Tab 1

**Peramaeters of gru with its explainantion and what if we change it?**

Excellent question! Let's break down the key **GRU parameters** (in Keras / TensorFlow) and what happens when you change them. This helps you tune the model more intelligently.

## **📦 GRU Layer Parameters (Keras/TensorFlow)**

GRU(

units,

activation='tanh',

recurrent\_activation='sigmoid',

use\_bias=True,

dropout=0.0,

recurrent\_dropout=0.0,

return\_sequences=False,

return\_state=False,

go\_backwards=False,

stateful=False,

unroll=False

)

### **🔑 Important Parameters (You Actually Tune)**

| **Parameter** | **Description** | **If You Increase...** | **If You Decrease...** |
| --- | --- | --- | --- |
| units | Number of GRU neurons (aka hidden state size) | Better capacity, may overfit or slow training | Lower capacity, may underfit |
| activation | Activation function for **cell output** ('tanh' is default) | Try 'relu' or 'swish' for non-saturated | May affect stability if changed carelessly |
| recurrent\_activation | Used for **gates** (default: 'sigmoid') | Usually keep as 'sigmoid' | Rarely changed; may destabilize gates |
| dropout | Dropout on **inputs** between GRU cells (not weights) | Helps prevent overfitting | May cause overfitting if too low |
| recurrent\_dropout | Dropout on **recurrent connections** (state-to-state) | Helps regularize sequence memory | Can lead to overfitting |
| return\_sequences | If True, returns output **for all time steps** (for stacking RNNs) | Required if stacking or attention layer follows | Outputs full sequence |
| go\_backwards | If True, processes sequence **backward** | Adds robustness if combined with forward layer | Can be confusing alone |
| stateful | Keeps hidden states across batches (like a memory) | Needed for time-dependent inference | Usually False unless special case |

### **✅ Common Configurations**

| **Task Type** | **units** | **dropout** | **return\_sequences** | **Notes** |
| --- | --- | --- | --- | --- |
| Simple time-series regression | 64–128 | 0.2–0.3 | False | Good for single-output predictions |
| Deep stacked GRU | 128–256 | 0.2–0.4 | True (except last) | Return full sequence in layers 1–N-1 |
| Multistep forecasting | 64–128 | 0.2 | True or False | Depends on how you're decoding |

### **📈 What Happens If You Change Them?**

#### **🔹 units=128 → 256**

* Pros: Learns more complex patterns
* Cons: More prone to overfitting, longer training

#### **🔹 dropout=0.1 → 0.4**

* Pros: Reduces overfitting
* Cons: May underfit if too high, especially with small data

#### **🔹 recurrent\_dropout=0.0 → 0.3**

* Pros: Adds regularization to memory connections
* Cons: Slower training, possible instability on small sequences

#### **🔹 return\_sequences=False → True**

* Pros: Enables stacking more GRU layers or using attention
* Cons: Increases output dimensionality; not compatible with some output layers

### **🎯 Example Optuna-Tuned GRU Layer**

model = Sequential()

model.add(GRU(units=128,

activation='tanh',

recurrent\_activation='sigmoid',

dropout=0.3,

recurrent\_dropout=0.2,

return\_sequences=False,

input\_shape=(X\_train.shape[1], X\_train.shape[2])))

model.add(Dense(1))

### **🧠 Tips for Tuning**

* **Start with units=64–128**, increase if model underfits
* **Use dropout ≥ 0.2** if validation loss diverges from training
* **Don't go crazy on recurrent\_dropout** — use ≤ 0.3
* If **stacking GRUs**, set return\_sequences=True in all but last

Would you like:

* A visual diagram of how GRU works internally?
* Suggested GRU configs for multi-step forecasting or classification?

30-07-2025

model = Sequential([

GRU(units=256, dropout=0.2, recurrent\_dropout=0.2,

activation='tanh', recurrent\_activation='sigmoid',

input\_shape=(X\_train.shape[1], X\_train.shape[2])),

Dense(1)

])

model.compile(optimizer='adam', loss='mse', metrics=['mae'])

model.summary() explain it

This code snippet is defining and compiling a **Recurrent Neural Network (RNN)** model using **GRU (Gated Recurrent Units)**, a type of RNN that is commonly used for time series forecasting. Let's break down each part of the model definition:

### **🧠 Model Architecture:**

model = Sequential([

GRU(units=256, dropout=0.2, recurrent\_dropout=0.2,

activation='tanh', recurrent\_activation='sigmoid',

input\_shape=(X\_train.shape[1], X\_train.shape[2])),

Dense(1)

])

#### **1. Sequential() Model:**

* Sequential() is a linear stack of layers in Keras.
* Each layer has exactly one input tensor and one output tensor, making it easier to build the model layer by layer.

#### **2. GRU Layer:**

GRU(units=256, dropout=0.2, recurrent\_dropout=0.2,

activation='tanh', recurrent\_activation='sigmoid',

input\_shape=(X\_train.shape[1], X\_train.shape[2]))

This is the **core recurrent layer** in the model and uses the **GRU** (Gated Recurrent Unit) architecture.

