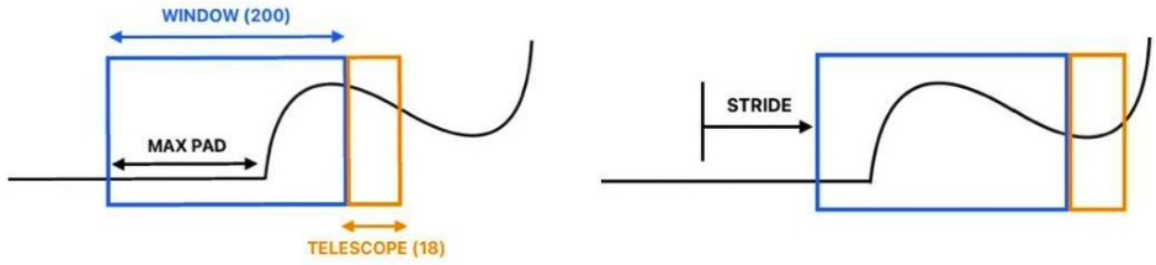
APPROACHING THE PROBLEM

Description of the problem:

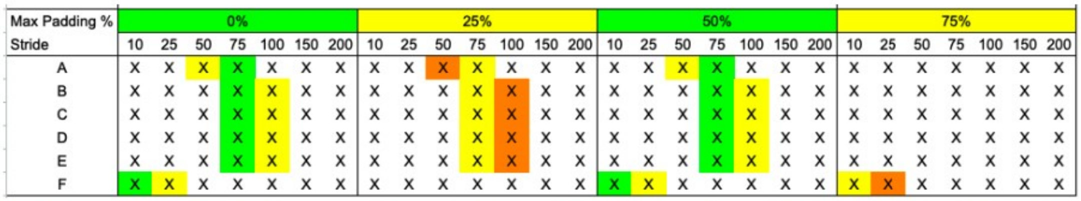
I implement a forecasting model to predict the behavior of several uncorrelated input time series. These are related to six different categories: A, B, C, D, E, F. It is available an initial dataset consisting of three numpy arrays. The first one contains 48000 time series of 2776 values between 0 and 1; the second specifies, for each of the time series, the start and end index, i.e. the part without padding; the last one provides the category of each time series.
The goal of the project is to build a model to predict the first 18 future samples of 60 time series of length 200, from a hidden test set. To do so, a hidden dataset is provided to test on the first 9 future samples intermediate models, before the choice of the best one.
It is worth noticing that the performance of our model is evaluated using the mean squared error, i.e. summing the squared residuals.

Approaching the problem:

The problem was approached step-by-step, developing a variety of models, which could fit the data better and better. I will now give an overview of the chronological steps followed on our way to the final model.
Part 1:
The very first step was loading, inspecting and visualizing raw data. It was straightforward to notice the wide variety of behaviors of the different categories. For example, it was evident how volatile where time series from category A with respect to more persistent ones from B. This led me to opt for a model capable of differentiating among different categories. How I implemented this idea will be explained below. The next step was data processing. For each category, and for each combination of stride (10, 25, 50, 75, 100, 150, 200) and maximum padding (0%, 25%, 50%, 75%), a pair of arrays was generated. These arrays, named ‘data’ and ‘out’, contain respectively input time series having length 200, i.e. the window of the problem, and output time series of length 18, i.e. the telescope of the model. How were these arrays generated? Let’s consider a stride of 75 and a maximum padding of 50%. For each time series provided, the last 18 time steps make the first output series; on the other hand, the previous 200 observations are the first input series. Because of a stride of 75, observations from 2753 to 2770 are the second output series and steps from 2553 to 2752 are the second input series. This procedure is reiterated until the last input time series has more than 50% of its observations different from zeros.
The illustrations below may clarify the generation of the clean datasets:



It is worth noticing that data was processed considering several combinations of strides and maximum paddings to find for each category the best-performing model.



