9. Supplementary material

9.1. *Adopted Deep Neural Network architectures*

We formulate sequence modelling as a regression task, i.e. lean to minimize the Euclidean distance between the true and the predicted values. Specifically, the sequence is a time series and the model needs to respect causality, i.e. in order to predict the output y_t for some time t, we can only use those inputs that have been previously observed $(x_{t-k}, ..., x_{t-1})$. The problem faced in prediction (discussed in 3.2) is supervised because given the input to the model, we know the output that it should produce and therefore we can compute gradient and train the network to turn out the right output. The forecasting explored in 3.3 is also a regression problem, but learn self-supervisedly since we aim to predict the future from the past: the model needs to learn data representations to solve the task.

We will briefly present here the theory behind the architecture adopted for this work.

9.1.1. *1D-CNN*

Convolutional neural network (CNN) is a class of deep and supervised models that was introduced for the first time by LeCun et al. in 1998 for processing data that has a grid-like topology (e.g. images): this first CNN was applied to digit recognition, using MNIST dataset. CNNs get a dominant class of deep learning methods after the ImageNet competition for image recognition (Krizhevsky et al., 2012), which has then become popular in the most varied applications. A typical architecture of a 2D convolutional network consists of a set of layers each of which contains several filters for detecting various features in the input image, the model performs convolutions using the chosen kernels and in doing this the procedure adds activation functions (i.e. activation when a feature is matched), constituting the so called feature map.
With 1D-CNN we can do the same for a 1-Dimensional input, e.g. a temporal series. We have a 1-Dimensional array in input and some 1-Dimensional kernels that we use to perform convolution and extract features, as in the 2D case. This is indeed the main

difference between 1D and 2D CNNs: 1D arrays replace 2D matrices for both kernels and feature maps. That result in a low computational complexity: $\mathcal{O}(k * n * d)$ for 1D CNNs, with respect to $\mathcal{O}(k * n * d^2)$ of 2D CNNs. Where k is the kernel size of the convolution, d is the representation dimension or embedding dimension of a word, n is the sequence length.

9.1.2. LSTM

In the past years, LSTM (Long Short-Term Memory network) (Hochreiter and Schmidhuber, 1997) has been successfully applied to a number of sequence model tasks, e.g. speech recognition, language modeling and translation, image captioning, trajectory forecasting and so on. In this work we apply it in geophysical problems.

LSTM it's a type of Recurrent Neural Network (RNN): RNNs are deep learning models that iteratively combines past informations with the present, to make them persist. Indeed, they have an “internal state” (hidden state) that can be seen as the memory: it is updated as a sequence is processed, by applying a recurrence formula at every time step, using function that combine the past information with the current input. In figure S4 left is represented the RNN if we unroll the loop. In figure S4 right is represented one iteration of the RNN, where:

$$
\begin{aligned}
h_t &= tanh(W_{hh}h_{t-1} + W_{xh}x_t) = \\
&= tanh(\begin{bmatrix} W_{hh}W_{xh} \end{bmatrix} \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix}) = \\
&= tanh(W \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix})
\end{aligned}
$$

here $tanh()$ is the non-linear function, W are the parameters, h_t and h_{t-1} are the hidden state at time t and $t-1$ and x_t is the input at time t.

LSTM works, for many tasks, much better than the RNN standard version. They were introduced in 1997 (Hochreiter and Schmidhuber, 1997), and were improved and popularized by many people in following works. The main problem of the standard RNNs is the difficulty to access information from many steps back. LSTM instead is explicitly designed to avoid the long-term dependency problem: they are capable of learning long-term dependencies, thanks to some internal mechanisms, called gates, that can regulate the flow of information. These gates can learn which data in a sequence are important to keep or throw away. Another important feature of LSTM is the Cell c_t that performs better in forward (direct connection with past element) and in backward (easy backward of the model and avoid gradient vanishing problem, that is a common problem of other RNNs).

