Workflow prediction on Alibaba Cluster Trace 2023

This project compares three architectures for time series prediction on the 2023 Alibaba Custerdata: a plain LSTMs, an Encoder-Decoder architecture, and an Encoder-Decoder architecture with attention module.

1 DATA DESCRIPTION AND VISUALIZATION

The 2023 Alibaba clusterdata can be obtained here. The dataset consists of 8152 tasks submitted to the GPU cluster of Alibaba's Cloud center. The tasks are specified by the requested resources, in *number of CPUs*, *number of GPUs*, and *amount of memory*. Further specifications are *Quality of Service*, *specific GPU*s requested, *pod phase* (specified as Succeeded, Running, Pending, Failed), and c*reation/deletion/scheduled* times. From this data, six six time series were obtained: *overall CPU*, *overall GPU*, and *overall memory usage* for a given time step, as well as the CPU, GPU, and memory requirements of pods that *started* at that time step (figure 1). Each time step is a millisecond starting from Jan 1st 2023.

A significant change in workload and frequency of newly arriving pods is observable after day 115. Therefore, for the time series prediction task, introduced in the next section, only the data after the 115th day was used. This data was collected into intervals of 2 minutes, and a moving average was calculated using a window of 15 days for both the collective and the newly scheduled requirements (figure 2).