* **units=256**: Specifies the number of **units** (also called neurons) in this GRU layer. Each unit will maintain its internal state and output a value based on the current input and its memory of the previous inputs. You have 256 units, which is a pretty good size for capturing complex temporal patterns.
* **dropout=0.2**: This is a **dropout rate** applied to the input connections of the GRU layer. It randomly sets 20% of the input features to 0 during each training iteration to prevent overfitting.
* **recurrent\_dropout=0.2**: Similar to regular dropout, but this is applied to the **recurrent connections** (connections between time steps) in the GRU layer. This helps prevent the model from memorizing the training data and encourages it to generalize better.
* **activation='tanh'**: Specifies the **activation function** used for the output of each GRU unit. tanh (hyperbolic tangent) squashes the output values to a range between -1 and 1, which helps the model to maintain gradients during backpropagation.
* **recurrent\_activation='sigmoid'**: This defines the activation function used for the recurrent step. sigmoid is commonly used in GRUs and LSTMs (Long Short-Term Memory networks) to regulate the flow of information (i.e., deciding how much of the previous state to remember or forget).
* **input\_shape=(X\_train.shape[1], X\_train.shape[2])**: This defines the shape of the input data. Since this is a time series problem:  
  + X\_train.shape[1]: The number of time steps in each sequence (this is the sequence length you used earlier, which is 10 in your case).
  + X\_train.shape[2]: The number of features at each time step (this would be the number of columns in your scaled feature data after preprocessing).

#### **3. Dense Layer:**

Dense(1)

* **Dense(1)**: This is a **fully connected layer** (dense layer) with a single output unit, which is appropriate for regression problems like this one (forecasting dengue cases).  
  + Since you're predicting a **single continuous value** (the number of dengue cases), you only need one output unit.
  + This layer will output the predicted value (scaled), and it will be compared to the true target (scaled) using the loss function.

### **🧑‍🏫 Model Compilation:**

model.compile(optimizer='adam', loss='mse', metrics=['mae'])

* **optimizer='adam'**:  
  + **Adam** is an efficient optimization algorithm that adjusts learning rates during training. It’s widely used because it adapts the learning rate dynamically based on the gradients.
  + Adam is a good choice for most tasks, especially with RNNs like GRU.
* **loss='mse'**:  
  + **MSE** (Mean Squared Error) is used as the loss function. It's a common loss function for regression problems, and it measures how close the predicted values are to the actual values.
  + The model will minimize this error during training.
* **metrics=['mae']**:  
  + **MAE** (Mean Absolute Error) is another evaluation metric. While the model optimizes for MSE during training, **MAE** gives you a more interpretable measure of the error (the average absolute difference between predicted and true values).
  + MAE is often easier to understand because it’s in the same units as your target variable (dengue cases).

### **📊 Model Summary:**

model.summary()

* This prints out a summary of the model architecture, including the number of parameters (weights and biases) in each layer and the total number of trainable parameters.  
  + This helps you understand the complexity of the model and how many parameters are being learned.

### **✅ Overall Explanation:**

* The model consists of a **GRU layer** that learns from sequences of historical data and a **Dense layer** that outputs the predicted number of dengue cases.
* The model is compiled using the **Adam optimizer**, **MSE** loss, and **MAE** metric for evaluation.
* After training, the model will output predictions that you can transform back to the original scale using the target\_scaler.

Let me know if you need help with training the model or evaluating its performance!

Model.summary

The output of model.summary() provides a detailed breakdown of your model's architecture. Let's break down each part of this summary:

### **🔍 Layer Summary Breakdown**

#### **1. GRU Layer:**

gru\_9 (GRU) │ (None, 256) │ 215,040 │

* **Layer Name**: gru\_9  
  + This is just the name of the GRU layer (it’s automatically named gru\_9 in this case, with 9 being the sequential identifier).
* **Output Shape**: (None, 256)  
  + **None**: This is the **batch size**. It’s None because the batch size is not specified when you define the model (it will be determined during training). The model can accept any batch size.
  + **256**: This is the **number of units** in the GRU layer, which is what the output shape refers to. Each of the 256 units will output a value at each time step, and the output for the entire sequence is the **last hidden state** from the GRU. So, after processing the entire sequence, this layer outputs a vector with 256 values.
* **Parameter Count**: 215,040  
  + This is the number of **trainable parameters** in the GRU layer.
  + The GRU layer has several components (gates for forgetting, updating, etc.), and the number of parameters includes the weights and biases for each of these components.
  + The formula for calculating the number of parameters in a GRU layer is:  
     Parameters=3×(input size+units)×units+3×units\text{Parameters} = 3 \times (\text{input size} + \text{units}) \times \text{units} + 3 \times \text{units}
    - This includes:  
      * Weights for input-to-hidden state connections.
      * Weights for hidden state-to-hidden state connections (recurrent).
      * Biases for each gate (reset, update, and output gates).

