**Deep Learning Model for Robot Tracking and Prediction**

horizontal line

[**Introduction 2**](#_6kiuklubajk9)

[**Neural Networks 2**](#_2bvkxmuzqatw)

[1. Sequential Data 2](#_ng2rb9tzvbd7)

[2. Recurrent Layers 2](#_jmvikp3yfsul)

[3. Hidden States (Specific to RNNs) 3](#_pp4tk0tvo8pz)

[4. RNN Variants 3](#_wybxbcwpra1y)

[5. Backpropagation Through Time (BPTT) (Specific to RNNs) 3](#_v1ic5tpgj7eh)

[**Key Concepts 4**](#_c26g4f31o8ft)

[Single Neuron Model 4](#_68cx28j0u7a)

[Key Differences: 4](#_oh1nleir799v)

[How It Works 5](#_rsgr6z9s1y6t)

[Data Structure: 5](#_l2z7s7v92tkv)

[Sequence of Inputs 6](#_vo1dwsxug5x8)

[Model Parameters 6](#_xpnr2ye5sfhp)

[**Pipeline 7**](#_gk3zv1wef3zl)

[Preprocessing 7](#_w5haxbiobvh5)

[Key Roles in Deep Learning: 7](#_qw674nzhjmto)

[Model Architecture 8](#_l6jgygc8217f)

[Model Training 9](#_hna4326t6i9e)

[Key Roles in Deep Learning: 9](#_pq110v8lhk3s)

[Model Evaluation 10](#_yknp9ak5af8f)

[Mean Absolute Error (MAE): 11](#_ku3jbzn2f1jm)

[Root Mean Squared Error (RMSE): 11](#_6628l5cq7fzj)

[Predictions and Visualization 12](#_smh9lzkeqfjk)

[**Result 12**](#_ej7cx9kj1e1m)

[**Conclusion 13**](#_dxgieq4hhp9y)

[References 13](#_ovrvr3acnoq3)

# 

# 

# Introduction

In this guide, we will learn how to perform a time series prediction of a robot's position using a single-neuron neural network. The model aims to predict the robot's next position based on its previous movements stored in a dataset. We'll use the movement data as our input features and the next position as our target output with a **fully connected neural network (single neuron)**.

# Neural Networks

**Recurrent Neural Networks (RNNs)** are commonly used for time-series tasks because they are designed to handle sequential data. In time series prediction, where past data influences future predictions, RNNs are ideal since they maintain a memory of previous inputs, allowing them to capture temporal dependencies more effectively.

**Why RNNs for Time-Series?**

Sequential Data: RNNs retain information from previous time steps, making them well-suited for tasks like robot movement prediction, stock forecasting, or any data where order and timing matter.

Temporal Patterns: Unlike a single neuron that looks at fixed-length windows of past data, RNNs maintain state across different time steps, allowing them to learn long-term dependencies.

For **Recurrent Neural Networks (RNNs)**, the key concepts we focus on are the architecture's ability to handle sequential data and learn temporal dependencies. Here's a brief breakdown of the key concepts:

### 1. Sequential Data

**Sequential Data** refers to the time-ordered data fed into RNNs. This could be time series, text, or any data where the order of the elements matters. For example, in the case of robot tracking, the sequence of positions over time is considered..

### 2. Recurrent Layers

The **Recurrent Layer** is the core of an RNN. It processes one element of the sequence at a time while maintaining a hidden state that carries information about prior elements.

This layer is designed to allow information to persist through time, capturing temporal dependencies and making it suitable for time series prediction.

### 3. Hidden States (Specific to RNNs)

**Hidden States** store the context of previous inputs. These states are updated with each new input in the sequence, allowing the network to remember information from previous time steps.

They are crucial for allowing the network to "remember" past data points while predicting future ones.

