[Draw your reader in with an engaging abstract. It is typically a short summary of the document. When you’re ready to add your content, just click here and start typing.]

DL-Ops

Assignment 2 - RNN, LSTM

Debonil Ghosh

Roll No: M21AIE225

Executive MTech

Artificial Intelligence

Indian Institute of Technology, Jodhpur

Problem Statement:

### **Question 1:**

You have been provided a DATASET, which contains pairs of the words (x,y) i.e. akhbaar

अख़बार in which the first word is a Latin word( words we usually type while chatting with friends

in WhatsApp) and the second word is its corresponding word in native script. Your main goal is

to train a seq2seq model which takes as input the romanized string and produces the

corresponding word in native script.

For Example, Jabki yah Jainon se km hai. ⇒ जबकि यह जनै ों सेकम है। [75]

a) Build a seq2seq model which contains the following layers - [20]

(i) input layer for character embeddings

(ii) one encoder which sequentially encodes the input character sequence (Latin)

(iii) one decoder which takes the last state of the encoder as an input and produces one

character output at a time (native).

Please note that the dimension of input character embeddings, the hidden state of

encoders and decoders, the cell(RNN and LSTM), and the number of layers in the encoder and

decoder should be passed as an argument.

(Note:- For Reference you may refer to this Blog, but the implementation must be in

PyTorch only.)

b) Now train your model using the standard train, test, and val data provided in the dataset.

Try below mentioned hyperparameters and draw the correlation table along with the plot

(loss/accuracy VS. hyperparameter) for both RNN and LSTM. [20]

(i) Input embedding size: 16, 64

(ii) number of encoder layers: 1,3

(iii) number of decoder layers: 1,3

(iv) hidden layer size: 16,64

c) Based on the above parts please try to answer the following- [10]

(i) Does RNN takes less time to converge than LSTM? Reason to support your answer.

(ii) Does dropout leads to better performance? Proof to support your answer.

(iii) Using a smaller size in hidden layer does not give good results? Proof to support your

answer.

d) Now add an attention network to your LSTM seq2seq model and train your model along

with the hyperparameter tuning. (you can use single or multiple attention layers) [25]

(i) Plot the accuracy/loss.

(ii) Report the test accuracy.

(iii) Does the attention-based model performs better than the LSTM? Proof to support

your answer.

### **Question 2:**

The dataset you have been given is Individual household electric power consumption

dataset. [25]

(i)Split the dataset into train and test (80:20) and do the basic preprocessing. [10]

(ii) Use LSTM to predict the global active power while keeping all other important

features and predict it for the testing days by training the model and plot the real global active

power and predicted global active power for the testing days and comparing the results. [8]

(iii) Now split the dataset in train and test (70:30) and predict the global active power for

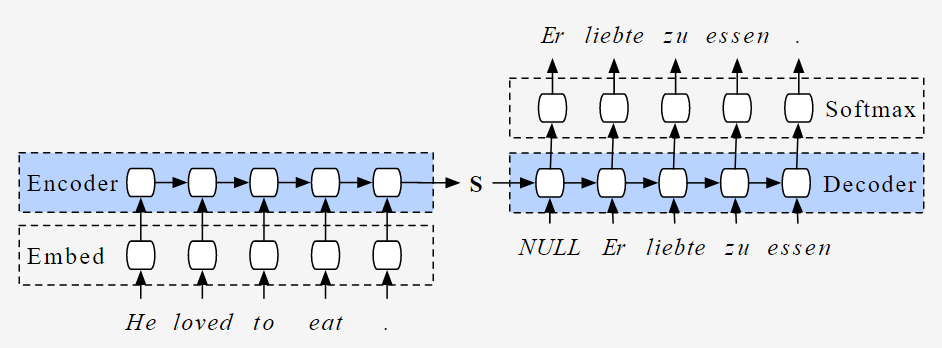
the testing days and compare the results with part (ii). [7]

# **Solution of Question 1:**

Sequence to Sequence (often abbreviated to seq2seq) models is a special class of Recurrent Neural Network architectures that we typically use (but not restricted) to solve complex Language problems like Machine Translation, Question Answering, creating Chatbots, Text Summarization, etc. In this question, we have to use a Seq2Seq model to transliterate text written in a Romanised script into the native script (Hindi – Devnagari).

## **Encoder-Decoder Architecture:**

The most common architecture used to build Seq2Seq models is Encoder-Decoder architecture



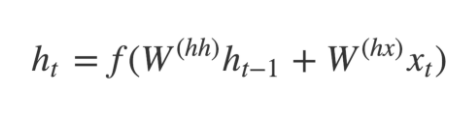
The encoder and decoder components of the Seq2Seq model are typically implemented using LSTM or GRU models. The encoder reads the input sequence and produces internal state vectors or a context vector, which summarize the input information and aid the decoder in generating accurate predictions. The encoder LSTM calculates hidden state vectors (h\_i) and cell state vectors (c\_i) at each time step, using an input sequence element (Xi), previous hidden and cell state vectors, and a formula that updates the internal state vectors.