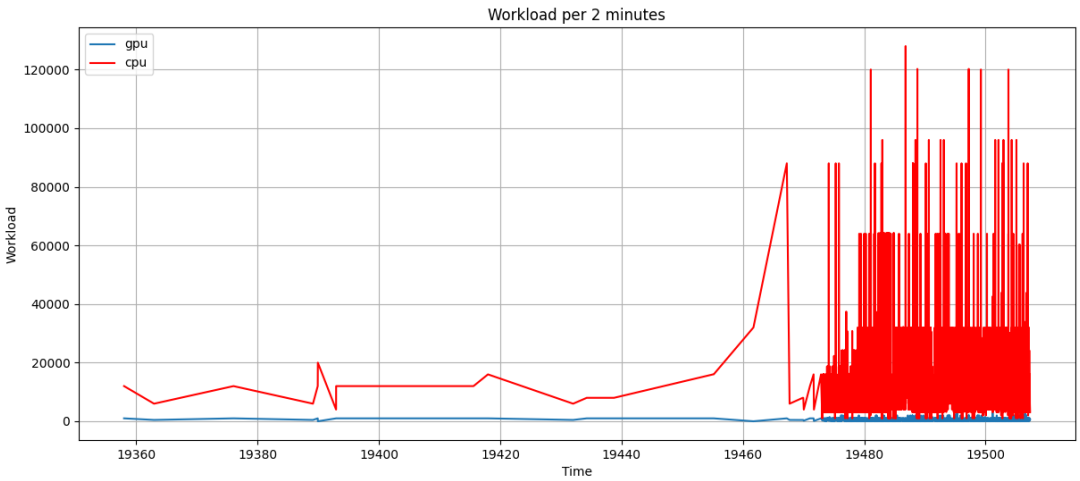

FIG. 1 – *Overall workload of GPU and CPU per 2 minute intervals.*

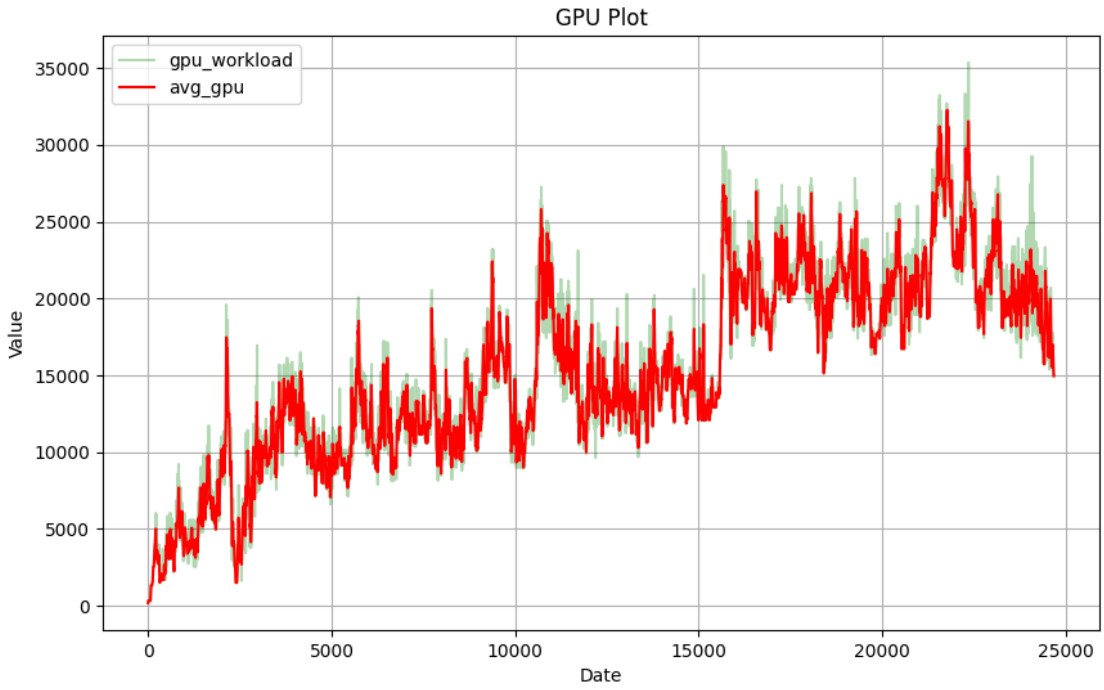

FIG. 2 – *Moving average workflow of a 15 time-step window*

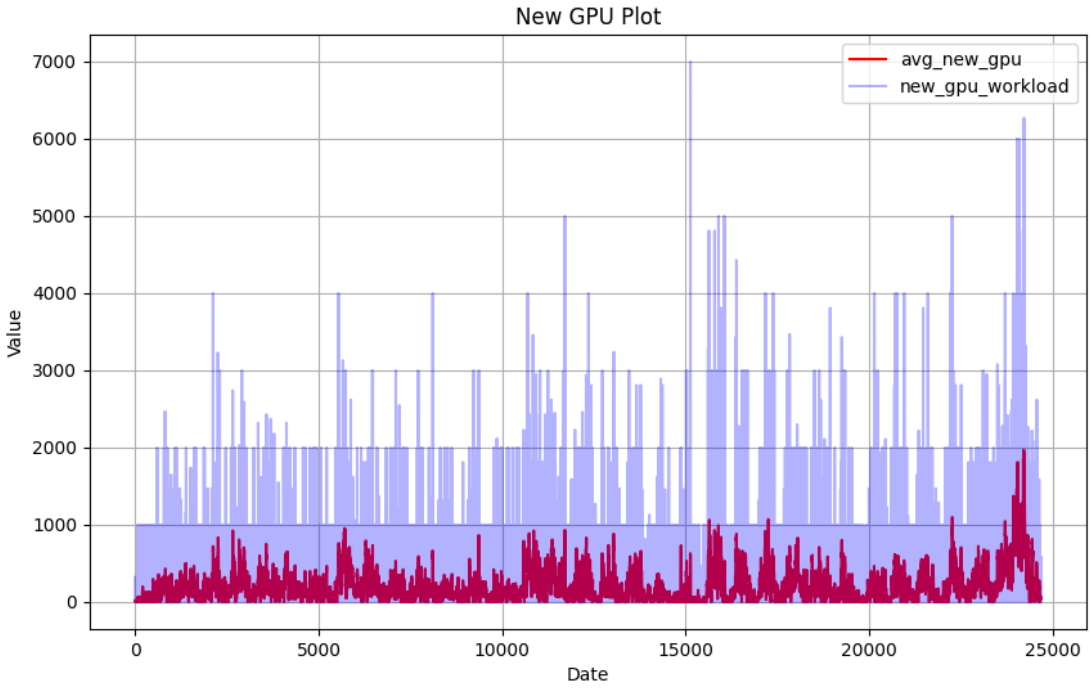

FIG. 3 – *Moving average of a newly scheduled gpu*

2 PROBLEM DESCRIPTION

The task is to predict the (moving average of the) workflow using time series analysis. The workload depends on two factors: The time it needs to process one pod, and the addition of new pods into the workload. The addition of new pods should follow randomly, because it depends on user's submissions. Yet it is surprisingly periodical, as can be seen from figure 3. In general, however, the number of newly arriving pods contains a random element and it is therefore not easily predictable using RNNs alone — which means that not too good results should be expected.

The time series values of overall workload of CPU, GPU, and memory of a number of past time steps $t_{i-n}, \dots t_i$ in a given window n will be combined with data of newly issued CPU, GPU, and memory at time t_{i+1} to predict the new overall workflow of CPU, GPU, and memory at t_{i+1}. Insofar as the request of new resources follows a brownian motion, adding these as explicit information should help the RNN to discern more periodicity (insofar as—and that is the hypothesis—duration of pods is determined by requested CPU, GPU, and memory allocation.

3 APPROACH AND EXPERIMENTAL SETUP

Following [1], the aim is to compare three approaches:

- using plain LSTMs
- using a Encoder-Decoder architecture, https://github.com/jpe9at/Alibaba-Workflow-Prediction-with-Encoder-Decoder-LSTM.
- using the same Encoder-Decoder architecture equipped with an attention module. https://github.com Workflow-Prediction-with-Attention-Module.