### 4. RNN Variants

There are different types of RNN architectures. Key variants include:

* **Vanilla RNN**: The simplest form, prone to vanishing gradients.
* **LSTM (Long Short-Term Memory)**: A more advanced version that handles long-term dependencies using internal gates.
* **GRU (Gated Recurrent Unit)**: A simpler variant of LSTM with fewer gates, but similarly powerful.

### 5. Backpropagation Through Time (BPTT) (Specific to RNNs)

**BPTT** is the algorithm used to train RNNs. It works similarly to backpropagation in feedforward networks but with the added complexity of unrolling the network through time, calculating the gradients across each time step.

In contrast, a single neuron neural network might suffice for simpler tasks where the patterns are linear and short-term, but for more complex, long-term dependencies, RNNs would perform better. Hence why it is used for the task ahead.

### 

### 

# 

# Key Concepts

### Single Neuron Model

Unlike deep neural networks or convolutional networks, this project uses a simple model: a single neuron. This neuron aims to learn the relationship between the robot's past positions and future movement.

#### **Key Differences:**

* A **dense layer** (as used in many neural networks) typically contains multiple neurons. Here, we use **one neuron** to map the input features to a single output (the next predicted position).
* Since the prediction task is relatively straightforward (predicting the next number in a sequence), a single neuron suffices to capture the movement pattern in this case.

