Data processing

- Replace the empty nan with artifact_value = 79.1318359375
- Interpolate
  is a method of estimating the unknown value between the known values
  Linear interpolation is like drawing line and just filling the place with forward near value or backward near value
- Normalization
  is a method to rescale feature to a standard range .
  StandardScaler: rescales features so they have zero mean and unit variance, improving model performance and stability.
  In this project ; Removes subject-specific scale differences and forces the model to learn **relative patterns** and makes it more generalised
- only on the training split** (the 80% in that fold), then used to transform validation and test. Fitting on validation or test would leak information

Subject	Raw Humidity	After LOSO-Normalization
A	40–50	−1 to +1
B	60–80	−1 to +1

Machine learning

- Binary classification : predict 1 or 2 possible outcomes
- Input data(feature extraction) -model (patter recognition) - output layer (categorize)
- In your CNN-LSTM model:
  Input → sequential features (sensor readings over time) + handcrafted features (mean, std, max, min, etc.)
  Output → probability of **class 1** (event occurs)
  Binary labels → 0 (no event), 1 (event)
  Threshold & smoothing → convert probabilities → final binary predictions
  Everything you do (focal loss, class weights, optimal threshold) is **focused on improving binary classification performance**, especially when the classes are **imbalanced**

Term	Meaning	Example in your code
Feature	Input data, model "looks at"	`X_seq_train`, `X_feat_train`
Label	Output data, model "predicts"	`y_train`, `y_test`

- Training vs testing : data i split into training data with features and with labels and testing data with feature and the labels that needed to be predicted by the model.

📌 Summary (Training)

Type	Variable	Meaning
Feature	`X_seq_train`	Sequential input data
Feature	`X_feat_train`	Handcrafted input features
Label	`y_train`	Ground-truth class (0 or 1)

📌 Summary (Testing)

Type	Variable	Meaning
Feature	`X_seq_test`	Sequential input data (unseen subject)
Feature	`X_feat_test`	Handcrafted features (unseen subject)
Label	`y_test`	True class for evaluation

🔷 4. Key Difference (Training vs Testing)

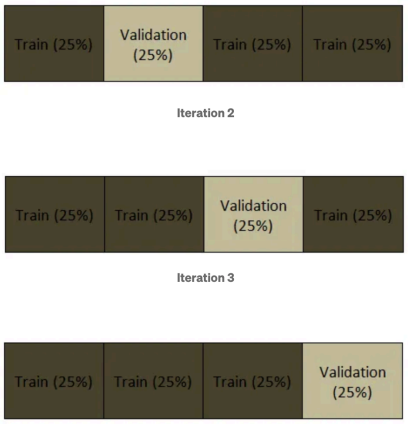
Aspect	Training	Testing
Purpose	Learn patterns	Evaluate generalization
Features	`X_seq_train`, `X_feat_train`	`X_seq_test`, `X_feat_test`
Labels	`y_train`	`y_test`
Model weights	Updated	Frozen
Subject data	All except test subject	Only test subject

- Overfitting means data leak
- Generalization : the model ability to perform well on unseen test data after learning from a train dataset .
  Because sometimes models perform great on training data but perform poorly on unseen test data so generalisation is important .
  For the model to be more generalized we should have proper bias-variance and use a cross validation method.

  Generalization was evaluated using Leave-One-Subject-Out cross-validation, where each subject was held out for testing while training on all remaining subjects. The model achieved an average accuracy of 97.0% and an average F1 score of 0.65, demonstrating robust generalization to unseen subjects despite class imbalance and inter-subject variability.

Cross-Validation

- K-fold : the dataset split into n folds and n-1 fold is for training and the 1 fold is set for testing and gives the reliable estimation of how the model will perform
- Training split : a set of training examples the network is trained on.
- Validation split : which is used to tune hyperparameters such as the number of hidden units, or the learning rate.
- VAL_FRACTION = 0.2 # 20% of training pool → validation
- Test split : which is used to measure the generalization performance
- But splitting into train/val/test wastes too much data, So we use cross-validation to reuse data efficient



- LOSO cross validation : the train data window is used for the learning weight. The validation set is also the test subject and its is used for early stop but it should be used for **tuning architecture per fold for the hyperparameter** but since all the hyperparameters are static and the subject level independence is more important.The test data is used for evaluation the generalisation.
- Fraction of the 19-subject training pool (in each LOSO fold) used as validation.

Dataset role	In your project
Training set	All windows from **non-test subjects**
Validation set	The **test subject** (used during training for early stopping)
Test set	Also the **test subject**, used for final evaluation

Class imbalance

- This happens when one class in the model over-weights compared to other class in the model, so this imbalance make the model to perform poorly on minor classes
- The solution is resampling : oversampling to minor classes and undersampling to major classes.
- Class weight: The idea is to assign higher weights to the samples of the minority class and lower weights to the majority class during the training process.

$$\text{weight}_c = \frac{n_{\text{samples}}}{n_{\text{classes}} \times n_c}$$

Where:

- n_{samples} = total number of samples
- n_{classes} = number of unique classes
- n_c = number of samples in class c

In this project class 0 - negative (major) , class 1- positive/event (minor) in the y_train (label train dataset) so the `class_weights`
In the code tries to find the the weight of class 0 occurrence and the weight of class 1 occurrence and map it like “{0: 1.5, 1: 0.75}”

Binary cross entropy 👍

- each data point truly belongs to only one class

- Cross-entropy loss is a way to measure how close a model's predictions are to the correct answers in classification problems.
- It quantifies the difference between the actual class labels (0 or 1) and the predicted probabilities output by the model. The lower the binary cross-entropy value, the better the model's predictions align with the true labels.

$$BCE = -\frac{1}{N}\Sigma_{i=1}^{N}(y_i.log(p_i) + (1-y_i)log(1-p_i))$$

where

- N is number of samples,
- y_i true label for sample i(0 or 1),
- p_i model-predicted probability for class 1 for sample i.

Observation	True Label (y)	Predicted Probability (p)
1	1	0.9
2	0	0.2
3	1	0.8
4	0	0.4

We will calculate the Binary Cross-Entropy loss for this set of observations step-by-step.

Observation 1:

Here, True label y_1=1 and Predicted probability p_1=0.1

$$\text{Loss}_1 = -\left(1 \cdot \log(0.9) + (1-1) \cdot \log(1-0.9)\right) = -\log(0.9) \approx -(-0.1054) = 0.1054$$

Similarly, for other classes,

1. Predicted probability $p2$ =0.2 and Loss2=0.223
2. Predicted probability p3 =0.8and Loss3=0.2231
3. Predicted probability p4 =0.4 and Loss4=0.5108

Next, we sum the individual losses and calculate the average:

$$\text{Total Loss} = 0.1054 + 0.2231 + 0.2231 + 0.5108 = 1.0624$$

$$\text{Average Loss (BCE)} = \frac{1.06244}{4} = 0.2656$$

Therefore, the Binary Cross-Entropy loss for these observations is approximately 0.2656.

Focal loss:

- Advance extension of multi-class cross entropy and BCE
- Used for balancing the cross loss entropy
- Reason : cross-entropy loss ends up overwhelmed by the dominant class, often leaving the minority classes in the dust.
- Two main hyper parameter = alpha , gamma

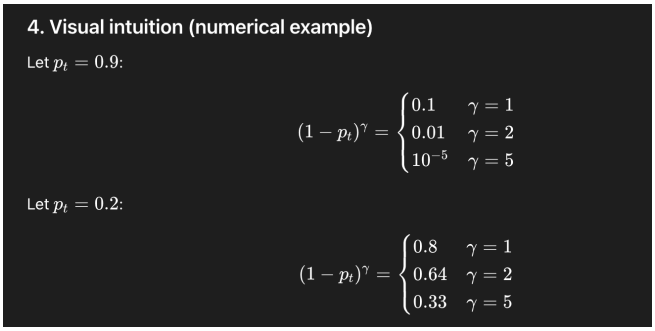
$$\text{FL}(p_t) = -\alpha\,(1-p_t)^{\gamma}\,\log(p_t)$$

Where:

- $p_t = p$ if $y = 1$, $p_t = 1 - p$ if $y = 0$
- $\alpha \in [0, 1]$ = weight for class imbalance (like class weight)
- $\gamma \geq 0$ = focusing parameter

- Pt : model prediction, its low if the model is uncertain
- Gamma : Focusing on Hard-to-Classify Samples

- So if p_t is large and if gamma is 0 to 5 the it suppress focus and if p_t in small and gamma is 0 to 5 it preserves the focus



- Alpha : Controlling Class Weights, This extra control allows you to further adjust for class imbalance beyond what gamma alone can do
- Focal Loss=BCE×(1−pt)γ×α (this project)

Technique	Purpose
Class weight	Handles **global imbalance**
Focal loss	Focuses on **hard samples**
Threshold tuning	Controls precision–recall tradeoff

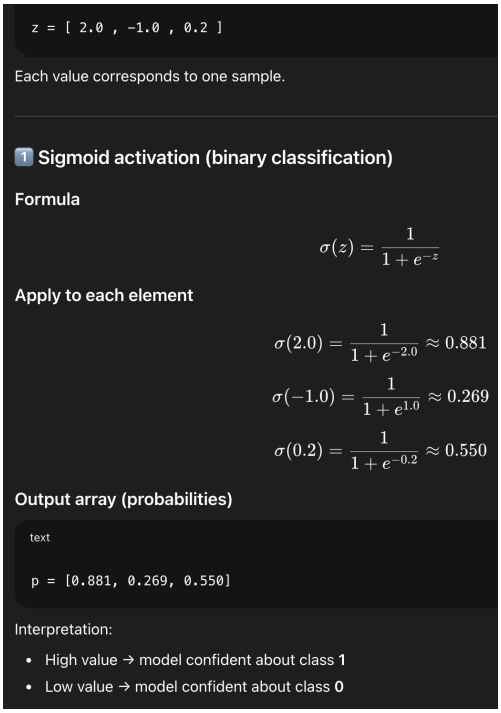
Deep learning

- Neurons: are nodes within the neural network that serve as connection points, it's a combination of data sets, weights, and biases that works to identify, classify, and describe objects within the data.
  Weights : **Weights** control **how important** each input is , and its learned automatically and optimized during backpropagation
  Heart rate (weight = 2.5) → very important
  Humidity (weight = 0.2) → less important
  Bias : extra learnable value added to the neuron , allows the neuron to **shift** the activation function left or right (+ve /-ve). It's also automatically done by keras
- f=wx
- Input layer - hidden layer - output layer
- X - $\sigma(W0,1x + b0,1)$ - f
- Feedforward neurons: connect in way that information flow forward from the start to
- Convolution neurons: Neurons within (CNNs) gradually improve and learn from their data and training
- Recurrent neurons: Neurons within (RNNs) take information from prior inputs (the neuron's "memory") to determine current inputs and outputs.
- Layer : deep learning layers, including:
- Input layer : receives data for processing.
- Output layer : final prediction based on the data or classification of the data.
- Synaptic weight: Weights refer to the strength of the connection between neurons.
- Activation function: Establishes the output of a neuron based on its input.
- My model uses **multiple layer types**:
  1) **CNN layer:** Extract **local patterns** from sequences,Neurons look at short time windows

  2) **LSTM layers :** Capture **temporal dependencies**, Neurons have memory (past information)

3) **Dense (fully connected) layers :** Combine learned features, Perform final decision

- weights determine feature importance, bias shifts the activation threshold, and layers organize neurons to learn hierarchical representations.
- Activation function : introduce nonlinearity to the model and transform the neuron input to output , in lament term “An activation function decides whether and how strongly a neuron “fires”.” and by doing so the model can learn new pattern
- Activation functions turn raw model scores into usable signals (probabilities or features), and loss functions measure how wrong those activated outputs are.
  In this project we are using relu and sigmoid activation function.
  Sigmoid : well suited for binary classification, threshold handling and class imbalance. It is used in the output layer , if used in a hidden layer : stuck at 0/1 and slower learning .



```
z = [ 2.0 , -1.0 , 0.2 ]
```

Each value corresponds to one sample.