In the encoder-decoder network, the encoder processes the time series and feeds a context vector to the decoder, which learns to reconstruct the time series and finally make a prediction (in concatenation with the newly arrived pods). This set-up is extended in two ways: (i) by equipping the context vector with an Attention module, which is supposed to learn which entries in the context vector are most relevant for the current prediction. Furthermore (ii), the vales of the encoder are are applied via the attention module at each time step in combination with the decoder output of the previous timestep to a new input. (Unlike in the Encoder-Decoder Structure without attention, where the context vector was only applied to the very first time step of the decoder. The details of the attention module are as follows:

For each time step, the whole encoder output is used. First, the previous decoder hidden state is permuted and repeated to match the sequence length, then encoder outputs and the repeated decoder hidden states are concatenated along the hidden dimension, i.e. the dimension of the LSTM (here hidden layer 1), to obtain a tensor **concat** of dimensions `batch_size` $\times$ `window_size` $\times$ (`hidden_size` $* 2$). The concatenated tensor is passed through a linear layer followed by a tanh activation function to compute the "energy" scores:

$$\mathbf{E} = \tanh(\mathbf{concat} \cdot \mathbf{W}_{\text{attn}}) \in \mathbb{R}^{\texttt{batch} \times \texttt{window} \times \texttt{hidden}}$$

The attention scores are then computed by taking the dot product between $\mathbf{E}$ and a learnable parameter vector $\mathbf{v}$. Finally, the attention scores are normalized using the softmax function to

get the attention weights:

$$\textbf{attn\_weights} = \text{softmax}(\textbf{attn\_scores})$$

The attention weights are a tensor of size `batch_size`×`window_size`, which contains the relative importances of each time step for the current time step.

3.1 Data Normalisation and weight initialisation

The data is normalised using the `StandardScaler` for each time series. The `StandardScaler` transforms the value x_i of a time step t_i using the following formula:

$$z_i = \frac{x_i - \mu}{\sigma}$$

where μ_i is the mean of the time series values, and σ_i their standard deviation.

This is used instead of MinMax Scaling, because standardized features help in faster convergence and stabilize the learning process for gradient-based optimization algorithms. It is furthermore less sensitive to outliers, which also seems suitable for the task at hand.

Futhermore, all weights in the LSTM (whether single or in the autoencoder framework) are initialised via Xavier/Glorot initialisation, which draws them from the uniform distribution

$$W \sim \mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{n_{\text{in}} + n_{\text{out}}}}, \frac{\sqrt{6}}{\sqrt{n_{\text{in}} + n_{\text{out}}}}\right)$$

where n_{in} and n_{out} are the number of input and output neurons of the layer. This is suitable for LSTMs, because it ensures that the variance of the activations and gradients remains roughly the same across all layers and thereby helps against exploding and vanishing gradients (especially in larger layers).

The weight of the linear layers are intitialized via Normal and He initialisation drawn from a normal distribution:

$$W \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{\text{in}}}}\right)$$

This is particularly suitable for use with ReLU activation functions.

4 NUMERICAL RESULTS

The three architectures were compared after hyperparameter optimization. The parameters that make up the search space are listed in table 1. Among the loss functions that were used in the hyperparameter set-up, Huber loss is a combination of Mean Squared Error and Mean Absolute Error. It is and said to be more stable than MAE for small values, and less sensitive to outliers than MSE. Huber loss is defined as

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \leq \delta \\ \delta(|a| - \frac{1}{2}\delta) & \text{for } |a| > \delta \end{cases}$$

Hyperparameter	Optimization Range/Values
Optimizer	SGD, Adam
Learning Rate	1×10^{-6} to 1×10^{-2}
Batch Size	64, 128, 256
Hidden Size	8, 32, 64, 128
Hidden Size 2	16, 32
L2 Regularization Rate	0.0, 0.0001, 0.005
Loss Function	MSE, MAE, Huber
Window Size	10, 15, 18, 20
Gradient Clip	0.0, 1.0
Scheduler	None, OnPlateau
Number of Layers	1, 2

TAB. 1 – *Overview of Optimized Hyperparameters*

The pytorch default value of δ is used, which is 1.

The best results obtained for all three architectures are listed in table 2. The metric to report the accuracy of the prediciton on the test data is in all cases MAE.

No l2 regularisation was need, since there was little chance of overfitting. In case of vanilla LSTM the the validation loss was even lower than the training loss. This could be due to the fact that the test set was in some sense easier (or more periodical) than the training set. Similarly, give the moderate layer size, no exploding and vanishing gradients were observed, such that gradient clipping was also not needed.