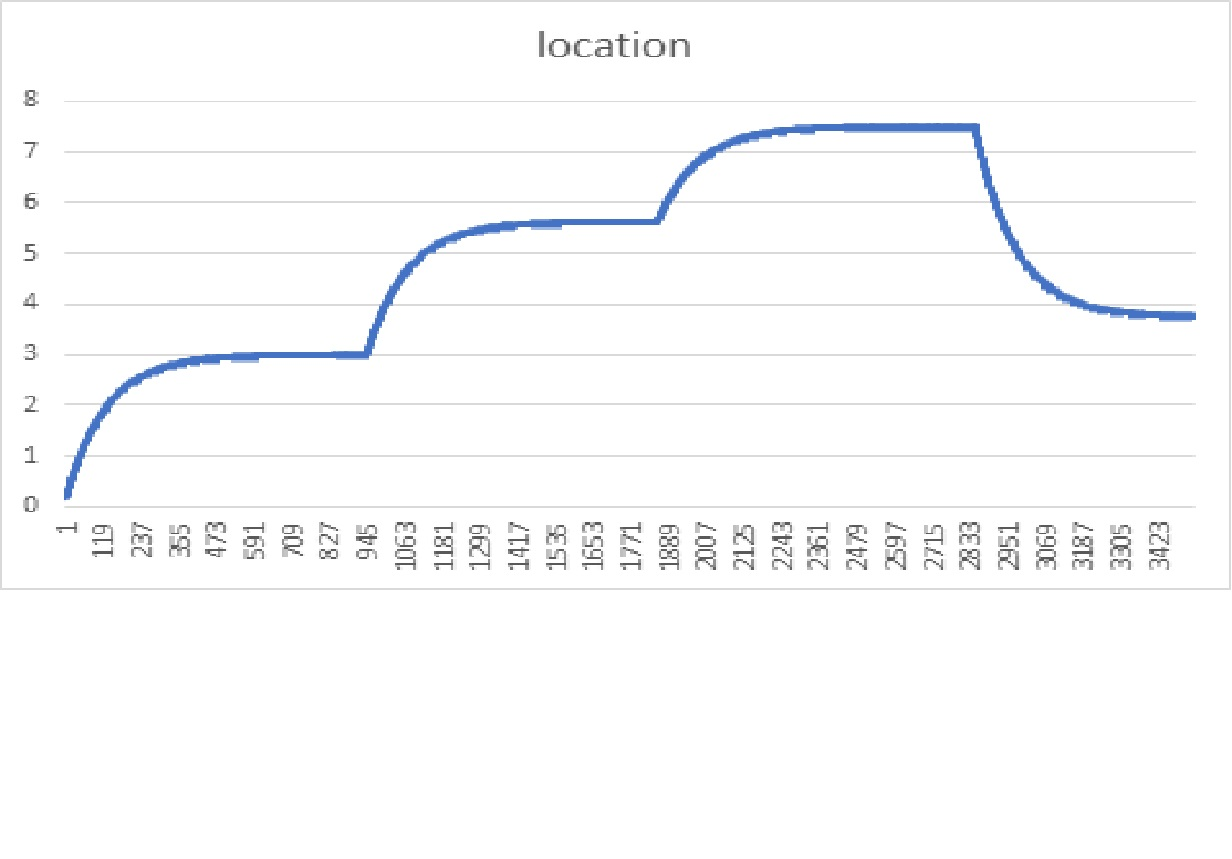
**Movement Data**

The data file is called ***track.txt***. This file contains the movements of a robot. The file has just one column of data – this generalised distance of the robot from you or a sensor (see fig below)

This is a snapshot of the single-column data containing the first couple of rows.

| location |
| --- |
| 0.2 |
| 0.2224 |
| 0.2546208 |
| 0.266663834 |
| 0.298530523 |
| 0.320222279 |

If you plot the positions against time (sample number), you will get the following graph.

The data represents the **robot's position** at specific time intervals. In this case, we treat it as a **time series**, where past data points (previous positions) are used to predict future data points (next position).

### How It Works

Our model uses the robot's previous positions as input to predict its next position. The key idea is that the neuron learns the pattern in the robot's movements over time.

For example, if we feed the neuron a sequence of 5 previous positions: Xt−4, Xt−3, Xt−2, Xt−1, XtX\_{t-4}, X\_{t-3}, X\_{t-2}, X\_{t-1}, X\_tXt−4​, Xt−3​, Xt−2​, Xt−1​, Xt​ The model will output a prediction for the next position, Xt+1X\_{t+1}Xt+1​.

#### **Data Structure:**

The input to the model consists of several past positions (e.g., 5 previous positions), and the output is the next position. This is repeated over the entire dataset to create training samples.

### Sequence of Inputs

The data is split into sliding windows of previous positions and corresponding next positions. We use a **window size of 5**, which means the model looks at the last 5 positions to predict the next one.

### 

### Model Parameters

This model uses a **dense layer** with:

* **1 neuron**: This is a single output neuron.
* **Sigmoid activation function**: The sigmoid activation function is useful in time series tasks because it introduces non-linearity, allowing the model to capture complex patterns and dependencies over time.
* **Mean Absolute Error**: The loss function is used to minimize the error between predicted and actual positions.
* **Adam Optimizer**: The Adam optimizer is well-suited for time series tasks due to its adaptive learning rate and momentum. It adjusts the learning rate for each parameter based on the gradient history, which helps in efficiently handling the often noisy and non-stationary nature of time series data2. This leads to faster convergence and better performance.

Now It is time to use the single neuron model, movement data, sequence of inputs, and Model parameters to predict the movement of the robot.

# Pipeline

**Data Loading**

The data is loaded from the track.txt file and a plot is generated showing the robot's position over time to understand the trend.

| # Load data  df = pd.read\_csv('/content/track (1).txt')  location = df.values[1:3531]  # Plot data  plt.plot(location) plt.title('location') plt.xlabel('Time')  plt.ylabel('Position') plt.legend('data set') |
| --- |

### Preprocessing

Preprocessing is a crucial step in preparing raw data to make it suitable for deep learning models. Properly preprocessed data ensures that the model learns patterns efficiently and reduces the risk of poor performance due to irrelevant variations or scale imbalances in the data.

#### **Key Roles in Deep Learning:**

**Normalization/Scaling**:

**Purpose**: To ensure that all input features have a similar scale, typically within a specific range like [0, 1].

**Why It Matters**: Deep learning models are sensitive to the scale of the input features. Without normalization, features with larger scales may dominate and influence the model disproportionately, leading to suboptimal learning.