1️⃣ **Sigmoid activation (binary classification)**

Formula

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Apply to each element

$$\sigma(2.0) = \frac{1}{1 + e^{-2.0}} \approx 0.881$$

$$\sigma(-1.0) = \frac{1}{1 + e^{1.0}} \approx 0.269$$

$$\sigma(0.2) = \frac{1}{1 + e^{-0.2}} \approx 0.550$$

Output array (probabilities)

```
text

p = [0.881, 0.269, 0.550]
```

Interpretation:

- High value → model confident about class **1**
- Low value → model confident about class **0**

Relu : Sensor signals are noisy and nonlinear CNN and Dense layers need to: detect patterns ignore irrelevant signals ReLU acts like a pattern detector fires only when a meaningful pattern exists,it use in hidden layer

3 ReLU activation (hidden layer)

Input array

```
text

z = [-2.0, 1.5, 0.0, 3.2]
```

ReLU formula

$$\text{ReLU}(z) = \max(0, z)$$

Output

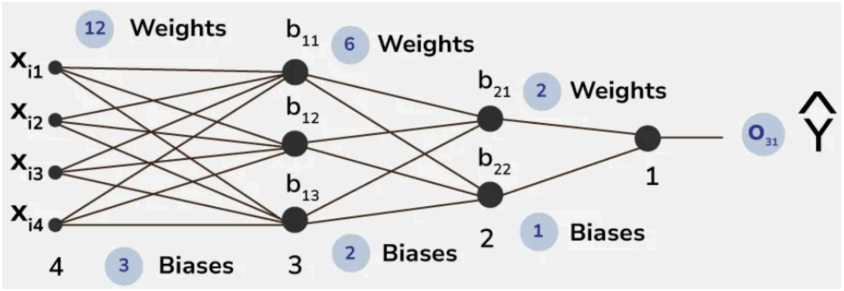
```
text

a = [0.0, 1.5, 0.0, 3.2]
```

Interpretation:

- Negative values → turned off
- Positive values → passed forward

Layer	Activation	Purpose
CNN	ReLU	Detect local patterns
LSTM	tanh + sigmoid	Control memory
Dense (hidden)	ReLU	Non-linearity
Output Dense	Sigmoid	Probability output

- Forward propagation : transforms raw inputs into predictions using weights, biases and activation functions. it produces **predictions** which loss function compares to labels



Input → Hidden

```
text

W₁ = [[0.1, 0.2],
      [0.3, 0.4]]

b₁ = [0.1, 0.1]
```

Hidden → Output

```
text

W₂ = [0.5, -0.6]
b₂ = 0.2
```

3 Hidden layer (Linear step)

$$z^{(1)} = W_1 x + b_1$$

$$z^{(1)} = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{bmatrix} \begin{bmatrix} 1.0 \\ 2.0 \end{bmatrix} + \begin{bmatrix} 0.1 \\ 0.1 \end{bmatrix}$$

$$z^{(1)} = \begin{bmatrix} 0.1(1) + 0.2(2) + 0.1 \\ 0.3(1) + 0.4(2) + 0.1 \end{bmatrix} = \begin{bmatrix} 0.6 \\ 1.2 \end{bmatrix}$$

4 Hidden layer (Activation: ReLU)

$$a^{(1)} = \text{ReLU}(z^{(1)})$$

$$a^{(1)} = [0.6, 1.2]$$

(ReLU keeps positives, removes negatives)

5 Output layer (Linear step)

$$z^{(2)} = W_2 a^{(1)} + b_2$$

$$z^{(2)} = 0.5(0.6) - 0.6(1.2) + 0.2$$

$$z^{(2)} = 0.3 - 0.72 + 0.2 = -0.22$$

6 Output layer (Activation: Sigmoid)

$$\hat{y} = \sigma(z^{(2)}) = \frac{1}{1 + e^{-z^{(2)}}}$$

$$\hat{y} = \frac{1}{1 + e^{0.22}} \approx 0.445$$

7 Final output

```
text

Predicted probability = 0.445
Predicted class = 0 (if threshold = 0.5)
```

- Backward propagation : after forward pass calculation the error is computed in the outer layer and by using focal loss the error i calculated (there is no learning until now) then the gradient of loss function is calculated w.r.t the network weight (if the gradient is +ve= decrease prediction, if the gradient is -ve = increase prediction) so in order to minimize the loss we use optimization algorithm called as gradient descent , it minimises the loss as possible by adjusting the weight parameter
- This cycle (forward propagation, error calculation, backpropagation, and weights & biases update) is repeated for many iterations (or epochs) during the training process, gradually improving the accuracy of the network's predictions.

1 Loss Function (what the model is minimizing)

For **binary classification**, the most common loss is **Binary Cross-Entropy (BCE)**.

Definition

Given:

- True label $y \in \{0, 1\}$
- Predicted probability $\hat{y} \in (0, 1)$

$$\boxed{\mathcal{L}(y, \hat{y}) = -\left[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})\right]}$$

Using our example

- True label: $y = 1$
- Prediction: $\hat{y} = 0.445$

$$\mathcal{L} = -\log(0.445) \approx 0.810$$

➡ **High loss** because the prediction is far from 1.

2 Gradient of the Loss (core of backpropagation)

The gradient tells us **how to change weights to reduce loss.**

3 Step 1: Gradient w.r.t. output activation $\hat{y}$

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = -\left(\frac{y}{\hat{y}} - \frac{1 - y}{1 - \hat{y}}\right)$$

For $y = 1$:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = -\frac{1}{\hat{y}} \approx -2.247$$

4 Step 2: Gradient of Sigmoid activation

Sigmoid function:

$$\hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}}$$

Derivative:

$$\boxed{\frac{d\hat{y}}{dz} = \hat{y}(1 - \hat{y})}$$

For our example:

$$\frac{d\hat{y}}{dz} = 0.445(1 - 0.445) \approx 0.247$$

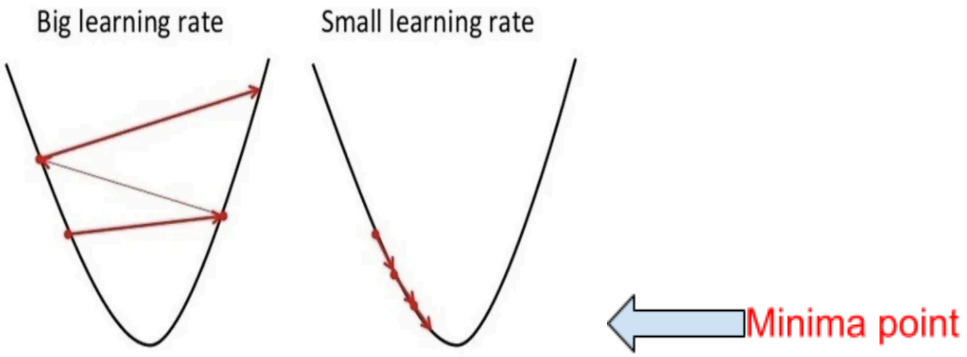
5 Step 3: Gradient of loss w.r.t. logit z

Using chain rule:

$$\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{d\hat{y}}{dz}$$

$$\frac{\partial \mathcal{L}}{\partial z} = (-2.247)(0.247) \approx -0.555$$

- Loss function : how close the output (prediction) is to the target label .
- Cost function: Used to refer to an average of the loss functions over an entire training data.
- Large loss means bad prediction therefore need to improve the training model
- Classification loss: binary cross entropy loss, multi-class cross entropy loss
- Multi-class cross entropy : the class with higher value is the correct prediction compared to the class with lower value.
- Binary cross entropy: measures the distance b/w the true label and the predicted label per epochs and if the BCE is high the prediction is wrong and if the BCE is low the prediction is correct
- To reduce the loss and reach the minima/ideal point the model need to move to the opposite direction and that direction is represented by the term called learning rate/step size " the size of the steps taken to reach the approximate minimum point , small step = slow learning , big step = converge and move pass the minimum point.



- The gradient decent aims to minimise the loss as there are 3 types of gradient decent

Gradient Descent	Stochastic Gradient Descent	Mini-batch gradient descent
Gradient Descent determines the cost function's gradient throughout the whole training dataset and updates the model's parameters based on the mean of all training examples across each epoch.	Stochastic gradient descent involves updating the model parameters and computing the gradient of the cost function for a single random training example at each iteration.	Mini-batch Gradient Descent updates the model parameters based on the mean gradient of the cost function with respect to the model parameters over a mini-batch, which is a smaller subset of the training dataset of equivalent size.
As each iteration of the approach requires computing the gradient of the cost function across the whole training dataset, GD takes some time to converge.	SGD adjusts the model parameters more often than GD, which causes it to converge more quickly	In order to strike a reasonable balance between speed and accuracy, the model parameters are changed more frequently than GD but less frequently than SGD.
Due to the requirement to retain the whole training dataset, GD consumes a lot of memory.	As just one training sample needs to be stored for each iteration, SGD requires less memory.	Just a percentage of the training samples had to be retained for each repetition, therefore the memory use is manageable.
GD is computationally expensive because the gradient of the cost function must be computed for the whole training dataset at each iteration.	As the cost function's gradient only needs to be calculated once for each repeat of training data, SGD is computationally efficient.	As the gradient of the cost function must be calculated for a portion of the training examples for each iteration, it is computationally efficient.
With little error, GD modifies the model's parameters based on the average of all training samples.	Due to the fact that SGD is updated using just one training sample, it has a lot of noise.	Mini-batch Gradient Descent has a significant amount of noise because the update is based on a small number of training examples.

- The project is using a mini batch gradient decent and the batch size is 32 i.e the model updates weights every 32 samples therefore the model updates weights ~1,400 times per epoch as mentioned in the output

Note : one epoch contains n number of samples

Optimization

- Technique used to adjust the parameter and minimise the error and improve accuracy
- First order optimization algorithm used in this project are mini-batch gradient descent (base moment) and Adam optimization(1st moment not 2nd derivative so it not 2nd order algo) and we are not using second order algorithm optimization technique.
- Adam (Adaptive Moment Estimation) optimization : is a combination of Momentum + RMS (root mean square) Propagation.
  Since mini-batch gradient (which is noisy) from back propagation as base moment ”($gt = \nabla \theta L(\theta; \text{mini-batch})$)” and some parameter need small updates and som parameter needs large update the learning rate need to adapt automatically

Momentum (1st moment) : Keeps a running average of gradients, Smooths noisy updates, Helps move consistently in the right direction
RMSProp (2nd moment) : Keeps a running average of squared gradients, Adapts learning rate per parameter, Prevents large, unstable updates , Think of it as automatic step-size control.

2 Adam update equations (mathematical form)

At time step t, for parameter θ with gradient $g_t = \nabla_\theta \mathcal{L}_t$:

Step 1: First moment (momentum)

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

Step 2: Second moment (RMS)

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

Step 3: Bias correction

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Step 4: Parameter update

$$\theta_t = \theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Typical hyperparameters

Symbol	Value
η (learning rate)	0.001
β_1	0.9
β_2	0.999
ϵ	10^{-8}

3 Numeric example (single parameter)

Assume:

- Initial parameter: $\theta_0 = 0.5$
- Gradient at step 1: $g_1 = -0.555$ (from our backprop example)
- $m_0 = 0, v_0 = 0$

Step 1: First moment

$$m_1 = 0.9(0) + 0.1(-0.555) = -0.0555$$

Step 2: Second moment

$$v_1 = 0.999(0) + 0.001(-0.555)^2 = 0.000308$$

Step 3: Bias correction

$$\hat{m}_1 = \frac{-0.0555}{1-0.9^1} = -0.555$$

$$\hat{v}_1 = \frac{0.000308}{1-0.999^1} = 0.308$$

Step 4: Update parameter

$$\theta_1 = 0.5 - 0.001 \cdot \frac{-0.555}{\sqrt{0.308} + 10^{-8}}$$

$$\sqrt{0.308} \approx 0.555$$

$$\theta_1 \approx 0.5 + 0.001 = 0.501$$

➡️ Parameter moves **in the correct direction**, with adaptive step size.

