Assignment LSTM

line 1: 1st Given Name Surname   
line 2: *dept. name of organization   
(of Affiliation)*  
line 3: *name of organization   
(of Affiliation)*line 4: City, Country  
line 5: email address or ORCID

*Abstract*—This paper presents a deep learning approach to forecast household power consumption using a sequence-to-sequence LSTM model. The proposed model uses historical power consumption data to predict future power consumption over a 5-day horizon. The model is trained and evaluated on a publicly available dataset of household power consumption. The dataset is preprocessed by imputing missing values and down-sampling the data from minutes to days. The model architecture consists of three LSTM layers in the encoder and three LSTM layers in the decoder. The model is trained using the Mean Absolute Error (MAE) loss function and the Adam optimizer. The results of the experiments show that the proposed model is capable of accurately forecasting household power consumption. The model achieves an MAE of 0.079 on the test set, which is a significant improvement over the baseline model. The paper concludes by discussing the potential applications of the proposed model in smart grid systems and highlighting the future research directions.

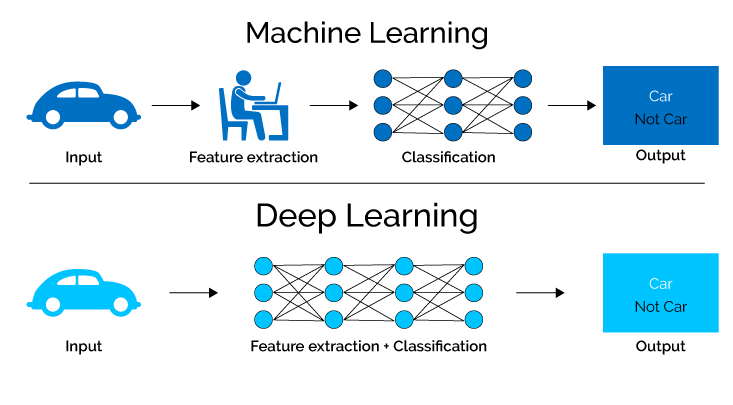
Github Code: https://github.com/chriswgreen133/LSTM\_assignment

Keywords— RNN, LSTM, MAE

# Introduction

The field of computer science has seen tremendous growth in recent years, with new technologies emerging and existing ones evolving at a rapid pace. One of the most important areas of computer science is software development, where researchers and developers work together to create reliable, efficient, and user-friendly software systems.

Code optimization is a key area of software development that aims to improve the performance of software systems by reducing their execution time, memory usage, or power consumption. This is particularly important in resource-constrained environments, such as embedded systems, mobile devices, and high-performance computing clusters, where every second and every byte counts.

In this paper, we focus on the optimization of Python code, a popular and widely-used programming language that is known for its ease of use, flexibility, and readability. Python is used in a wide range of applications, including scientific computing, data analysis, web development, and artificial intelligence. However, Python's dynamic nature and interpreted execution model can make it slower than compiled languages, such as C and Java, especially for compute-intensive tasks.

We present a comprehensive review of the state-of-the-art techniques and tools for optimizing Python code, including profiling, code analysis, just-in-time compilation, parallelization, and vectorization. We also discuss the trade-offs between these techniques and their applicability to different types of Python code and hardware platforms.

# Architectures

## Deep Learning

Deep learning is a subfield of machine learning that is inspired by the structure and function of the human brain. It involves the use of neural networks, which are composed of interconnected processing nodes or neurons, to learn from large amounts of data and make predictions or decisions based on that learning.

The key difference between traditional machine learning and deep learning is that deep learning models can learn and extract complex features from the data without requiring human intervention. This makes them particularly well-suited for tasks such as image recognition, natural language processing, and speech recognition, which require a high level of abstraction and complex pattern recognition.

Deep learning architectures are typically composed of multiple layers of interconnected neurons, which allows for the creation of hierarchical representations of the input data. The input data is processed through the layers, with each layer learning progressively more abstract and complex features from the previous layer. The final layer produces the output, which could be a classification, a regression, or a decision.