	LSTM	**Encoder-Decoder**	**Attention**
Learning Rate (rounded)	0.000096	0.000124	0.003259
Number of LSTM Layers	2	1	1
Hidden Size	128	64	128
Hidden Size 2	32	32	8
Hidden Size 3	8	x	x
Batch Size	128	32	32
Gradient Clip	0	0	0
L2 Regularization	0	0	0
Loss Function	Huber	MSE	Huber
Optimizer	Adam	Adam	SGD
Window Size	18	10	15
MAE Loss Value	1.9741	2.007	0.9844

TAB. 2 – *Comparison of LSTM, Encoder-Decoder, and Attention mechanisms.*

5 FURTHER EXPERIMENTS

The number of layers could be included in the hyperparameter optimization, there could be other loss functions considered, and other activation functions, and different methods of weight

initialisation. But it seems that what lstm's can do with this type of data is, by the simple fact that it contains a random element, quite maxed out.

Furthermore, predictions for two minute intervals are insufficient to program a scheduler. For this, workload would need to be predicted close up to the millisecond. In this respect, the intervals for the time series predictions could be reduced, but in this case, even worse results are to be expected.

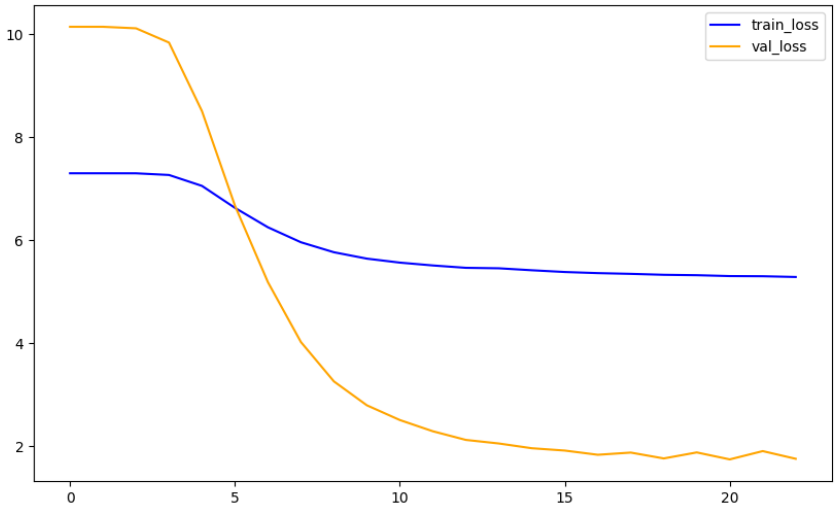

FIG. 4 – *LSTM Training and Validation Loss*

Références

[1] Yihai Chen Yonghua Zhu Weilin Zhang and Honghao Gao. "A novel approach to workload prediction using attention-based LSTM encoder- decoder network in cloud environment". In: *Eurasip Journal on Wireless Communications and Networking* 274 (2019). DOI: `https://doi.org/10.1186/s13638-019-1605-z`.