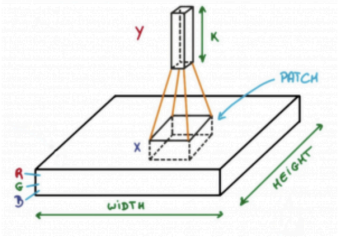
6 Final comparison (THIS is the key insight)

Stage	Prediction
Before update	**0.445**
After update	**0.533**
True label	**1.0**

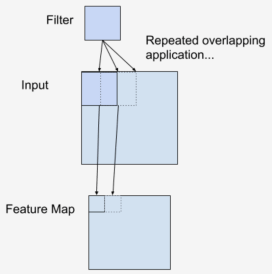
✅ **Prediction moved closer to the true label**

Convolution Neural Network (CNN)

- Convolution : is where we extract features from a large input data by taking a small sliding window and extracting a small patch of the data and



- Convolution layer : this process repeats by sliding across the table and each patch from the input dataset is multiplied with a filter/kernel (i.e weight) via dot product matrix multiplication (element-wise multiplication) which is then summed which always gives single value therefore this operation is called scalar product . Using a filter/kernel (weight) smaller than the input is intentional as it allows the same filter (set of weights) to be multiplied by the input array multiple times at different points on the input. Specifically, the filter is applied systematically to each overlapping part or filter-sized patch of the input data, left to right, top to bottom. This systematic application of the same filter across an image is a powerful idea. If the filter is designed to detect a specific type of feature in the input, then the application of that filter systematically across the entire input image allows the filter an opportunity to discover that feature anywhere in the image. This capability is commonly referred to as translation invariance, e.g. the general interest in whether the feature is present rather than where it was present. And The output of the convolution layer is referred as feature maps



- Kernel/filter : is small sliding matrix fo weight used for feature extraction from the input data, Values of kernels are initially random and then learned during training process using backpropagation that updates the kernel values to better detect important features like (Sudden spikes Ramps / trends Oscillations/ Transitions Frequency-like patterns),
- the kernel used in this project is custom kernel/data-driven kernel/learning filter,
- The kernels operate in the time domain but may implicitly capture frequency-related features by using band pass filter in-spite of it not being a Frequency-Specific Kernels
- Kernels evolve to detect minority-class patterns Prevents majority-class dominance with goes well with focal loss.

2 What is Convolution?

Convolution = sliding the kernel across the input and computing **dot products.**

$$y_i = \sum_{j=0}^{k-1} w_j \cdot x_{i+j}$$

Where:

- y_i = output at position i (feature map)
- x = input array
- w = kernel of size k

1D Example

Input features:

$$x = [1, 2, 3, 4, 5]$$

Kernel:

$$w = [0.2, 0.5, -0.3]$$

Convolution calculation:

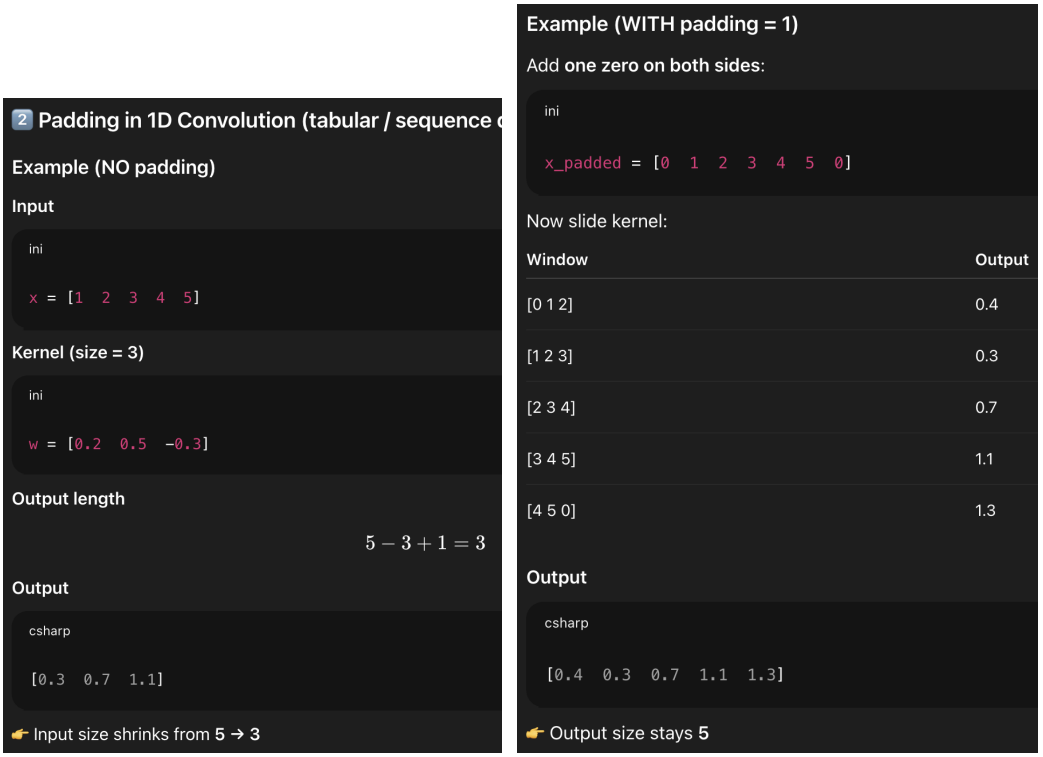
1. First window `[1 2 3]` → $y_0 = 0.2 * 1 + 0.5 * 2 + (-0.3) * 3 = 0.3$
2. Second window `[2 3 4]` → $y_1 = 0.2 * 2 + 0.5 * 3 + (-0.3) * 4 = 0.7$
3. Third window `[3 4 5]` → $y_2 = 0.2 * 3 + 0.5 * 4 + (-0.3) * 5 = 1.1$

Output feature map:

$$y = [0.3, 0.7, 1.1]$$

✅ This is exactly how a CNN extracts **local patterns.**

- Padding : it usually add 0 to the edges on an image in order to preserve the input data from being minimised due to the smaller kernel value multiplication is such thing happens the value in the border will be less compared to the value in the center (the pixels present at the edges get operated with kernels fewer times compared to the central ones). usually padding i applied along the height/width of an image .
- Padding applies to any convolution, including 1D time-series convolution. Padding is applied along the time axis therefore padding add extra time series “0 “ at the beginning and the end , in this project zero padding (“padding= same” = it Does not introduce artificial signal) is used
- The reason for having padding is to match the input and output size , avoid shrinking, and maybe for the model to identify the start and the end of the window (boundary pattern learning).
- Length_out = Length_in − kernel_size + 1



- Pooling : down-sampling operation applied after convolution, purpose : Reduce sequence length by keeping the most important features , Add robustness to small temporal shifts, Reduce overfitting .ther are 3 pooling techniques: max pooling, min pooling, average pooling .
- Max pooling is used in most of the cases. When we use 2x2 pooling, the image size reduces four times. This helps us retain the highest value i.e., features as we shrink down the image size. In this project it Detects whether an important pattern occurred and Keeps peak activations (note : Average pooling would: Smooth out strong signals, Reduce event sharpness)
- Stride how far the kernel moves each step (if stride =2 Skips positions during convolution) this project use implicit stride =1 cause having stride > 1 means missing out on some patterns .
- Padding = adding margins so text near the page edges is not cut. Pooling = summarizing paragraphs into key points

4 1D Pooling Example (tabular / sequence data)

Input (feature map after convolution)

```
[0.4  0.3  0.7  1.1  1.3  0.9]
```

Max Pooling

- Pool size = 2
- Stride = 2

Pooling steps

Window	Output
[0.4 0.3]	0.4
[0.7 1.1]	1.1
[1.3 0.9]	1.3

Output

```
[0.4  1.1  1.3]
```

Average Pooling (same input)

Window	Output
[0.4 0.3]	0.35
[0.7 1.1]	0.9
[1.3 0.9]	1.1

2 1D Min Pooling Example (tabular / sequence)

Input feature map

```
[0.4  0.3  0.7  1.1  1.3  0.9]
```

Pool size = 2, stride = 2

Window	Min
[0.4 0.3]	0.3
[0.7 1.1]	0.7
[1.3 0.9]	0.9

Output

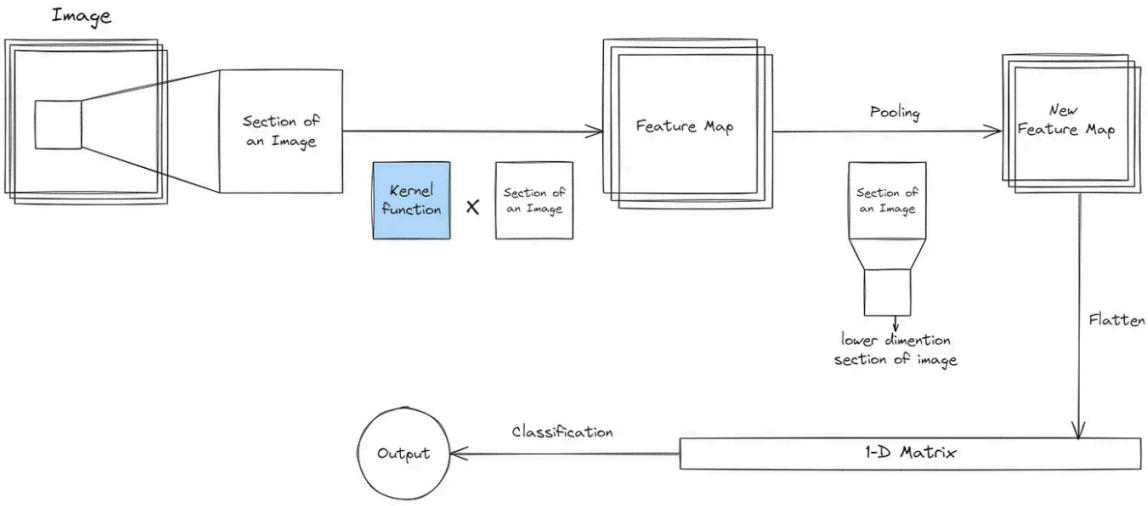
```
[0.3  0.7  0.9]
```

- Feature map :

3 Example shape flow

Layer	Output Shape	Notes
Input	(100, 1)	1D sequence, 100 timesteps
Conv1D 32 filters	(100, 32)	32 feature maps, stride=1, padding=same
MaxPooling1D pool=2	(50, 32)	Sequence downsampled, 32 feature maps
Conv1D 64 filters	(50, 64)	64 new feature maps
MaxPooling1D pool=2	(25, 64)	Downsampled again

> So **feature maps = number of filters** at that layer.



RNN

- Rnn: handle sequential data by taking the output from previous steps and using it as input for the current step. They are essential for tasks that involve time series data, and they recognize patterns in sequences of data. They have a “memory” that captures information about what has been calculated so far. This memory allows RNNs to use prior inputs to influence the current output.
- Sequence : is an ordered list of data points where the order (time) matters and in rnn each element are dependent,

- Rnn process by iterating through elements, where the hidden state at time t is updated using the current input and the previous state t-1. They handle variable-length sequences by sharing weights across time steps, allowing them to capture temporal dependencies and context.

$$X = \{x_1, x_2, x_3, \ldots, x_T\}$$

Where:

- T = sequence length (timesteps)
- x_t = feature vector at time step t

- Hidden state : RNN tracks the context by maintaining a hidden state at each time step. A feedback loop is created by passing the hidden state from one-time step to the next. The hidden state acts as a memory that stores information about previous inputs. At each time step, the RNN processes the current input along with the hidden state from the previous time step. This allows the RNN to "remember" previous data points and use that information to influence the current output.
- Having hidden states in this project helps with retaining the memory of the model and in doing so a sequence of spikes indicates an event and (1 spike != event)

3 Mathematical view (simple RNN)

$$h_t = f(W_x x_t + W_h h_{t-1} + b)$$

Where:

- x_t = input at time step t
- h_{t-1} = previous hidden state
- h_t = new hidden state
- $f(\cdot)$ = activation function

5 Simple numeric example (step by step)

Assume:

- x_t is scalar
- h_t is scalar
- $f = \tanh$

Parameters:

$$W_x = 0.5, \quad W_h = 0.8, \quad b = 0$$

Sequence:

$$x = [1.0,\ 2.0,\ 1.5]$$

Initial hidden state:

$$h_0 = 0$$

Step 1: $t = 1$

$$h_1 = \tanh(0.5(1.0) + 0.8(0)) = \tanh(0.5) \approx 0.462$$

Step 2: $t = 2$

$$h_2 = \tanh(0.5(2.0) + 0.8(0.462))$$
$$= \tanh(1.0 + 0.3696) = \tanh(1.3696) \approx 0.878$$

Step 3: $t = 3$

$$h_3 = \tanh(0.5(1.5) + 0.8(0.878))$$
$$= \tanh(0.75 + 0.702) = \tanh(1.452) \approx 0.896$$

6 What does the hidden state mean?

Time	Hidden state
h_1	Remembers x_1
h_2	Remembers x_1, x_2
h_3	Remembers entire sequence

➡️ Hidden state is a **compressed memory** of the past.