Max Padding %	0%							25%							50%							75%						
Stride	10	25	50	75	100	150	200	10	25	50	75	100	150	200	10	25	50	75	100	150	200	10	25	50	75	100	150	200
A	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
B	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
C	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
D	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
E	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Table 1. Summary of all datasets created for each category. Green datasets have high priority to be tested, yellow medium, orange law.

The very next step was further normalizing time series. Each observation was adjusted by subtracting the median and dividing it by the interquartile range of points from the 'data' array (i.e. the difference between the 75% percentile and the 25% percentile). I applied the Robust Scaling method since this normalization is less affected by outliers. It must be noticed that data provided as a hidden test, before making predictions, has to be normalized using the training data median and iqr too. Results are eventually de-normalized before being returned by the 'predict' function.

Part 2:

I experimented with the new data files creating and training different models. The first architectures were sequences of LSTM and Conv1D layers, designed from scratch and varying parameters and hyperparameters at each training. Nonetheless, all these networks presented two common layers at the end: a Conv1D and a Cropping1D layer. The first one was added to match the corresponding output shape, meaning a depth of one. The second to crop the output to the desired time-step length, which in my case is 18. In fact, I decided to design my model to predict 18 timesteps, cropping them to 9 subsequently in the predict function, for the submissions of phase 1 of the challenge. In this way, phase 1 models can be easily adapted to the requests of phase 2 with minimal code modification.

For efficiency reasons, a first block of code containing global variables was added and modified as needed to manage the setting of data, model and training parameters. The submission procedure was also automated so that each model, after training, together with the metadata file and the model.py, was saved in a folder on the drive, ready for testing.

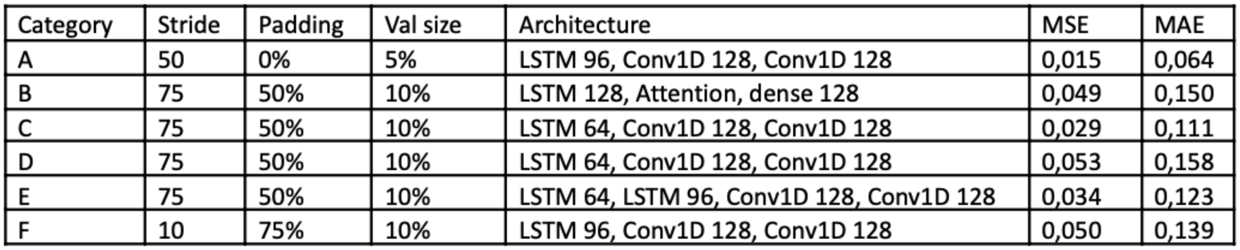
It is worth highlighting that I was not interested in finding a unique model exhibiting forecasting capabilities in all categories, but the best-performing model for each categorical domain.

After playing with these architectures, I tried to include in my networks resblock and resblock_lstm layers *[resblock and resblock_lstm are here called layers since they are in the code keys of the dictionary layers. Actually, they are both sequences of layers, i.e. blocks. For details on implementation, see section Details on Implemented Layers]*. From a theoretical point of view, these layers which are inspired by the classical Residual Neural Networks could be suitable for time series forecasting since they allow me to create models that learn to predict the next element of a sequence of inputs as the variation of the previous timesteps. The residual blocks are composed of sequences of convolutional layers (Conv1D) or LSTMs respectively, batch normalizations and activation functions, repeated as many times as selected by the hyperparameter 'units'. Moreover, there exists a shortcut connection between the extreme layers of the block so that a copy of the input is propagated through a skip connection up to the end of the block to be added to the other processed input.

Unfortunately, neither of these layers helped significantly improve the MSE of the trained networks. Hence, they were excluded from the final model.

Another layer I tried to use in standard architectures is an attention layer. The basic intuition behind the attention mechanism is to identify which are the most relevant parts of the input sequence on which my model should focus to predict the output. In my model, I decided to implement an attention layer ourselves defining the new class attention, derived from the parent Keras class Layer. The attention layer was implemented according to Bahdanau's method *[For details on implementation, see section Details on Implemented Layers]*. It is worth noticing that this attention layer generated relevant results mainly for categories B and D. The MSE of category E was just slightly improved. A possible reason for it, for category D, could be its quite repetitive patterns.