During training, we discard the outputs of the encoder and only retain the internal states to generate the context vector. The LSTM reads the input sequence one character at a time, and if the input is a sequence of length 't', the LSTM reads it in 't' time steps.

The decoder uses the context vector to generate the output sequence, one character at a time. The LSTM in the decoder also maintains hidden state vectors (h) and cell state vectors (c) at each time step. The output sequence element at each time step (Yi) is a probability distribution over the entire vocabulary, generated by applying a softmax activation function to the output of the decoder LSTM cell. This output sequence element is a vector of size "vocab\_size" representing the probability distribution of the next native script character .

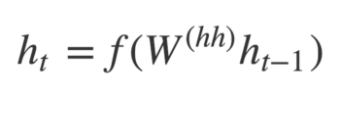
**Encoder :**

* Both encoder and the decoder are LSTM models (or sometimes GRU models)
* Encoder reads the input sequence and summarizes the information in something called the **internal state vectors** or **context vector** (in case of LSTM these are called the hidden state and cell state vectors). We discard the outputs of the encoder and only preserve the internal states. This context vector aims to encapsulate the information for all input elements in order to help the decoder make accurate predictions.
* The hidden states h\_i are computed using the formula:

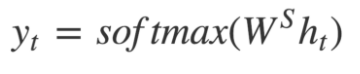


**Decoder :**

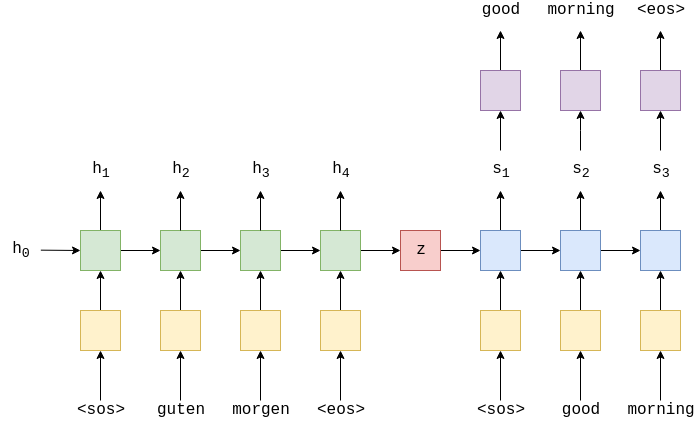
* The decoder is an LSTM whose initial states are initialized to the final states of the Encoder LSTM, i.e. the context vector of the encoder’s final cell is input to the first cell of the decoder network. Using these initial states, the decoder starts generating the output sequence, and these outputs are also taken into consideration for future outputs.
* A stack of several LSTM units where each predicts an output y\_t at a time step t.
* Each recurrent unit accepts a hidden state from the previous unit and produces and output as well as its own hidden state.
* Any hidden state *h\_i*is computed using the formula:



* The output *y\_t* at time step *t* is computed using the formula:



* The outputs are calculated using the hidden state at the current time step together with the respective weight W(S). Softmax is used to create a probability vector which will help to determine the final output (e.g. word in the question-answering problem).



**Given Dataset:** Dakshina Dataset

The Dakshina dataset is a collection of text in both Latin and native scripts for 12 South Asian languages. For each language, the dataset includes a large collection of native script Wikipedia text, a romanization lexicon which consists of words in the native script with attested romanizations, and some full sentence parallel data in both a native script of the language and the basic Latin alphabet.

Dataset URL: <https://github.com/google-research-datasets/dakshina>

There are 12 languages represented in the dataset: Bangla (bn), Gujarati (gu), Hindi (hi), Kannada (kn), Malayalam (ml), Marathi (mr), Punjabi (pa), Sindhi (sd), Sinhala (si), Tamil (ta), Telugu (te) and Urdu (ur).

All data is derived from Wikipedia text. Each language has its own directory, in which there are three subdirectories:

**In current problem, only Hindi will be used**

# Question 1. (a)

# Model Training:

# 

# 

# Test accuracy: 0.694876372814178

# Question 1. (b)

# Hyper Parameter Tuning Results:



## Best Result Found:

|  |  |
| --- | --- |
| Cell type | Lstm |
| Latent dim | 256 |
| Hidden dim | 16 |
| Num encoders | 1 |
| Num decoders | 1 |
| Loss | 1.9077255 |
| Accuracy | 0.7253369 |

# c) Based on the above parts please try to answer the following- [10]

# (i) Does RNN takes less time to converge than LSTM? Reason to support your answer.

Yes

RNNs are simpler than LSTMs and have a smaller number of parameters, which can make them faster to train and converge. RNNs also have a simpler gating mechanism that can be less prone to overfitting. However, RNNs can suffer from the vanishing gradient problem, where gradients can become very small and cause the network to have difficulty learning long-term dependencies.

LSTMs, on the other hand, have a more complex architecture with more parameters and a more sophisticated gating mechanism. LSTMs are designed to address the vanishing gradient problem by using a set of gates to selectively forget or remember information from previous time steps. While this can improve the network's ability to learn long-term dependencies, it can also make LSTMs slower to converge and more prone to overfitting.