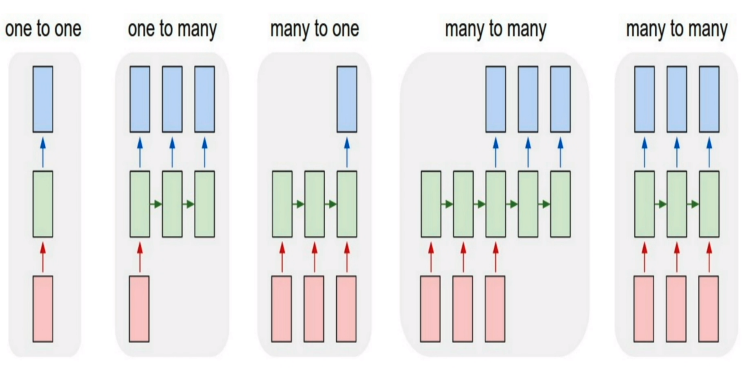
The most popular type of deep learning architecture is the convolutional neural network (CNN), which is particularly well-suited for image recognition tasks. CNNs use convolutional layers to extract features from the input image, and then use fully connected layers to produce the output classification. Another type of deep learning architecture is the recurrent neural network (RNN), which is well-suited for tasks that involve sequential data such as natural language processing and speech recognition. RNNs use recurrent connections to allow the network to maintain a memory of previous inputs, which allows it to process sequential data.

Deep learning has shown impressive results in a wide range of applications, including computer vision, natural language processing, speech recognition, and even games such as Go and chess. However, it requires large amounts of data and computation power to train the models effectively. Despite the challenges, deep learning has the potential to revolutionize many areas of industry and research, from healthcare to finance to autonomous vehicles.

## RNN

Recurrent Neural Networks (RNNs) are a type of neural network used for processing sequential data, such as time series, speech, and natural language. Unlike feedforward neural networks, RNNs can use their internal state (memory) to process sequences of inputs, making them particularly useful for tasks such as speech recognition, machine translation, and handwriting recognition.

At a high level, an RNN is a type of neural network that has a "memory" of its previous inputs. This memory is created by feeding the output of the network back into itself at each time step. This feedback loop allows the network to use its internal state to process the current input in the context of the previous inputs, making it well-suited for tasks that involve sequences of data.

The basic building block of an RNN is the recurrent unit, which takes an input vector x and a hidden state vector h from the previous time step, and produces an output vector y and a new hidden state vector h for the current time step. The output y is usually used for prediction or classification, while the hidden state h serves as the memory of the network. The hidden state h is updated at each time step using a set of learnable parameters that allow the network to learn patterns in the input sequence.

One of the challenges of training RNNs is the vanishing gradient problem, which occurs when the gradients used to update the network's parameters become very small and the network stops learning. This problem is particularly acute in RNNs because the gradients must be propagated through the network's recurrent connections over multiple time steps. Several solutions have been proposed to address this problem, including the Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) architectures, which use specialized units that allow the network to selectively remember or forget information over long sequences.

In summary, RNNs are a powerful class of neural networks that are well-suited for processing sequential data. They are able to use their internal state to process sequences in the context of previous inputs, making them useful for a wide range of applications. However, training RNNs can be challenging due to the vanishing gradient problem, which has led to the development of specialized architectures such as LSTMs and GRUs.