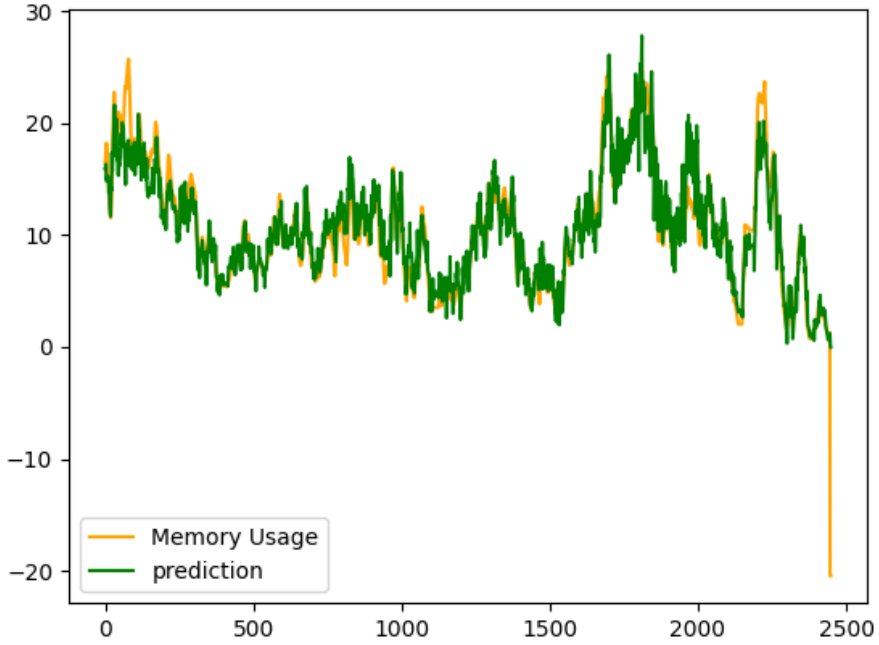

FIG. 5 – *LSTM Prediction of Memory Workflow*

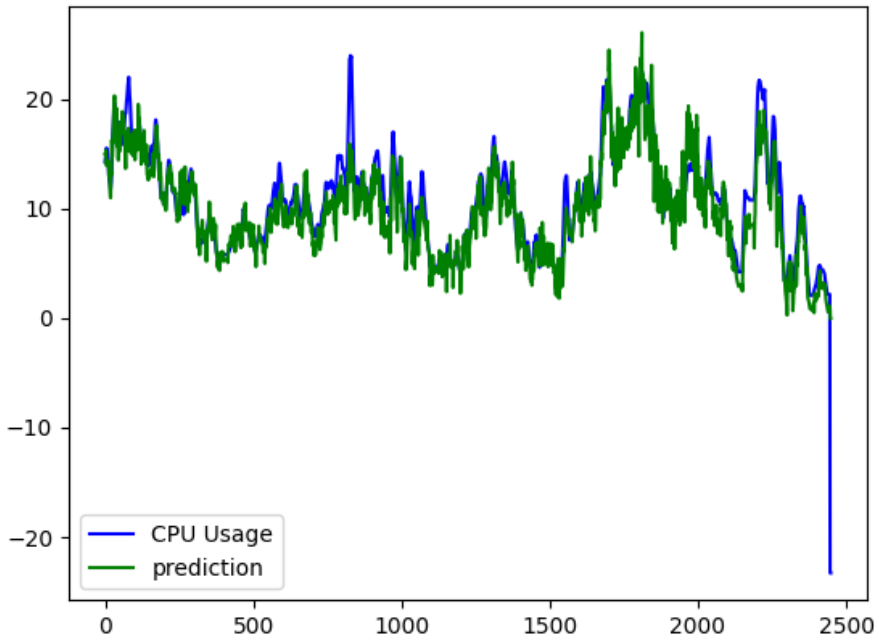

FIG. 6 – *LSTM Prediction of CPU Workflow*

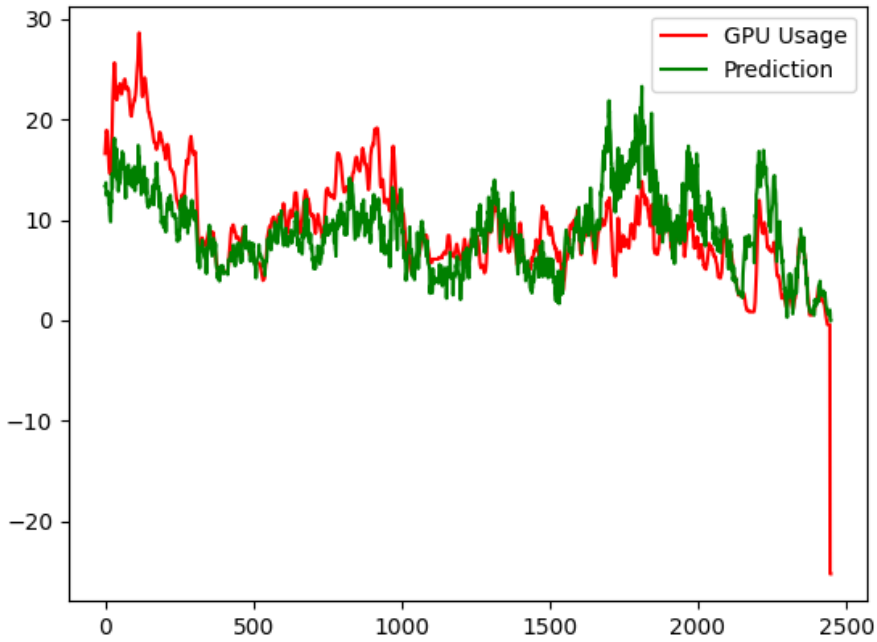

FIG. 7 – *LSTM Prediction of GPU Workflow*

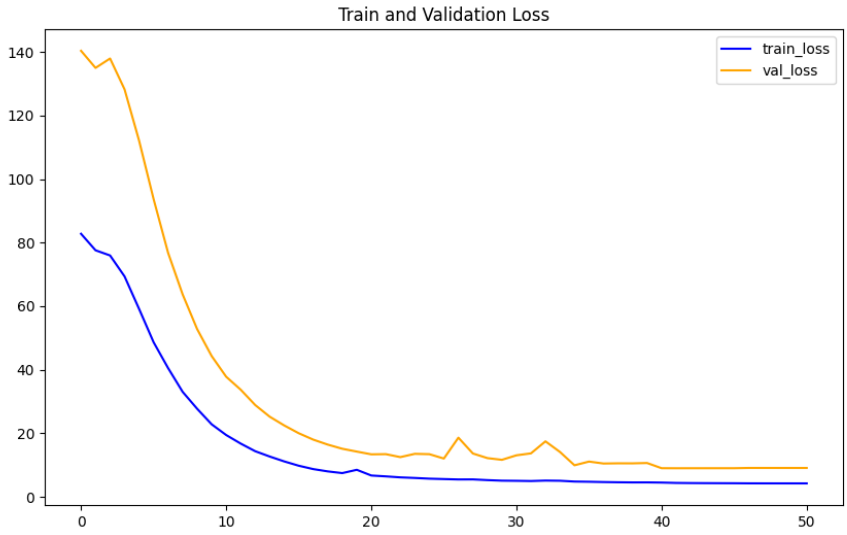

