When Larger Isn't Better: Lightweight CNNs Outperform Large Time-Series Models in Classification of Oil and Gas Drilling Data

Abdallah Zakaria Benzine [*]
AIQ, Abu Dhabi, UAE
`abdallah.benzine@g42.ai`

Sebastiaan Buiting[*]
AIQ, Abu Dhabi, UAE
`sebastiaan.buiting@g42.ai`

Soumyadipta Sengupta[*]
AIQ, Abu Dhabi
`soumyadipta.sengupta@g42.ai`

Badal Gupta[*]
AIQ, Abu Dhabi, UAE
`trbadal.gupta@g42.ai`

Youssef Tamaazousti
AIQ, Abu Dhabi, UAE
`Youssef.Tamaazousti@g42.ai`

Abstract

Large models have transformed various fields, particularly in forecasting, but their effectiveness in classification remains limited, especially for specialized domains like oil and gas drilling. This paper evaluates the performance of large models in time series classification tasks, highlighting their challenges in handling real-world univariate and multi-variates real-world time series data. Through comprehensive experiments, we show that these models are outperformed by lightweight convolutional baselines in both accuracy and efficiency. While large models like Chronos and Moments demonstrate some success, they require significantly more computational resources to achieve optimal classification performance. Our results suggest that lighter CNNs are better suited for time series classification in industrial applications, where both accuracy and computational efficiency are critical.

1 Introduction

Time series analysis has become a cornerstone in many domains, enabling the extraction of valuable insights from sequential data. The rise of large models has brought about significant advancements in time series forecasting (Ansari et al. [2024], Goswami et al. [2024], Götz et al. [2024], Zeng et al. [2024], Li et al. [2024]). However, the applicability of these models to time series classification remains less explored. In this paper, we explore this aspect for the drilling domain of the oil and gas industry. Accurate classification is crucial for optimizing drilling performance and ensuring safety, as demonstrated in prior work on automated quality control and anomaly detection in the drilling domain Benzine et al. [2024a,b]. Specifically, here we focus on the classification of drilling activities from sensor data, with the objective of monitoring and optimizing drilling performance.

Classifying time-series data in the drilling domain presents several technical challenges: (i) sensor noise, often introduced by environmental factors, complicates accurate data interpretation; (ii) high intra-class variance from diverse nature of drilling activities, further increases classification complexity, while (iii) low inter-class variance leads to difficulties in distinguishing between classes, resulting in false positives; (iv) data contains complex patterns that go beyond simple trends, periodicity, or spike detection, requiring models to capture both short-and long-term dependencies; (v) the scarcity

[*]**Equal Contributions**

of labeled data requiring drilling SMEs for annotation; and last but not least (vi) the class imbalance aspect, where the events of interest are rare and underrepresented in the dataset. All those technical challenges created the need to assess state-of-the-art approaches on real-world drilling time-series data; with a focus on Large Models for Time Series (LM4TS) - *i.e.,* Chronos Ansari et al. [2024] and Moment Goswami et al. [2024] - and Convolutional Neural Networks Kiranyaz et al. [2021].

Large models for time series are often seen as universal solutions due to their broad applicability across domains. However, they face significant limitations when confronted with real-world, real-time challenges. First, most of these models are optimized for forecasting rather than handling tasks like time series classification, which involves recognizing complex patterns instead of predicting future values. Additionally, these models often struggle to process the complex and noisy patterns commonly found in industrial time series data, which limits their practical utility in real-time applications.

In this paper, we conducted a comparison study of convolutional neural netwerk (CNNs) models and large models in time series (LM4TS), with a focus on classification of oil and gas drilling data. We evaluated the approaches in two scenarios: (1) training from scratch and (2) fine-tuning on the downstream task. Our study spans two classification tasks — one univariate and one multivariate —reflecting the diverse challenges of real-world data. We also assess the impact of different amounts of training data on final classification accuracy as well as the runtime of each model. The results demonstrate that large models can be outperformed by lighter, more efficient convolutional baselines. Not only do these simpler models achieve better accuracy, but they also exhibit significantly lower computational overhead, faster running times, and greater robustness when dealing with limited data. Through this series of experiments, we highlight that convolutional models offer a practical and efficient solution for time series classification in industrial settings like oil and gas drilling.

The paper is organized as follows: Section 2 describes the drilling classification tasks and data. Section 3 outlines the models evaluated. Section 4 presents the results of our empirical study.

2 Drilling classification tasks

Two tasks have been considered in this paper: (1) downlinking classification and (2) casing classification. These are two important tasks of interest in drilling monitoring and are described below.

2.1 Downlinking classification task

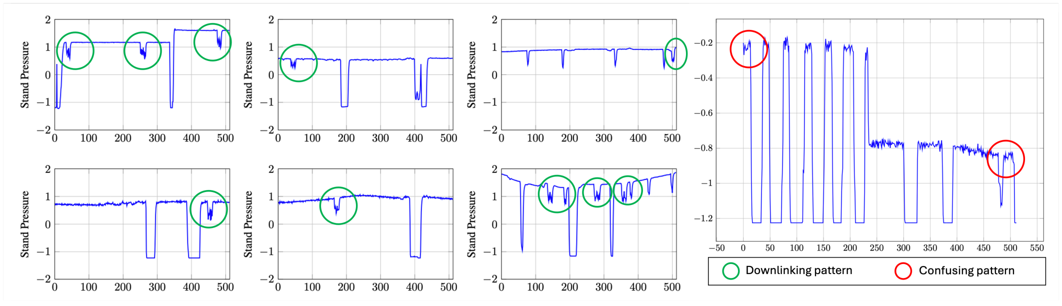

Figure 1: Example of positive (left) and negative (most-right) downlinkings data.

The downlinking classification task is a uni-variate problem involving the analysis of Surface Pressure Pulse signals to classify downlinking events during drilling operations. Downlinking, which typically lasts a few minutes, involves sending commands from the surface to downhole tools via pressure pulses in the drilling fluid (see appendix A.1 for more details). The task uses 8192 secs windows, resampled to 512 steps, to identify these downlinking events within the standpipe pressure signal, which is often noisy and mixed with other operational data. The dataset contains 46,825 windows total, with 38% positive samples, split into train (62%), validation (20%), and test (19%) sets.

2.2 Running casing classification task

The running casing classification task is a multi-variate problem involving the analysis of various drilling signals to detect and classify running casing events during drilling operations. Running

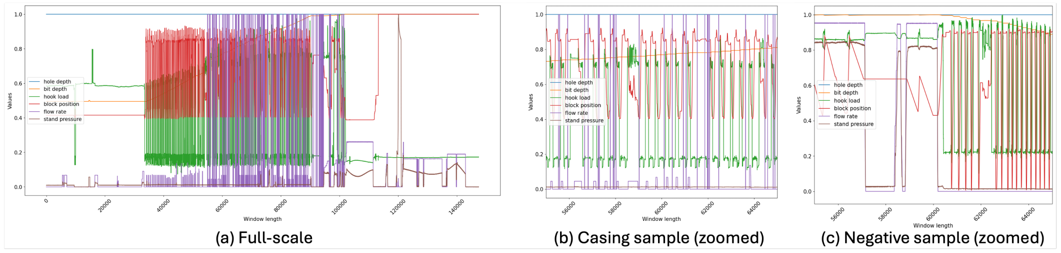

Figure 2: Example of data (left) with positive (middle) and negative (most-right) Casing samples.

casing, which involves lowering and installing large-diameter pipes to stabilize the well, typically lasts several hours (see appendix A.2 for more details). The task uses 40-hour windows at second resolution, resampled to 512 steps, to identify these events within multiple signals including hook load, block position, bit depth, hole depth, flow rate, and standpipe pressure. The dataset contains 17,751 windows total, with 33% positive samples, split into train (57%), validation (19%), and test (24%) sets. This split maintains the integrity of longer time series to prevent information leakage between sets, while improving class balance through selective sampling of negative class windows.

3 Benchmarked Methods

Baseline: A simple ML (*i.e.*, Xgboost) classifier trained on top of raw data.

LM4TS - Chronos [Ansari et al., 2024] is a T5-based model for time series forecasting that tokenizes univariate time series into a fixed vocabulary, transforming regression tasks into classification problems.

LM4TS - MOMENT [Goswami et al., 2024] is a transformer encoder pre-trained on diverse datasets for general-purpose analysis. It supports multiple tasks, handling univariate and multivariate data with models ranging from 40M to 385M parameters.

The 1D Convolutional Neural Network (CNN) Model Kiranyaz et al. [2021] is a simpler alternative to large models, designed for time series classification. It stacks Bottleneck1D blocks with max pooling to capture key sequential features, using 1D convolutions, residual connections, global average pooling, and a fully connected output layer. This lightweight model is easy to train and ideal for quick benchmarking against more complex architectures (details in Appendix F).

4 Experimental Results

In this section, we present the results of several experiments to evaluate different time series models.

Model	Weights Init.	Params (M)	Donwlinking F1 (%)	Casing F1 (%)	Inference Time (ms) GPU	Inference Time (ms) CPU
XGboost	n/a	n/a	61.3	92	n/a	**0.001***
CNN	From Scratch	**15.5**	**95.61**	**94.57***	**3.4**	**3.2**
Moments	From Scratch	385	90.0	90.8	10	140
Chronos	From Scratch	710	83.0	79.0	40	3154
Moments	Pre-trained	385	94.0	88.0	10	140
Chronos	Pre-trained	710	**97.0***	**93.0**	40	3154

Table 1: Comparison of various models for downlinking and running casing tasks. Top: Baseline of simple classfier (xgboost) trained on raw data. Middle: CNN vs LM4TS when trained from scratch. Bottom: Fine-tunning of pre-trained LM4TS. Best score in bold and with star. 2nd best in bold.

4.1 Large Model For Time-Series (LM4TS) vs Convolutional Neural Networks (CNN)

In this comparison, we evaluate LM4TS against the CNN. The CNN outperformed most models across all metrics, achieving an F1 score of 95.61% for the downlinking classification task and 94.57% for the running casing task. In particular, it outperforms all LM4TS models trained from scratch or with pre-trained weight initialization except in the downlinking task where it is beaten by the Chronos-large model. Nevertheless, the CNN is more than 900x faster on CPU. This makes it the most efficient and effective model in the evaluation, with fewer parameters and faster inference times.

4.2 Impact of Training Data Proportion

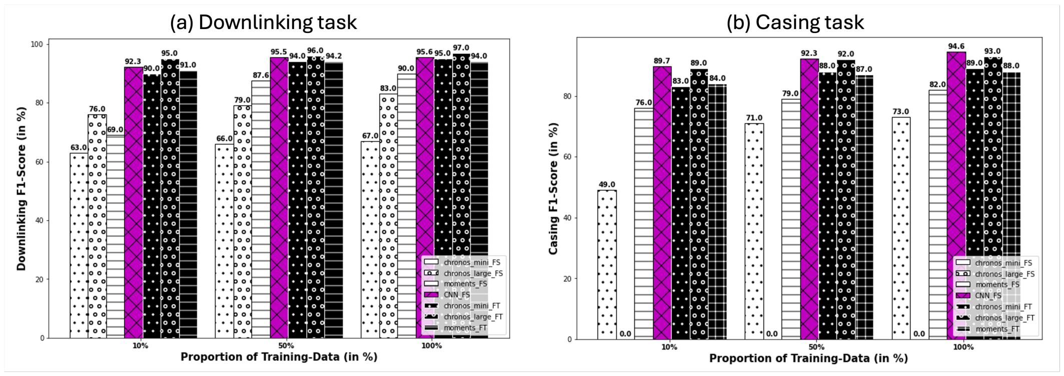

Figure 3: Comparison between LM4TS trained from scratch (white bars), CNN(magenta) and pretrained LM4TS(black) on downlinking and casing tasks using different proportion of training data.

The plots in Figure 3 clearly demonstrate that the CNN baseline consistently outperforms all LM4TS models trained from scratch, regardless of the training data proportion used. Remarkably, the CNN, trained from scratch, achieves the same or even better accuracy compared to pre-trained LM4TS models. When compared with Chronos-mini, which has a comparable number of parameters (20M), the CNN outperforms this model both with and without pretraining, highlighting that even with equal capacity, the CNN is a more effective and efficient choice. This comparison underscores the superior performance of CNN architectures in both low-data and full-data scenarios, proving it to be the better choice for time-series classification tasks. Appendix I contain more detailed results in particular for MINIROCKET Dempster et al. [2021] and LSTM Hochreiter [1997].