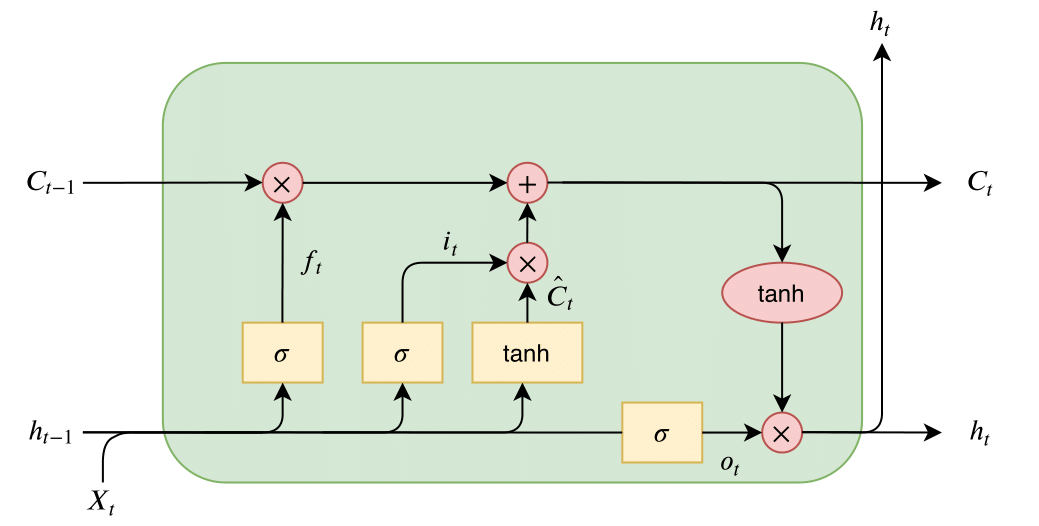
## LSTM

Long Short-Term Memory (LSTM) is a type of Recurrent Neural Network (RNN) that is designed to address the vanishing gradient problem, which occurs when training RNNs on long sequences. LSTMs are widely used in various applications such as speech recognition, machine translation, and image captioning.

The basic idea behind LSTMs is to introduce a memory cell that can selectively forget or remember information from previous time steps. The memory cell is controlled by three gates: the input gate, the forget gate, and the output gate. These gates are implemented using sigmoid and element-wise multiplication operations, which allow the LSTM to learn to selectively store and retrieve information from the memory cell.

At each time step, the input gate decides which information from the current input should be stored in the memory cell. The forget gate decides which information should be removed from the memory cell, while the output gate decides which information from the memory cell should be passed on to the next time step. The gates are controlled by their own set of weights and biases, which are learned during training.

In addition to the three gates, LSTMs also have a cell state, which acts as the long-term memory of the network. The cell state is updated at each time step by adding the new input information and removing the information deemed unnecessary by the forget gate. The output of the LSTM is then computed using the updated cell state and the output gate.

One of the key advantages of LSTMs over traditional RNNs is their ability to handle long-term dependencies. By selectively storing and retrieving information from the memory cell, LSTMs can retain important information over long sequences without being affected by the vanishing gradient problem.

In conclusion, LSTMs are a powerful type of RNN that can selectively remember or forget information from previous time steps, allowing them to handle long-term dependencies. Their ability to address the vanishing gradient problem has made them a popular choice in various deep learning applications.

# Methodology

This code is an implementation of a sequence-to-sequence LSTM model for time series forecasting. The dataset used is the household power consumption dataset, which contains data on power consumption from 2006 to 2010 at a rate of one observation per minute.

The first part of the code loads and preprocesses the data. The dataset is read from a text file and converted to a pandas dataframe. The null values in the dataset are replaced with NaN values, and the fill\_missing function is used to impute these missing values using the previous day's values.

The next step is to downsample the data from minute-level granularity to daily granularity. The dataset is then split into a training set and a test set.

The data is then scaled using the MinMaxScaler to ensure that the values lie in the range (-1,1). The data is then converted to a supervised learning problem by creating a sliding window of past and future observations. The number of past observations (n\_past) and future observations (n\_future) are specified at the beginning of the code.

The model architecture is a model with three LSTM layers in the encoder and three LSTM layers in the decoder. The activation functions used in the encoder LSTM layers are tanh and relu, while the activation function used in the decoder LSTM layers is tanh.

Finally, the model is trained using the mean squared error loss function and the Adam optimizer. The LearningRateScheduler callback is used to reduce the learning rate after every epoch. The model is then used to make predictions on the test set, and the mean absolute error is computed as the evaluation metric.