Term	Meaning in your project
Sample	One window (one training example)
Sequence	Ordered time steps inside that sample
Time step	One CNN feature vector
Feature	Learned pattern value

- Recurrence unit : the rnn utilised recurrence condition i.e the output of the hidden state t-1 is the feed as input of the current state t , and by doing so the rnn captures temporal dependencies and patterns within sequences.

- It like a loop , The same network repeats (recurs) at every time step , The same weights are reused across time and Only the hidden state changes

4 Numeric example (simple recurrence)

Parameters:

$$W_x = 0.6, \quad W_h = 0.7, \quad b = 0$$

Sequence:

$$x = [1,\ 0.5,\ 2]$$

Initial state:

$$h_0 = 0$$

Step-by-step recurrence

$t = 1$

$$h_1 = \tanh(0.6(1) + 0.7(0)) = \tanh(0.6) = 0.537$$

$t = 2$

$$h_2 = \tanh(0.6(0.5) + 0.7(0.537))$$
$$= \tanh(0.3 + 0.376) = \tanh(0.676) = 0.589$$

$t = 3$

$$h_3 = \tanh(0.6(2) + 0.7(0.589))$$
$$= \tanh(1.2 + 0.412) = \tanh(1.612) = 0.923$$

3 Mathematical definition of recurrence

$$h_t = f(x_t, h_{t-1})$$

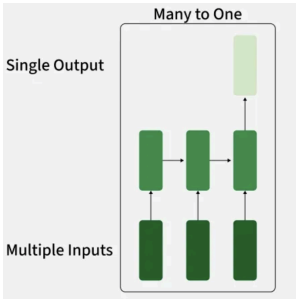
Where:

- x_t = input at time step t
- h_{t-1} = hidden state from previous step
- h_t = updated hidden state

📌 The dependence on h_{t-1} is what makes the model *recurrent*.

Property	Explanation
Same weights	W_x, W_h reused
Feedback loop	$h_{t-1} \rightarrow h_t$
Temporal dependency	Past affects future

- this project is many-to-one the output is either even/no event



.

- The vanishing gradient problem occurs in deep neural networks when gradients used for weight updates shrink exponentially during backpropagation, rendering early layers unable to learn

LSTM

- LSTM : designed to remember important patterns over long sequences
- Forget gate : decides which parts of the previous cell state (C□-1) should be forgotten. It takes the current input (x□) and the previous hidden state (h□-1) to produce a value between 0 and 1 for each unit . If f□ is close to 0, the information is mostly forgotten; close to 1 means it's retained. And it help to preserve gradient .this influence the input gate

3 Mathematical definition

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

Where:

- f_t = forget gate vector
- h_{t-1} = previous hidden state
- x_t = current input
- σ = sigmoid activation → outputs values in $[0, 1]$

Interpretation:

- $f_t = 1$ → keep memory
- $f_t = 0$ → forget memory

5 How forget gate acts on memory

$$\boxed{c_t = f_t \odot c_{t-1} + (\text{new information})}$$

The symbol $\odot$ = element-wise multiplication.

6 Numeric example (very important)

Assume:

- Previous cell state:

$$c_{t-1} = [2.0,\ -1.0,\ 0.5]$$

Forget gate output:

$$f_t = [0.9,\ 0.1,\ 0.5]$$

Apply forget gate:

$$f_t \odot c_{t-1} = [1.8,\ -0.1,\ 0.25]$$

Interpretation:

- Keep **90%** of first memory
- Almost forget second memory
- Keep half of third memory

- Input gate : determines what new information from the current input and previous hidden state is stored in the cell state. Using a sigmoid layer (0 to 1) to filter information and a tanh layer to create candidate values, it selectively updates the memory, preventing irrelevant data from impacting long-term storage.
- Output gate : deciding what part of the current cell state should be sent as the hidden state (output) for this time step, This hidden state h^t is then passed to the next time step and can also be used for generating the output of the network
- How much of the LSTM's internal memory (cell state) should be exposed as the hidden state (output) at the current time step.
- Needed as the cell state stored all the memory it should not be fully revealed so output gate is necessary
- In this project the output gate Revealing event-related temporal patterns and Suppressing irrelevant background information and Producing a clean hidden state for classification..

2 Mathematical equations

Output gate

$$\boxed{o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)}$$

Hidden state

$$\boxed{h_t = o_t \odot \tanh(c_t)}$$

Where:

- σ → sigmoid (0 to 1)
- $\tanh$ → squashes memory to $[-1, 1]$

4 Numeric example (using previous cell state)

From earlier steps, we had:

Cell state

$$c_t = [2.15,\ 0.06,\ -0.11]$$

Apply tanh

$$\tanh(c_t) \approx [0.973,\ 0.060,\ -0.110]$$

Output gate values

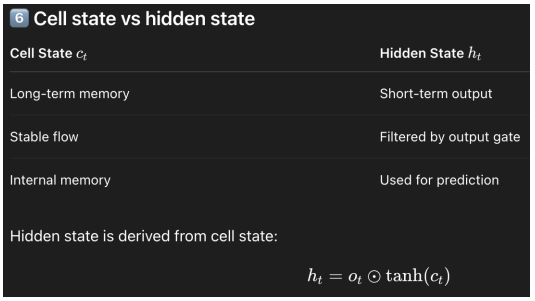
$$o_t = [0.6,\ 0.9,\ 0.2]$$

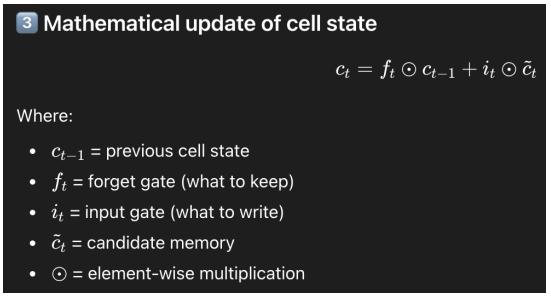
Compute hidden state

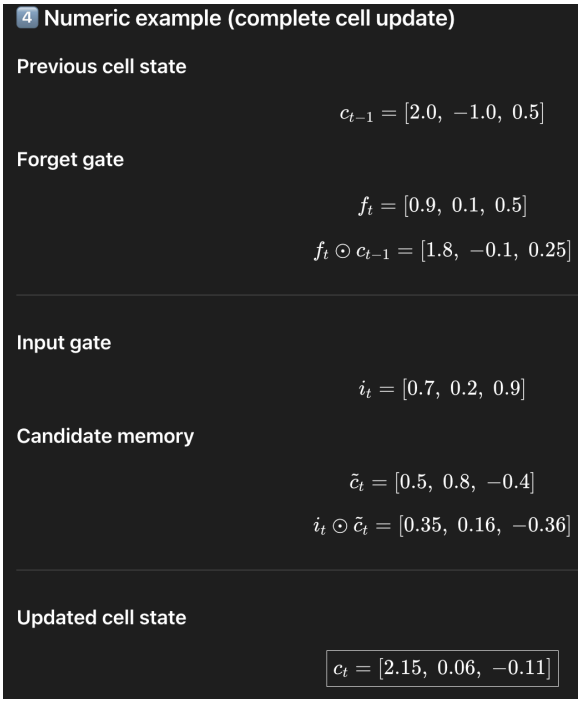
$$h_t = o_t \odot \tanh(c_t)$$
$$= [0.6 \cdot 0.973,\ 0.9 \cdot 0.060,\ 0.2 \cdot (-0.110)]$$
$$= [0.584,\ 0.054,\ -0.022]$$

- Cell state: acts as a conveyor belt for long-term memory, carrying relevant information across time steps with minimal interaction. It enables the network to

remember distant past information, mitigating vanishing gradient problems, and is updated by forget and input gates to add or remove information.

- 



- 

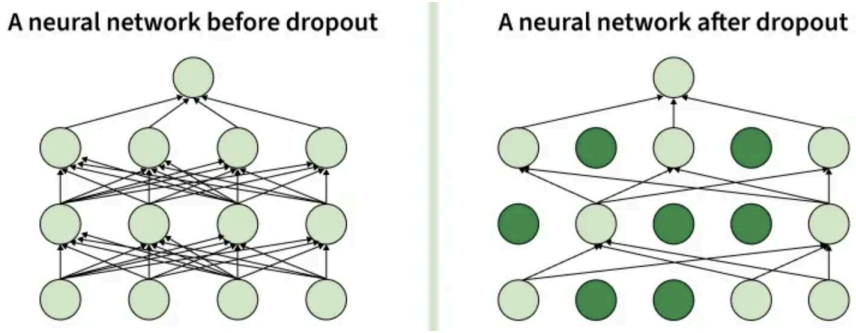
- Time series : A sequence of measurements collected in time order, where each value depends on previous values. Examples: Sensor readings over time
- Raw data is converted to widow by sliding window so each sample become “One window = one time-series segment”
- (samples, time_steps, channels)before cnn => (samples, reduced_time_steps, feature_maps) after cnn + pooling .
- In this project, the LSTM processes time-series segments formed by sliding windows over sensor data, modeling temporal dependencies across learned CNN feature sequences to perform binary classification.
- Temporal dependencies : situations where the current state, event, or data point is influenced by or dependent on previous ones, forming sequential patterns over time.
- {The temporal dependencies in this project arise from events that develop, persist, and decay over time, requiring the model to learn relationships across consecutive sensor measurements rather than isolated time steps.}
- Therefore it means one spike is not event , a consistent sequence of spike is event

Regularization technique

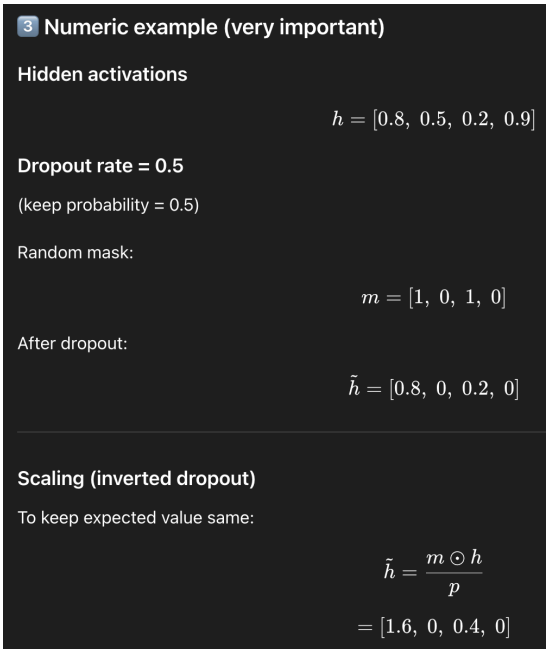
- Regularization introduces a penalty term to the loss function, discouraging the model from fitting the training data too closely and promoting simpler or more regular patterns in the learned parameters. This helps prevent the model from capturing

noise or irrelevant details in the training data, leading to better performance on new, unseen data

- Dropout: randomly deactivating a portion of neurons during training hence forcing the model to learn more robust and independent features. And prevent from over relying on specific neurons



- In this project dropout is used after convolution layer to Regularizes feature maps and Prevents CNN filters from over-specializing, and dropout → applied to input connections, recurrent_dropout → applied to recurrent (hidden-to-hidden) connections
- During training Random neurons are dropped and Different subnetworks are trained each batch . During testing all neurons are active, and weights are automatically scaled therefore no dropout is applied during inference.



- internal covariate shift (“distribution of activations (outputs of neurons) changes during the training process due to updates in network parameters. This shift means that each layer in the network has to continuously adapt to the changing distributions of inputs it receives from the previous layers. This phenomenon can slow down the training process and make it more difficult for the model to converge”
- Batch normalisation : is used to reduce the problem of internal covariate shift, It works by normalizing the data within each mini-batch. This means it calculates the mean and variance of data in a batch and then adjusts the values so that they have similar range. After that it scales and shifts the values so that the model learns effectively and learns stability .