5 Conclusion & Perspectives

In this paper, we evaluated the performance of large models, in time-series classification tasks within the oil and gas drilling domain. While large models have shown success in time-series forecasting, our findings demonstrate that they can be outperformed by lightweight CNN models in classification. This superiority was evident in terms of accuracy, the amount of training data , and inference time.

Our experiments, conducted on two drilling classification tasks highlighted several key insights: (1) *CNN models trained from scratch consistently achieved better accuracy than Transformers trained from scratch*, with up to 5% higher performance on downstream tasks; (2) *CNN models were competitive, and in some cases, outperformed pre-trained Transformers*, showcasing their robustness in classification tasks – These findings suggest that CNNs offer a strong alternative to large models, even without the benefit of pre-training; (3) *CNN models demonstrated comparable abilities to perform well with limited data* when compared to pre-trained Transformers. This proves their effectiveness in learning from smaller datasets, a valuable trait in practical applications where extensive labeled data is often unavailable; (4) *CNN architectures were up to 900x faster* in inference than the best-performing Transformer models. These results suggest that lightweight CNN models are not only more accurate but also more efficient than large models for time-series classification in the oil and gas drilling domain. Their balance of performance, reduced data requirements, and fast inference time makes CNNs a more practical choice for real-world, real-time industrial applications.

Looking forward, future research will focus on expanding the evaluation to a broader set of time-series tasks (segmentation, anomaly detection and forecasting) and more drilling events. Additionally, as pre-training has been shown to be beneficial Tamaazousti et al. [2020], we aim to explore the potential of pre-training CNNs on large time-series datasets to enable a fairer comparison with LM4TS, particularly in scenarios where embeddings are frozen or fine-tuned. This will provide deeper insights into the feature learning capabilities of each architecture.

References

Abdul Fatir Ansari, Lorenzo Stella, Caner Turkmen, Xiyuan Zhang, Pedro Mercado, Huibin Shen, Oleksandr Shchur, Syama Sundar Rangapuram, Sebastian Pineda Arango, Shubham Kapoor, et al. Chronos: Learning the language of time series. *arXiv preprint arXiv:2403.07815*, 2024.

Abdallah Benzine, Amine El Khair, Sebastiaan Buiting, ? Soumyadipta Sengupta, Badal Gupta, Ufaq Khan, Youssef Tamaazousti, Sudheesh Vadakkekalam, ? Sreejith Balakrishnan, Ahmed Jhinaoui, et al. Ai-automated drilling rtoc with ml and deep learning anomaly detection approach. In *Abu Dhabi International Petroleum Exhibition and Conference*, page D041S140R001. SPE, 2024a.

Abdallah Benzine, Youssef Tamaazousti, Sudheesh Vadakkekalam, Sreejith Balakrishnan, Imane Chraibi, Dhaker Ezzeddine, Arghad Arnaout, Shreepad Purushottam Khambete, Paulinus Bimastianto, Shahid Duivala Muhammad, et al. Ai-automated codification-qc model for daily drilling reports. In *Abu Dhabi International Petroleum Exhibition and Conference*, page D041S154R001. SPE, 2024b.

Angus Dempster, Daniel F Schmidt, and Geoffrey I Webb. Minirocket: A very fast (almost) deterministic transform for time series classification. In *Proceedings of the 27th ACM SIGKDD conference on knowledge discovery & data mining*, pages 248–257, 2021.

Mononito Goswami, Konrad Szafer, Arjun Choudhry, Yifu Cai, Shuo Li, and Artur Dubrawski. Moment: A family of open time-series foundation models. *arXiv preprint arXiv:2402.03885*, 2024.

Leon Götz, Marcel Kollovieh, Stephan Günnemann, and Leo Schwinn. Efficient time series processing for transformers and state-space models through token merging. *arXiv preprint arXiv:2405.17951*, 2024.

S Hochreiter. Long short-term memory. *Neural Computation MIT-Press*, 1997.

Serkan Kiranyaz, Onur Avci, Osama Abdeljaber, Turker Ince, Moncef Gabbouj, and Daniel J. Inman. 1d convolutional neural networks and applications: A survey. *Mechanical Systems and Signal Processing*, 151:107398, 2021. ISSN 0888-3270. doi: https://doi.org/10.1016/j.ymssp.2020.107398. URL `https://www.sciencedirect.com/science/article/pii/S0888327020307846`.

Jiawei Li, Jingshu Peng, Haoyang Li, and Lei Chen. Unicl: A universal contrastive learning framework for large time series models. *arXiv preprint arXiv:2405.10597*, 2024.

Youssef Tamaazousti, Herve Le Borgne, Celine Hudelot, Mohamed-El-Amine Seddik, and Mohamed Tamaazousti. Learning more universal representations for transfer-learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(9):2212–2224, 2020.

Chaolv Zeng, Zhanyu Liu, Guanjie Zheng, and Linghe Kong. C-mamba: Channel correlation enhanced state space models for multivariate time series forecasting. *arXiv preprint arXiv:2406.05316*, 2024.

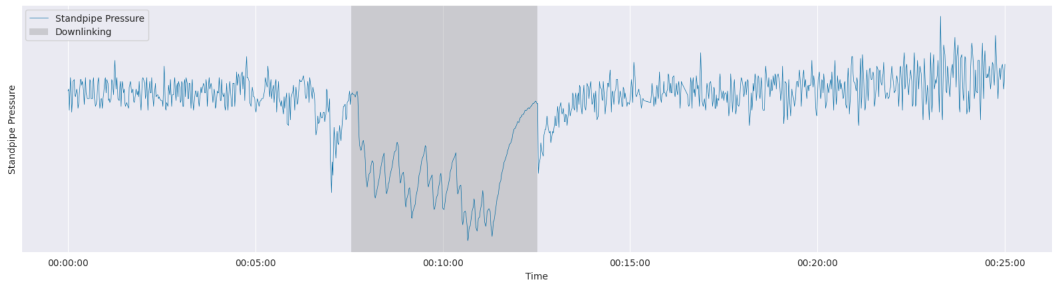

Figure 4: Example of a downlinking in fairly noisy data

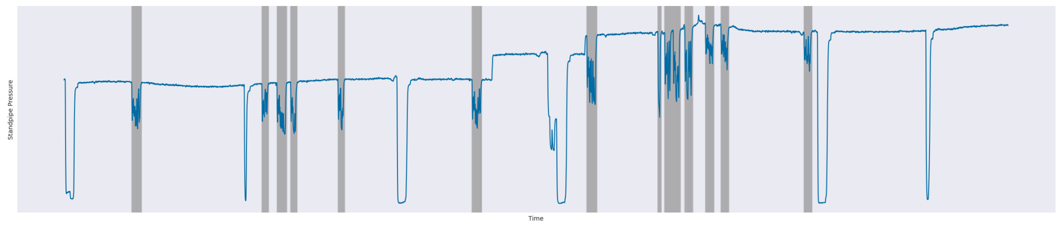

Figure 5: Example of repeated downlinking signatures in the pressure signal

A Appendix - Use cases

A.1 Downlinking

Downlinking in drilling operations refers to the transmission of commands from the surface to downhole tools, crucial for real-time control and optimization of the drilling process. Characterized by specific patterns in drilling sensor data, such as repeated signatures in standpipe pressure or pipe rotation speed, downlinking events can be challenging to distinguish from pressure anomalies (see fig 4 and figure 5).

A.2 Running casing

Casing is run after a section of the oil/gas well is drilled. Casing helps in protecting the well from collapsing and provides zonal isolation when drilling subsequent deeper sections of the well is drilled. Usually, a typical casing run can last from 24-50 hours. In addition, a single casing run might not be continuous due to multiple stoppages during casing running. Parameters like bit depth increases during casing run and hole depth stays constant. Other parameters like block position can indicate the length of the pipes that are being run which can be indicative of casing pipes. Hook load is another important parameter to consider since casing pipes could have different weights than drill pipes. Typically, the flow rate is near zero during a casing run and hence flow rate is another indicator for casing run. Behavior of all these indicators/features is shown in 6

B Appendix - Baseline - Linear Model

The Linear Model is a simple yet effective baseline, implemented as a single fully connected layer:

```python
class BaselineModel(nn.Module):
    def __init__(self, input_size):
        super().__init__()
        self.fc = nn.Linear(input_size, 1)

    def forward(self, x):
        x = x.reshape(x.size(0), -1)
        return self.fc(x)
```

Listing 1: Implementation of the linear model class.

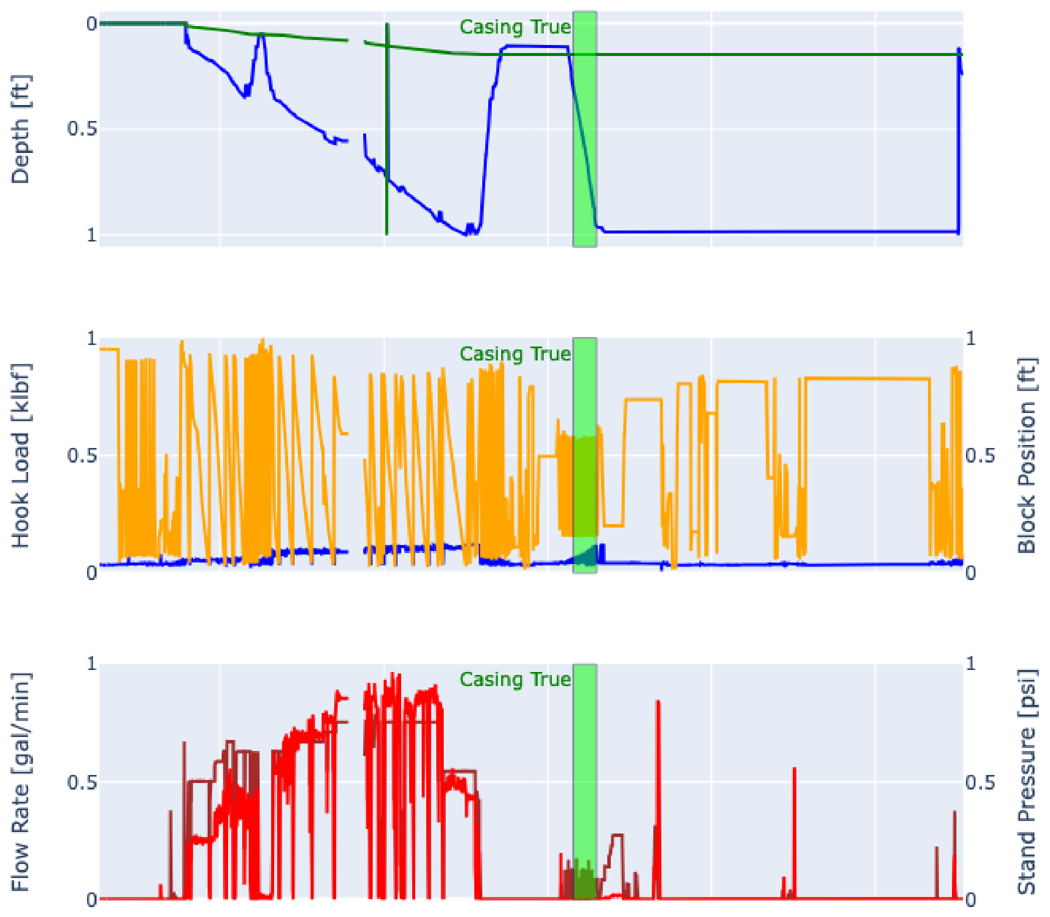

Figure 6: Behavior of hole depth, bit depth, hook load, block position, stand pressure and flow rate during and before running casing event

The training setup for this linear model utilized the Adam optimizer with default parameters (i.e. learning rate of 0.001). The model was trained for a maximum of 20 epochs, with an early stopping patience of 5 epochs for no improvement in validation loss. The loss function used was Binary Cross-Entropy with Logits (BCEWithLogitsLoss).

B.1 Results and analysis

We conducted experiments using different proportions of the training data, ranging from 10% to 100%. We experimented with two distinct use cases, running casing and downlinking classification.

B.1.1 Downlinking

The Linear Model, with 513 trainable parameters using only stand pressure and a window length of 512 steps, was surprisingly able to capture a significant portion of the downlinking signatures, achieving a peak F1 score of 57% with only 30% of the training data. The model shows little improvement in accuracy as the proportion of training data increases (figure 7). There is even a slight reduction in F1 Score when 50% or more of the training data was included.