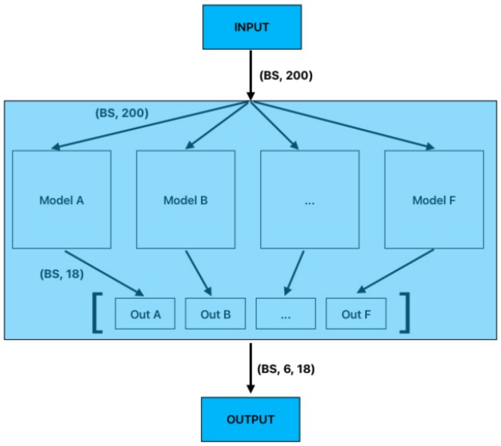
By now, the best models for each category are summarized in the table below:



Category	Stride	Padding	Val size	Architecture	MSE	MAE
A	50	0%	5%	LSTM 96, Conv1D 128, Conv1D 128	0,015	0,064
B	75	50%	10%	LSTM 128, Attention, dense 128	0,049	0,150
C	75	50%	10%	LSTM 64, Conv1D 128, Conv1D 128	0,029	0,111
D	75	50%	10%	LSTM 64, Conv1D 128, Conv1D 128	0,053	0,158
E	75	50%	10%	LSTM 64, LSTM 96, Conv1D 128, Conv1D 128	0,034	0,123
F	10	75%	10%	LSTM 96, Conv1D 128, Conv1D 128	0,050	0,139

Table 2. In all these models, training parameters are: epochs 400, batch size 64, learning rate 0,001, early stopping patience 120, plateau patience 100.

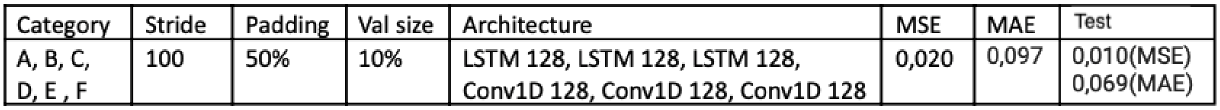
Since I'm requested to generate a unique model, the six categorical models above have been merged into a unique final model. To do so, the input layer, identical among the six architectures, is linked in a parallel way to the characterizing layers of each of the six networks. The output layer of this model is therefore a list of the output layers of the networks. This implies that the predictions on an input time series of a given category are the corresponding element in the list output_layer of the model. The illustration on the right provides the intuition on how these models were merged. The model explained above is my best model in phase 1, but not the final best.



Part 3:

In this part, I tried to build models not for individual categories but for a unique dataset consisting of time series for A, B, C, D, E and F. I opted for considering quite large strides (50, 75, 100) and the standard first implemented architectures of LSTMs and Conv1Ds, having to deal with many more sequences of data. Some of these models, when tested, achieved better performances than my best part 2 model, making me realize, even though quite late, that the subdivision of data in categorical domains to be trained separately was not helpful after all.

The network that obtained the best MSE and MAE (mean absolute error) when trained and resulted to be my best model in phase 2 is:



Category	Stride	Padding	Val size	Architecture	MSE	MAE	Test
A, B, C, D, E , F	100	50%	10%	LSTM 128, LSTM 128, LSTM 128, Conv1D 128, Conv1D 128, Conv1D 128	0,020	0,097	0,010(MSE) 0,069(MAE)

Table 3. Training parameters are: epochs 400, batch size 64, learning rate 0,001, early stopping patience 120, plateau patience 100.

Conclusion: To conclude, I would like to point out that, although the performance of my model isn't exceptional, this challenge was the chance to make mistakes and work to solve them. I approached the problem by implementing different strategies: from data generation to network design. It was an extremely formative project.

