AN2DL - First Challenge Report
The Big Batch Theory

Benedetta Mussini, Andrea Rossi, Fabio Rossi, Francesco Sarra

benedettamussini, Redsss, poliReus, fsarra

278186, 287278, 286400, 279755

November 17, 2025

1 Introduction

This report details the methodology and results for the First challenge of the AN2DL course of the group **TheBigBatchTheory** by team members *Benedetta Mussini, Andrea Rossi, Fabio Rossi, and Francesco Sarra.*

The project's objective was to address a multi-class, time-series classification task: predicting a subject's true pain status - `no_pain, low_pain, or high_pain` - based on sensor data.

To solve this, we built a full deep learning setup. We spent a lot of time preparing the data, creating useful features, and designing a recurrent neural network (RNN) to make predictions. The final solution employs a 5-fold cross-validation strategy and an ensemble of the resulting models to generate the final predictions.

2 Problem Analysis

The dataset consists of time-series data from multiple subjects, identified by a `sample_index`. The feature set is a mix of numerical sensor readings (`joint_00`, ..., `joint_30`) and categorical descriptors (e.g., `n_legs`, `pain_survey_1`).

Our initial data profiling and exploration (using tools like *ydata-profiling* and correlation analysis) revealed some potential challenges:

- subject-dependent data (risk of data leakage),
- high correlation in `joint_` features,
- irrelevant/noisy features,
- class imbalance.

Our early tests showed right away that overfitting was a big problem. The models did almost perfectly on the training data but performed poorly on new data. So, preventing overfitting became the main focus of our project.

3 Method

Data Preprocessing

Preprocessing includes mapping categorical attributes to numeric values, one-hot encoding `pain_survey` features, removing zero-variance and weakly correlated `joint` signals, and applying PCA to reduce redundancy among `joint` groups. Data is split by subject (`sample_index`) to avoid leakage, and, while we experimented with StandardScaler, we found that a fold-specific Min-Max scaler provided more stable results for our final models. Time-series sequences are generated through a sliding-window strategy with stride and zero-padding, converting each subject's data into fixed-length inputs. We systematically explored various combinations of window_size and stride, as this proved to be the most impactful factor in controlling the trade-off between data volume and data redundancy.

Model Architecture and Training

We defined a flexible `RecurrentClassifier` class (supporting *RNN*, *LSTM* and *GRU* layers). The core of this model is a multi-layer recurrent module with dropout for regularization, followed by a single `Linear` layer for classification.

The model is trained using an AdamW optimizer (which includes L2 regularization) and a CrossEntropyLoss. This loss function was enhanced with class weights to combat dataset imbalance and label smoothing to prevent overfitting. In fact, while class weights alone showed mixed results, the combination of it with label smoothing, and others regularization technique effectively constrained our models and forced them to generalize.

Evaluation and Ensemble-Based Inference

Our evaluation process began with a grid search over a set of hyperparameters. We used *Tensorboard* to compare models, selecting for a high validation `F1` score while ensuring the training error did not drop to zero, thereby avoiding overfitted solutions.

For robust validation, we ensembled the models from a *K-fold cross-validation*. During inference, we averaged the logits from all K models, with each model predicting on test data normalized by its own fold's scaler. The final label for each subject (`sample_index`) was determined by a majority vote of the argmax predictions from these averaged-logit windows.

4 Experiments

All experiments were conducted with a fixed random seed (`SEED=42`) for reproducibility.

Our experimental process began with simpler models and incrementally explored the hyperparameter space, using results from one set of experiments to guide the next. This allowed us to manually narrow down the most impactful parameters and refine our model's architecture.

In our experiments, we kept trying to find the right balance between how big our model was and how much regularization we used. The biggest problem from the start was overfitting. Even the simplest models got almost perfect accuracy on the training data, but the validation results barely improved, so we knew the model was just memorizing.

A lot of effort went into choosing the right `window_size` and `stride`. These settings control how much of the signal the model sees at once. We learned that bigger windows gave more context, but if the stride was too small, the samples overlapped too much and the model memorized instead of learning. After many tests, we found that `window_size` = 25 and `stride` = 2 gave the best balance.

Once we had the data prepared correctly, we tried different model sizes and regularization strengths. Bigger models could work well, but only when we added enough regularization like `dropout` or `l2_lambda`. Smaller models, on the other hand, didn't overfit as much because they naturally couldn't memorize everything.