3 BatchNorm formula (core math)

Given activations x in a mini-batch:

Step 1: Mean

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$$

Step 2: Variance

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

Step 3: Normalize

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

Step 4: Scale & shift

$$\boxed{y_i = \gamma \hat{x}_i + \beta}$$

- γ, β are **learnable**
- ϵ prevents division by zero

4 Numeric example

Mini-batch:

$$x = [2,\ 4,\ 6]$$

Mean

$$\mu = 4$$

Variance

$$\sigma^2 = \frac{(2-4)^2 + (4-4)^2 + (6-4)^2}{3} = \frac{8}{3}$$

Normalize

$$\hat{x} = [-1.22,\ 0,\ 1.22]$$

Scale & shift (assume $\gamma = 1, \beta = 0$)

$$y = [-1.22,\ 0,\ 1.22]$$

5 Where BatchNorm is applied

$$\text{Linear} \rightarrow \text{BatchNorm} \rightarrow \text{Activation}$$

Not after activation (usually).

-

-

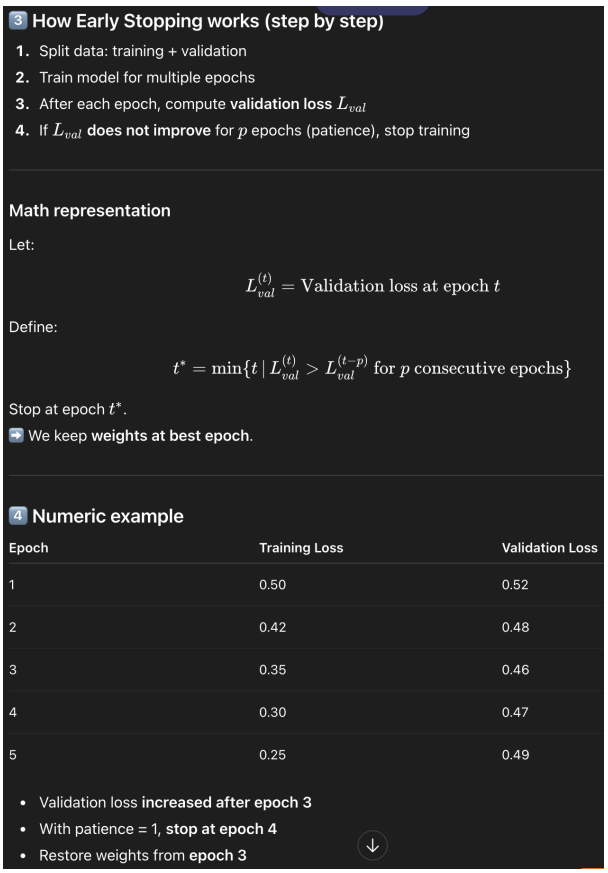
6 BatchNorm vs Dropout

Aspect	BatchNorm	Dropout
Purpose	Stabilize training	Prevent overfitting
Uses noise	❌ No	✅ Yes
Improves gradients	✅ Yes	❌ No
Training speed	Faster	Slower

➡️ Often **BatchNorm replaces Dropout** in deep networks.

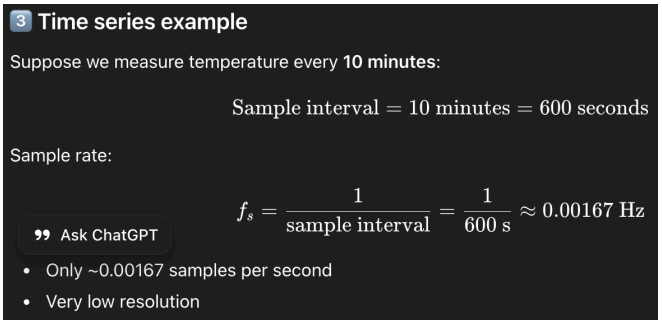
-

- Used after cnn to Normalizes feature maps and Stabilizes learning across subjects. Not used in lstm cause it tricky and the used inn dense layer to Ensures inputs to classifier are normalized
- Early stopping: technique that stops model training when overfitting signs appear. prevents the model from performing well on the training set when underperforming on unseen data i.e validation set (Training stops when performance improves on the training set but degrades on the validation set,)
- Monitors both training and validation data , If the validation performance worsens, training stops and the model retains the best weights from the period of optimal validation performance.
- Hyperparameter inn early stopping : Patience: The number of epochs to wait for validation improvement before stopping, typically between 5 to 10 epochs. Monitor Metric: The metric to track during training, often validation loss or validation accuracy. Restore Best Weights: After stopping, the model reverts to the weights from the epoch with the best validation performance.

- 

Time series analysis

- Sample rate : The number of data points collected per second from a continuous signal. Example: `50 Hz` → 50 data points per second. In **time series data**, it defines **temporal resolution**.
- raw signals are collected at a fixed sample rate , then used in window sliding and step and also in temporal smoothing.
- Sample rate is not used as a model parameter — it's used **pre-model** to define how sequences are sliced, how long each segment is, and how smoothing is applied. Without knowing the sample rate, your sliding windows and smoothing would not correspond to real time.
- It also Determines **sequence length** per time unit in lstm



- Sample rate determines how often you measure your time series. It controls resolution, sequence length, and the ability of models like LSTM to detect patterns.
- Sliding window : maintains a range (or "window") that moves step-by-step through the data, updating results incrementally. The main idea is to use the results of

previous window to do computations for the next window .

3 How it works (numeric example)

Time series:

$$X = [10, 12, 11, 13, 15]$$

Window size = 3

Stride = 1 (window moves 1 step at a time)

Step 1: Create sequences

Input (X)	Target (y)
[10, 12, 11]	13
[12, 11, 13]	15

- Window moves → input sequences overlap
- Each sequence length = window size
- Target = value after the window

Step 2: Represent as LSTM input

- Shape for LSTM: **(num_sequences, sequence_length, features)**

For univariate series:

$$X_{LSTM} = \begin{bmatrix} [10, 12, 11] \\ [12, 11, 13] \end{bmatrix}, \quad y = [13, 15]$$

- num_sequences = 2
- sequence_length = 3
- features = 1

- Convert continuous time series into fixed-length samples for training. And Captures temporal dependencies for the model and Makes sure each segment has enough context to train on with.

2. How It Works

1. **Window Size (W):**
   - Duration of each segment in number of samples.
   - Depends on **sample rate**:

$$W = \text{desired duration (s)} \times \text{sample rate (Hz)}$$

   Example: 0.5 s window at 100 Hz → 50 samples per window.
2. **Stride (S):**
   - How much the window moves forward each time.
   - Smaller stride → more overlapping windows → more training data.
   - Larger stride → less overlap → fewer windows, less redundancy.
3. **Sequence Generation:**
   - Move the window across your dataset from start to end.
   - Each window becomes one **input sample** for the CNN/LSTM.

-

3. Example

- Sample rate = 100 Hz (100 samples per second)
- Window size = 0.5 s → 50 samples
- Stride = 0.1 s → 10 samples

Sliding the window over the data:

Window #	Samples Used
1	0–49
2	10–59
3	20–69

Each window is fed to the model as a **sequence**.

- Temporal ferature : Temporal features are features that capture information across time
- temporal features are created from sliding windows over the signals, **Sliding window segments** generate sequences of raw signal values., then Each window is a **temporal snapshot**: contains multiple time steps. And The CNN extracts **local temporal patterns** within the window (e.g., quick spikes or local changes). And The LSTM captures **longer-term temporal dependencies** across window
- It is used like Feed the CNN-LSTM model as input sequences. Then CNN learns **short-term temporal relationships** (like a “local pattern detector”). And LSTM learns **long-term dependencies** (how patterns evolve across time). Thererfore Model predicts target labels based on both **instantaneous patterns** and **temporal trends.**

✅ **Summary Table**

Concept	General Deep Learning	In This Project	Purpose
Temporal feature	Features dependent on time	Sliding window sequences of signal values	Capture order, trends, patterns
Short-term dependency	Local signal changes	CNN kernels on window	Detect local spikes/trends
Long-term dependency	Pattern over longer periods	LSTM hidden states	Track temporal evolution
Duration & frequency	How long events last / how often they occur	Derived from windowed sequences	Improve event detection accuracy

- create_windows ->takes your raw time-series and splits it into overlapping temporal sequences of length WINDOW_SIZE and Each sequence retains time order, so later processing can learn temporal patterns.
- `Conv1D` uses kernels that slide along the time axis → picks up local temporal patterns (e.g., quick rises, pulses). `MaxPooling1D` down-samples along time → preserves strongest patterns, reduces sequence length before LSTM.
- LSTM takes the sequence of CNN features and models how they evolve over time. It remembers patterns across time steps and passes processed temporal context to the classifier.

Temporal Mechanism	Implementation in Your Code
Sliding window creation	`mag_seq = create_windows(df, ...)` (before training)
Short-term temporal feature extraction	`Conv1D` + `MaxPooling1D` in model
Long-term temporal dependency modeling	`LSTM(...)` in model
Sequence shape preservation	`X_seq_train = ...reshape(...)` before feeding to model

Feature engineering : transforming raw data into meaningful representations to improve model performance

- There are 2 typed : hand crafted feature and automatic learning
- Statistical feature: is a summary of the numbers computed from raw data that describe its: Central tendency ,Dispersion, Shape ,Dynamics
- Its a Numerical summaries that describe the distribution of values within a window of data. They compress a time window into a few informative numbers.
- this is a common way of grouping data
- 4 Numeric example (sliding window)

  Ask ChatGPT

  $$X = [10,\ 12,\ 11,\ 13,\ 15]$$

  Window size = 3

  Window 1: [10, 12, 11]
  - Mean = 11
  - Std = 0.82
  - Min = 10
  - Max = 12

  Window 2: [12, 11, 13]
  - Mean = 12
  - Std = 0.82
  - Min = 11
  - Max = 13

  5 Table: Statistical features per window

Mean	Std	Min	Max	Range
11	0.82	10	12	2
12	0.82	11	13	2

  🔷 Common Statistical Features

Feature	Meaning
Mean	Average value
Standard deviation	Variability
Variance	Spread of values
Min / Max	Extreme values
RMS	Signal energy
Skewness	Asymmetry
Kurtosis	Peakedness

- In this project, statistical features are explicitly engineered here :

  `mag_seq` → **raw temporal sequences** (for CNN + LSTM)

  `mag_feat` → **statistical features from magnitude**

  `hum_feat` → **statistical features from humidity**

  `labels` → target class
- Magnitude statistical features (mag_feat) Likely include: Mean magnitude Standard deviation Min / max Energy or RMS Range (handcrafted features per sliding window)
- Humidity statistical features (hum_feat) Likely include: Mean humidity Variance Trend or difference Min / max (hand crafted feature per sliding window)
- Statistical features give the model **summary context by provide global information about a window**, while CNN-LSTM gives **fine temporal detail**.
- in this project, feature engineering is performed by extracting statistical features such as mean, variance, and energy from sliding windows of sensor data, while temporal features are automatically learned using CNN and LSTM layers, forming a hybrid feature learning approach.