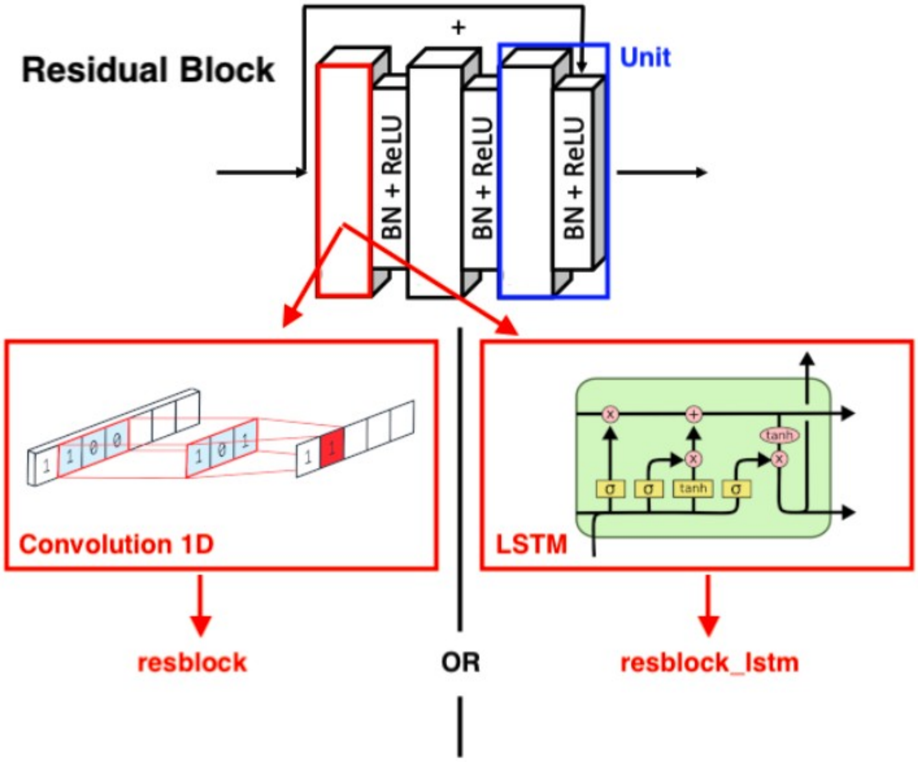
References:

How to Choose the Best Model for Time Series Forecasting: ARIMA, Prophet, or mSSa (ikigailabs.io)
le reti residue
1611.06455.pdf (arxiv.org)
Previsione di serie temporali | TensorFlow Core
Time Series Prediction with LSTM Recurrent Neural Networks in Python with Keras - MachineLearningMastery.com
Robust Scaling: Why and How to Use It to Handle Outliers | Proclus Academy
sklearn.preprocessing.RobustScaler — scikit-learn 1.3.2 documentation
The Attention Mechanism from Scratch - MachineLearningMastery.com
Attention layer (keras.io)
https://proclusacademy.com/blog/robust-scaler-outliers/
https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html#sklearn.pre processing.RobustScaler.inverse_transform https://machinelearningmastery.com/adding-a-custom-attention-layer-to-recurrent-neural-network-in-keras/
https://machinelearningmastery.com/the-attention-mechanism-from-scratch/
https://keras.io/api/layers/attention_layers/attention/
https://www.tensorflow.org/api_docs/python/tf/keras/layers/Attention
https://arxiv.org/abs/1512.03385
https://arxiv.org/pdf/1611.06455.pdf https://stackoverflow.com/questions/63284471/tensorflow-use-model-inside-another-model-as-layer https://keras.io/examples/timeseries/
https://keras.io/examples/timeseries/timeseries_traffic_forecasting/
https://keras.io/examples/timeseries/timeseries_weather_forecasting/
https://www.tensorflow.org/tutorials/structured_data/time_series?hl=it
https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/
https://www.ikigailabs.io/blog/how-to-choose-the-best-model-for-time-series-forecasting-arima-prophe t-or-mssa

Details on Implemented Layers:

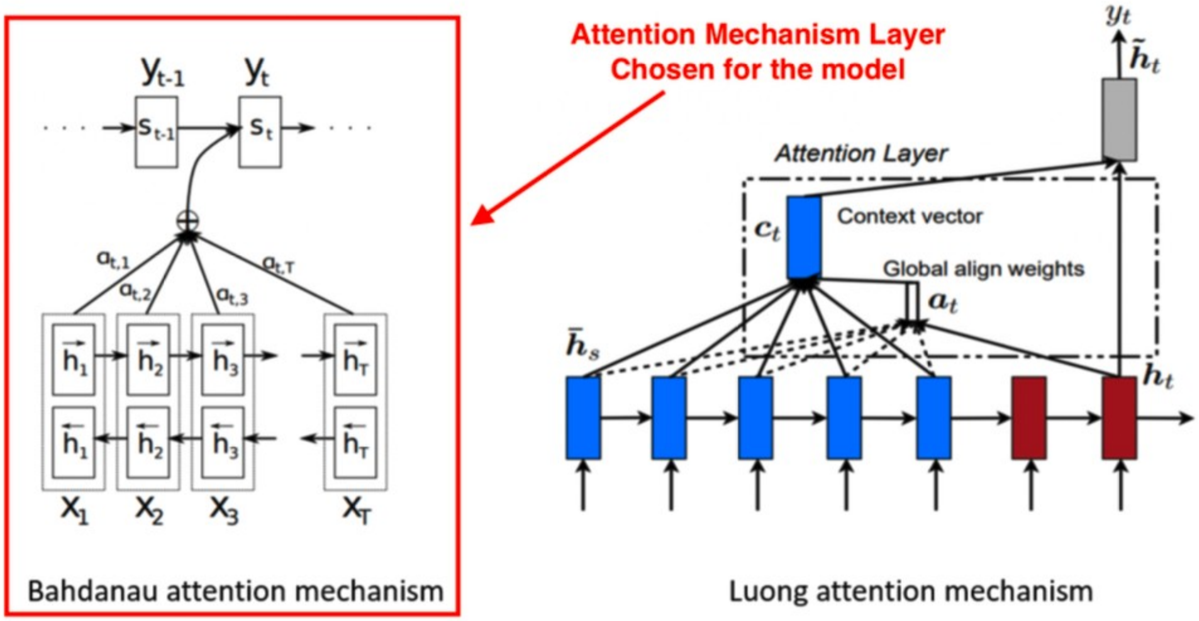
resblock and resblock_lstm layers:

From a theoretical point of view, these layers which are inspired by the classical Residual Neural Networks could be suitable for time series forecasting since they allow me to create models that learn to predict the next element of a sequence of inputs as the variation of the previous timesteps. Let's start with resblock, whose architecture is inspired by the following paper: 'Time Series Classification from Scratch with Deep Neural Networks: A Strong Baseline', Zhiguang Wang, Weizhong Yan, Tim Oates, 14 December 2016. The residual block is composed of a sequence of convolutional layers (Conv1D), batch normalizations and activation functions repeated as many times as selected by the hyperparameter 'units'. Moreover, there exists a shortcut connection between the extreme layers of the block. To be precise, the input is duplicated before entering each block: one instance goes through the whole block, whereas the other copy is propagated through a skip connection up to the end of the block to be added to the previous one. Before being added, the input copy is adjusted to match the last dimension of the processed one, changed according to the number of filters of the convolutional layers. This is done by another convolutional layer with a unitary kernel and a number of filters equal to those of the layers inside the block. In my code, an additional block is implemented: the resblock_lstm. This is identical to resblock, apart from having as first layer of each unit of the block a LSTM instead of a Conv1D. The reason behind the use in my models of this changed resblock was trying to exploit the long-term memory effect of the LSTM.



Bahdanau's method for implementing the attention layer:

This layer has a vector of weights W and a bias vector b, randomly initialized from a normal distribution and from a vector of zeros, respectively, and then modified during the training phase through backpropagation. First, I compute the expression W * X + b (where X is the input); then the hyperbolic tangent is applied to get the so-called Alignment scores; finally, the softmax of the Alignment scores is calculated and multiplied by the input X to get the context vector, which is the final output of our attention layer.