In general, the choice between RNNs and LSTMs depends on the specific problem and dataset being used. While RNNs can be faster to train and converge, LSTMs may be more effective at learning long-term dependencies and avoiding overfitting.

# (ii) Does dropout leads to better performance? Proof to support your answer.

Yes.

Dropout is a regularization technique commonly used in deep learning to prevent overfitting. Dropout randomly drops out a proportion of the neurons in a neural network during training, forcing the remaining neurons to learn more robust representations. This can help the network generalize better to new data and improve performance on the test set.

There have been several studies that have shown that dropout can lead to better performance. For example, in a 2014 paper titled "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", the authors conducted experiments on several benchmark datasets and showed that dropout improved performance on all of them. Another study from 2017 titled "Dropout Improves Recurrent Neural Networks for Handwriting Recognition" showed that dropout can improve performance on sequence prediction tasks, such as handwriting recognition.

Moreover, dropout has become a widely used technique in deep learning and is included in many state-of-the-art models. For example, dropout is used in the popular ResNet and Inception architectures for image classification, as well as in the LSTM-based models for sequence prediction.

In summary, there is empirical evidence to support that dropout can lead to better performance in deep learning models. However, the extent of the improvement and the optimal dropout rate may depend on the specific dataset and problem being addressed, and may require some experimentation to determine the best hyperparameters for the model.

# (iii) Using a smaller size in hidden layer does not give good results? Proof to support your answer.

Not always.

Increasing the size of the hidden layer in a neural network can improve its performance by increasing its capacity to learn complex patterns in the data, but can also increase the risk of overfitting. Using a smaller size in the hidden layer can reduce the capacity of the model and prevent overfitting, but may not be able to capture complex patterns as well. The choice of the hidden layer size depends on the complexity of the problem and amount of training data available, and may require experimentation to find the optimal hyperparameters.

# **Solution of Question 2:**

Time series prediction is one application of LSTM (Long Short-Term Memory), which are a type of recurrent neural network (RNN) that is well-suited to sequence data such as time series. Here are the steps to implement time series prediction using LSTM:

1. Data Preparation: The first step is to prepare your time series data by splitting it into training and test sets, normalizing the data, and formatting it into a suitable shape for the LSTM model. This often involves transforming the time series data into a supervised learning problem by framing the data as input-output pairs.
2. Model Architecture: The next step is to define the LSTM model architecture, which typically involves specifying the number of LSTM layers, the number of hidden units per layer, the activation functions, and the dropout rate. You can also experiment with different architectures to find the best one for your data.
3. Model Training: Once the model architecture is defined, you can train the LSTM model using the training set. During training, the model updates its weights and biases to minimize the prediction error. You can use a variety of loss functions and optimization algorithms to train the model.
4. Model Evaluation: After training, you can evaluate the performance of the LSTM model using the test set. Common metrics for evaluating time series prediction models include mean absolute error (MAE), root mean squared error (RMSE), and coefficient of determination (R-squared).
5. Prediction: Once the model is trained and evaluated, you can use it to make predictions on new, unseen data. This involves feeding the input data to the model and obtaining the predicted output.

Overall, time series prediction using LSTM is a complex process that requires careful data preparation, model architecture design, and training and evaluation. However, it can be a powerful tool for making accurate predictions on time series data.

**Given Dataset:**

**Dataset Name: Individual household electric power consumption Data Set**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Data Set Characteristics:** | Multivariate, Time-Series | **Number of Instances:** | 2075259 | **Area:** | Physical |
| **Attribute Characteristics:** | Real | **Number of Attributes:** | 9 | **Missing Values?** | Yes |

**Data Set Information:**

This archive contains 2075259 measurements gathered in a house located in Sceaux (7km of Paris, France) between December 2006 and November 2010 (47 months).  
Notes:  
1.(global\_active\_power\*1000/60 - sub\_metering\_1 - sub\_metering\_2 - sub\_metering\_3) represents the active energy consumed every minute (in watt hour) in the household by electrical equipment not measured in sub-meterings 1, 2 and 3.  
2.The dataset contains some missing values in the measurements (nearly 1,25% of the rows). All calendar timestamps are present in the dataset but for some timestamps, the measurement values are missing: a missing value is represented by the absence of value between two consecutive semi-colon attribute separators. For instance, the dataset shows missing values on April 28, 2007.