Here there are two internal states: c_t and h_t that proceed in parallel, and represent respectively the long and the short term memory. There is a complex mechanism to manage memory, by using four gates:

- Input gate (i): whether to write to cell
- Forget gate (f): whether to erase cell
- Output gate (o): how much to reveal cell
- Gate gate (g): how much to write to cell

In Figure 4a is represented one iteration of the LSTM. Details are provided by Formula 1, where $tanh()$ and $\sigma = sigmoid()$ are the non-linear functions, W are the parameters, h_t and h_{t-1} are the hidden state at time t and $t-1$, c_t and c_{t-1} are the cell state at time t and $t-1$ and x_t is the input at time t. The "forget gate" say how much we should be forgetting about the previous cell information (c_{t-1} is the memory of our system) and then, once decided what to forget we would be modulating with an "input gate" how much we want to memorize from the current input x_t.

$$
\begin{bmatrix} i \\ f \\ o \\ g \end{bmatrix} = \begin{bmatrix} \sigma \\ \sigma \\ \sigma \\ tanh \end{bmatrix} W \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} \tag{1}
$$

$$
\begin{aligned}
& where: \\
& c_t = f \odot c_{t-1} + i \odot g \\
& h_t = o \odot tanh(c_t)
\end{aligned}
$$

To better understand the behavior of the memory, let's assume we are at time t, then the LSTM memory explicitly consider all the information from time $t-k$ to t. The best length of the long term memory (k) is not known a priori: we further analyse it in the dedicated section below 4.1.1.

Here the computational complexity is: $\mathcal{O}(n * d^2)$, where d is the representation dimension or embedding dimension of a word, n is the sequence length.

9.1.3. Transformer Network

This model was introduced in 2017 by Vaswani et al. (2017) and it was born mainly for common natural language processing, but nowadays is successfully used in a variety of different sequence modeling tasks (e.g. video, audio and so on). It has an encoder-decoder structure where the encoder maps an input sequence of symbol representations $(x_1, ..., x_n)$ to a sequence of continuous representations $\mathbf{z} = (z_1, ..., z_n)$. The decoder uses $\mathbf{z}$ to generates autoregressively an output sequence $(y_1, ..., y_m)$ of symbols. The Transformer Network (TF) is implicitly autoregressive, in that we use the predicted output in the input of the next step (auto means that it feeds its own prediciton). In particular, in order to let the transformer deal with the input, this is embedded onto a higher D'-dimensional space using a linear projection with a matrix of weights. In the same way, the output is a D”-dimensional vector prediction, which is back-projected to the original 1-D space. Differently from RNNs that receive one input at a time, TF receives all inputs one-shot. The TF uses a "positional encoding" to encode time for each past and future time instant t. Positional encoding is necessary to give an ordered context to the non-recurrent architecture of multi-head attention, because without it the model is permutation invariant. Sine/cosine functions are used to define positional encoding vector, that is: we represent the time in a sine/cosine basis.

The Transformer has 3 fundamental modules (attention, fully connected, residual connections). The attention modules are 2: self-attention and encoder-decoder attention. The encoder (Figure 4c, left) has six identical layers, where each layer has two sub-layers: a multi-head self-attention mechanism and a position-wise fully connected feed-forward network. All the outputs have a dimension of $d_{model} = 512$. The decoder (Figure 4c, right) has six identical layers and it has an additional layer in addition to the two sub-layers already described in the encoder: this performs multi-head attention over the output of the encoder stack. Decoder uses both self-attention and encoder-decoder attention, but the self-attention sub-layer in the decoder uses a masking mechanism to prevent positions from attending to subsequent positions. This ensures that the predictions for position t_i can depend only on the known outputs at positions before t_i. To start forecasting it uses a special token that indicate the start of the sequence. It is shown with <S> in 4c. The "Add & Norm" layers in figure 4c refer to Residual Connections (that sum the output of each layer with the input, to avoid vanishing gradient problem) and Layer Normalization.

The attention function maps a query and a set of key-value pairs to an output, where the query Q (dimension $d_N \times d_k$, where d_N is the number of element in the sequence and

d_k the latent dimension), keys K (dimension $d_N \times d_k$), values V (dimension $d_N \times d_v$), and output are all vectors. Q is related with what we encode (it can be output of encoder layer or decoder layer); K is related with what we use as input to output; V is related with input, as a result of calculations, and it is a learned vector. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2}$$

Instead of performing a single attention function, the model linearly project the queries, keys and values h times with different, learned linear projections to d_k, d_k and d_v dimensions, respectively, then performing the attention function in parallel for each projected query, key and value. This allows the model to jointly attend to information from different representation subspaces at different positions: those are the heads, and we need more than one because each of these capture specific characteristic of the features.
For TF the computational complexity is: $\mathcal{O}(n^2 * d)$, where d is the representation dimension or embedding dimension of a word, n is the sequence length.