FIG. 8 – *Encoder-Decoder Structure Training and Validation Loss*

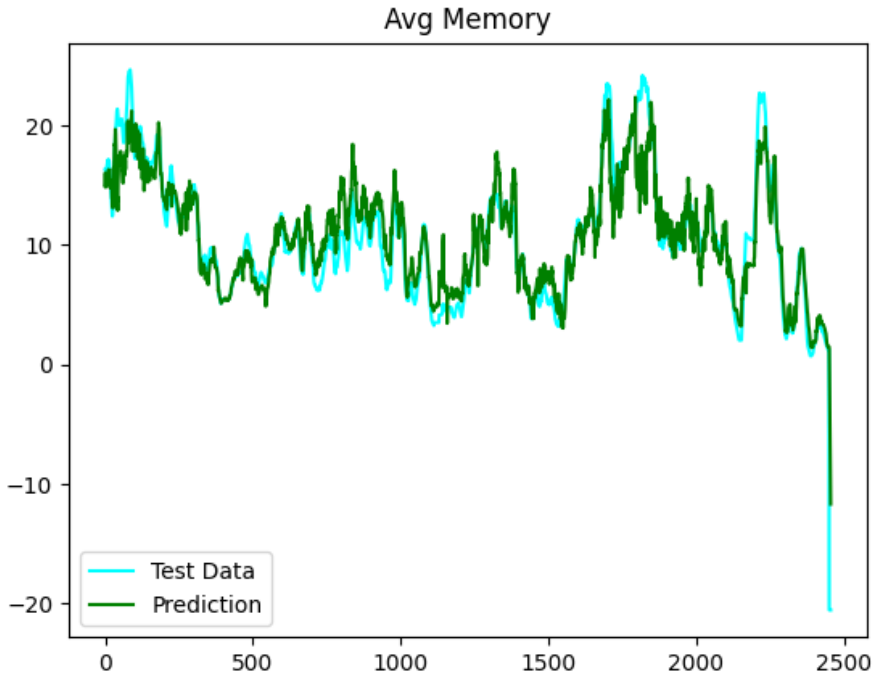

FIG. 9 – *Encoder-Decoder Structure Prediction of Memory Workflow*

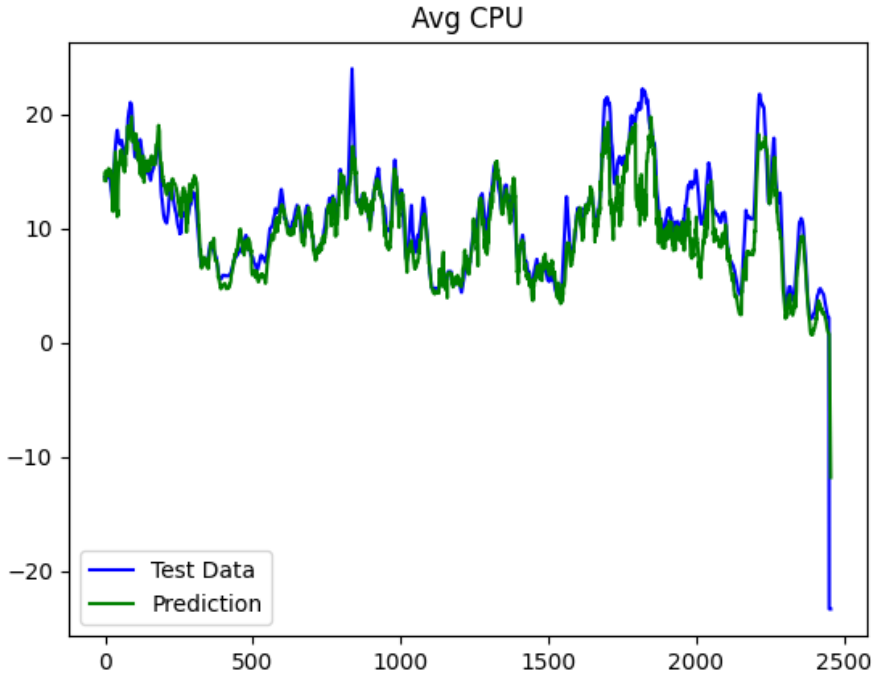

FIG. 10 – *Encoder-Decoder Structure Prediction of CPU Workflow*

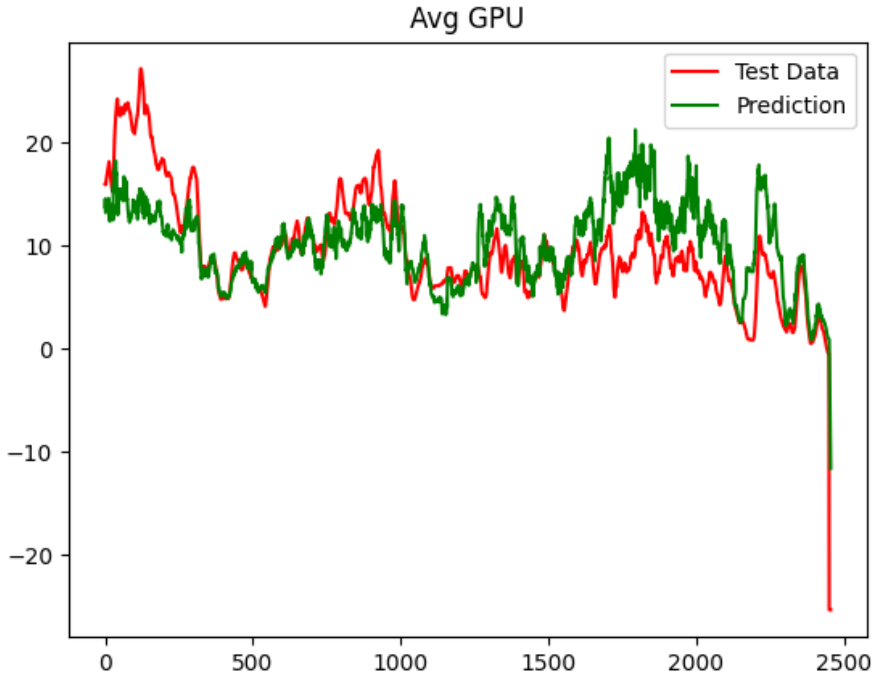