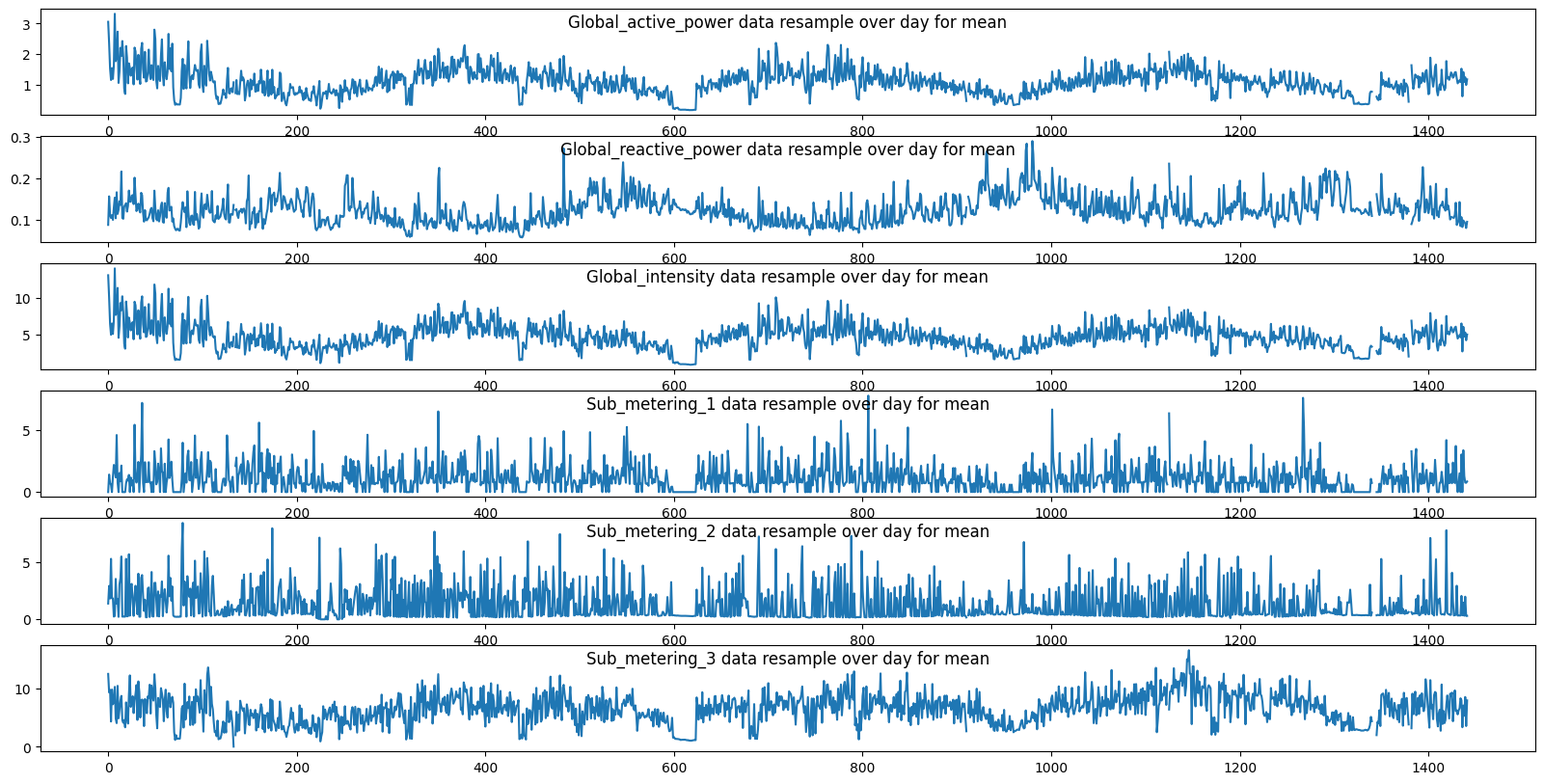
**Attribute Information:**

1.date: Date in format dd/mm/yyyy  
2.time: time in format hh:mm:ss  
3.global\_active\_power: household global minute-averaged active power (in kilowatt)  
4.global\_reactive\_power: household global minute-averaged reactive power (in kilowatt)  
5.voltage: minute-averaged voltage (in volt)  
6.global\_intensity: household global minute-averaged current intensity (in ampere)  
7.sub\_metering\_1: energy sub-metering No. 1 (in watt-hour of active energy). It corresponds to the kitchen, containing mainly a dishwasher, an oven and a microwave (hot plates are not electric but gas powered).  
8.sub\_metering\_2: energy sub-metering No. 2 (in watt-hour of active energy). It corresponds to the laundry room, containing a washing-machine, a tumble-drier, a refrigerator and a light.  
9.sub\_metering\_3: energy sub-metering No. 3 (in watt-hour of active energy). It corresponds to an electric water-heater and an air-conditioner.

# Data Exploration:

**Monthly Mean of different parameters:**

# 

**Daily Mean of different parameters:**

# Data Pre-processing:

**MinMaxScaler** is a data normalization technique used in machine learning to transform the data in a way that the values are between 0 and 1. This technique is commonly used to rescale numerical data, especially in deep learning models. The MinMaxScaler is used to transform the data so that it is in the range of [0, 1] using the following formula:

X\_scaled = (X - X\_min) / (X\_max - X\_min)

where X is the original data, X\_min and X\_max are the minimum and maximum values in the data set, and X\_scaled is the normalized data.

The main purpose of using the MinMaxScaler is to scale the data to the same range to avoid any feature dominating over the other. For example, if we have two features, one having a range of [0, 1] and another having a range of [0, 100], then the second feature may dominate the first feature, causing the model to perform poorly.