- 🔷 **5. How These Features Are Used in the Model**

  You **combine statistical features** like this:

  ```python
  X_feat_train.append(np.hstack([mag_feat, hum_feat]))
  ```

  Then normalize:

  ```python
  X_feat_train = scaler_feat.fit_transform(X_feat_train)
  ```

  And feed them to the model:

  ```python
  model.fit(
      [X_seq_train, X_feat_train],
      y_train,
      ...
  )
  ```

- Temporal feature : A temporal feature is any feature that: Depends on time order , Captures patterns across multiple time steps , Cannot be understood from a single data point
- Temporal features in your project are the time-ordered sequences of sensor values extracted using sliding windows and learned automatically by the CNN and LSTM layers to capture how the signal evolves over time
- This “mag_seq” sequence is a **temporal feature** because: The order matters, The spacing is defined by sample rate and It represents signal behavior over time
- It depends on: Event duration, Order of signal changes ,Temporal continuity
- In this project, temporal features are the time-ordered sequences of sensor measurements extracted using sliding windows and automatically learned by CNN and LSTM layers to model short-term and long-term temporal dependencies

Feature Type	Example	Used Where
Temporal feature	Sequence of magnitude values	CNN + LSTM branch
Statistical feature	Mean, std, energy	Feature branch
Combined	Fused representation	Final classifier

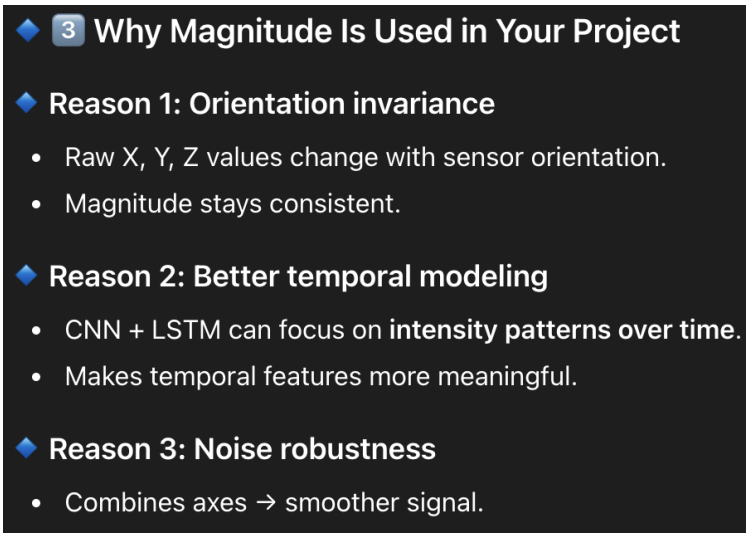
- Magnitude: A single scalar value that represents the **overall intensity** of a multi-axis sensor signal. If your sensor has multiple axes (e.g., X, Y, Z), magnitude combines them into **one meaningful value.**

  Mathematical form (common):

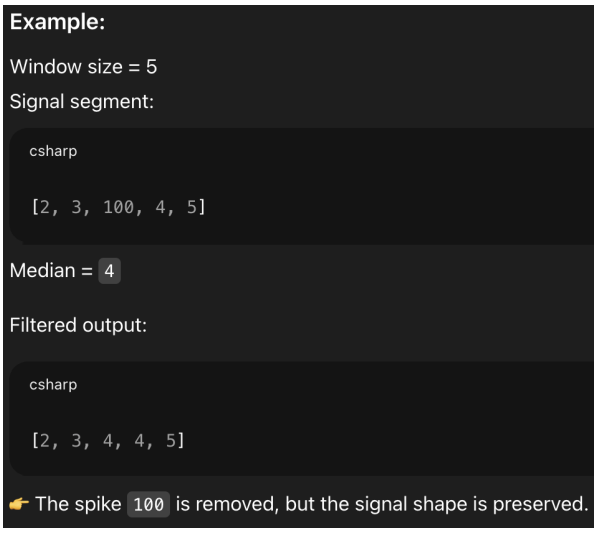
  $$\text{Magnitude} = \sqrt{x^2 + y^2 + z^2}$$

  This removes:

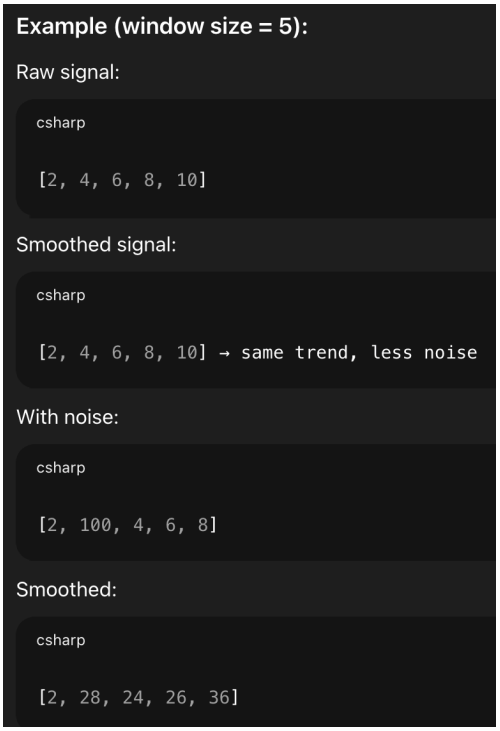
  - Orientation dependency
  - Axis-specific noise

- Magnitude is handled in two ways in my project. : As a Temporal Feature (Sequence)=> `mag_seq` is a sequence of magnitude values over time. Fed into CNN + LSTM.
  As Statistical Magnitude Features => These are summary statistics of magnitude within each window. Examples: Mean magnitude, Standard deviation, Energy / RMS, Min / max. This is your **hand-crafted magnitude feature**
- In this project, the magnitude feature represents the overall intensity of the sensor signal, derived by combining multi-axis measurements, and is used both as a time-series input to CNN-LSTM layers and as statistical features for classification.
- 

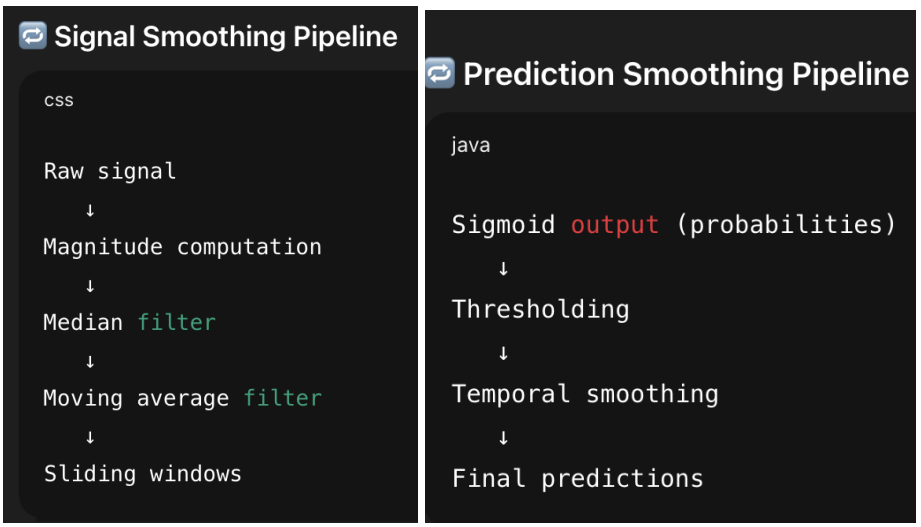
Signal processing

- Median Filter : **non-linear smoothing filter** used to remove **impulsive noise (spikes/outliers)** from a signal **without blurring edges**.
- 

- The median filter is applied during **preprocessing**, **before** feature extraction and model training , Raw sensor data → magnitude calculation, Median filter applied to magnitude time series, Result: smoother, cleaner temporal signal
- Therefore Remove sudden noise spikes an letting cnn and lstm learn correctly , Make sliding windows more stable, Improve CNN + LSTM learning
- Moving average filter : A **moving average (MA) filter** is a **linear smoothing filter** that reduces short-term fluctuations by averaging neighboring values.

- 

- **preprocessing**, **after median filtering** and **before sliding window creation**.
- Purpose is to Smooth **high-frequency noise,** Reduce jitter in magnitude signal, Stabilize temporal patterns, Improve window-based feature extraction.
- A moving average filter is a linear preprocessing technique used in this project to smooth high-frequency noise in the magnitude signal after median filtering and before sliding-window segmentation, improving temporal feature quality for CNN-LSTM learning.
- Smoothie technique : **it** is used to: Reduce noise and short-term fluctuations in data while preserving the underlying signal trend., it is a **preprocessing ad postprocessing** step in this project
- **TWO types of smoothing** in this project : Signal-Level Smoothing (Before the Model) => median filter and moving average . Prediction-Level Smoothing (After the Model) => temporal smoothing of predictions
- Temporal smoothing : A post-processing technique that enforces **time consistency** in model predictions by removing short, isolated prediction changes that are unlikely to represent real events. it Remove isolated false positives, Enforce minimum event duration, Improve F1-score.

- 

- Noise reduction : Noise reduction in this project is achieved through signal-level preprocessing using median and moving average filters, model-level regularization, and prediction-level temporal smoothing, ensuring robust temporal feature learning and improved generalization

.

Evaluation metrics

- Accuracy : evaluating the performance of a classification model. It tells us the proportion of correct predictions made by the model out of all predictions. But it can be misleading in cases of imbalanced datasets by giving a False Positive sense of achieving high accuracy.
- $$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$
- Precision: measures how many of the positive predictions made by the model are actually correct . helps ensure that when the model predicts a positive outcome, it's likely to be correct

  $$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

  Where:

  - TP = True Positives
  - FP = False Positives
- Recall/sensitivity : measures how many of the actual positive cases were correctly identified by the model. It is important not to miss a positive case

  $$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

  Where:

  - FN = False Negatives

- F1 score : harmonic mean of precision and recall. It is useful when we need a balance between precision and recall as it combines both into a single number. A high F1 score means the model performs well on both metrics.

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

- The project evaluates the model using **F1-score, precision, and recall** to handle class imbalance and temporal sequence predictions, with temporal smoothing and thresholding improving event-level performance.
- Confusion matrix : A confusion matrix is a table that summarizes **predictions vs. true labels**:

n=165	Predicted No	Predicted Yes
Actual No	50	10
Actual Yes	5	100

-
-

Validation-Only Threshold Selection

- We use validation only selection : since the model 0.5 threshold is often not optimal, especially with imbalanced datasets for imbalanced problems, you often want: Higher recall, Higher precision, Best F1, Custom tradeoff
- where the minority class hand washing has lower threshold. And Using the same data for training and thresholding leads to overly optimistic performance, which this method avoids.
- Converting Probabilities to Class Labels : Prediction < 0.5 = Class 0, Prediction >= 0.5 = Class 1
- Threshold-Moving for Imbalanced Classification : imbalanced classification problem, such as resampling the training dataset and developing customized version of machine learning algorithms. Solution : search a range of threshold values in order to find the best threshold.

  We can summarize this procedure below.

  - 1. Fit Model on the Training Dataset.
  - 2. Predict Probabilities on the Test Dataset.
  - 3. For each threshold in Thresholds:
    - 3a. Convert probabilities to Class Labels using the threshold.
    - 3b. Evaluate Class Labels.
    - 3c. If Score is Better than Best Score.
      - 3ci. Adopt Threshold.
    - 4. Use Adopted Threshold When Making Class Predictions on New Data.

- Optimal Threshold for ROC Curve : A ROC curve is a diagnostic plot that evaluates a set of probability predictions made by a model on a test dataset. A set of different thresholds are used to interpret the true positive rate and the false positive rate of the

predictions on the positive (minority) class, and the scores are plotted in a line of increasing thresholds to create a curve. The false-positive rate is plotted on the x-axis and the true positive rate is plotted on the y-axis and the plot is referred to as the Receiver Operating Characteristic curve, or ROC curve. A diagonal line on the plot from the bottom-left to top-right indicates the “curve” for a no-skill classifier (predicts the majority class in all cases), and a point in the top left of the plot indicates a model with perfect skill. The curve is useful to understand the trade-off in the true-positive rate and false-positive rate for different thresholds. The area under the ROC Curve, so-called ROC AUC, provides a single number to summarize the performance of a model in terms of its ROC Curve with a value between 0.5 (no-skill) and 1.0 (perfect skill). The ROC Curve is a useful diagnostic tool for understanding the trade-off for different thresholds and the ROC AUC provides a useful number for comparing models based on their general capabilities.

- Optimal Threshold for Precision-Recall Curve : Validation-only threshold selection using a precision-recall (PR) curve involves using a held-out validation dataset—not the training data or final test data—to identify the optimal probability threshold for a binary classifier. This process maximizes model performance (e.g., F1-score) for specific business requirements by balancing precision and recall.
- validation only precision recall if done in validation data set where th model predict the label of positive true minor class by predict 1 if probability > τ and compute precision and recall; F1 = 2·P·R/(P+R). Pick τ that gives maximum F1 on **validation**. therefore test data set is only used once for the evaluation and accuracy checking of the model by comparing the validation only threshold selection using the precision recall curve .
- If you later: Retrain model on train + validation combined, Then you must:
    1) Either re-select threshold via cross-validation,
    2) freeze the threshold from earlier validation (But if you keep original split, you're perfectly fine)
- performed after model training and hyperparameter tuning, but before final testing . The model makes predictions on a validation set, producing probability scores rather than hard class labels
- During Evaluation: The PR curve is calculated by scanning through various potential threshold values (0.0 to 1.0) on the validation set, and the threshold that maximizes a specific metric (often F1-score or a tailored cost-sensitive metric) is selected.
- the Threshold is treated like hyperparameter

2️⃣ **On validation set**

You:

- Compute probabilities $p = P(y = 1|x)$
- Try different thresholds τ
- Convert probabilities to labels:

$$\hat{y} = \begin{cases} 1 & \text{if } p \geq \tau \\ 0 & \text{otherwise} \end{cases}$$

- Compute:

$$Precision(\tau)$$

$$Recall(\tau)$$

$$F1(\tau) = \frac{2PR}{P+R}$$

- Select:

$$\tau^* = \arg\max_{\tau} F1(\tau)$$

- In A paper, they propose a secure and efficient threshold-based event validation protocol for MH-relevant applications. We convert probabilistic counting to threshold-based validation, and show that threshold-based validation schemes yield significant savings compared to just counting accurately and comparing to the threshold, because threshold-based validation schemes can output an accurate decision based on an inaccurate estimate.
- Most classifiers output **probabilities**:

$$\hat{y} = P(y = 1 \mid x)$$

Default decision rule:

$$\hat{y}_{class} = \begin{cases} 1 & \text{if } \hat{y} \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

But **0.5 is arbitrary**.