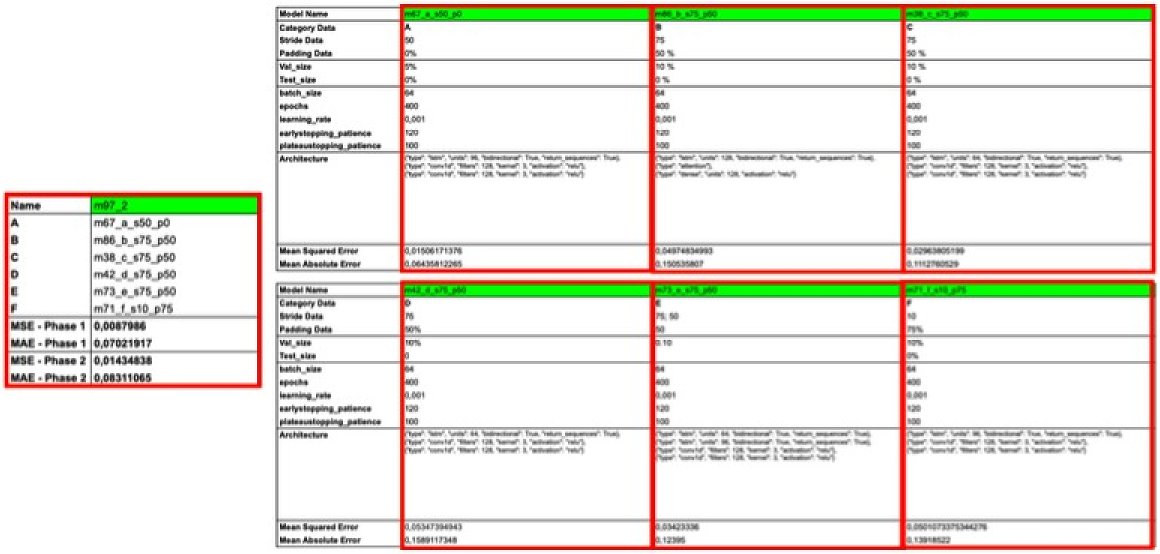
Attention Mechanism Layer
Chosen for the model

Bahdanau attention mechanism

Luong attention mechanism

Final Best models:

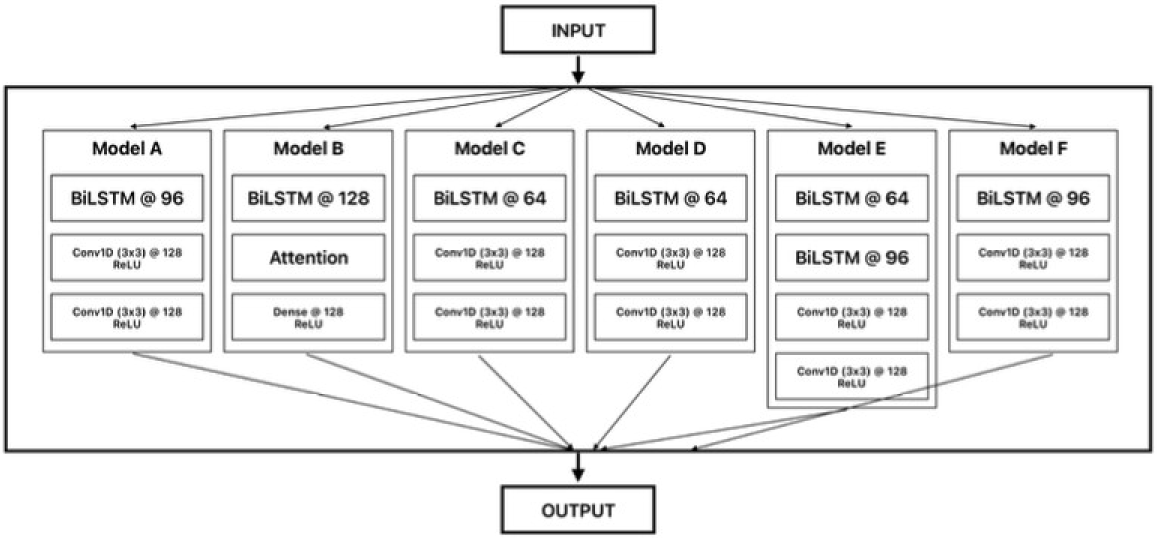
PHASE 1



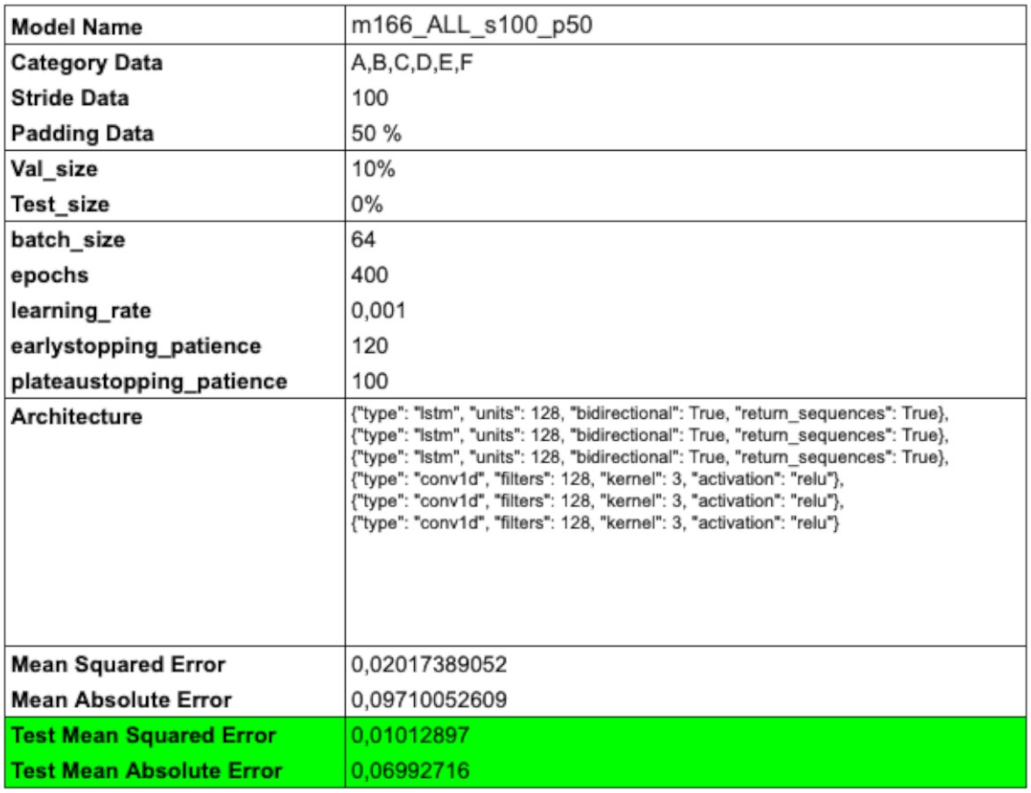
Name	m97_2
A	m67_a_s50_p0
B	m86_b_s75_p50
C	m38_c_s75_p50
D	m42_d_s75_p50
E	m73_e_s75_p50
F	m71_f_s10_p75
MSE - Phase 1	0,0087986
MAE - Phase 1	0,07021917
MSE - Phase 2	0,01434838
MAE - Phase 2	0,08311065