# Model Architecture

LSTMModel(

(lstm): LSTM(7, 100, batch\_first=True)

(dropout): Dropout(p=0.1, inplace=False)

(fc): Linear(in\_features=100, out\_features=1, bias=True)

)

# Hyper Parameters for LSTM:

# In Dimension : 7

# Out Dimension : 100

# Loss Function: MSE Loss

# Optimiser: Adam

# Learning Rate: 0.0001

# Drop Out : 0.1

# Number of epochs: 200

# Model with 80-20 Training

Log:

Epoch 10, Train Loss: 0.031362, Test Loss: 0.019585

Epoch 20, Train Loss: 0.020634, Test Loss: 0.012282

Epoch 30, Train Loss: 0.017757, Test Loss: 0.012019

Epoch 40, Train Loss: 0.016601, Test Loss: 0.011427

Epoch 50, Train Loss: 0.014911, Test Loss: 0.009778

Epoch 60, Train Loss: 0.013675, Test Loss: 0.008931

Epoch 70, Train Loss: 0.012630, Test Loss: 0.008755

Epoch 80, Train Loss: 0.012001, Test Loss: 0.008560

Epoch 90, Train Loss: 0.011679, Test Loss: 0.008519

Epoch 100, Train Loss: 0.011559, Test Loss: 0.008572

Epoch 110, Train Loss: 0.011439, Test Loss: 0.008451

Epoch 120, Train Loss: 0.011289, Test Loss: 0.008316

Epoch 130, Train Loss: 0.011172, Test Loss: 0.008199

Epoch 140, Train Loss: 0.011060, Test Loss: 0.008128

Epoch 150, Train Loss: 0.010961, Test Loss: 0.008041

Epoch 160, Train Loss: 0.010868, Test Loss: 0.007978

Epoch 170, Train Loss: 0.010758, Test Loss: 0.007887

Epoch 180, Train Loss: 0.010666, Test Loss: 0.007830

Epoch 190, Train Loss: 0.010566, Test Loss: 0.007794

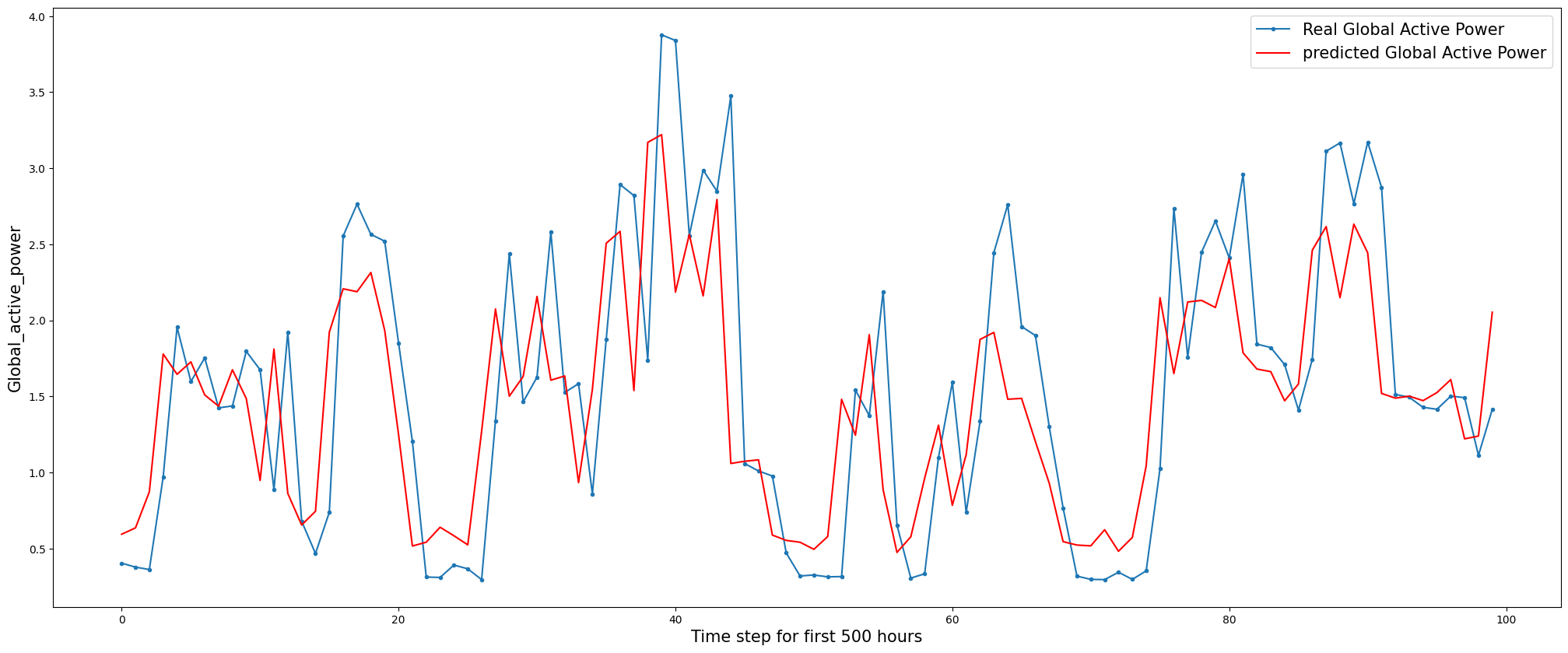
Epoch 200, Train Loss: 0.010473, Test Loss: 0.007740