9.1.4. Transformer Network not pretrained forecasting results

As explained in 4.2.4, TF is good in learning the aperiodicity and the singularities, however the common feature of all the experiments is the oscillatory behaviour of the signal. Without the pretraining with the sine wave, TF can't predict properly the target. Moreover TF is the most complex among the tested model in optimization and training and it requires a lot of data and computing to start working. We have quite small dataset though, that are not enough in training properly the model. As shown in Table S4, the results of TF not pretrained are always worst than the pretrained TF. Some windows of example for the three experiments are in Figure S5.

9.2. Networks training details

We train the model for 120 epochs in the case of prediction and for 30 epochs in the case of forecasting. In both cases we use the validation dataset to pick the best epoch and use it in testing phase, in order to avoid overfitting.

The size of each batch is 256 or 32 for prediction or forecasting, respectively.

As optimizer we use NAdam in the case of prediction: this is like Adam optimizer with the difference that it uses Nesterov momentum. In the case of forecasting we use Noam: this is like Adam optimizer with the difference that it increases the learning rate linearly for the first steps, and decreases it after that proportionally to the inverse square root of the step number (Vaswani et al., 2017).

Table S2 summarizes the number of data samples we have for each experiment, for training, validation and testing datasets. As explained in subsections 4.1.2 and 4.2.1, we adopt different dataset splits. This is because we use different frameworks (i.e. TensorFlow and PyTorch), which means different preimplemented functions. TensorFlow allows the user to select the validation part from the training data (so we take 10% from the 70% of the dataset used as training data). With PyTorch we can explicitly set train, val, test datasets sections (so we choose 70%, 10%, 20%, respectively). The reason why we use two different framework is that the model from LANL competition was in TensorFlow, then we keep this choice. Then we move to Pytorch for the forecasting part, since it's more straightforward and it makes model and functions editing easier.

9.3. Supplementary tables

	Exp p4581		Exp p5198		Exp p4679	
	Length	Shift	Length	Shift	Length	Shift
One-point prediction	1.0	0.1	1.0	0.1	0.01	0.003
Sequence forecasting	1.0	0.1	1.0	0.1	1.0	0.1

Table S1: Data preprocessing is done using overlapped moving windows to calculate statistical features from the original data. Here, "Length" refers to time in seconds of the windows length and "Shift" is the time in seconds by which windows are shifted. Acoustic data are recorded at 4 MHz, thus a 1 s window with a 0.1 s shift means that we produce 10 statistical features per second. We varied window size for each experiment and chose values that produced optimum results. "One-point prediction" refers to the first part of our work where we use LSTM+CNN model to predict one point at a time based on the prior signal variance. "Sequence forecasting" refers to the second part of the work where we use AR models (LSTM, TCN or TF) to forecast a sequence of values at future times in an auto-regressive fashion.

Dataset	Task	p4581	p5198	p4679
Train	Prediction	1829	1832	38237
	Forecasting	17300	18100	18000
Validation	Prediction	203	203	4248
	Forecasting	1900	2100	2000
Test	Prediction	902	903	19066
	Forecasting	3800	4100	4000

Table S2: Training, validation and testing dataset sizes. For prediction this is the number of datapoints, for forecasting this is the number of windows (each window includes past-input and future-output)

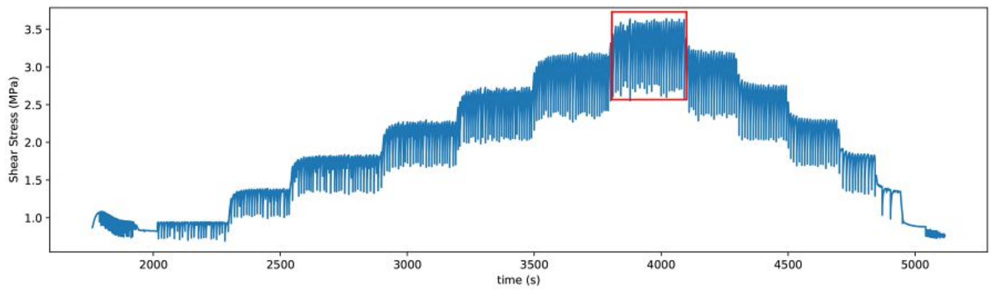
Target	R^2	p4581 Glass beads	p5198 Quartz powder	p4679 Quartz powder
Shear Stress	LSTM+CNN XGBoost	0.9254 0.73	0.9884 0.83	0.9574 —
Time To Start Of Failures	LSTM+CNN XGBoost	0.6317 —	0.9313 0.85	0.8229 0.70
Time To End Of Failures	LSTM+CNN XGBoost	0.8721 —	0.9697 —	0.9200 0.86

Table S3: Comparison between our results obtained with NN model vs. available results from the literature obtained with ML (XGBoost model) (Hulbert et al., 2018; Rouet-Leduc et al., 2017). For each target we show R^2, since RSME is not available from the literature. Our procedure outperforms the state-of-the-art in all the available occurrences.

