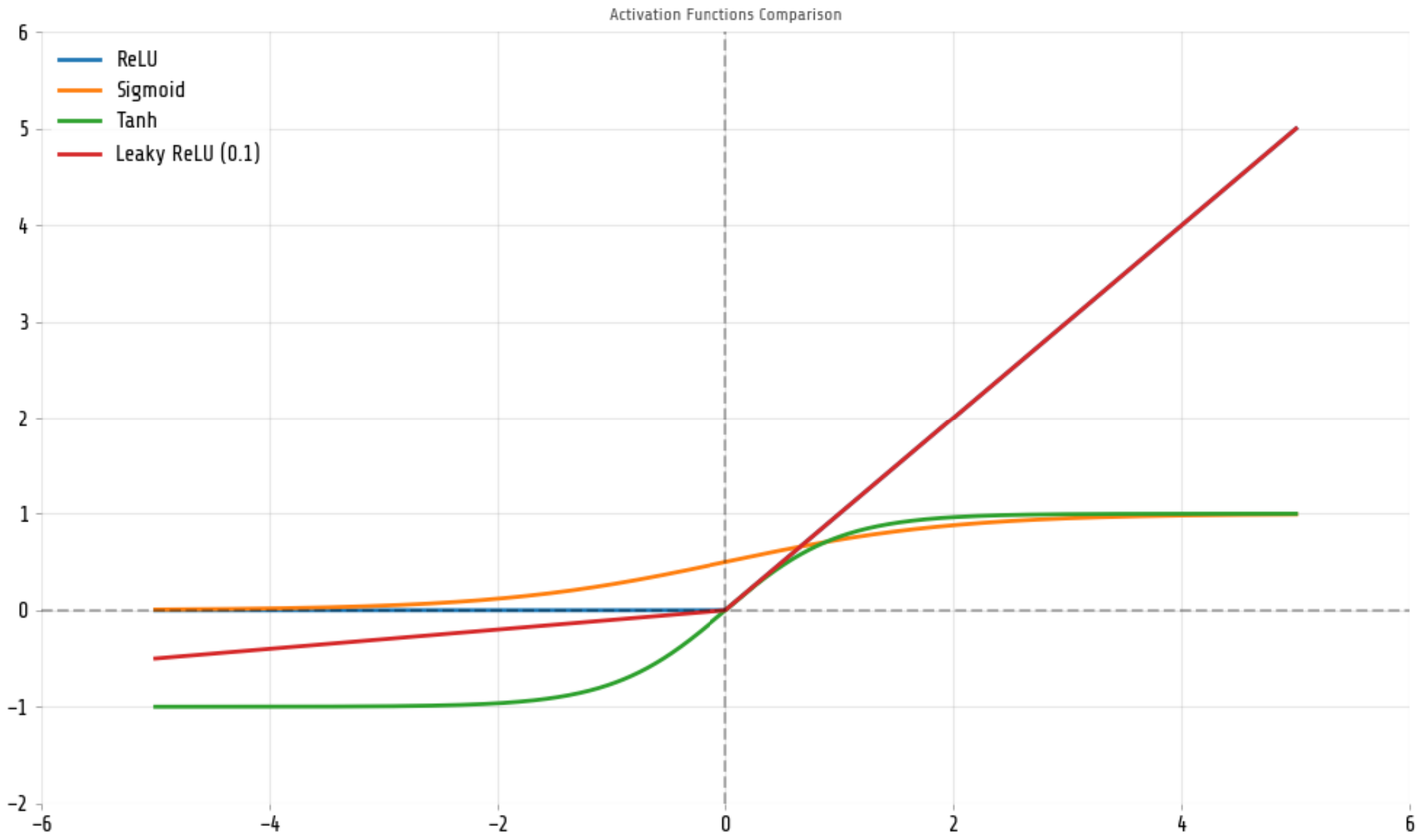
ACTIVATION FUNCTIONS

purpose:

- transform linear input to non-linear output
- enable networks to learn complex patterns and relationships

key properties:

- differentiable
- non-linear
- computationally efficient



NETWORK BUILDING

using nn.Sequential: to keep our code clean, we can group our layers together using pytorch's nn.Sequential container.

this saves us from having to manually pass the data through every single layer and activation function in the forward pass.