# Training vs Validation Loss

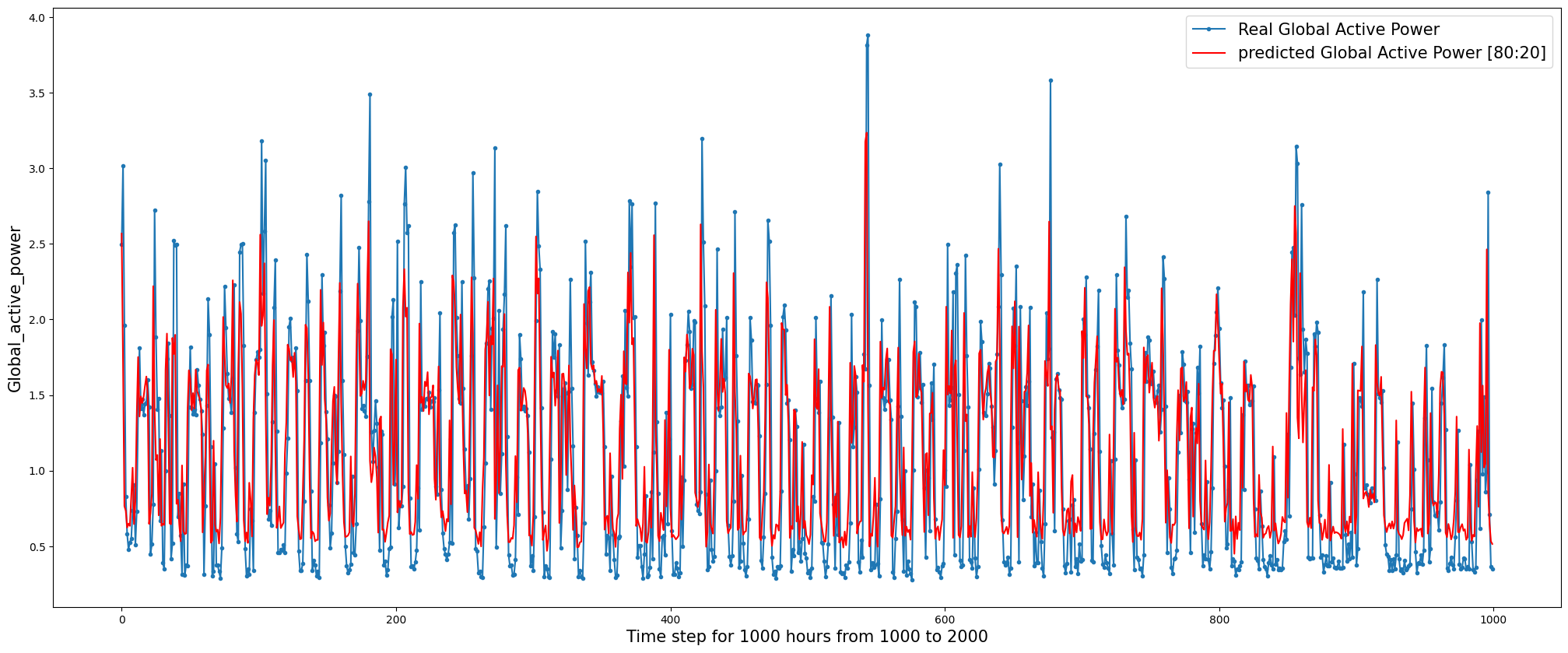
# 

# Test RMSE: **0.565**

# Time step for first 100 hours



# Time step for 1000 hours from 1000 to 2000



# Data Split 70-30:

**Model Training Log:**

Epoch 10, Train Loss: 0.032408, Test Loss: 0.026220

Epoch 20, Train Loss: 0.019658, Test Loss: 0.014716

Epoch 30, Train Loss: 0.017375, Test Loss: 0.013495

Epoch 40, Train Loss: 0.016209, Test Loss: 0.012465

Epoch 50, Train Loss: 0.014673, Test Loss: 0.011144

Epoch 60, Train Loss: 0.013643, Test Loss: 0.010372

Epoch 70, Train Loss: 0.012667, Test Loss: 0.009834