B.1.2 Running Casing

The Linear Model for Running Casing model has 3,073 trainable parameters and uses a window size of 512 for input data. The features used include bit depth, hole depth, block position, hook load, stand pressure, and flow rate. The model is trained with a fairly strict early stopping procedure to prevent

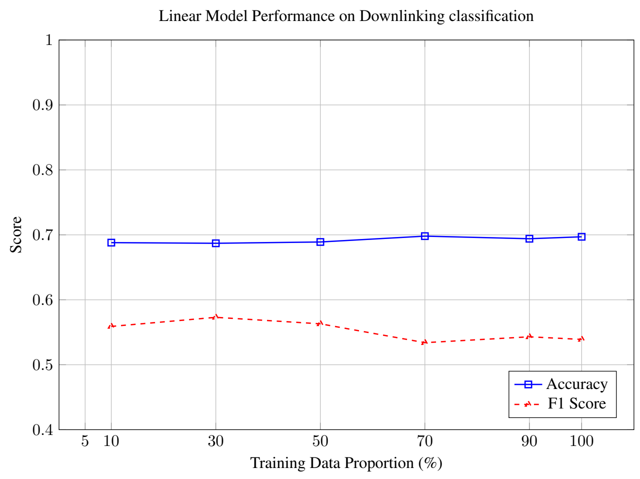

Figure 7: Impact of Training Data Proportion on Accuracy and F1 Score for the Linear Model on Downlinking classification.

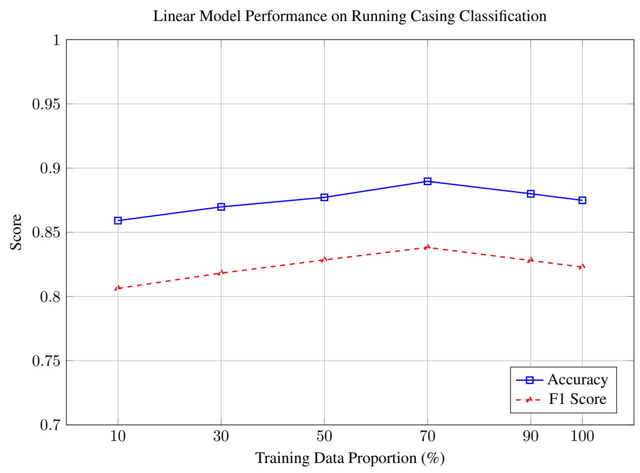

Figure 8: Impact of Training Data Proportion on Accuracy and F1 Score for Linear Model on Running Casing classification.

overfitting. The results show good performance for running casing classification with the highest accuracy of 89% and F1 score of almost 83% are achieved at 70% of the training data, suggesting that some signatures are easily captured (see figure 8). A gradual improvement can be seen in both accuracy and F1 score as the proportion of training data increases.

C Appendix - Baseline - LSTM

```
class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size=128, num_layers=1):
        super(LSTMModel, self).__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers,
            batch_first=True)
        self.fc = nn.Linear(hidden_size, 1)

    def forward(self, x):
        # x is of shape (batch_size, seq_length, input_size)
        out, _ = self.lstm(x)
        out = out[:, -1, :]  # Take the output of the last time step
        out = self.fc(out)
        return out
```

Listing 2: Implementation of the LSTM class.

The LSTM model employs a single-layer LSTM with a hidden size of 128, followed by a fully connected layer for binary classification. The model captures temporal dependencies through recurrent processing and outputs predictions based on the final time step. It was trained using the Adam optimizer with a learning rate of 0.001 for a maximum of 20 epochs, with early stopping patience of 5 epochs. The loss function used was Binary Cross-Entropy with Logits (BCEWithLogitsLoss).

C.1 Results and analysis

We conducted experiments using different proportions of the training data, ranging from 10% to 100%. We experimented with two distinct use cases, running casing and downlinking classification.

C.1.1 Downlinking

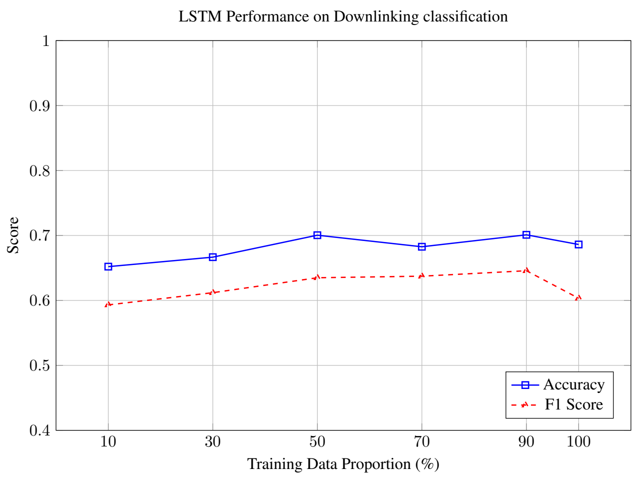

Figure 9: Impact of Training Data Proportion on Accuracy and F1 Score for the LSTM on Downlinking classification.

The LSTM model shows improvement for the classification of downlinking signatures in both accuracy and F1 score as training data increases, reaching a peak at 90% of the data (see figure 9). Beyond that, performance slightly drops. With 67,201 trainable parameters, the model captures key patterns but shows diminishing returns with larger data sets, similar to the behavior observed in the linear model, but with two orders of magnitude more parameters.

C.1.2 Running Casing

The LSTM model for running casing classification initially struggles with a small fraction of the data, failing to converge and producing an F1 score of 0 at 10% training data (see figure 10). As the training data increases, both accuracy and F1 score improve, peaking at 70% of the data with an accuracy of 90.2% and an F1 score of 85.1%. Beyond 70%, performance stabilizes, with only slight reductions in the following data fractions. With 69,761 trainable parameters, the model captures key patterns but shows diminishing returns with larger data sets.

D Appendix - Baseline - MINIROCKET

```
def train_minirocket(X_train_scaled, y_train):
    from sklearn.linear_model import RidgeClassifierCV
    from sktime.transformations.panel.rocket import MiniRocket

    X_train_df = convert_to_dataframe(X_train_scaled)
```

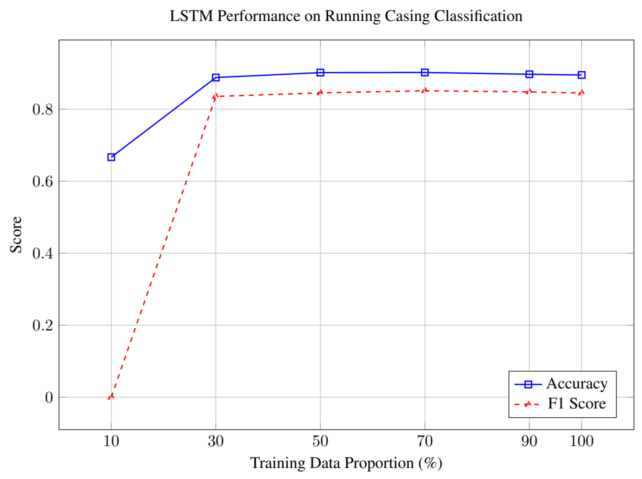

Figure 10: Impact of Training Data Proportion on Accuracy and F1 Score for LSTM on Running Casing classification.

```
    minirocket = MiniRocket()
    minirocket.fit(X_train_df)

    X_train_transform = minirocket.transform(X_train_df)

    y_train = y_train.ravel() if y_train.ndim > 1 else y_train

    classifier = RidgeClassifierCV(alphas=np.logspace(-3, 3, 10))
    classifier.fit(X_train_transform, y_train)

    return minirocket, classifier
```

Listing 3: Implementation of the MINIROCKET class.

The MiniRocket pipeline provides an efficient approach for time-series classification. Using default settings, MiniRocket extracts 9,996 features from the time-series data through convolutional operations, mapping the data to higher-dimensional which enables a Ridge Classifier to effectively model a potential linear decision boundary. This combination efficiently captures temporal patterns while exploiting the linear separability of the transformed feature space.

D.1 Results and analysis

We conducted experiments using different proportions of the training data, ranging from 10% to 100%. We experimented with two distinct use cases, running casing and downlinking classification.

D.1.1 Downlinking

The MINIROCKET pipeline demonstrates consistently high performance for downlinking classification across all training data proportions (see figure 11). Accuracy starts of high and peaks at 30% of the data (89.8%) and stabilizes with only minor variations thereafter. Similarly, the F1 score reaches its highest value (85.1%) at 30% but remains steady across other proportions. MiniRocket efficiently captures temporal patterns and achieves robust performance even with limited data, showing minimal sensitivity to increasing training data.

D.1.2 Running Casing

MiniRocket performs exceptionally well for running casing classification, achieving over 95% accuracy and 92.5% F1 score even at just 10% of the data (see figure 12). Both metrics peak at 70% of the data, with an accuracy of 96.2% and an F1 score of 94.1%, before slightly declining at higher

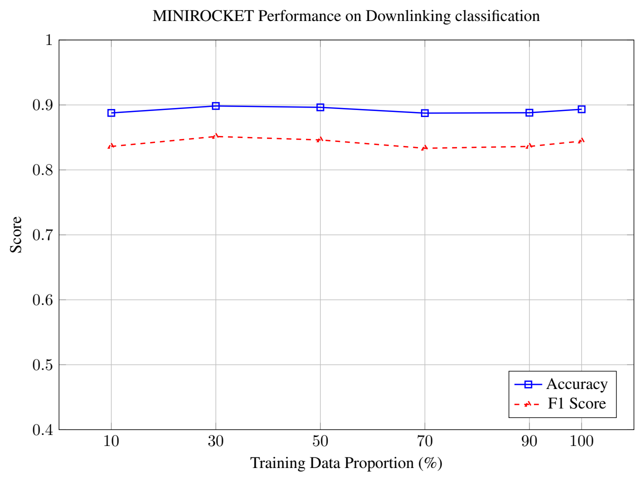

Figure 11: Impact of Training Data Proportion on Accuracy and F1 Score for the MINIROCKET on Downlinking classification.

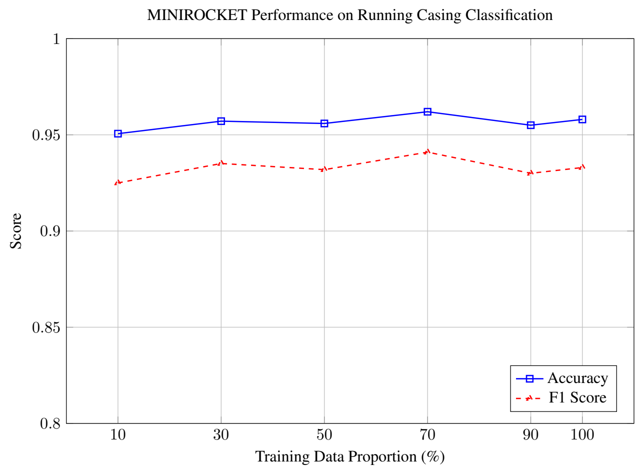

Figure 12: Impact of Training Data Proportion on Accuracy and F1 Score for MINIROCKET on Running Casing classification.

proportions. The model's strong performance with limited data highlights the effectiveness of its feature extraction and Ridge Classifier combination for this task.

E Appendix - Baseline - XGBoost Model

The XGBoost model serves as another robust baseline, utilizing default hyperparameters (i.e. 100 estimators and a maximum tree depth of 6). The windows were fed directly to the model. This model leverages decision trees to capture complex patterns in the data.

```
def train_xgboost(X_train, y_train, X_val, y_val):

    from xgboost import XGBClassifier

    model = XGBClassifier(
        objective="binary:logistic",
        eval_metric="logloss",
        use_label_encoder=False,
    )
    eval_set = [(X_val, y_val)]
    model.fit(X_train, y_train, eval_set=eval_set, verbose=False)
    return model
```

Listing 4: Implementation of the XGBoost model training.

E.1 Results and analysis

We conducted experiments using different proportions of the training data, ranging from 10% to 100%.

E.1.1 Downlinking

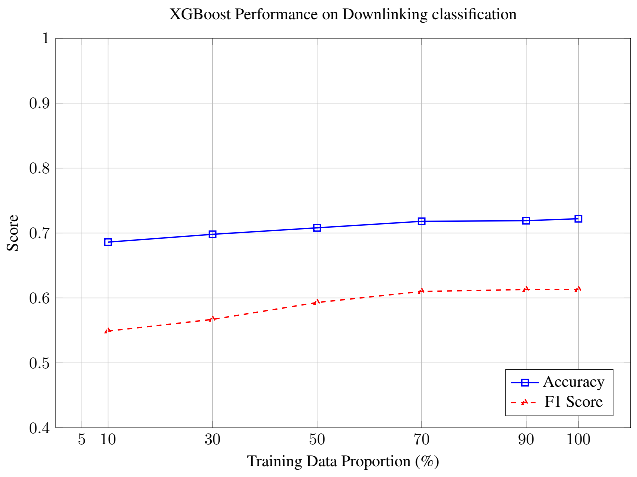

Figure 13: Impact of Training Data Proportion on Accuracy and F1 Score for XGBoost on Downlinking classification.