symmetry: notice how the decoder is generally a perfect mirror image of the encoder. if we compress from 128 down to 32, we reconstruct from 32 back up to 128.

```python
import torch.nn as nn

class AudioAutoencoder(nn.Module):
    def __init__(self, input_dim):
        super().__init__()

        # 1. The Encoder (Compressing)
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 32)      # The Bottleneck
        )

        # 2. The Decoder (Reconstructing)
        self.decoder = nn.Sequential(
            nn.Linear(32, 128),
            nn.ReLU(),
            nn.Linear(128, input_dim) # Output must match Input
dimension
        )

    def forward(self, x):
        # Compress, then immediately reconstruct
        compressed_state = self.encoder(x)
        reconstructed_state = self.decoder(compressed_state)
        return reconstructed_state
```

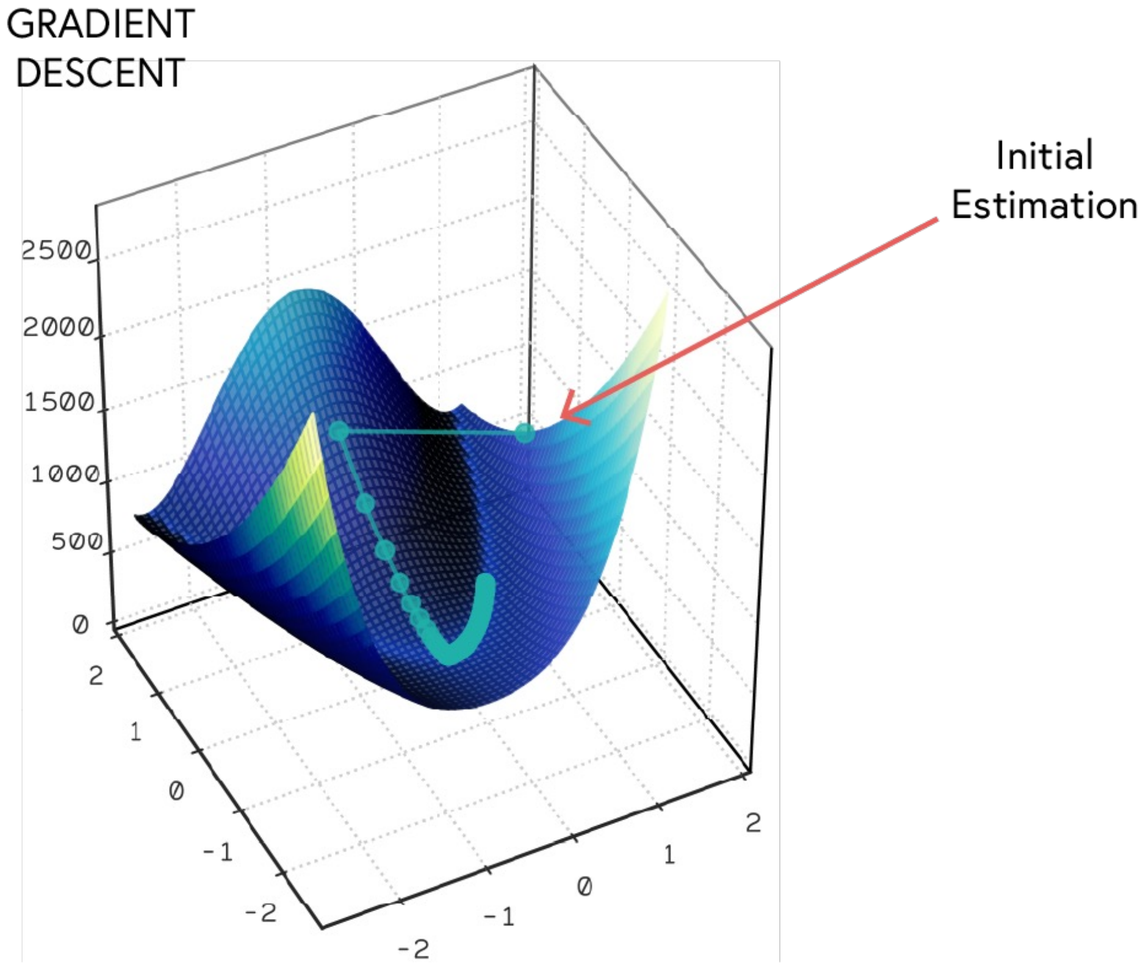
OPTIMISER & LOSS

the target is the input: in unsupervised learning with autoencoders, we don't have y labels. our target is simply our input x.

the loss function (MSE) nn.MSELoss: it calculates the mathematical difference between the original audio features and the reconstructed audio features.

- **high MSE = poor reconstruction** (the network failed to compress/decompress accurately).
- **low MSE = good reconstruction** (the network successfully learned the healthy patterns).

the optimizer (adam): we use the adam optimizer (optim.Adam). because the loss landscape of autoencoders can be complex.