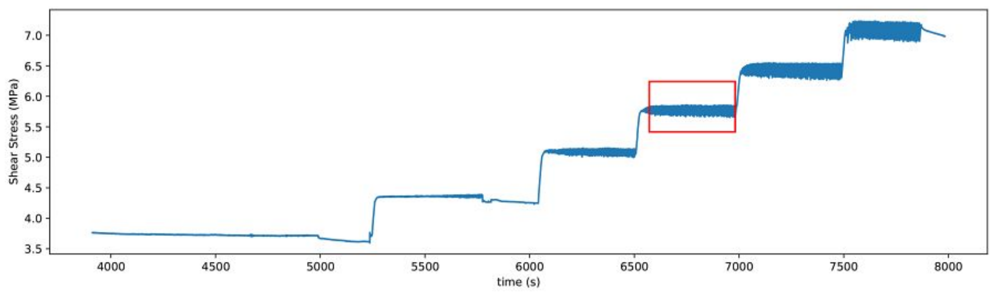
Model	GOF Material	p4581 Glass beads	p5198 Quartz powder	p4679 Quartz powder
TF pretrained	R^2 $RMSE$	0.1172 0.1460	0.8914 0.0707	0.7940 0.0738
TF not pretrained	R^2 $RMSE$	-0.3410 0.1510	0.6376 0.1247	0.6061 0.0997

Table S4: Experimental results for autoregressive forecasting. This is a comparison between the TF models, when pretrained and when not. The goodness of fit (GOF) is an average computed among all the tested windows. Figures for TF pretrained, together with all the tested models are in Figure 6. Illustration for TF is in Figure S5.