FIG. 11 – *Encoder-Decoder Structure Prediction of GPU Workflow*

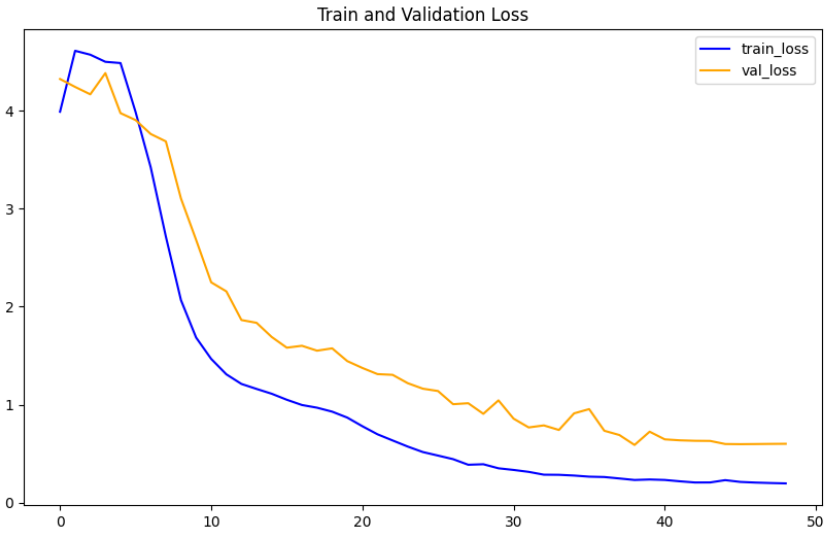

FIG. 12 – *Encoder-Decoder Structure with Attention Training and Validation Loss*

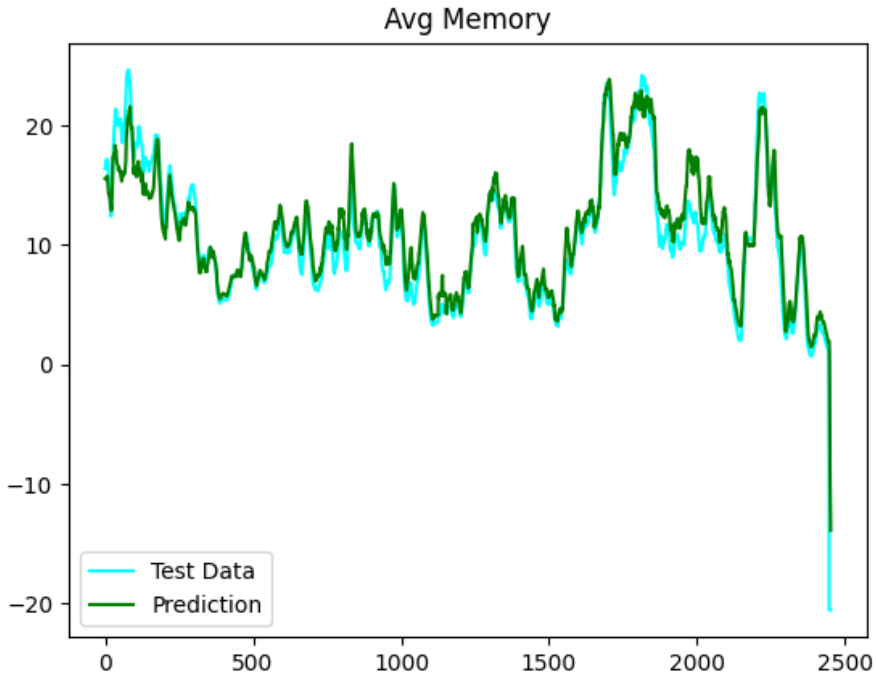

FIG. 13 – *Encoder-Decoder Structure with Attention Prediction of Memory Workflow*