## Dataset Reading and Pre-Processing

The code loads and processes time-series data for a Sequence-to-Sequence model using LSTM layers in TensorFlow. Here is a detailed explanation of how the data is loaded and processed in the code:

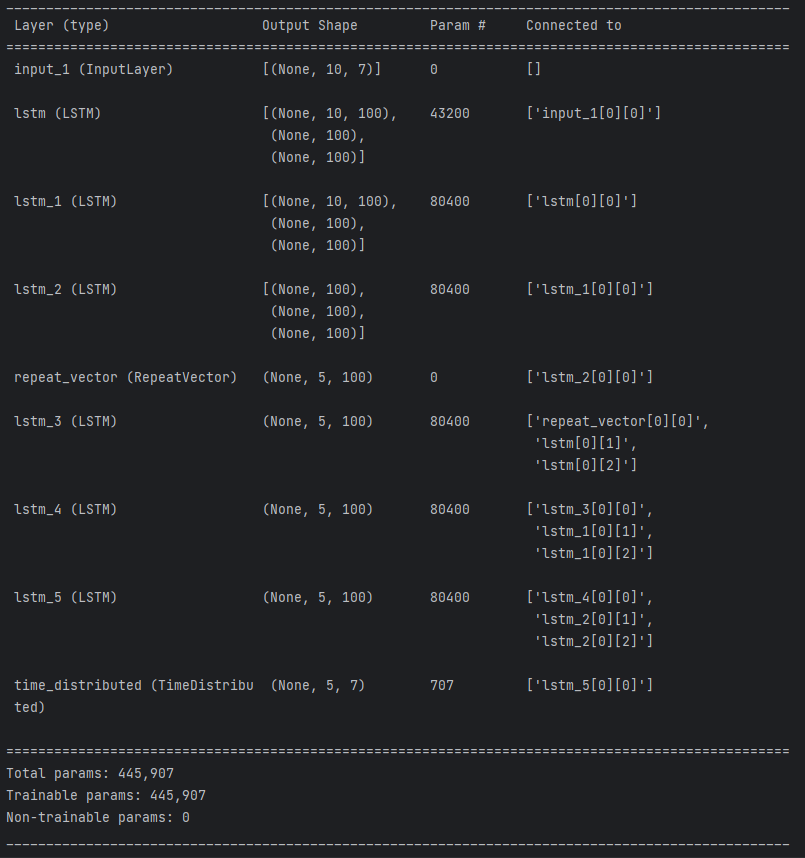
* Importing Libraries: The code starts with importing the required libraries, such as pandas, numpy, Matplotlib, TensorFlow, and MinMaxScaler from sklearn.preprocessing, etc.
* Loading the Data: The dataset used in this code is 'household\_power\_consumption.txt'. The dataset is loaded into a pandas dataframe using the read\_csv() function, which reads the dataset from a file in the directory and converts it to a pandas dataframe. The 'sep' parameter is set to ";" as the data is separated by a semicolon in the dataset. The 'header' parameter is set to 0, indicating that the first row contains the column names. The 'low\_memory' parameter is set to False to avoid memory-related warnings. The 'infer\_datetime\_format' parameter is set to True to infer the format of datetime automatically. The 'parse\_dates' parameter is set to a dictionary with two keys 'datetime' and 'time', where the values are the column indices (0 and 1) for datetime and time, respectively. The 'index\_col' parameter is set to 'datetime' to use datetime as the index of the dataframe.
* Imputing Missing Values: The code replaces all the '?' values in the dataframe with NaN values using the replace() function. After replacing the '?' with NaN, the isnull() function is used to count the number of missing values in each column. Then, a custom function fill\_missing() is defined, which takes the values from the dataframe as input and fills the missing values using the previous day's values (assuming that the missing values are due to power cuts). Finally, the fill\_missing() function is called with the values of the dataframe, and the isnull() function is used again to confirm if all the missing values have been filled.
* Down Sampling the Data: The data is downsampled from minute-wise to day-wise using the resample() function. The 'D' parameter is used to resample the data to daily frequency. The daily dataframe is stored in 'daily\_df', and the shape of both the original and downsampled dataframes is printed for comparison.
* Defining Input Parameters for the Model: The code sets the values for n\_past, n\_future, and n\_features. n\_past represents the number of past observations that will be used to predict the future, n\_future represents the number of future observations that will be predicted, and n\_features represent the number of features at each timestep in the data.
* Train-Test Split: The code splits the data into a training set and a test set using pandas' indexing. The first 1081 rows of the downsampled data are used as the training set, and the remaining rows are used as the test set.
* Scaling the Data: The code scales the training and test data using MinMaxScaler from sklearn.preprocessing. For each column of the training set, a scaler object is created, and the fit\_transform() function is used to scale the values between -1 and 1. The scaled values are reshaped to the original length of the array. The scaler objects are stored in a dictionary 'scalers'. The same scaler objects are then used to scale the test set using the transform() function. The scaled values are again reshaped to the original length of the array.
* Converting the Series to Samples for Supervised Learning: The code defines the split\_series() function to split the time-series data into samples for supervised learning. This function takes the time-series data, n\_past, and n\_future as input and returns