Epoch 80, Train Loss: 0.012007, Test Loss: 0.009468

Epoch 90, Train Loss: 0.011619, Test Loss: 0.009249

Epoch 100, Train Loss: 0.011380, Test Loss: 0.009247

Epoch 110, Train Loss: 0.011268, Test Loss: 0.009224

Epoch 120, Train Loss: 0.011161, Test Loss: 0.009143

Epoch 130, Train Loss: 0.011042, Test Loss: 0.009115

Epoch 140, Train Loss: 0.010937, Test Loss: 0.008970

Epoch 150, Train Loss: 0.010825, Test Loss: 0.008929

Epoch 160, Train Loss: 0.010749, Test Loss: 0.008843

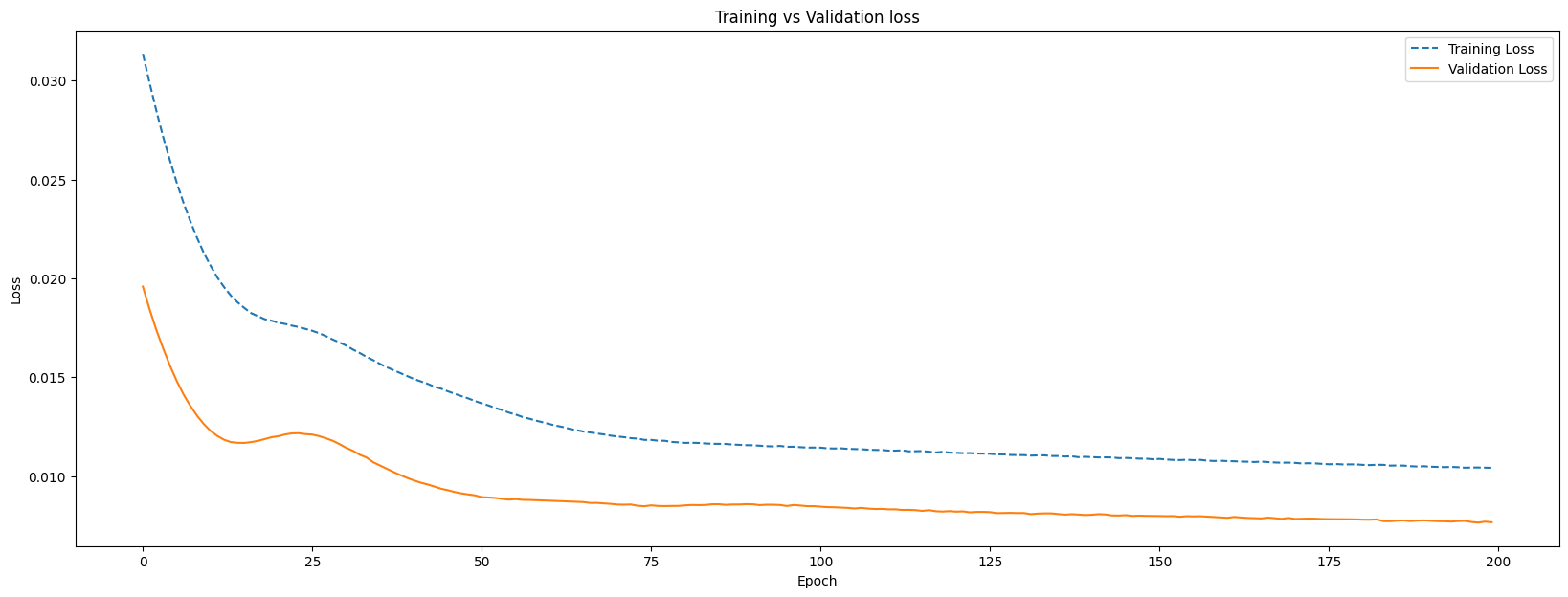
Epoch 170, Train Loss: 0.010617, Test Loss: 0.008838