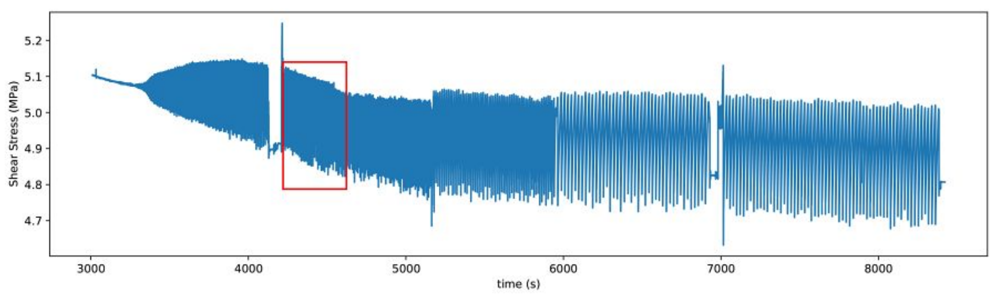
9.4. Supplementary Figures



(a) p4581



(b) p5198



(c) p4679

Figure S1: Full experiments, red box is for the subsection adopted in this work

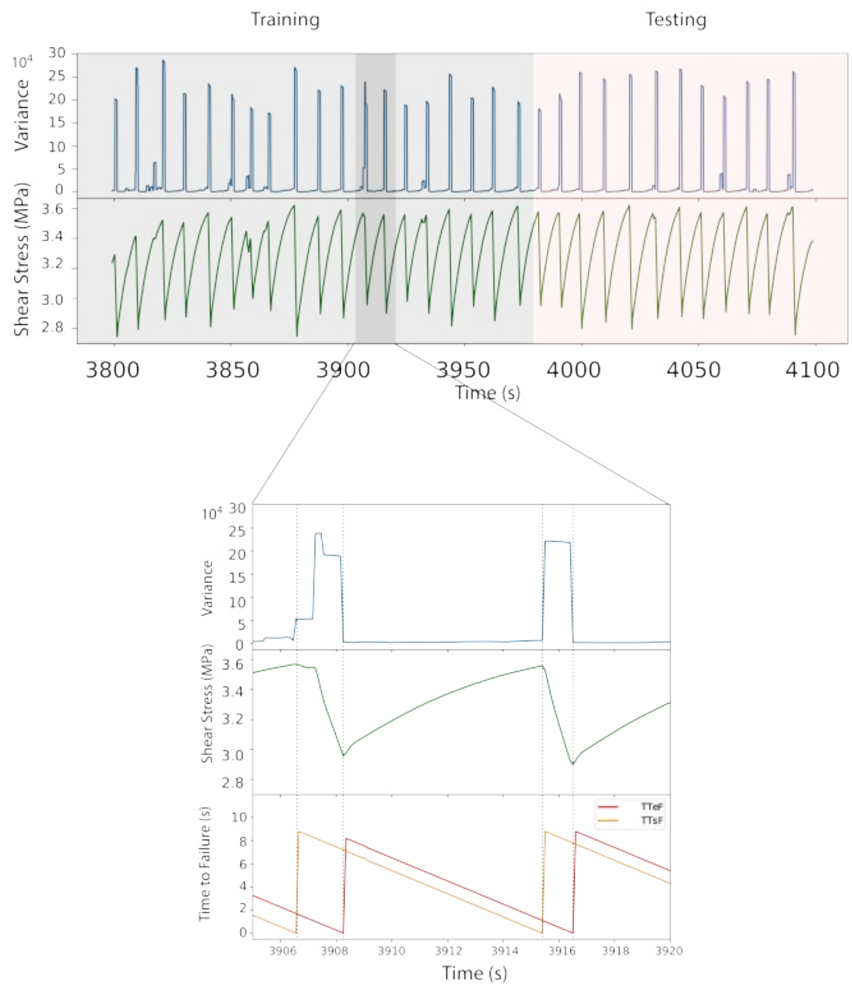


Figure S2: Signals' shape for experiment p4581, glass beads. For plot details see Figure 2

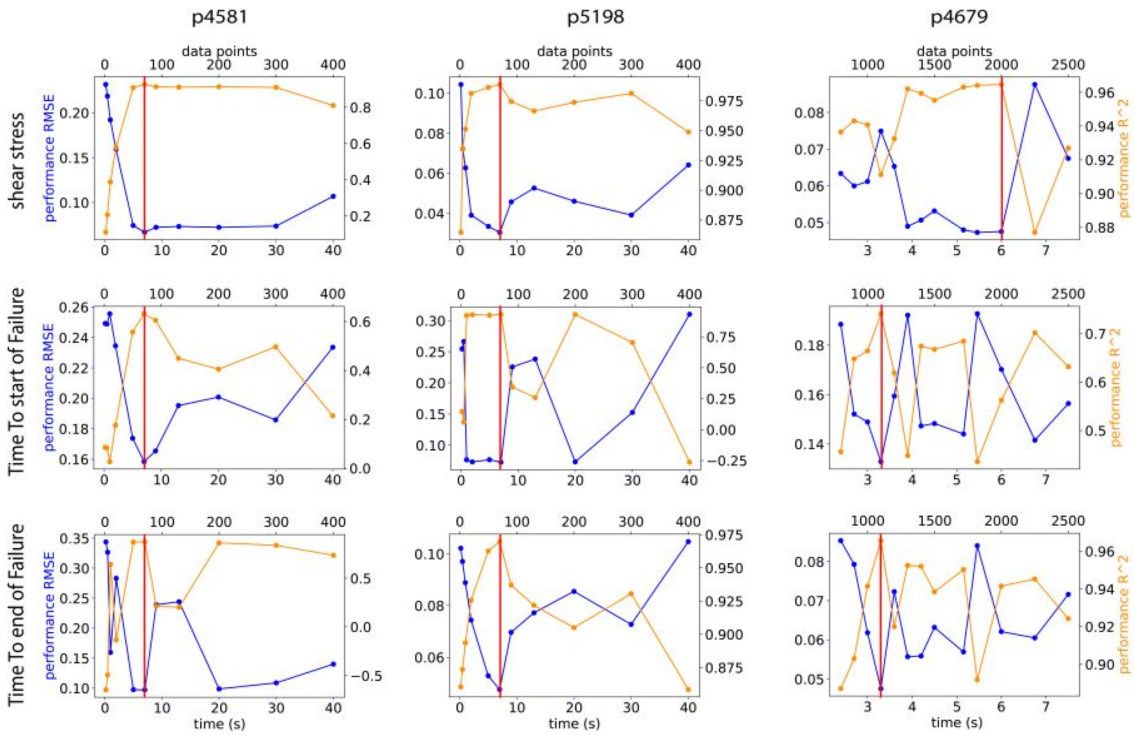


Figure S3: Variation in performance for different values of the LSTM memory length k. Each column shows results for one experiment. Red line shows the optimum memory length time. For experiments p4581 and p5198 the optimum is about $k = 70$, which corresponds more or less to one seismic cycle. For experiment p4679 the optimum value is $k = 2000$ when the target is shear stress while it is $k = 1100$ when target is TTF. Here, one seismic cycle is about 1700 data points.