The XGBoost model for running casing classification shows gradual improvement as training data increases. Starting with 68.6% accuracy and 54.9% F1 score at 10% data, it reaches peak performance of 72.2% accuracy and 61.3% F1 score using 100% data (see figure 13). Notably, reasonable performance is achieved even with limited data, indicating robust feature capture. Performance gains diminish at higher data proportions, with only marginal improvements between 70% and 100%.

E.1.2 Running Casing

For the running casing task using the XGBoost model, the results improve in both accuracy and F1 score as the proportion of training data increases (see figure 14). Notably, the model achieves strong performance even with limited data, indicating that the features effectively capture relevant patterns early on. The highest accuracy of almost 95% and F1 score of 92% at 100% of the training data. This suggests that XGBoost is effective at identifying key patterns for the running casing task, demonstrating robust performance with minimal data and delivering near-optimal results as more data is introduced.

F Appendix - Baseline - CNN

The CNN Baseline model used in this study is a highly efficient time series classification architecture, designed with 1D convolutions and residual connections to extract meaningful features from sequential data. It is specifically optimized for tasks such as downlinking and running casing classification in the oil and gas drilling domain. The model's simplicity and computational efficiency make it ideal for real-time processing.

Architecture Overview: The CNN Baseline consists of several components that collectively process the time series data in a hierarchical manner. The model primarily utilizes *Bottleneck1D* blocks to capture local features and apply downsampling, ensuring computational efficiency.

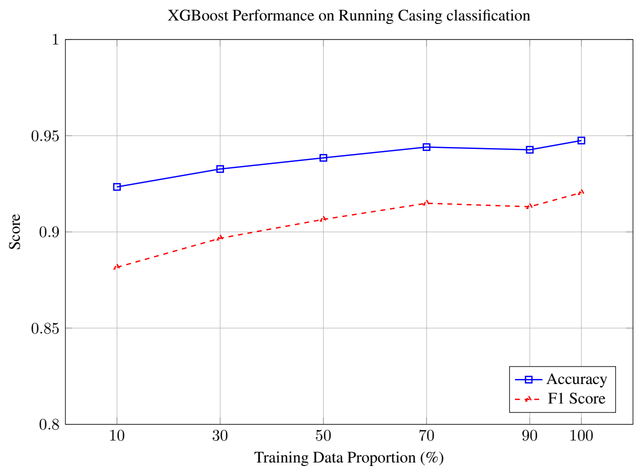

Figure 14: Impact of Training Data Proportion on Accuracy and F1 Score for XGBoost on Running Casing classification.

Key Components and Hyperparameters:

- **Bottleneck1D Block:**
  - **Convolutional Layers:**
    - **1x1 Convolution:** Reduces the dimensionality of the input and transforms features.
    - **3x3 Convolution:** Captures local patterns in the time series data.
    - **1x1 Convolution:** Restores the dimensionality post-processing.
  - **Batch Normalization:** Applied after each convolution to stabilize training and normalize the activations.
  - **ReLU Activation:** A non-linear activation function applied after every convolution, enabling the model to learn complex, non-linear patterns.
  - **Residual Connections:** Used to add the input back into the output, improving gradient flow and training speed.

  Hyperparameters:
  - **Input Channels:** Varies depending on the task (e.g., 5 for downlinking and 6 for running casing).
  - **Expansion Factor:** 2 (expands the output size after the bottleneck block).
  - **Kernel Size:** 1x1, 3x3 for convolutions.
  - **Stride:** 1 (for most layers, except for downsampling).
  - **Number of Bottleneck Blocks:** 1 block in each layer.
- **Encoder Architecture:** The encoder progressively downsamples the input sequence through a series of *Bottleneck1D* blocks and pooling operations. This downsampling allows the model to focus on important features while reducing computational complexity.

  Hyperparameters:
  - **Depth:** 7 layers of bottleneck blocks.
  - **Max-Pooling:** Applied after each bottleneck block to reduce temporal resolution and capture high-level features.
  - **Number of Features:** 256 (initial number of features processed by the first convolutional layer).
  - **Kernel Size:** 7x1 for the initial convolutional layer.
  - **Stride:** 2 (to ensure downsampling at appropriate intervals).
  - **Pooling Kernel Size:** 2 (for max-pooling).
- **Global Average Pooling:** After the encoder processes the sequence, global average pooling is applied to reduce the high-dimensional feature map into a fixed-size vector. This vector is then passed to a fully connected layer for final classification.

Hyperparameters:

  - **Global Pooling:** Adaptive, ensures that the output dimension is fixed, regardless of input size.

- **Fully Connected Layer:** The pooled features are passed to a fully connected layer, which outputs a single logit for binary classification (i.e., whether an event occurs or not).

  Hyperparameters:

  - **Input Features:** 512 (from the global pooling layer).
  - **Output Features:** 1 (for binary classification).

Training and Optimization:

- **Optimizer:** Adam optimizer is used with an initial learning rate of 0.001.
- **Batch Size:** 32 (used during training).
- **Epochs:** 50 (early stopping is applied if no improvement is observed after 10 epochs).

G Appendix - Moment

MOMENT is a transformer encoder-based model designed for general-purpose time series analysis. Pretrained on a wide variety of time series data, MOMENT is capable of handling multiple tasks, including classification, forecasting, anomaly detection, and imputation. It is versatile in managing both univariate and multivariate time series data. The model family varies in size, ranging from 40 million to 385 million parameters, making it adaptable to different computational and performance needs.

```
from momentfm import MOMENTPipeline
from xgboost import XGBClassifier


model = MOMENTPipeline.from_pre-trained(
    "AutonLab/MOMENT-1-large",
    model_kwargs={"task_name": "embedding"},
    # We are loading the model in "embedding" mode
)
model.init()
train_embeddings, train_labels = get_embedding(model, train_dataloader
    )
model_xgb = XGBClassifier(use_label_encoder=False,eval_metric="logloss
    ")
model_xgb.fit(train_embeddings, train_labels)
\label{fig:embedding-moment}
```

Listing 5: Embeding Extraction Moment

```
from momentfm import MOMENTPipeline

model = MOMENTPipeline.from_pre-trained(
    "AutonLab/MOMENT-1-large",
    model_kwargs={"task_name": "classification"},
    # We are loading the model in "classification" mode
)
model.init()
```

Listing 6: Moment Full Finetuning

```
from momentfm import MOMENTPipeline
model = MOMENTPipeline.from_pre-trained(
    "AutonLab/MOMENT-1-large",
    model_kwargs={"task_name": "classification"},
    # We are loading the model in "classification" mode
)
model.init()
for i,layer in enumerate(self.model.modules()):
    if hasattr(layer, "reset_parameters"):
        layer.reset_parameters()
```

Listing 7: Moment Full Finetuning Weights Reset

The training process employed the Adam optimizer with an initial learning rate of 1e-6, coupled with a learning rate scheduler. The model was trained for a maximum of 20 epochs, with early stopping implemented to prevent overfitting, using a patience of 3 epochs. Cross-entropy loss was used as the loss function. For the Running Casing problem, a batch size of 20 was utilized, while a larger batch size of 128 was employed for the Downlinking problem

G.1 Results and analysis

We conducted experiments using varying proportions of the training data, ranging from 10% to 100%, to evaluate the performance of the MOMENT model on two distinct use cases: Downlinking and Running Casing. For these experiments, we selected the largest variant of the MOMENT model, with 385 million parameters.

G.1.1 Downlinking

Moment Embedding experiment Embedding are extracted from the pre trained moments model for downlinking use case for the we trained a Xgboost classifer to on top of the embeddings. In this experiment we did not observe a significant increase in performance with the use of a larger proportion of training data. Using only 10% of the data, the model achieved an F1-score of 72%. When the training data was increased to 100%, the F1-score improved marginally to 76%.(Refer Figure 15)

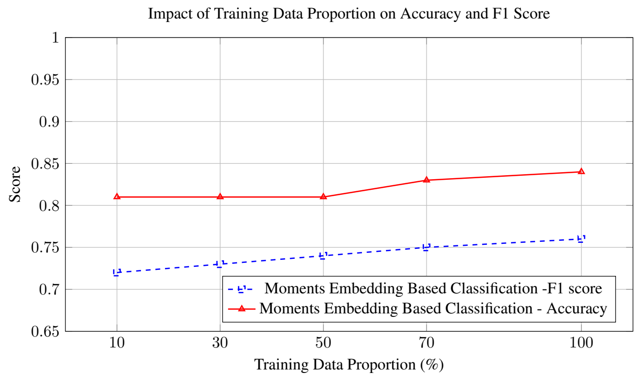

Figure 15: Testing Accuracy and F1 score using Moments Embedding Based Classification on Downlinking

Moment Full Fine tuning We fine-tuned the complete MOMENT model by updating the pre-trained weights based on the Downlinking data, which is univariate. The results show a significant improvement compared to the embedding-based classification mentioned earlier. After full fine-tuning, the model achieved an F1-score of 94%, a substantial increase from the 76% F1-score observed in the previous experiment with embedding-based classification. (Refer Figure 16)

Moment Full Finetuning after Weights Reset We fine-tuned the complete MOMENT model on Downlinking data after reinitializing the weights of the layers. The results were comparable to those achieved in the previous experiment with the fully pre-trained model. Notably, with the pre-trained weights, the model performed well even with only 10% of the data, highlighting the advantage of pre-training over starting with random weights. (Refer Figure 17)

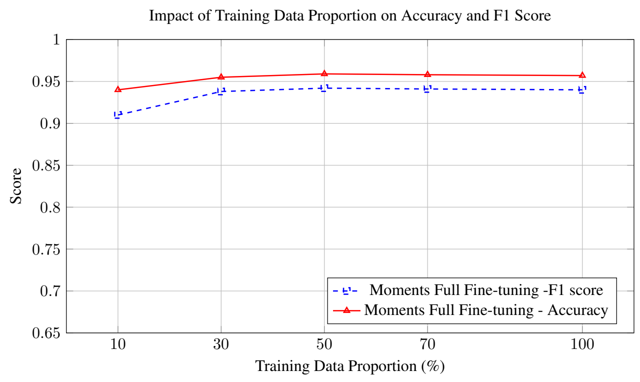

Figure 16: Testing Accuracy and F1score using Moments Full Fine tuning on Downlinking

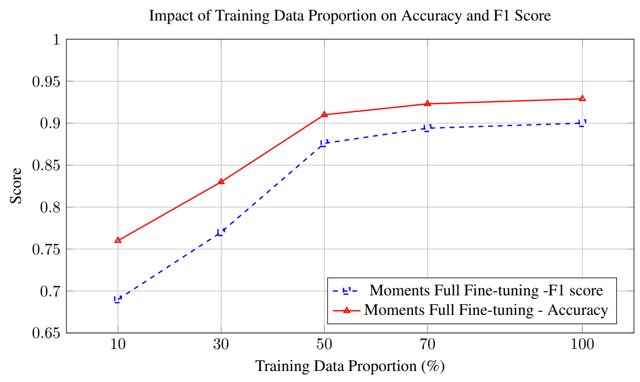

Figure 17: Testing Accuracy and F1score using Moments Full Fine tuning Weight Reset on Downlinking

G.1.2 Running Casing

Moment Embedding Experiment We extracted the embedding for Running casing data which is a multivariate problem In the case of Running casing, Results show a clear trend of improved performance as the dataset size increases. Accuracy scores range from 85% to 89%, while F1 scores improve from 75% to 83%. Notably, performance gains plateau around the 70% dataset mark, with minimal improvements observed beyond this point. The 100% dataset yielded the best results with an accuracy of 89% and an F1 score of 83%.(Refer Figure 18)

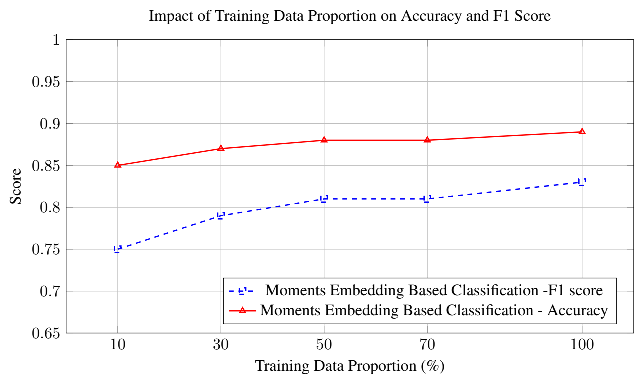

Figure 18: Testing Accuracy and F1 score using Moments Embedding Based Classification on Running Casing