Fair_f1 fixed and fair_f1

1. Fair F1 (fixed 0.5):Median filter + threshold 0.5 + min duration. No data-driven threshold; directly comparable to papers that use a fixed rule. 2. Reproducible for research paper ; Fixed 0.5 → “How good is the model under standard rule?”

$$\hat{y} = \begin{cases} 1 & p \geq 0.5 \\ 0 & p < 0.5 \end{cases}$$

2. Fair F1 (validation-chosen threshold): Same pipeline, but threshold = value that maximized F1 on validation. Still no use of test for threshold choice. ; Validation threshold → “How good is the model when optimally tuned?”

Pipeline:

- Model → probabilities
- Median filter
- Choose τ^* on **validation**:

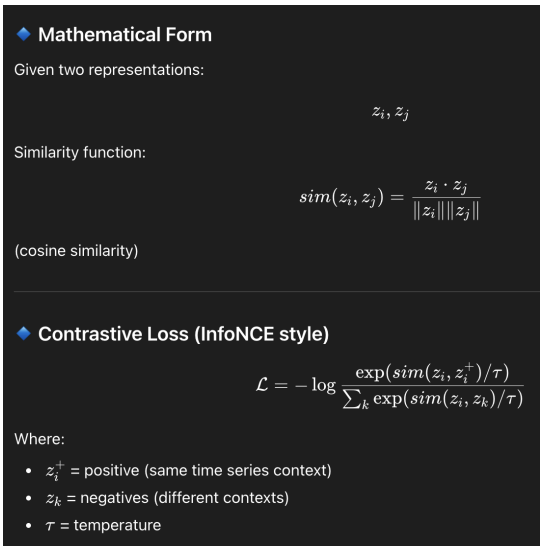
$$\tau^* = \arg\max_{\tau} F1_{val}(\tau)$$

- Apply same τ^* on test
- Evaluate once

- Predictions come from a **fixed** rule (0.5) or a threshold chosen only on **validation**; test is used only to evaluate.
-
-

Self-supervised learning

- SSL, a subset of unsupervised learning, aims to learn discriminative features from unlabeled data without relying on human-annotated labels.
- So it used pseudo task (:where a model is trained on unlabeled data by creating its own artificial labels ("pseudo-labels") to learn useful representations) as data
- Example :
    1) Predict masked part: "mask" a portion of the input data (text, image, or 3D cloud) and force the network to reconstruct the missing information based on the context of the visible, unmasked parts.
    2) Match two augmented views : aim to generate pseudo-labels for unlabeled data by enforcing consistency between different transformations (flipping , cropping , augmenting) of the same image.
-
- Ssl Contrastive learning: minimise the distance between positive sample pair(similar samples) and increase the distance between the negative paris(different samples) by distinguishing between similar and different samples or objects.
- Washing motion windows → similar embeddings; Positive pair: Two augmented versions of the same washing window
- Random motion windows → different embeddings ; Washing window vs random motion window
-

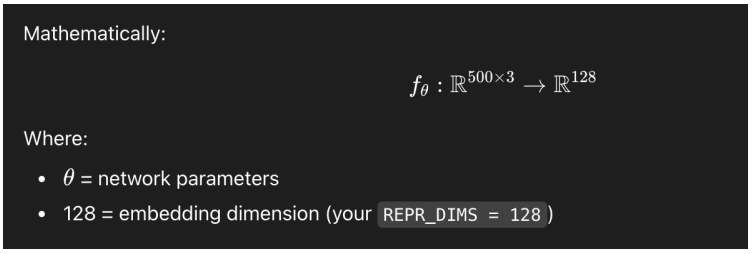


FLOW

- In my model i have used dense 128, dense 64, dense 1 (in the output), therefore the model does matric multiplication to complicate and help the model to learn a new pattern.
- Formula : y=Wx+b
  W= weight matrix
  x= input vector

b= bias

- R^d means a d-dimensional space of real numbers.
- Therefore inn dense layer **x ∈ R^d (**That means a 1D feature vector, not a 2D time series.**)** → **R^(500*3)**; window size=500 , channel = 3; (its a 2d matrix) cant handle → convert **R^128** That 128-dimensional vector is the embedding.
- Encoder :a fundamental component in machine learning and data processing that transforms raw input data into a compressed, high-level representation, often referred to as a latent space or latent vector. This process is crucial for reducing data dimensionality, extracting meaningful features, and filtering out noise

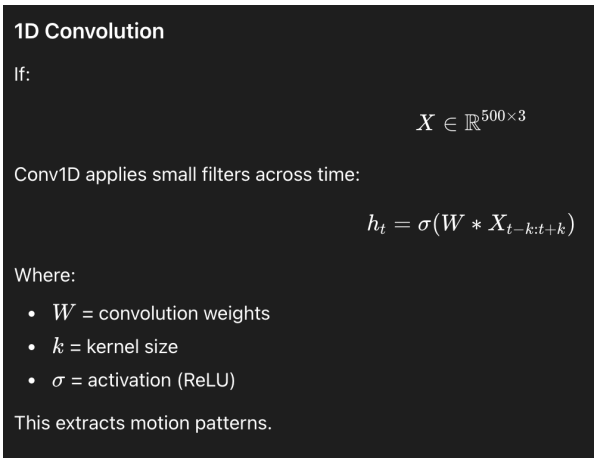
Example: 

fθ(X)=z

Temporal convolutional network

- a specialized deep learning architecture designed for sequence modeling, such as time-series forecasting, audio synthesis, and action detection.
- they use causal, dilated 1D convolutional layers to capture long-range dependencies, offering superior parallelization, lower memory usage during training, and more stable gradients compared to traditional Recurrent Neural Networks (RNNs) like LSTMs or GRUs.

T2Vec (time series to vector) (SSL-TCN)

- a state-of-the-art, universal framework for learning unsupervised representations of time series data. designed to learn high-quality, fine-grained rep at the timestamp level that are effective for various downstream tasks, including time series classification, forecasting, and anomaly detection.
- Conv1D → Conv1D → Conv1D → ... → pooling → vector
- 

- Pretraining no labels used , It learns by contrastive learning. Using raw imu data . the main object of pretraining is to have minimum contrastive loss.

- You have:

  $$X_{all} \in \mathbb{R}^{N \times 500 \times 3}$$

  Where:

  - N = total number of IMU windows
  - 500 = time steps
  - 3 = accelerometer channels

- ## 🔷3️⃣ What TS2Vec Learns During Pretraining

  Let encoder be:

  $$z = f_\theta(X)$$

  TS2Vec learns parameters θ such that:

  If two windows represent similar motion:

  $$f_\theta(X_i) \approx f_\theta(X_j)$$

  If motions are different:

  $$f_\theta(X_i) \text{ far from } f_\theta(X_k)$$

  So it learns **motion structure**, not labels.

- It tries to make:

  $$f_\theta(X^{(1)}) \approx f_\theta(X^{(2)})$$

  If both come from same window.

  Loss (simplified):

  $$\mathcal{L} = -\log \frac{\exp(sim(z_i, z_i^+)/\tau)}{\sum_j \exp(sim(z_i, z_j)/\tau)}$$

  Don't worry about formula — concept is:

  > Similar motion → similar vector
  > Different motion → different vector

- Pretraining objective:

  $$\min_\theta \mathcal{L}_{contrastive}$$

  After training:

  Encoder fixed:

  $$z_i = f_\theta(X_i)$$

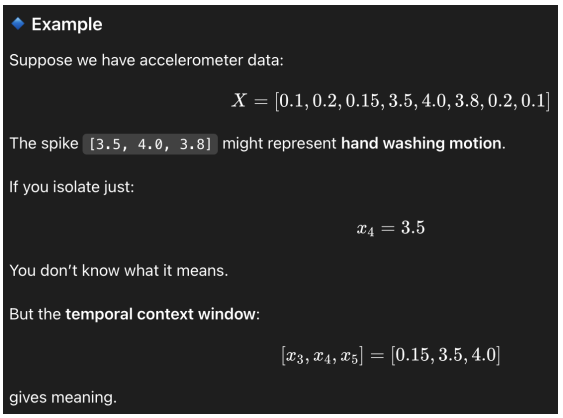
  Then classifier:

  $$\hat{y} = \sigma(Wz + b)$$

  So total training is separated:

  Stage 1 (SSL): learn good f_θ
  Stage 2 (supervised): learn decision boundary

- Encoder weights are frozen and Only classifier is trained per fold.
- The encoder learns the parameter θ and that defines z=fθ(X)
- Why Freeze It? Because: Pretraining stage already learned general motion features. During LOSO folds, we want: Fair evaluation Avoid overfitting per subject Keep representation consistent across folds If we did NOT freeze: Each fold would slightly change the encoder → Representations would be different per fold → Harder to compare fairly.
- TS2Vec internally produces timestamp-level features: H∈R^(n×500×128)
- It performs pooling across time dimension. zi=Pooling(Hi) ;l final output Z∈R^(n×128)
- **encoding_window='full_series'** => Because your window already represents one activity segment (500 samples). So i want: One embedding per window. Not one embedding per timestamp. So: (500 × 3) → (128)
-
- The embedding captures: Periodicity Motion, intensity ,Acceleration, rhythm Temporal consistency, Long-range, patterns. Not the classigfication of hand washing or not
- Without pretraining: Your MLP must learn everything from scratch. With pretraining: TS2Vec already understands: What smooth motion looks like What repetitive patterns look like What sudden spikes look like
-
- **vector embeddings** :is numerical rep of the text data in semantic meaning or context in a format that computers can process
- Temporal context:
    1) surrounding time information around a timestamp or segment
    2) the incorporation of time-related information—such as past behaviors, sequences, or specific time stamps—to enhance the interpretation, prediction, or recall of events, data, and communication



- It learns vector embeddings of time series without labels.