OPTIMISER & LOSS

the target is the input: in unsupervised learning with autoencoders, we don't have y labels. our target is simply our input x.

the loss function (MSE) nn.MSELoss: it calculates the mathematical difference between the original audio features and the reconstructed audio features.

- **high MSE = poor reconstruction** (the network failed to compress/decompress accurately).
- **low MSE = good reconstruction** (the network successfully learned the healthy patterns).

the optimizer (adam): we use the adam optimizer (optim.Adam). because the loss landscape of autoencoders can be complex.

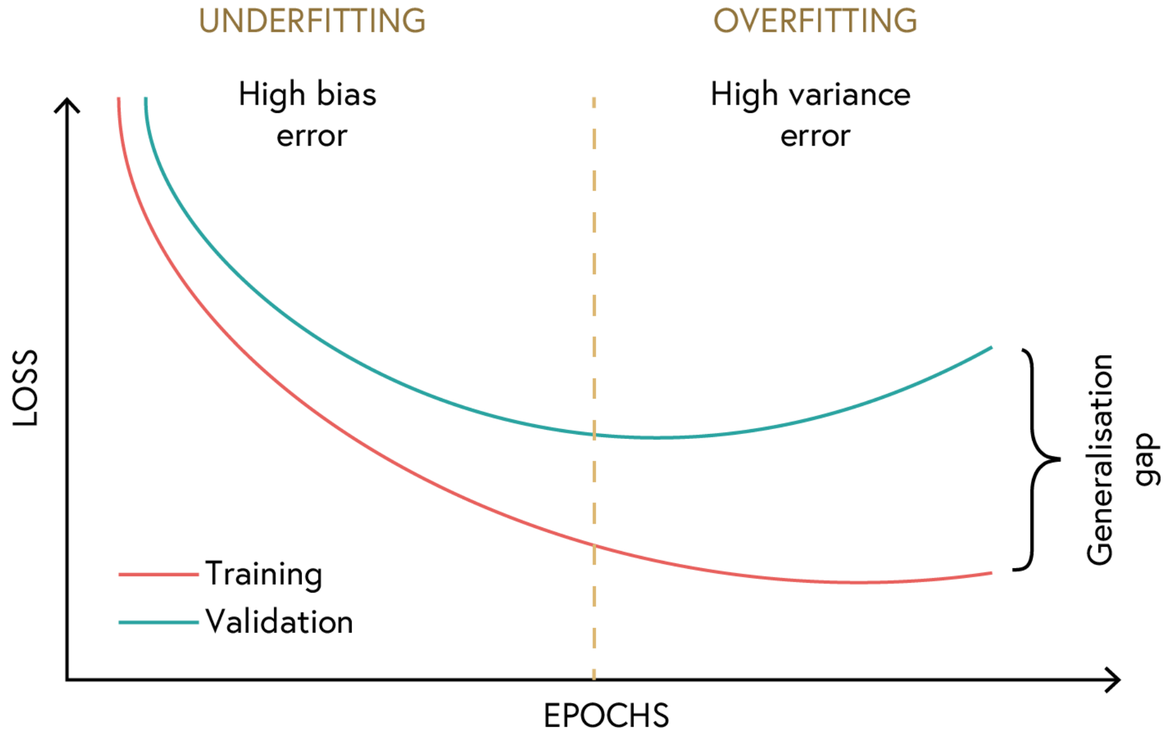
```
# Instantiate the Autoencoder
autoencoder = AudioAutoencoder(input_dim=X_train.shape[1])

# Define the Loss Function (MSE for reconstruction)
criterion = nn.MSELoss()

# Define the Optimizer (Adam for adaptive learning rates)
optimizer = torch.optim.Adam(autoencoder.parameters(), lr=0.001)
```