## Modal Architecture

The given model architecture is a sequence-to-sequence model with three encoder LSTM layers and two decoder LSTM layers.

The input to the model is a sequence of past data points, where each data point has n\_features features. The shape of the input is (n\_past, n\_features).

Here's a detailed explanation of the model architecture:

* The encoder\_inputs layer is created using the tf.keras.layers.Input function. It takes the shape of (n\_past, n\_features), which represents the input sequence.
* The first LSTM layer is created with tf.keras.layers.LSTM function. It has 100 units and is set to return sequences, state\_h, and state\_c. return\_sequences=True indicates that the layer returns the full sequence of outputs. return\_state=True indicates that the layer returns the final hidden and cell state output. The activation function used for this layer is tanh.
* The output of the first LSTM layer is passed to the second LSTM layer, which is similar to the first one. This layer also has 100 units, return\_sequences=True, return\_state=True, and an activation function of tanh. The encoder\_outputs1 variable stores the output of this layer.
* The third LSTM layer is created similarly to the first two layers. However, it has only one output, which is the final hidden state output. The activation function used for this layer is relu.
* The RepeatVector layer is added to repeat the final hidden state output from the encoder, n\_future number of times, which is the number of predicted future values.
* The decoder layers are created using the LSTM layers. The first LSTM layer takes the decoder\_inputs and the hidden and cell states from the first encoder layer. The second LSTM layer takes the output from the first LSTM layer and the hidden and cell states from the second encoder layer. The third LSTM layer takes the output from the second LSTM layer and the hidden and cell states from the third encoder layer.
* Finally, a TimeDistributed dense layer is used to produce the output for each time step in the output sequence. It applies the same dense layer to every time step of the output sequence.
* The model is created using tf.keras.models.Model function by specifying the inputs and outputs of the model. The input is the encoder\_inputs layer, and the output is the decoder\_outputs layer.

## Testing and Results

* The model is trained using the Adam optimizer and the mean absolute error (MAE) loss function.
* A learning rate scheduler is used to reduce the learning rate by a factor of 0.9 after each epoch.
* The trained model is then used to predict the test set, and the MAE between the predicted and actual values is calculated.

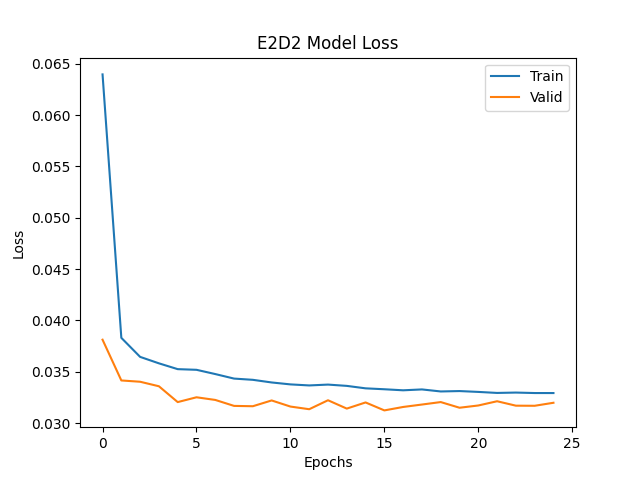
# Results

This section will provide the results for the proposed LSTM architecture.