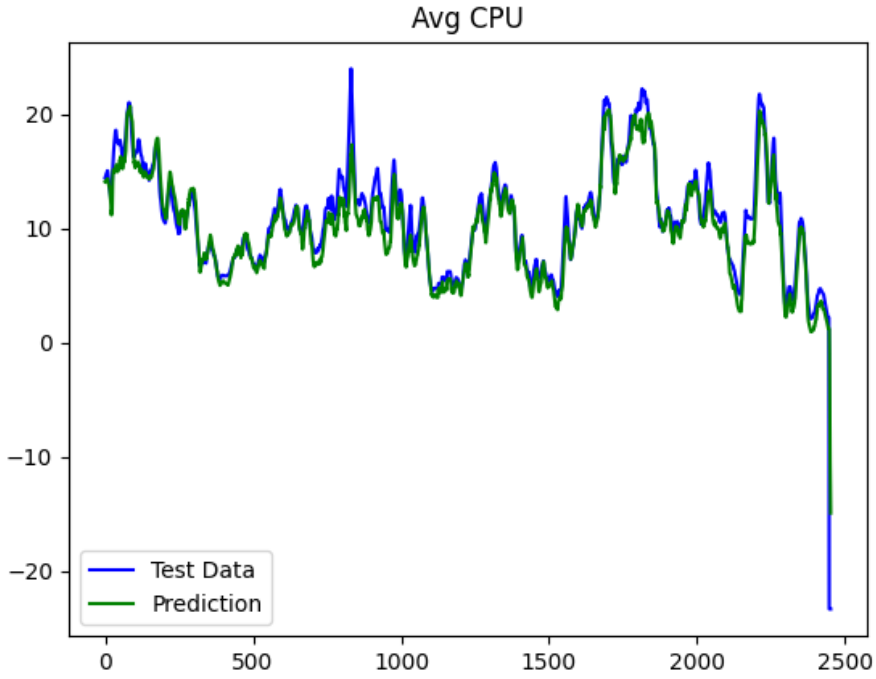

FIG. 14 – *Encoder-Decoder Structure with Attention Prediction of CPU Workflow*

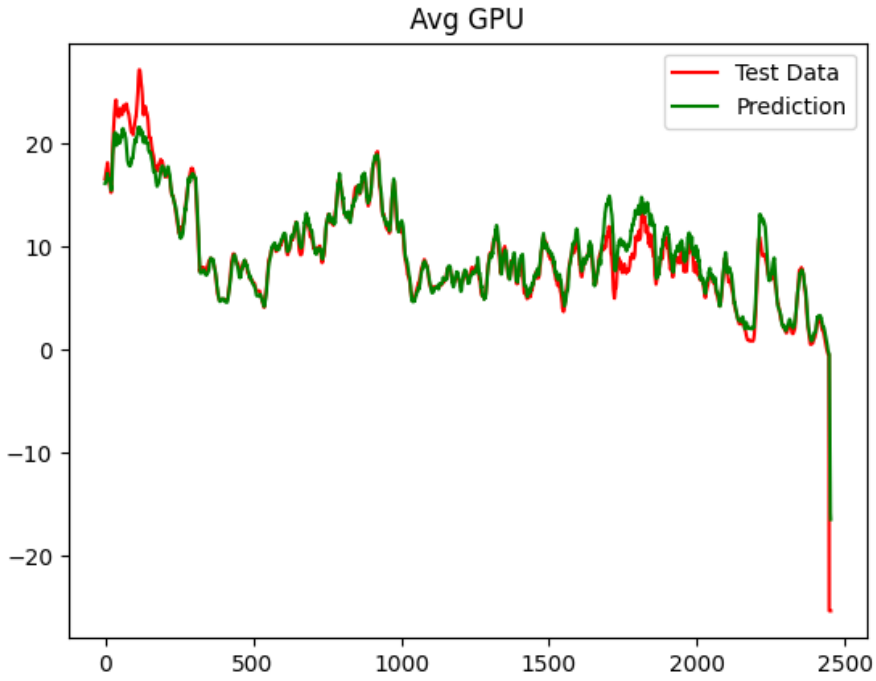

FIG. 15 – *Encoder-Decoder Structure with Attention Prediction of GPU Workflow*