Model Name	m67_a_s50_p0	m86_b_s75_p50	m38_c_s75_p50
Category Data	A	B	C
Stride Data	50	75	75
Padding Data	0%	50 %	50 %
Val_size	5%	10 %	10 %
Test_size	0%	0 %	0 %
batch_size	64	64	64
epochs	400	400	400
learning_rate	0,001	0,001	0,001
earlystopping_patience	120	120	120
plateaustopping_patience	100	100	100
Architecture	{"type": "lstm", "units": 96, "bidirectional": True, "return_sequences": True}, {"type": "conv1d", "filters": 128, "kernel": 3, "activation": "relu"}, {"type": "conv1d", "filters": 128, "kernel": 3, "activation": "relu"}	{"type": "lstm", "units": 128, "bidirectional": True, "return_sequences": True}, {"type": "attention"}, {"type": "dense", "units": 128, "activation": "relu"}	{"type": "lstm", "units": 64, "bidirectional": True, "return_sequences": True}, {"type": "conv1d", "filters": 128, "kernel": 3, "activation": "relu"}, {"type": "conv1d", "filters": 128, "kernel": 3, "activation": "relu"}
Mean Squared Error	0,01506171376	0,04974834993	0,02963805199
Mean Absolute Error	0,06435812265	0,150535807	0,1112760529

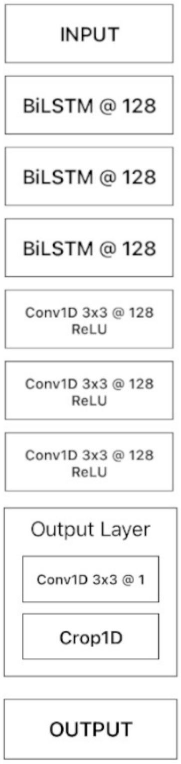
Model Name	m42_d_s75_p50	m73_e_s75_p50	m71_f_s10_p75
Category Data	D	E	F
Stride Data	75	75; 50	10
Padding Data	50%	50	75%
Val_size	10%	0.10	10%
Test_size	0		0%
batch_size	64	64	64
epochs	400	400	400
learning_rate	0,001	0,001	0,001
earlystopping_patience	120	120	120
plateaustopping_patience	100	100	100
Architecture	{"type": "lstm", "units": 64, "bidirectional": True, "return_sequences": True}, {"type": "conv1d", "filters": 128, "kernel": 3, "activation": "relu"}, {"type": "conv1d", "filters": 128, "kernel": 3, "activation": "relu"}	{"type": "lstm", "units": 64, "bidirectional": True, "return_sequences": True}, {"type": "lstm", "units": 96, "bidirectional": True, "return_sequences": True}, {"type": "conv1d", "filters": 128, "kernel": 3, "activation": "relu"}, {"type": "conv1d", "filters": 128, "kernel": 3, "activation": "relu"}	{"type": "lstm", "units": 96, "bidirectional": True, "return_sequences": True}, {"type": "conv1d", "filters": 128, "kernel": 3, "activation": "relu"}, {"type": "conv1d", "filters": 128, "kernel": 3, "activation": "relu"}
Mean Squared Error	0,05347394943	0,03423336	0,0501073375344276
Mean Absolute Error	0,1589117348	0,12395	0,13918522



PHASE 2



Model Name	m166_ALL_s100_p50
Category Data	A,B,C,D,E,F
Stride Data	100
Padding Data	50 %
Val_size	10%
Test_size	0%
batch_size	64
epochs	400
learning_rate	0,001
earlystopping_patience	120
plateaustopping_patience	100
Architecture	{"type": "lstm", "units": 128, "bidirectional": True, "return_sequences": True}, {"type": "lstm", "units": 128, "bidirectional": True, "return_sequences": True}, {"type": "lstm", "units": 128, "bidirectional": True, "return_sequences": True}, {"type": "conv1d", "filters": 128, "kernel": 3, "activation": "relu"}, {"type": "conv1d", "filters": 128, "kernel": 3, "activation": "relu"}, {"type": "conv1d", "filters": 128, "kernel": 3, "activation": "relu"}
Mean Squared Error	0,02017389052
Mean Absolute Error	0,09710052609
Test Mean Squared Error	0,01012897
Test Mean Absolute Error	0,06992716



INPUT

BiLSTM @ 128

BiLSTM @ 128

BiLSTM @ 128

Conv1D 3x3 @ 128 ReLU

Conv1D 3x3 @ 128 ReLU

Conv1D 3x3 @ 128 ReLU

Output Layer

Conv1D 3x3 @ 1

Crop1D

OUTPUT