Moment Full Fine tuning We conducted a full fine-tuning of the MOMENT model using the Running Casing data. Performance metrics remained robust across all dataset proportions, with accuracy consistently at or above 90% and F1-scores ranging from 84% to 88%. The use of pre-trained weights contributed to achieving better results compared to the embedding-based architecture, highlighting the effectiveness of full fine-tuning with pre-trained models.(Refer Figure 19)

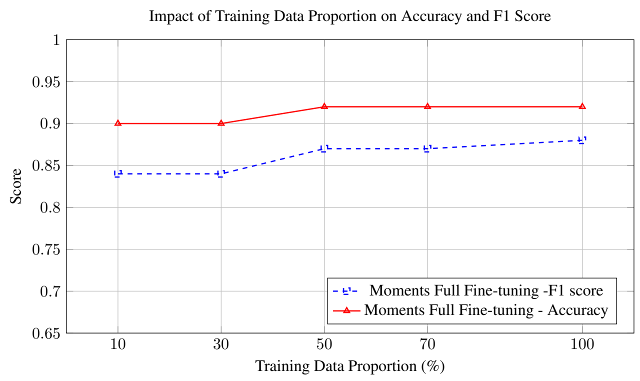

Figure 19: Testing Accuracy and F1score using Moments Full Fine tuning on Running Casing

Moment Full Finetuning after Weights Reset The performance of the MOMENT model with random weights was considerably lower compared to using pre-trained weights, especially when training with smaller amounts of data. With only 10% of the data, the model with pre-trained weights achieved an F1-score close to 85%, whereas the model with random weights achieved only 76%. While this performance gap narrowed as the data proportion increased, the model with pre-trained weights consistently outperformed the model with random weights across all dataset sizes.(Refer Figure 20)

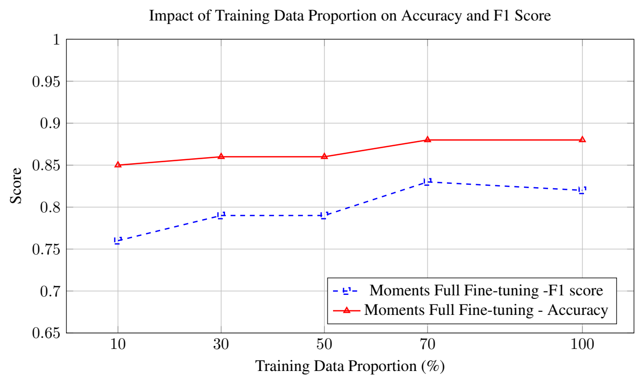

Figure 20: Testing Accuracy and F1score using Moments Full Fine tuning Weight Reset on Running Casing

H Appendix - Chronos Model

The Chronos model is a open source large time series forecasting open source model Ansari et al. [2024]. It is an encoder decoder architecture based on the T5 architecture. It comes with various number of parameters i.e. mini (8 M), tiny (20 M), small (46 M), base (210 M) and large (710 M). The Chronos model was trained on forecasting and is primarily used for forecasting. However, the model was adapted to perform classification task on both uni-variate and multivariate inputs. Embedding, full model fine tuning with pre-trained weights and random weights based initialisation were the different approaches experimented as with MoMeNt model. For full model fine-tuning in the univariate version, a classification head is attached at the end of the model. In the multivariate model full model finetuning 21 , all the covariates are passed through the encoder and the embeddings summed up before passing them through a classification head.

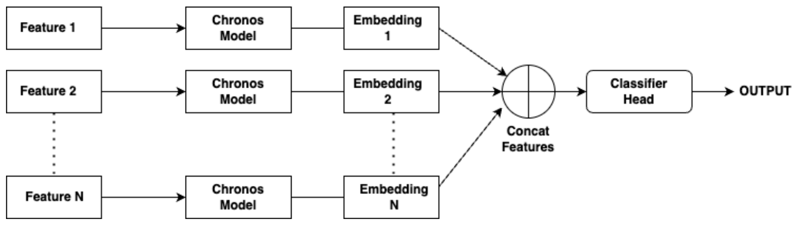

Figure 21: Chronos classification model accepting multivariate inputs

```
num_labels = 2
checkpoint = "amazon/chronos-t5-" + model_type
classfn_model = AutoModelForSequenceClassification.from_pre-trained(
    checkpoint, num_labels=num_labels)
pipeline = ChronosPipeline.from_pre-trained(
    checkpoint,
    device_map="cpu",
    torch_dtype=data_type,
)
tokenizer = pipeline.tokenizer
```

Listing 8: Implementation of the Chronos model (univariate input) for Classification

```
class ChronosMultiVariateModel(nn.Module):
    def __init__(self, classfn_model, emb_agg_mode="last"):
        super(ChronosMultiVariateModel, self).__init__()
        self.encoder = classfn_model.transformer.encoder
        self.classification_head = classfn_model.classification_head
        self.emb_agg_mode = emb_agg_mode
        self.sigmoid = nn.Sigmoid()

    def forward(self, input_ids):
        # input_ids - N*feats* (window_len+1)
        if input_ids.shape[1]!= len(features):
            raise ValueError("Dim=1 must equal number of features")
        if input_ids.shape[2]!= target_win_len+1:
            raise ValueError("Dim=2 must equal target window length +1
                ")

        emb_sum = 0
        for feat_i in range(input_ids.shape[1]):
            x = input_ids[:, feat_i, :]
            # Forward pass
            x = self.encoder(x)
            if self.emb_agg_mode == "sum":
                emb_sum += torch.sum(x["last_hidden_state"][:, :-1,
                    :], dim=1)
            elif self.emb_agg_mode == "last":
                emb_sum += x["last_hidden_state"][:, -1, :]

        x = self.classification_head(emb_sum)
        x = self.sigmoid(x)
        return x
```

Listing 9: Implementation of the Chronos model (multivariate input) for Classification

Like other models mentioned before, it uses a window size of 512 for the input features: bit depth, hole depth, block position, hook load, stand pressure, and flow rate.

H.1 Results and analysis

We conducted experiments using different proportions of the training data, ranging from 10% to 100%. Stand pressure was used as the input feature to the model.

H.1.1 Downlinking

Chronos Embedding Experiment Embeddings were generated for downlinking using the frozen Chronos model for the 6 features mentioned above on 512 length windows. A batch size of 64 was used for generating embeddings. XGBoost classifier (with same configuration as mentioned before) was trained on the embeddings. The embedding based classification showed gradual improvement in accuracy and f1 score with increasing proportion of training data (22. The highest accuracy and f1 score were 85% and 78% respectively at 100% training data proportion.

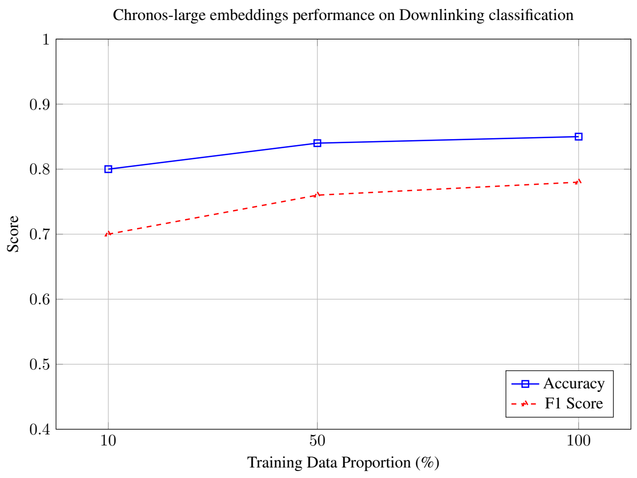

Figure 22: Impact of Training Data Proportion on Accuracy and F1 Score for Chronos-large embeddings on Downlinking classification. F1 scores are for the positive class

Chronos Full Fine Tuning Chronos-large model, accepting univariate input, adapted for classification was fully fine tuned i.e. all weights of the model were updated during training. The initial values of the weights in the model were kept the same as in the open source model and the classification head was randomly initialised. The training setup utilized the Adam optimizer with learning rate 0.0001. The scheduler would decrease the learning rate by a factor of 0.8 after every epoch. The batch size was 16. The model was trained for a maximum of 100 epochs, with an early stopping patience of 6 epochs for less than 1% improvement in validation F1 score. The loss function used was CrossEntropyLoss.

The full fine-tuning on Chronos-large with pre-trained weights yielded much improved performance 23 as compared to the embedding based classification. The accuracy and f1 score for pre-trained weight full model fine-tuning showed less sensitivity to training data proportion as compared to the embedding based classification.

Chronos Full Fine Tuning after Weights Reset In an attempt to assess the effect of using pre-trained weights, experiments were run with randomly initialised weights for full Chronos-large model fine-tuning 24 The training setup was the same as with the Chronos full fine tuning for downlinking with pretrained weights based initialisation. The random weight initialisation reduced both f1 scores and accuracy as compared to using pre-trained weights for full fine-tuning. The random weights initialisation based full fine-tuning yielded comparable results to the embeddings based classification method. Also, the sensitivity to proportion of training data used for training was noticeably higher than that on using pre-trained weights for full fine-tuning.

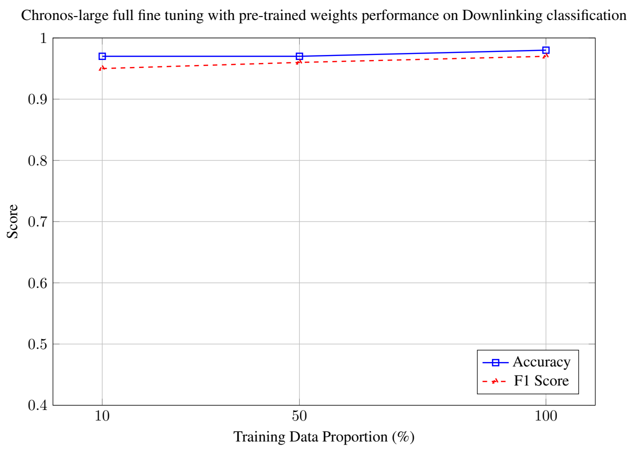

Figure 23: Impact of Training Data Proportion on Accuracy and F1 Score for Chronos-large full fine tuning with pre-trained weights on Downlinking classification. F1 scores are for the positive class.

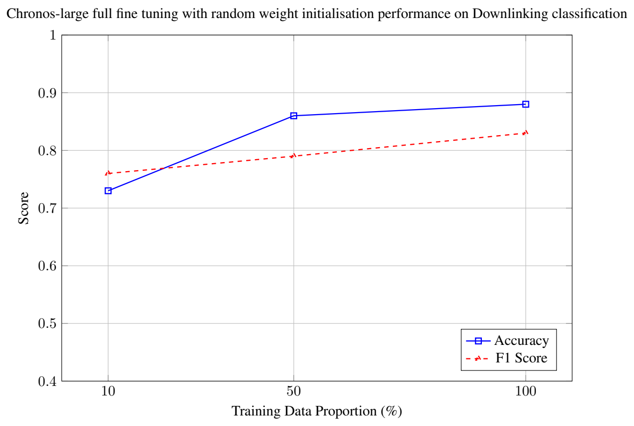

Figure 24: Impact of Training Data Proportion on Accuracy and F1 Score for Chronos-large full fine tuning with random weight initialisation on Downlinking classification. F1 scores are for the positive class

H.1.2 Running Casing

Chronos Embedding Experiment For running casing classification task, the first set of experiments were run using embeddings generated using the pre-trained Chronos-large model for six features (hole depth, flow rate, bit depth, stand pressure, block position and hook load) using 512 length windows. The batch size was 64 for generating the embeddings. The embeddings were summed on which an XGBoost classification model (with configuration mentioned before) was trained.

The accuracy and f1 scores improve with increasing percentage of training data 25. The highest f1 score was 88% at 100% training data.