OVER/UNDER-FITTING

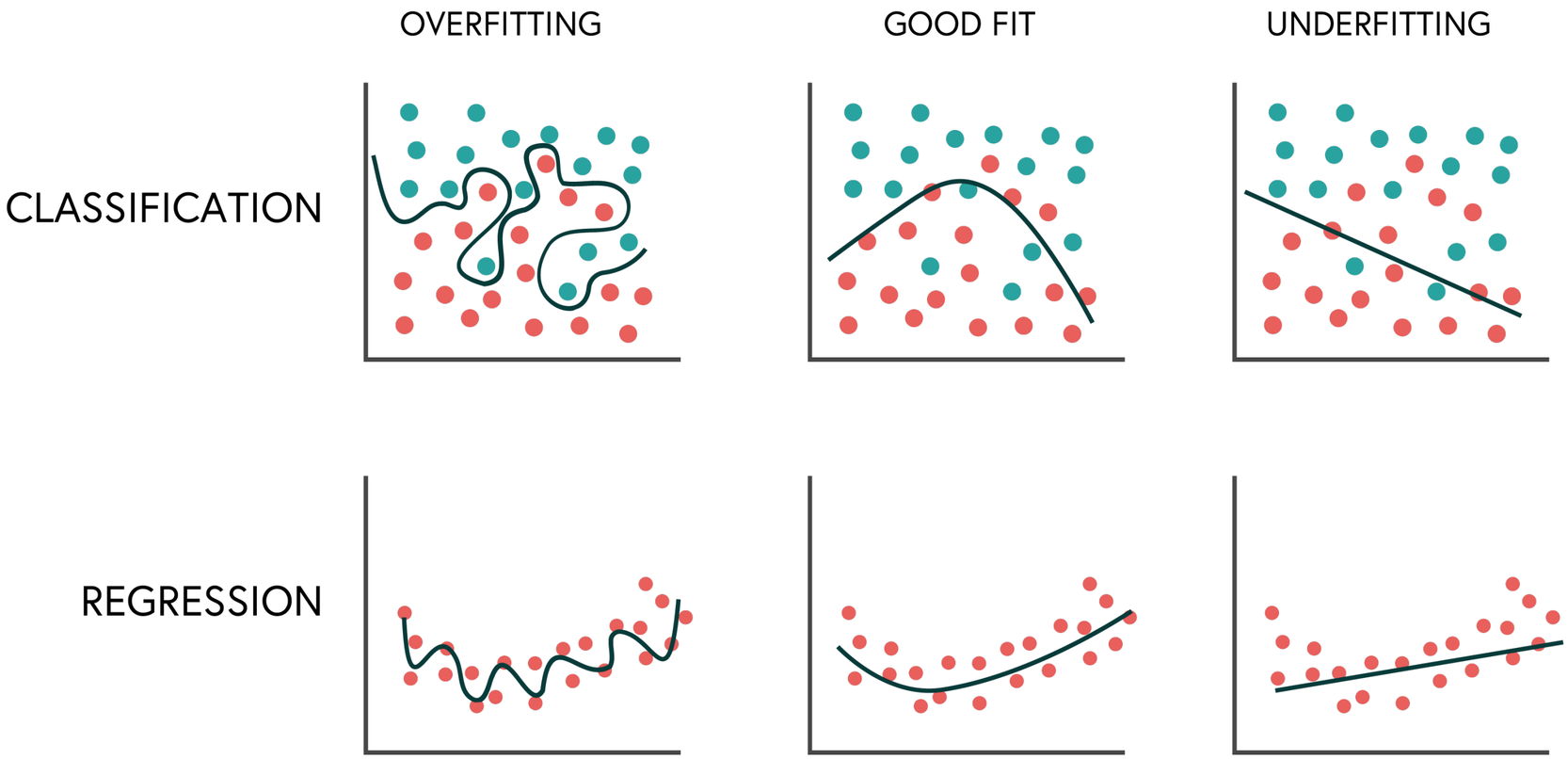
- **underfitting:** the model is too simple or hasn't trained long enough to learn the patterns of a normal heartbeat (both training and validation loss are high).

- **overfitting:** the model memorizes the exact audio files in the training set, but fails to generalize to new, unseen normal sounds.

- **a solution – early stopping:** we monitor the validation loss at every epoch. we stop training the moment the validation loss stops improving, ensuring our model generalizes perfectly to new patients.



OVER/UNDER -FITTING



EVALUATION

we now take our isolated **test set** and pass it through the trained autoencoder.

the results:

- **normal sounds:** the model recognizes the patterns, compresses them, and reconstructs them easily
- **abnormal sounds:** the model's bottleneck destroys the unknown diseased patterns, resulting in a failed reconstruction.
- **setting the threshold:** by establishing a cut-off threshold on the reconstruction error, we successfully identify medical anomalies without ever giving the network labeled diseased data!

```
📊 Key findings:
   • Normal sounds: Mean reconstruction error = 0.0142
   • Abnormal sounds: Mean reconstruction error = 0.0685
   • Separation: 4.8x higher for abnormal

✅ The autoencoder successfully learned to distinguish normal
from abnormal heart sounds!
```