**Example in Your Project**: MinMaxScaler is used to scale the robot's positional data into a range of [0, 1], making it easier for the model to interpret and learn patterns from this sequence.

**Sliding Window/Sequence Creation**:

**Purpose**: To convert continuous time-series data into meaningful input/output pairs.

**Why It Matters**: The model needs past information to predict future values. By breaking down the data into smaller sequences (e.g., using 5 previous positions to predict the next), you provide the model with the necessary context to learn dependencies and relationships.

**Example in Your Project**: The robot’s positions are divided into sequences of 5 previous positions as input and the next position as the output, helping the model understand the time-based dependencies of movement.

| # Normalize data scaler = MinMaxScaler(feature\_range=(0, 1)) dataset = scaler.fit\_transform(location.reshape(-1, 1))  # Split the sequence into samples number\_steps = 5 def split\_sequence(sequence, number\_steps):  X, y = [], []  for i in range(len(sequence) - number\_steps):  X.append(sequence[i:i+number\_steps])  y.append(sequence[i+number\_steps])  return np.array(X), np.array(y)  X, y = split\_sequence(location, number\_steps) |
| --- |

### Model Architecture

* A single neuron is adequate for this task because the dataset's sequential nature involves a simple, direct relationship between past and future positions. The task doesn't require complex feature extraction or pattern recognition, so a single neuron can effectively model the linear or predictable trend in the robot's movement without unnecessary complexity, reducing the risk of overfitting.
* A sequential neural network model is built using Keras. The model has:
  + A dense layer with 50 units and a sigmoid activation function.
  + A dropout layer to prevent overfitting by randomly disabling some neurons during training.
  + Another dense layer with 1 unit (output layer) to predict the next robot position.

| # Build the model model = Sequential() model.add(Dense(50, activation='sigmoid', kernel\_regularizer='l2', input\_shape=(5,))) model.add(Dropout(0.2)) model.add(Dense(1, kernel\_regularizer='l2')) |
| --- |

### Model Training

Model training is the process through which the neural network learns patterns and relationships in the data. It adjusts its internal parameters (weights and biases) to minimize errors, improving its predictions over time.

#### **Key Roles in Deep Learning:**

**Optimisation with Adam**:

**Purpose**: To adjust the model’s weights to minimize the loss function during training.

**Why It Matters**: Adam is an efficient gradient-based optimiser that adapts the learning rate during training, making it faster and more effective for complex problems. It also incorporates both momentum and adaptive learning rates, allowing the model to converge to an optimal solution faster.

**Example in the Project**: Adam is chosen to optimize the model’s weights, with a learning rate of 0.001, which ensures smooth updates to the model parameters, speeding up the learning process.

**Loss Function**:

**Purpose**: The loss function (MSE in this case) quantifies the error between predicted and actual values. It guides the optimiser on how much to adjust the weights during each iteration.

**Why It Matters**: A lower loss means better performance. The RMSE is chosen here because it penalises larger errors more, making it effective for time-series tasks where precision is important.

**Example in the Project**: By minimising the RMSE, the model improves its accuracy in predicting the robot’s next position.

**Early Stopping**:

**Purpose**: To prevent overfitting by halting training once the model stops improving on the validation set.

**Why It Matters**: Overfitting occurs when the model becomes too specialized to the training data, performing poorly on unseen data. Early stopping ensures the model generalises well by stopping training when no further improvements are made to validation data.