Chronos Full Fine Tuning Similar to downlinking, all weights in the Chronos-large model for multivariate input classification weights were updated by training and the weights were initialised with pre-trained weights for the encoder and random initialisation for the classification head. The training setup utilized the Adam optimizer with learning rate 0.0001. The scheduler would decrease the learning rate by a factor of 0.8 after every epoch. The batch size was 8. The model was trained for a maximum of 100 epochs, with an early stopping patience of 6 epochs for less than 1% improvement in validation F1 score. The loss function used was BinearyCrossEntropyLoss (BCELoss). The Chronos-large full model fine tuning with pre-trained weights 26 showed less sensitivity to increasing training data proportion compared to the embedding based classification. This is similar to what was observed for downlinking use case. This could be implying that pre-trained weights are very

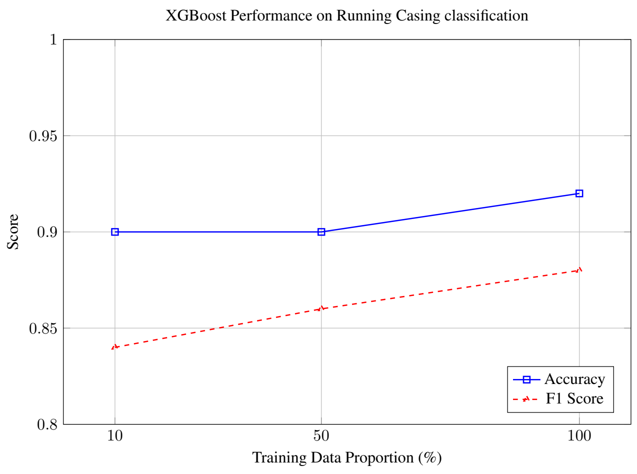

Figure 25: Impact of Training Data Proportion on Accuracy and F1 Score for Chronos-large embeddings on Running Casing classification. F1 scores are for the positive class

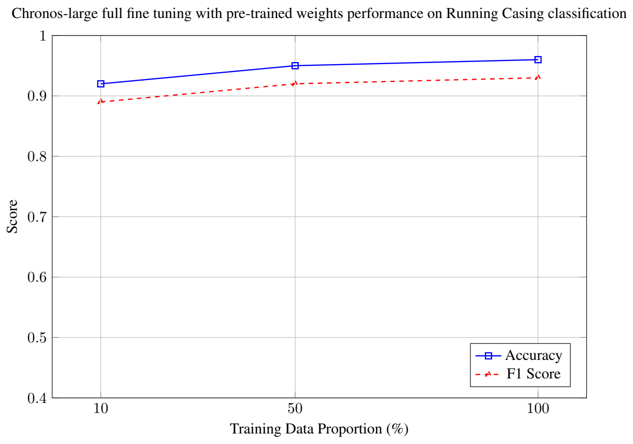

Figure 26: Impact of Training Data Proportion on Accuracy and F1 Score for Chronos-large full fine tuning with pre-trained weights on Running Casing classification. F1 scores are for the positive class

suitable for the two use cases mentioned. In the running casing case, the full model fine-tuning did not offer any large improvement over the embedding based approach unlike in the downlinking case. Therefore, simpler and time effective embedding based approach might be useful for some use cases.

Chronos Full Fine Tuning after Weights Reset The training configuration was identical to the full model fine-tuning with pre-trained weights for running casing use case. Similar to downlinking, the Chronos-large model was fully fine-tuned with random weights initialisation for the running casing use case. This approach yielded extremely poor results for running case in contrast with downlinking. No convergence was observed and the f1 scores were 0% for different training data proportions 27

I Appendix - Quantitative Results

I.1 Comparing Frozen Embeddings vs. Raw Data for Classification (XGBoost)

In this analysis, we compare two approaches for classification using XGBoost:

1. Training XGBoost directly on raw features.
2. Using final-layer embeddings from pretrained LM4TS as input to XGBoost.

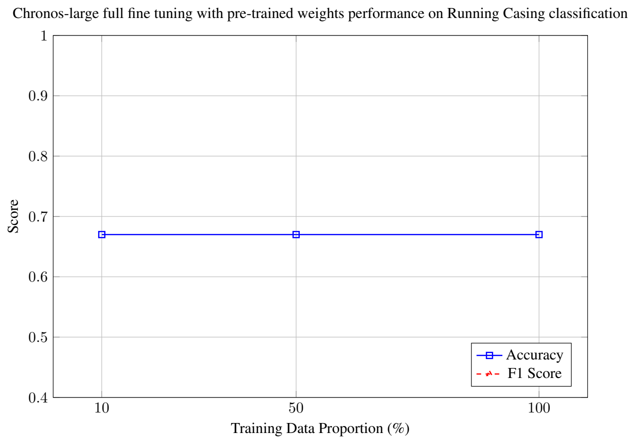

Figure 27: Impact of Training Data Proportion on Accuracy and F1 Score for Chronos-large full fine tuning with random weights initialisation on Running Casing classification. No convergence was observed. F1 scores are for the positive class

For the second approach, we first pass the raw features through a pretrained large model (e.g., Chronos, Moments) to obtain embeddings from the final layers. These embeddings are then used as input features for XGBoost classification. Importantly, the large models remain frozen during this process, serving only to transform the raw features into a potentially more informative representation.

This comparison allows us to assess whether the pretrained embeddings from large models offer any advantage over using the raw features directly for these specific classification tasks.

XGBoost, when trained on raw data, surpasses Chronos-Large in the downlinking task and outperforms all large models in the running casing task (see table 2). This suggests that large models and their pretraining approaches do not add more information to the representation.

Backbone	**Weights**	**Frozen**	**Classifier**	**Parameters (M)**	**Downl. F1 (%)**	**Casing F1 (%)**	**Inference Time (ms)**
Raw Data	-	No	XGBoost	-	61.3	92.0	**0.001**
Chronos	Pre-trained	Yes	XGBoost	710	46.0	90.0	40
Moments	Pre-trained	Yes	XGBoost	385	76.0	83.0	10

Table 2: Performance comparison of various models for downlinking and running casing classification tasks. The table presents F1 scores for each task, the number of model parameters, and inference times in milliseconds.

J Appendix - Qualitative results

J.1 Downlinking

J.1.1 Correct classifications across all models

Across the test set evaluation of all five downlinking classification models on over 8,700 windows, 58% were correct classified across all models. This suggests a moderate proportion of windows and downlinking signatures were easily classified. See figure 28 for examples of downlinking signatures that all models classified correctly.

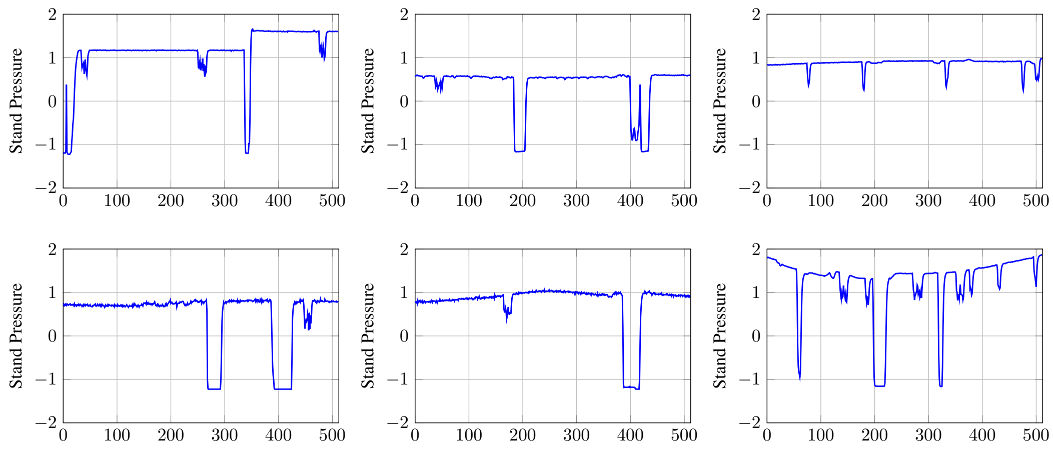

Figure 28: Examples of windows with downlinking signatures that all model predict correctly.

J.1.2 Wrong classification across all models

Only 0.5% of the windows were predicted wrongly by all models. Not much commonality was found between these windows. See figure 29 for an example in which the downlinking signal (pressure dip near the end) is short and not fluctuating.

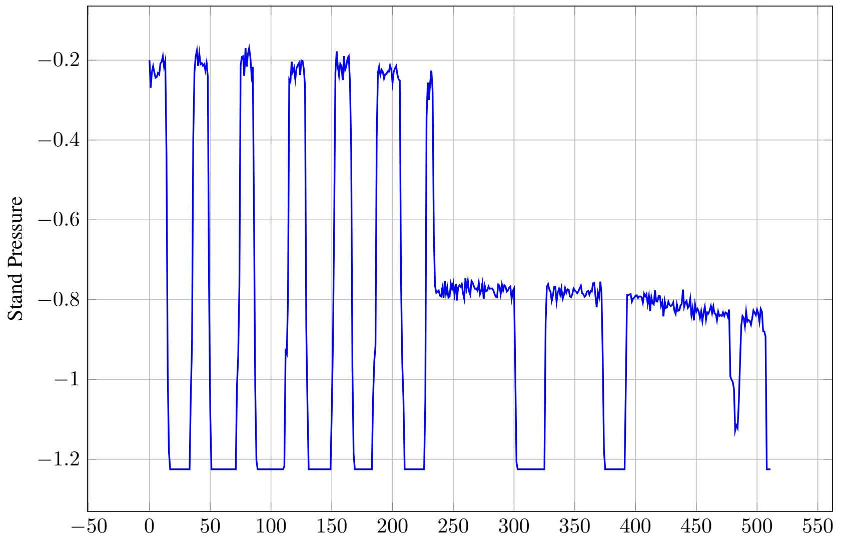

Figure 29: Example of a hard downlinking signature which all models classified incorrectly

J.1.3 Cases which only LM4TS got right

Only 0.54% of the windows in the test set only the LM4TS got right and the baselines got wrong (vice versa only 0.16% only all the baselines were right). Surprisingly, it seems that LM4TS are especially good at detecting downlinking signatures with relatively small drop in pressure and fluctuations (figure 30).

J.1.4 Cases only CNN got right

For 0.39% of the classified windows in the test set only the best baseline, CNN, got it right comparing to the large models. There are two commonalities found in those cases. First being windows which have signatures that look like downlinking, but are not. The CNN was successfully able to classify those as non-downlinking (see figure 31 left side). The second being windows which don't have some stable stand pressure to begin with, thus the stand pressure gradually moves over time, during which

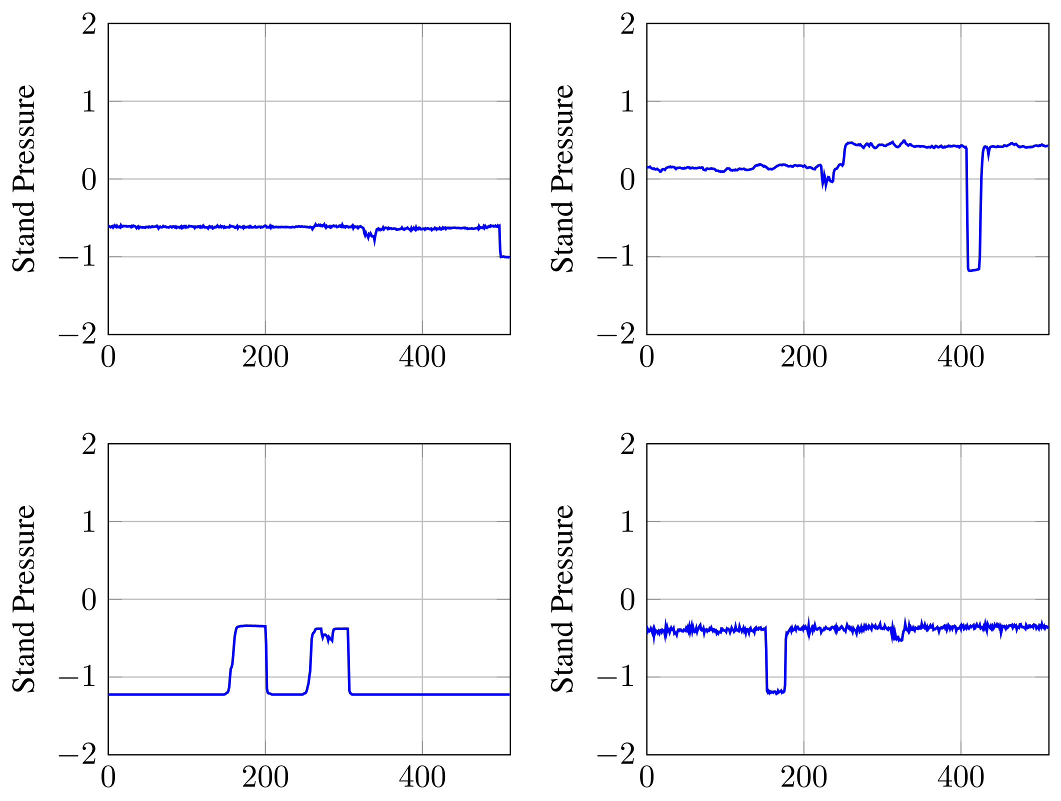

Figure 30: Examples of a downlinking signatures which only the large models classified correctly. Note the subtle fluctuations (downlinkings) near the middle of the plots for all three.

downlinkings are sent. Interestingly, the large models are not able to pick this up, but the CNN is (see figure 31 right side).