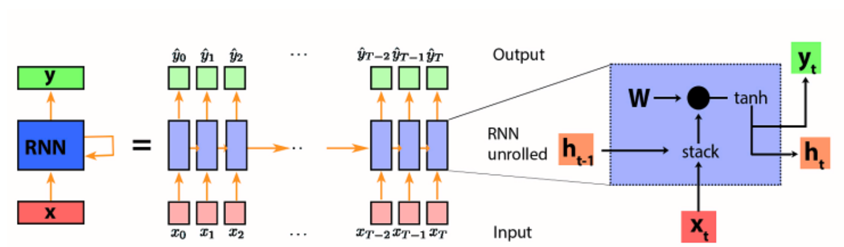


Figure S4: Recurrent neural network representation. The state-vector (h_t) is recursively updated at each t, including the new information coming from the current input.

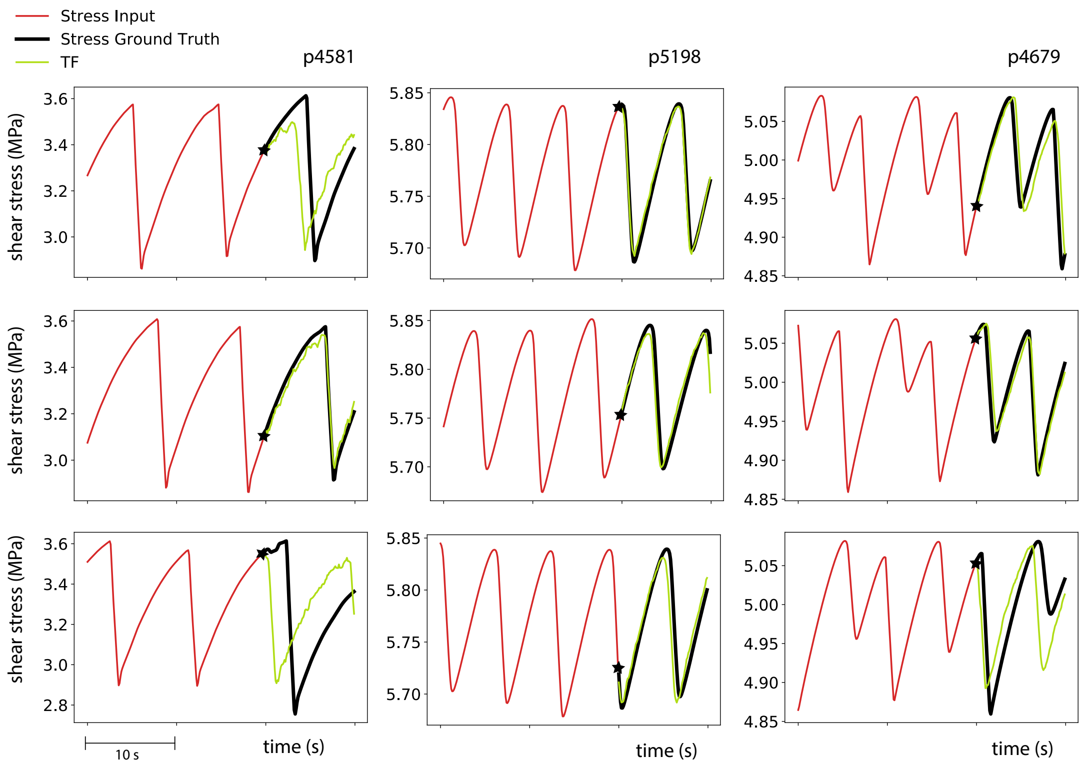


Figure S5: Results of forecasting models for the basic (not pretrained) Transformer Network. Each column shows a separate experiment. Red lines show input data and black lines show ground truth data for forecast testing. Green lines represent the output curves inferred from the model. X axis shows relative time, and Y axes are the target compared with model output. The results are not too bad for p5198 and p4679 and in general these results are worse than those for pretrained TF models.