In the end, we discovered that there wasn't just one best model. We actually found two different approaches that both performed really well: a larger LSTM model with strong regularization, and a smaller, simpler RNN that stayed stable because of its limited size. Both reached similar results, just in different ways.

The performed grid search explored a subset of combinations of the following parameters:

Parameter	**Values**
rnn_type	RNN, LSTM, GRU
bidirection	True, False
hidden layers	1, 2, 3, 4
hidden size	64, 128, 192, 256
window size	10, 15, 20, 24, 25, 30
stride	2, 5
batch size	32, 64, 128, 256, 512
learning rate	1e-3, 5e-4, 1e-4
dropout rate	0.2, 0.3
l1_lambda	0, 1e-6
l2_lambda	0, 1.5e-3, 1e-4, 5e-5, 1e-5

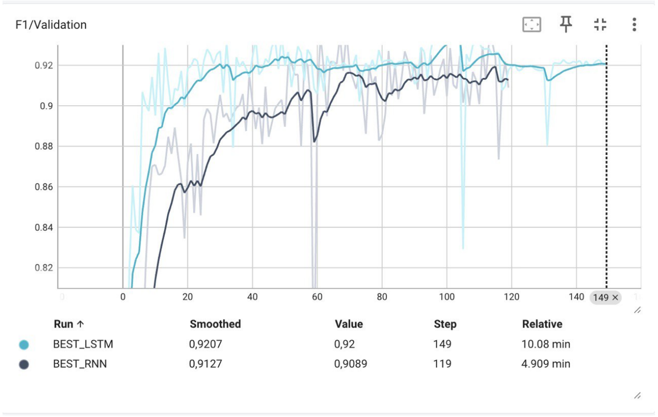
5 Results

This structured exploration led us to identify two top-performing configurations that provided a strong balance of high F1 score on the validation set without overfitting to the training data.

The two best models found were:

Parameter	**Model 1**	**Model 2**
rnn_type	LSTM	RNN
bidirection	False	False
hidden layers	2	1
hidden size	256	64
window size	25	25
stride	2	2
batch size	128	128
learning rate	1e-3	1e-3
dropout rate	0.3	0.3
l1_lambda	0	0
l2_lambda	0	0
F1 validation	**0.9384**	**0.9363**
F1 test	**0.9567**	**0.9558**

Here is the result from *Tensorboard*.



	Run ↑	Smoothed	Value	Step	Relative
●	BEST_LSTM	0,9207	0,92	149	10.08 min
●	BEST_RNN	0,9127	0,9089	119	4.909 min

These configurations were then selected for the final, robust K-fold cross-validation and ensembling, as described in the previous section.

6 Discussion

Here are some observations that emerged during development.

As mentioned in the *Model Architecture and Training* section, to address class imbalance we assigned class weights in the CrossEntropyLoss. We first used the simplest approach, where each weight was the inverse of the class frequency in the training set, but the results were inconsistent. We therefore adopted a logarithmic scaling instead: each weight is computed as

$$weight(class) = \frac{1}{log(1 + count(class))}$$

This compresses extreme values, giving the rarest class an advantage without granting it disproportionate influence.

Overall, we learned that our best results came from balancing how the data was prepared and how much the model was allowed to learn. There wasn't one perfect fix for overfitting,we had to carefully adjust different parts of the system to work together.

One of the most important things we found was the best window settings. Using window_size = 25 and stride = 2 gave the model enough useful information without making the samples too similar. This showed us that the way the model views the data can be just as important as the model itself.

We also discovered that there are two different types of models that can both work very well:

- A larger LSTM model that can learn complex patterns, but only if we use dropout to stop it from memorizing
- A smaller RNN model that avoids overfitting simply by being less powerful

Both of these models ended up reaching almost the same strong performance. This suggests that the important patterns in the data were clear enough to be learned either by a big model with strong regularization or a small model that naturally generalizes.

7 Conclusions

Both the simple RNN and the much larger LSTM reach similar performance because the task, after preprocessing, does not require complex temporal modeling. The window size is short, the informative features are largely static or linearly compressed, and the dataset is not large enough to benefit from higher model capacity. As a result, simpler architectures generalize just as well as deeper ones, while larger models do not exploit their additional representational power.