J.2 Running Casing

J.2.1 Correct classifications across all models

3233shows examples of correct classification by all models. The running casing exists near the middle of the window. The block position and hook load exhibit very distinct patterns during running casing as compared to other times in these examples.

J.2.2 Wrong classification across all models

34 show an example where all models made wrong prediction. The running casing event is at the beginning of the window. The block position patterns are very similar at the beginning and end of the window which could be the reason for the error.

J.2.3 Cases which only LM4TS got right

35 shows an example of running casing where only large models predicted correctly. The running casing event is at the end of the window which might be confusing the simpler models since they might not be capturing the long range temporal relationship across the whole window as effectively as large models.

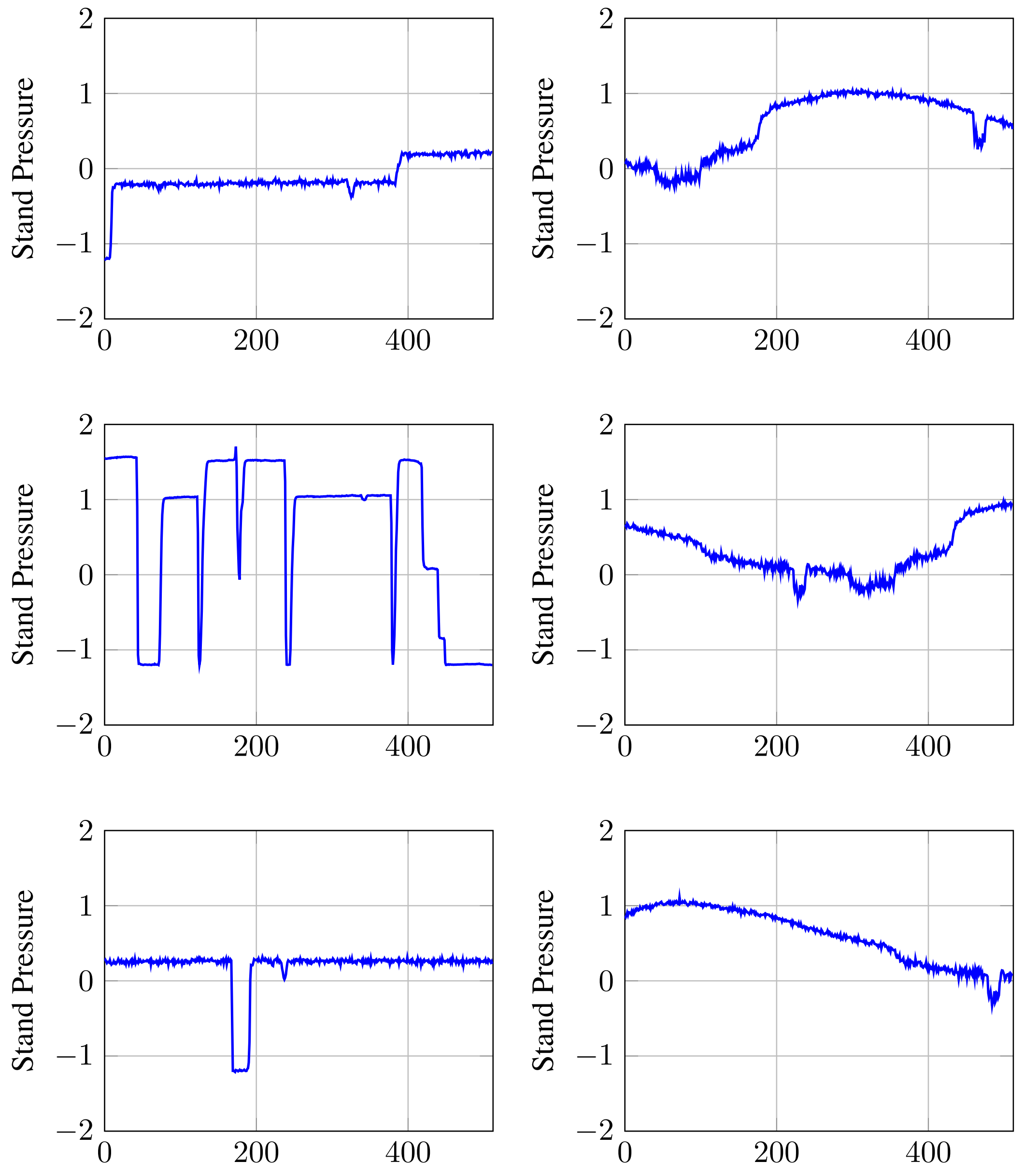

Figure 31: Examples of a downlinking signatures which only the CNN classified correctly. Left three graphs display windows with only a signature that have some resemblance to downlinking but is not. Right three graphs show windows with downlinking signatures as part of a continually changing pressure.

J.2.4 Case only CNN got right

36 shows an example of running casing where only large models predicted correctly. Again, the running casing event is at the end of the window. But this time the CNN captures the casing event but large models fail. This shows that CNN is also capable of capturing long time events.

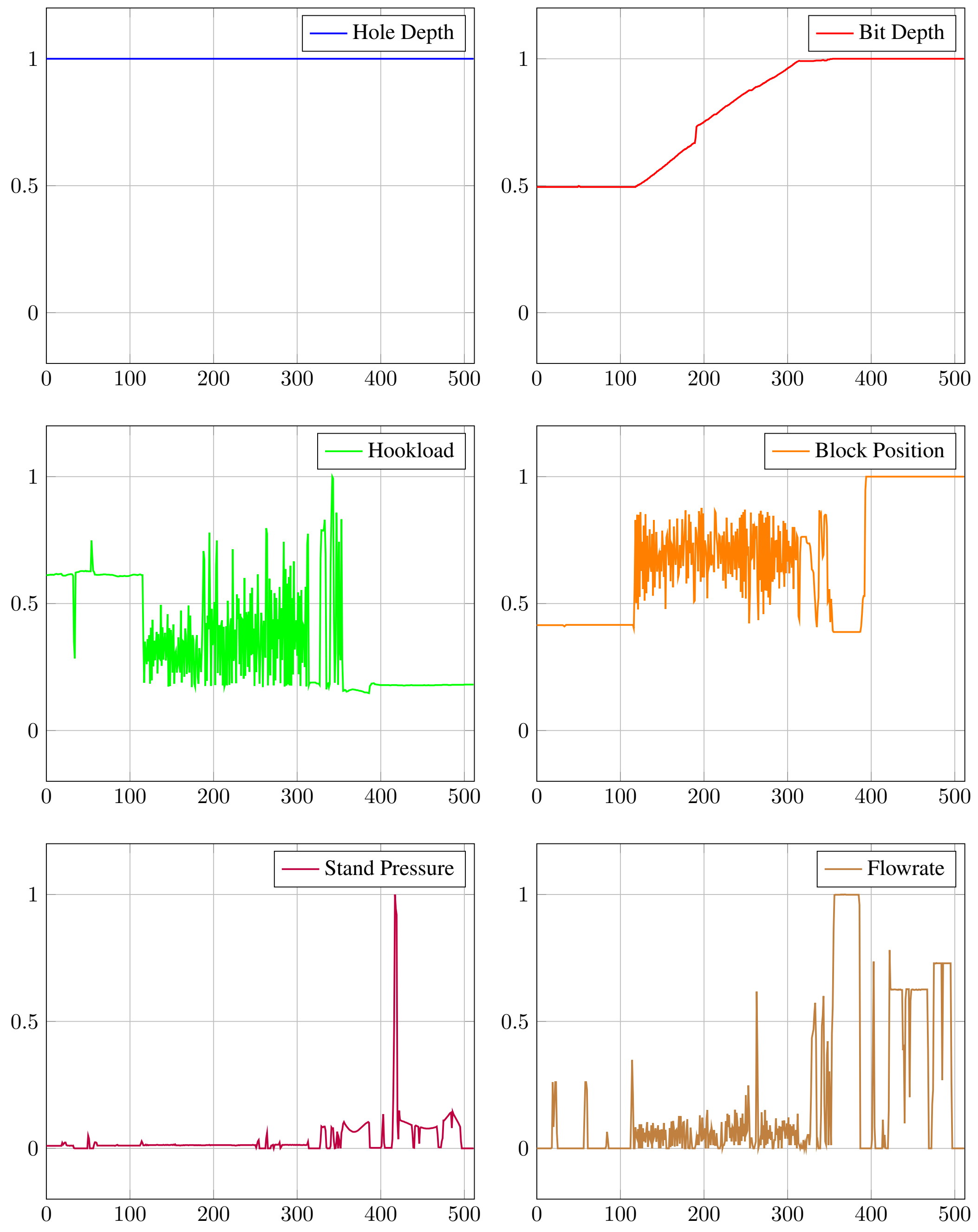

Figure 32: Example-1 of correct predictions by all models for running casing. Casing run extends from around step=100 upto step=300

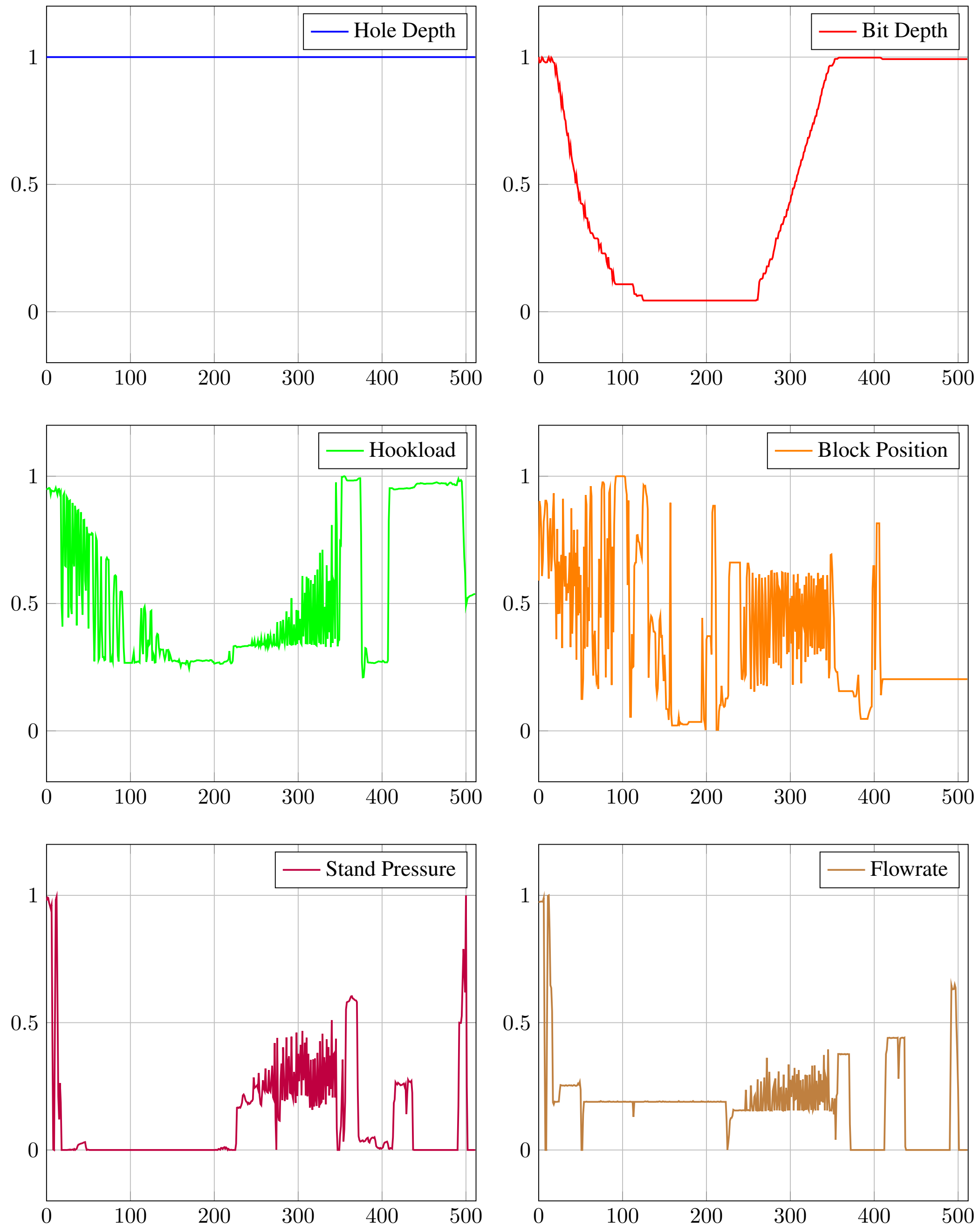

Figure 33: Example-2 of correct predictions by all models for running casing. Casing runnng extends from around step=280 upto step=340