Epoch 180, Train Loss: 0.010557, Test Loss: 0.008749

Epoch 190, Train Loss: 0.010485, Test Loss: 0.008721

Epoch 200, Train Loss: 0.010394, Test Loss: 0.008659

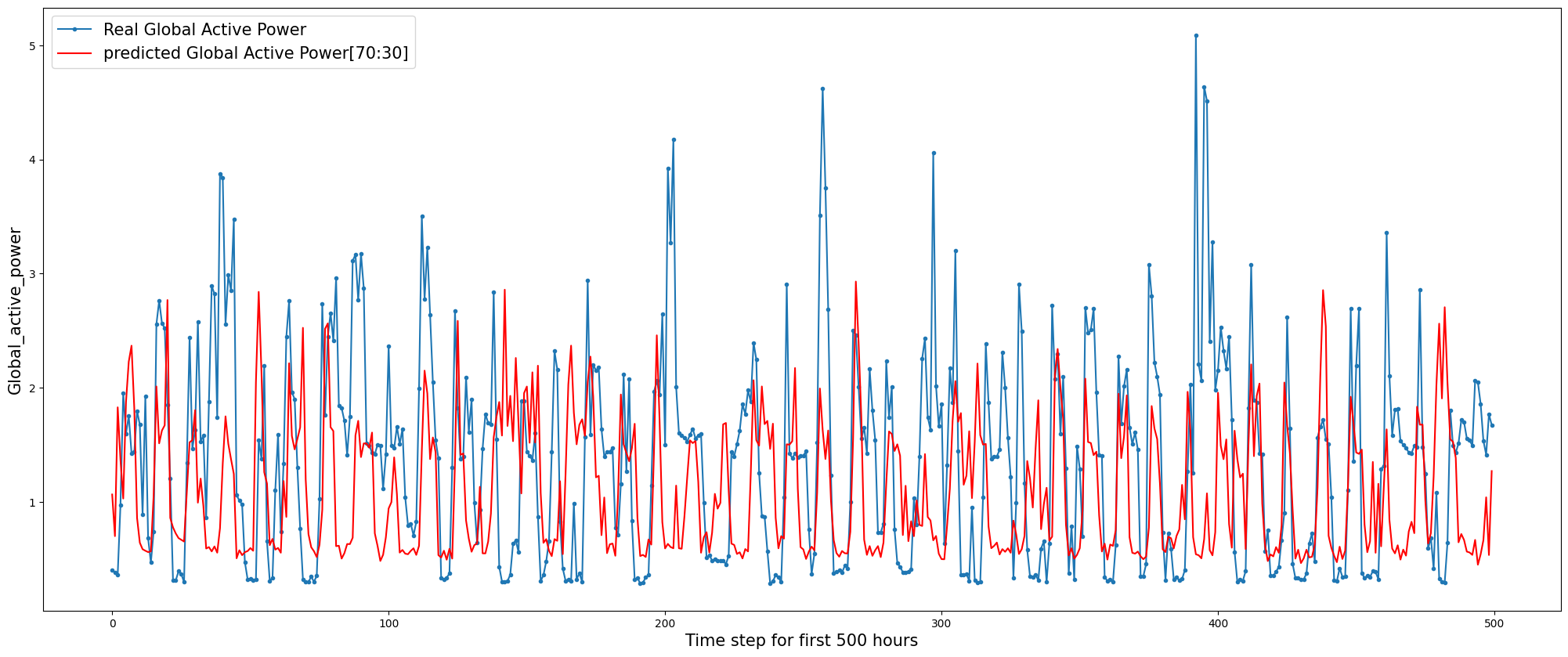
**Training vs Validation Loss:**

****

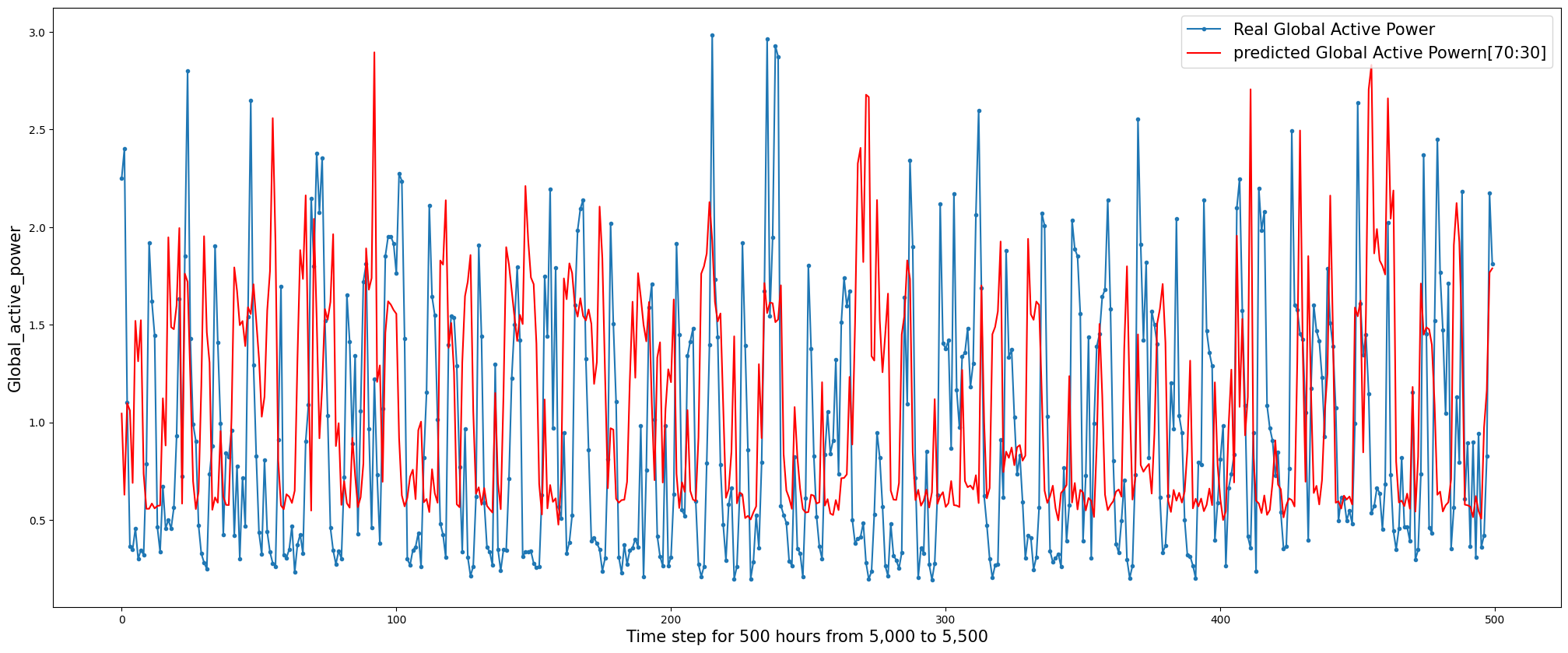
Test RMSE: 0.598

**Predictions:**

Real Global Active Power [70:30] for first 500 hours



# Real Global Active Power [70:30] for 500 hours from 5,000 to 5,500



# **Comparison**

# 

# Conclusion:

80:20 splits MSE error is lower than the same of 70:30 split. Model 80:20 performs better as it gets more exposer to training data.