T2SVec architecture

- It uses dialect convolution layer (normal convolution overlapping is done and in the dilated cnn the skip connection is done)

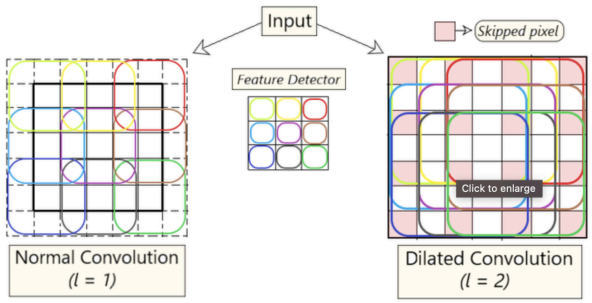
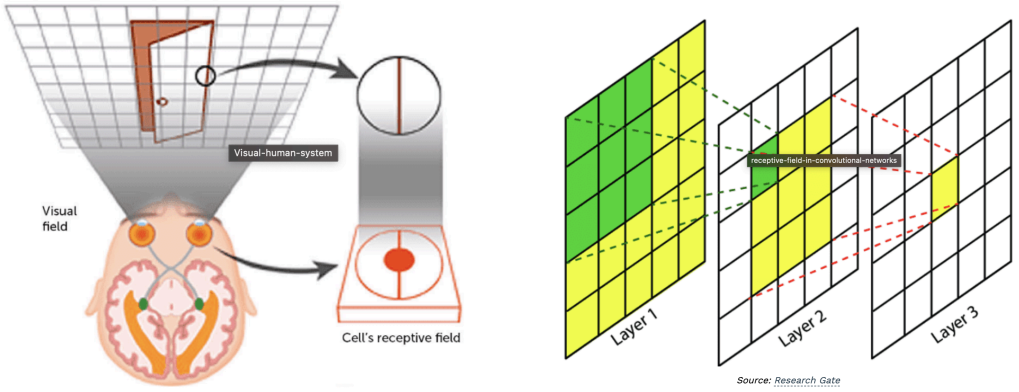
- 

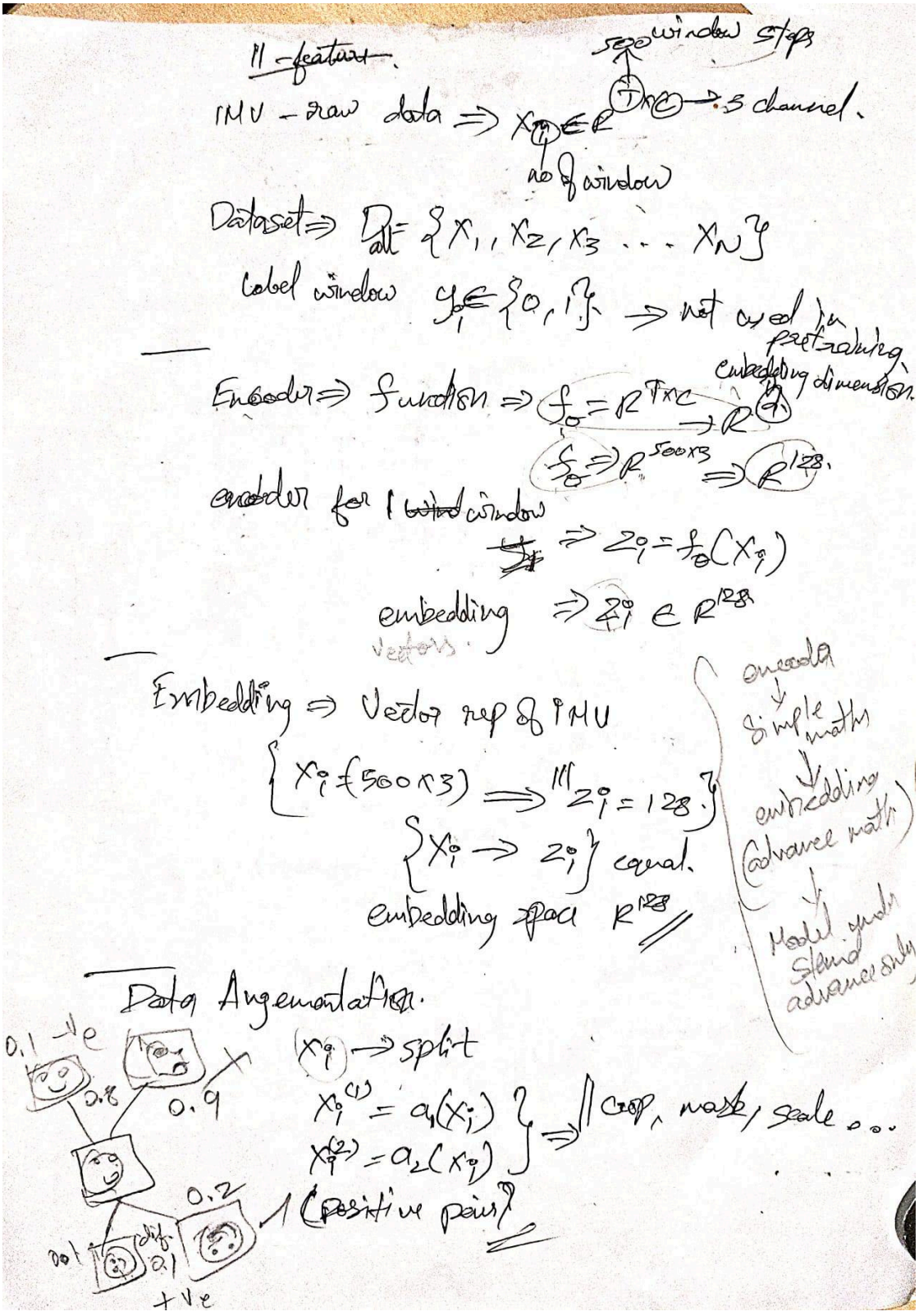
Fig 1: Normal Convolution vs Dilated Convolution
- Receptive field : The human visual system consists of millions of neurons, where each one captures different information. We define the neuron's receptive field as the patch of the total field of view. In other words, what information a single neuron has access to. That single cell is similar to receptive field and by increasing the size of receptive field as shown in the image , layer 3 covers a large area compared to layer 2 and layer 1 by using skip connection and increasing the field (optimization - dilated cnn) .
- 

Source: Research Gate

Inductive vs transductive

- Inductive: The model is trained only on the training set and is meant to generalize to any new, unseen data.
- Train only on the training subjects.
- The test subject is never used during training.
- Pure inductive learning
- Transductive: The model is trained using both labeled training data and unlabeled test data, and is optimized for predictions on that specific test set.
- TS2Vec is pretrained on all subjects, including the one that will be the test subject in each LOSO fold. That means: The TS2Vec encoder has seen the test subject's IMU data during pretraining (unsupervised). The encoder's representations are influenced by the test subject's distribution. The MLP classifier is still trained only on the other subjects (inductive). So the representation learning step is transductive (uses test data), while the classifier is inductive.

The entire code flow :

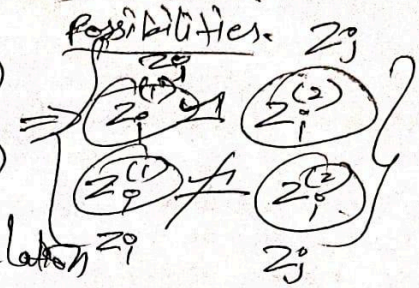


Pretrain Stage Contrastive learning

augmentation → passed → encoder.

$z_i^{(1)} = f_\theta(x_i^{(1)})$

$z_i^{(2)} = f_\theta(x_i^{(2)})$



Contrastive loss calculation

$\Downarrow$

$sim(z_i, z_j) = \frac{(z_i) \cdot (z_j)}{\|z_i\| \|z_j\|}$ (similarity identify)

$\Downarrow$

$L_i = -\log \frac{\exp(sim(z_i^{(1)}, z_i^{(2)})/t)}{\sum_{k=1}^{N} \exp(sim(z_i^{(1)}, z_k^{(2)})/t)}$

temperature

positive pair

positive + Negative pairs

$\Downarrow$

$L_{SSL} = \sum_i L_i$

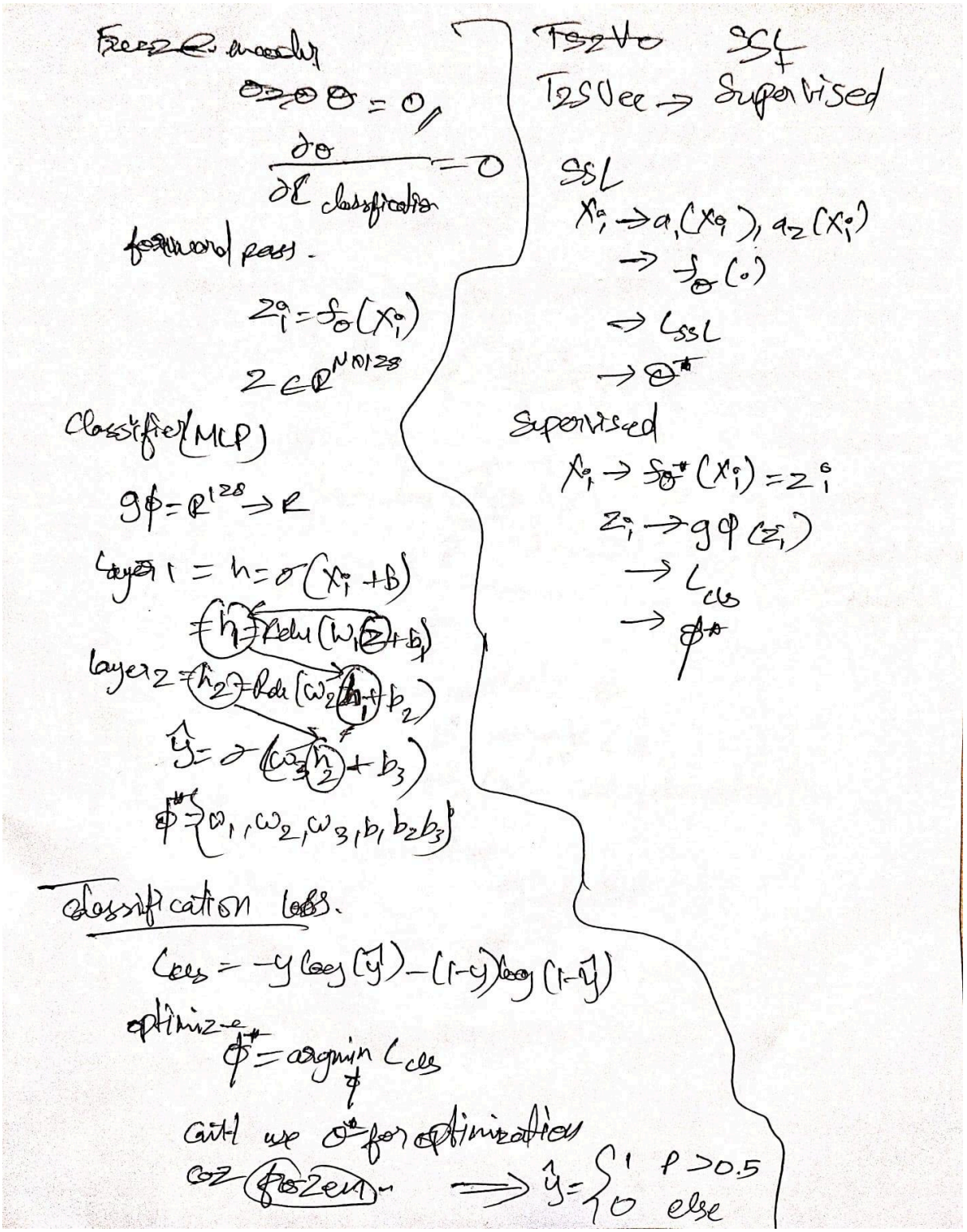
optimize ⇒ update θ

$\theta^* = \arg\min_\theta L_{SS}$

$\theta^* =$ (min Contrastive loss)

Transductive Vs Inductive

Transductive	Inductive
$\theta^* = \arg\min_\theta L_{SS}(D_{all})$	$\phi^* = \arg\min_\phi L_{SS}(D_{train})$
pretraining	~~Position~~ Decision classifier.



Multimodal fusion

- s a technique for integrating information from various modalities(perspective) , such as text, images, audio, and video, to enhance model performance and generalization.

Why Not Train Deep Network for Humidity?

- Because humidity signal may be:

- Low complexity Slow-changing
- Well summarized by statistics So handcrafted features are sufficient.

- Multimodal fusion does NOT mean humidity becomes the major feature. It means: The classifier is allowed to use both modalities. If performance improves, it shows that humidity adds useful complementary information. The model itself decides how much importance to assign to humidity via learned weights.
- Incorporating humidity features significantly improves F1-score (+9.6 points), indicating better detection of washing events. This suggests that humidity provides complementary information beyond motion dynamics captured by TS2Vec embeddings.

HYPER PARAMETERS
Sample rate :

1) https://www.sciencedirect.com/science/article/pii/S1746809425015460

- Included in sensor data to capture per second/sample
- Higher the sample meas more accurate data point (more detail)
- Higher sampling frequency for consistently improving classification accuracy

Window duration
Window size
Step size
OBSERVATION WINDOW

- Parameters : WINDOW_DURATION, WINDOW_SIZE, STEP_SIZE = 75
- Given these last 3 seconds of data, is this window handwashing or not?"
- sliding-window setup is **doing exactly what it should** for handwashing detection: each 3-second window is observed and classified, no need for a future prediction window.
-

Key Takeaways

1) https://ieeexplore.ieee.org/abstract/document/9006304

- Yes, your current pipeline **can implement ideas from the attack detection paper**:
    - Variable / multiple window durations
    - Feature evaluation per window length
    - Optimizing window size for performance (F1)
    - Window stride tuning
- This could **improve detection** and give better temporal resolution for handwashing events