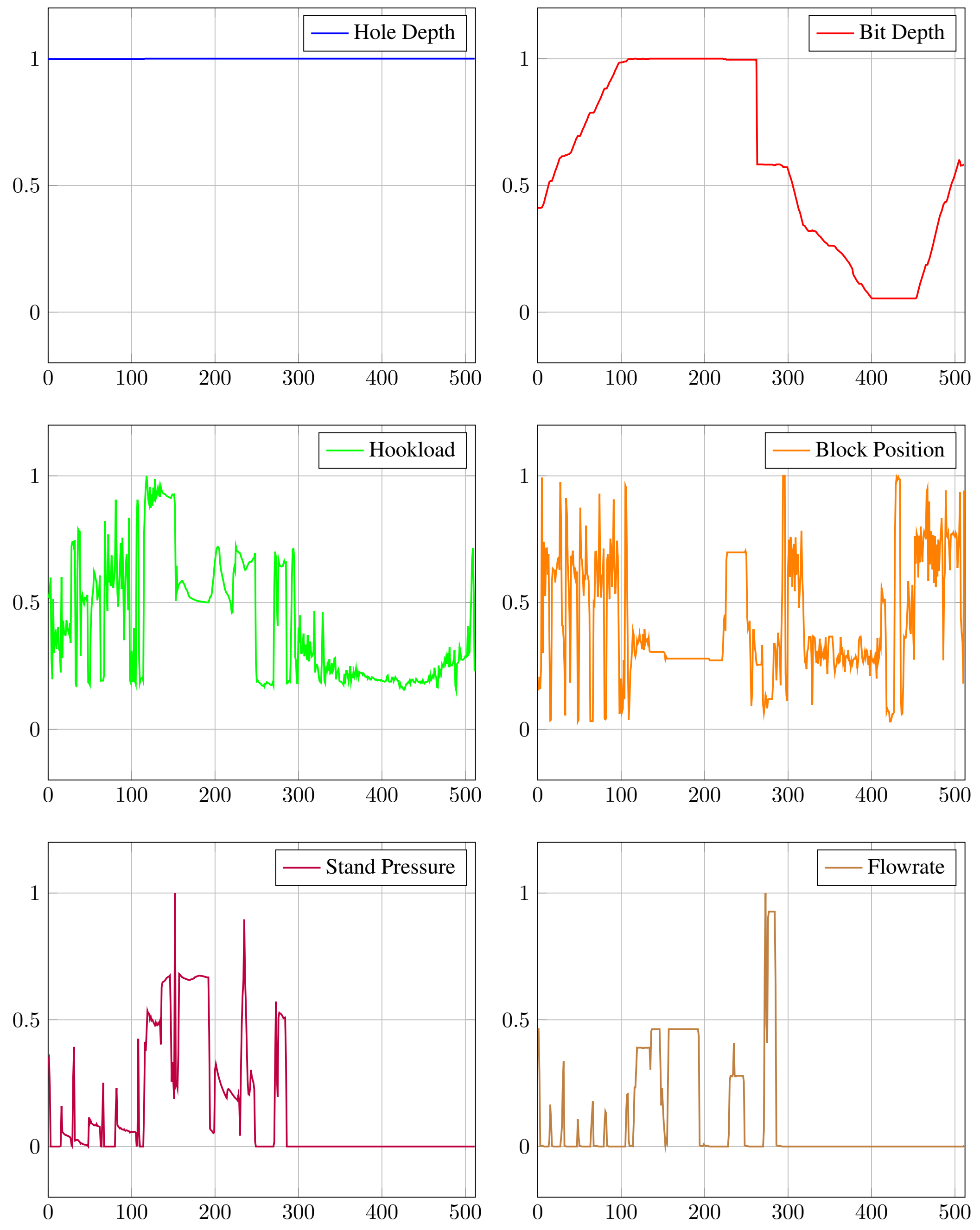

Figure 34: Example where all models made incorrect prediction for running casing. Casing running extends from around step=2 upto step=100

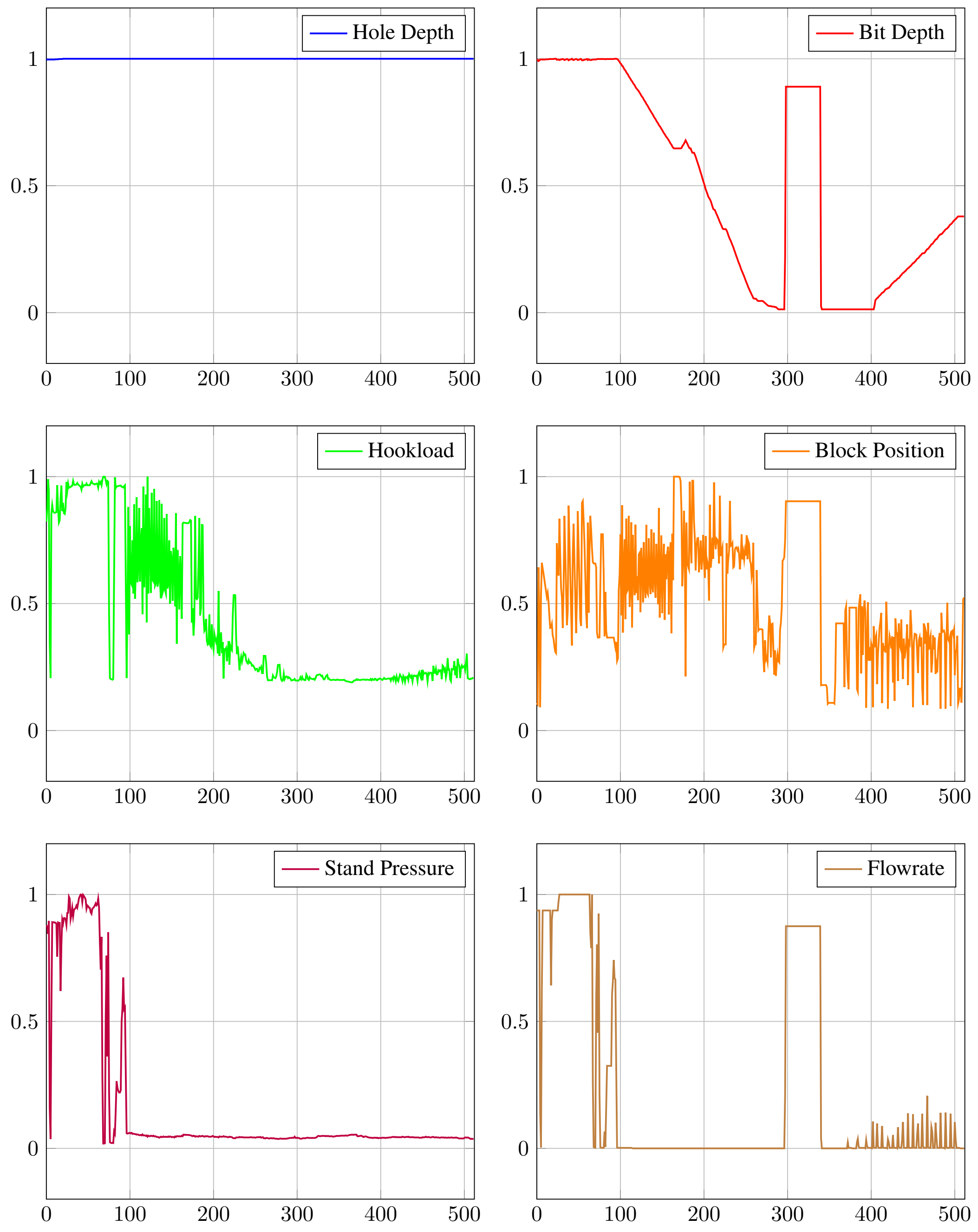

Figure 35: Example where only large models predicted correctly for running casing. Casing running extends from around step=400 upto step=500

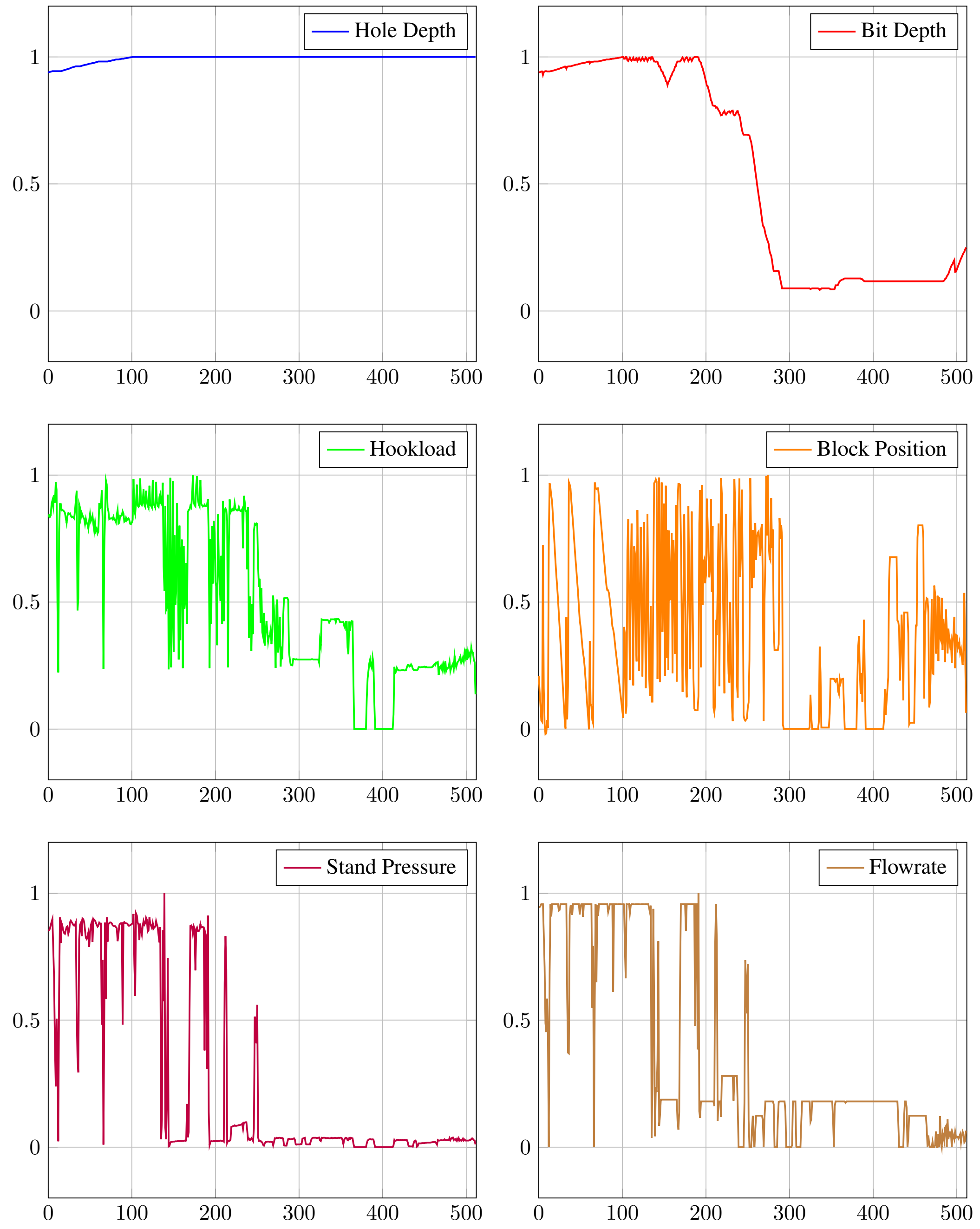

Figure 36: Example where only CNN predicted correctly for running casing. Casing running extends from around step=480 upto step=510