**Example in the Project**: Early stopping is set to 10 epochs, meaning the model will stop training if there is no improvement in validation loss for 10 consecutive epochs, protecting against overfitting.

| # Compile the model learning\_rate = 0.001 model.compile(loss='mean\_squared\_error', optimizer=Adam(learning\_rate=learning\_rate), metrics=['mse', 'mae'])  # Implement Early Stopping early\_stopping = EarlyStopping(patience=10, restore\_best\_weights=True)  # Train the model history = model.fit(X\_train, y\_train, epochs=1000, batch\_size=64, verbose=0,  validation\_data=(X\_test, y\_test), callbacks=[early\_stopping]) |
| --- |

### Model Evaluation

* The model’s performance is evaluated using Root Mean Squared Error (RMSE) and Mean Absolute Error (MAE) for both the training and testing sets.
* These metrics help assess how well the model predicts the robot’s movement.

| # Evaluate the model train\_score = model.evaluate(X\_train, y\_train, verbose=0) test\_score = model.evaluate(X\_test, y\_test, verbose=0) print("Train RMSE: %.2f; Train MAE: %.2f" % (np.sqrt(train\_score[1]), train\_score[2])) print("Test RMSE: %.2f; Test MAE: %.2f" % (np.sqrt(test\_score[1]), test\_score[2])) |
| --- |

#### Mean Absolute Error (MAE):

MAE is the average of the absolute differences between predicted and actual values. It gives a straightforward measure of error by treating all differences equally.

Formula:

Mean Absolute Error = (1/n) \* ∑|yi – xi|

where,

Σ: Greek symbol for summation

yi: Actual value for the ith observation

xi: Calculated value for the ith observation

n: Total number of observations

Interpretation: MAE represents the average magnitude of error, without considering the direction of errors (positive or negative).

#### Root Mean Squared Error (RMSE):

RMSE is the square root of the average of the squared differences between predicted and actual values. It penalizes larger errors more heavily than MAE because it squares the error before averaging.

Formula:

RMSE = √Σ(Pi – Oi)2 / n

where,

Σ is a fancy symbol that means “sum”

Pi is the predicted value for the ith observation in the dataset

Oi is the observed value for the ith observation in the dataset

n is the sample size

Interpretation: RMSE is more sensitive to outliers or large errors because it squares the differences. A lower RMSE indicates a better fit, but it can be more impacted by larger errors compared to MAE.

Both evaluation metrics are used as RMSE gives higher weight to large errors (due to squaring), while MAE treats all errors equally.

#### Predictions and Visualization

* The model predicts the robot’s future positions based on the input data.
* The predicted values are compared to the actual values and plotted to visualize the accuracy of the predictions.

| # Predict the entire sequence predicted\_location = model.predict(X)  # Plot the predicted and actual location plt.plot(location, label='Actual Location') plt.plot(predicted\_location, label='Predicted Location') plt.title('Robot Location Predicted') plt.xlabel('Time') plt.ylabel('Position') plt.legend() plt.show() |
| --- |

# Result

The following results show the RMSE and MAE for the training and testing sets.

Train RMSE: 0.31; Train MAE: 0.23

Test RMSE: 0.22; Test MAE: 0.20

Lower values indicate better performance, this indicates that the model generalizes well to unseen data and based on the results, the model does not appear to overfit since the test error is lower than the training error.

# Conclusion

The model effectively predicts the robot’s movements based on previous positions. However, potential improvements could involve tuning hyperparameters or using more advanced architectures such as but not limited to;

**Recurrent Neural Networks (RNNs):**

Why: RNNs are specifically designed to handle sequential data like time series by retaining information from previous time steps. This allows the model to efficiently capture temporal patterns in the robot's movement.

How: Unlike a dense layer that looks at static windows, RNNs have loops that allow information to persist over time, making them ideal for time-dependent tasks.

**Long Short-Term Memory (LSTM) Networks:**

Why: LSTMs are a specialized form of RNN that overcome the problem of vanishing gradients, making them better at learning long-term dependencies in data.

How: LSTMs have internal mechanisms (gates) to regulate the flow of information, allowing them to learn which past information is important for predicting future positions.

# References

* Keras library for neural network implementation.
* Matplotlib for data visualization.
* Scikit-learn for data preprocessing.