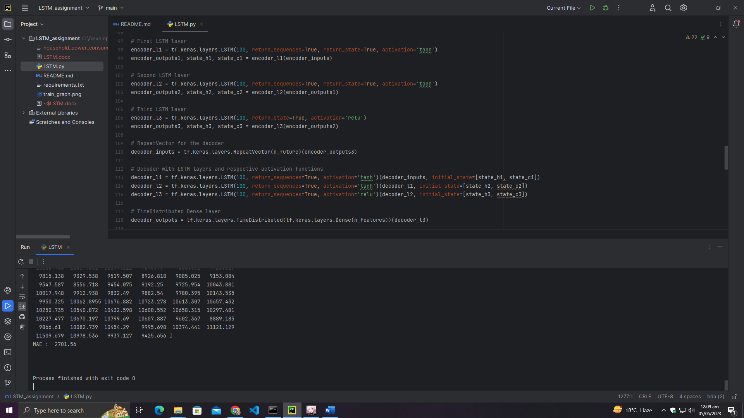
1. Results

| Attribute | MSE (Mean Squared Error) | | |
| --- | --- | --- | --- |
| Day 1 | Day 2 | Day 4 |
| Global\_active\_power | 249.27524 | 254.91064 | 267.10507 |
| Global\_reactive\_power | 32.16445 | 32.636368 | 32.78077 |
| Voltage | 1366.1659 | 1422.1338 | 1468.2426 |
| Global\_intensity | 1076.7666 | 1089.7192 | 1127.3901 |
| Sub\_metering\_1 | 984.3554 | 995.534 | 1000.3653 |
| Sub\_metering\_2 | 1348.1737 | 1358.3717 | 1365.2977 |
| Sub\_metering\_3 | 2470.8508 | 2484.8223 | 2558.2317 |

1. MSE results table

The above table shows the mean squared errors we get from the predicted values of different attributes for 3 days.

# Screenshots



# Conclusion

In this assignment, we first read in a time-series data of household power consumption and preprocessed it by replacing the missing values, downsampling the data from minutes to days, and scaling the values. Then, we converted the time-series data into supervised learning data by splitting it into past observations and future observations.

We then built a sequence-to-sequence LSTM model with three encoder LSTM layers and two decoder LSTM layers to forecast the next 3 days power consumption given the past 10 days of power consumption. The model architecture includes the RepeatVector layer for the decoder and TimeDistributed Dense layer for outputting predictions at each time step.

Finally, we trained the model on the training data and evaluated it on the test data. The model's performance was evaluated using mean absolute error (MAE), and the learning rate was reduced by 10% after every epoch using the LearningRateScheduler callback.

Overall, this code demonstrates how to build a sequence-to-sequence LSTM model for forecasting time-series data using TensorFlow and scikit-learn libraries.

##### References

1. LSTM: https://intellipaat.com/blog/what-is-lstm/?US#:~:text=LSTM%20Explained,-Now%2C%20let's%20understand&text=First%2C%20you%20must%20be%20wondering,especially%20in%20sequence%20prediction%20problems *(Website)*
2. RNN: https://www.ibm.com/topics/recurrent-neural-networks#:~:text=A%20recurrent%20neural%20network%20(RNN,data%20or%20time%20series%20data *(Website)*
3. Deep Learning: https://www.mathworks.com/discovery/deep-learning.html#:~:text=Deep%20learning%20is%20a%20machine,a%20pedestrian%20from%20a%20lamppost. *(Website)*
4. Keras LSTM: https://keras.io/api/layers/recurrent\_layers/lstm/ *(Website)*
5. Keras LSTM: https://towardsdatascience.com/understanding-lstm-and-its-quick-implementation-in-keras-for-sentiment-analysis-af410fd85b47 *(Website)*