#### **2. Dense Layer:**

dense\_7 (Dense) │ (None, 1) │ 257 │

* **Layer Name**: dense\_7  
  + This is the name of the Dense layer (automatically named dense\_7 here).
* **Output Shape**: (None, 1)  
  + **None**: Again, this refers to the batch size, which is flexible and not fixed.
  + **1**: The Dense layer outputs a single value per example. In your case, this will be the **predicted dengue cases** (after applying a linear activation, since there’s no activation function specified).
* **Parameter Count**: 257  
  + This is the number of trainable parameters in the Dense layer. It is calculated based on:  
    - The **number of units** in the previous layer (256 from GRU).
    - The **number of units** in this Dense layer (1).
    - **Weight matrix** size: (256 units from GRU) × (1 output) = 256 weights
    - **Bias**: 1 bias term for the output.
  + Total parameters = 256 (weights) + 1 (bias) = 257 parameters.

### **🧑‍🏫 Summary:**

* The model consists of:  
  1. A **GRU layer** with 256 units, which outputs a sequence of 256 values for each input sequence.
  2. A **Dense layer** that takes these 256 values and outputs a single predicted value (the predicted dengue cases).
* **Total Parameters**: The total number of parameters in the model is the sum of the parameters in the GRU and Dense layers:  
   215,040 (GRU)+257 (Dense)=215,297 parameters215,040 \text{ (GRU)} + 257 \text{ (Dense)} = 215,297 \text{ parameters}

These parameters are learned during training to minimize the loss function and improve the model's ability to make predictions.

Let me know if you want further clarification or more details about training!

Tab 4

early\_stop = EarlyStopping(monitor='val\_loss', patience=20, restore\_best\_weights=True)

history = model.fit(

X\_train, y\_train,

epochs=100,

batch\_size=64,

validation\_split=0.2,

callbacks=[early\_stop],

verbose=1

) explain code

This code snippet is used to **train the GRU-based model** with early stopping, which helps prevent **overfitting** by stopping training once the model’s performance on a validation set stops improving.

### **Explanation of Code**

### **1. EarlyStopping Callback:**

early\_stop = EarlyStopping(monitor='val\_loss', patience=20, restore\_best\_weights=True)

* **monitor='val\_loss'**:  
  + This tells the **EarlyStopping** callback to monitor the **validation loss** during training.
  + The validation loss is the loss computed on the validation set (validation\_split=0.2 in this case). The model uses this metric to evaluate how well it's generalizing to unseen data (not the data it’s trained on).
* **patience=20**:  
  + This defines how many epochs the model will continue to train **without improvement** in the monitored metric (val\_loss here) before stopping.
  + **Patience of 20** means the training will stop if the validation loss doesn’t improve for **20 consecutive epochs**.
* **restore\_best\_weights=True**:  
  + This ensures that once training is stopped, the model **restores the best weights** (i.e., the model weights corresponding to the epoch with the lowest validation loss).
  + This is important because sometimes the validation loss may start increasing slightly towards the end of training, but the model would have performed best at an earlier epoch.

### **2. Model Training:**

history = model.fit(

X\_train, y\_train,

epochs=100,

batch\_size=64,

validation\_split=0.2,

callbacks=[early\_stop],

verbose=1

)

* **X\_train** and **y\_train**:  
  + These are your **training data** (features and target values) that the model will use to learn the patterns.
* **epochs=100**:  
  + This specifies the **maximum number of training epochs** (iterations through the entire training dataset). The model will train for up to **100 epochs**, but early stopping may stop it earlier if there’s no improvement in the validation loss for 20 epochs in a row.
* **batch\_size=64**:  
  + This is the number of samples used in one forward and backward pass during training.
  + A **batch size of 64** means that the model will process 64 samples at once before updating the weights. Smaller batch sizes often allow faster training, but larger batches tend to provide more stable gradients.
* **validation\_split=0.2**:  
  + This sets aside **20%** of the training data for validation. During each epoch, the model will evaluate its performance on this **validation set** to determine if it is overfitting (if validation loss starts increasing).
* **callbacks=[early\_stop]**:  
  + This passes the **EarlyStopping** callback you defined earlier. This callback monitors the validation loss and will stop training early if needed.
* **verbose=1**:  
  + This controls how much information is displayed during training.
  + **verbose=1** means it will display progress updates with one line per epoch. It includes metrics like loss and accuracy (if defined) for both the training and validation sets.

### **What Happens During Training?**

1. **Epoch 1**: The model trains on the training data for the first epoch, calculating both the **training loss** and **validation loss**.
2. **Subsequent Epochs**: The model continues training and evaluates its performance on the validation set at the end of each epoch. The validation loss is used to monitor if the model is improving or not.
3. **Early Stopping**: If the validation loss stops improving for **20 consecutive epochs**, the training stops early, and the model’s weights from the best epoch (lowest validation loss) are restored.
4. **Training History**:  
   * The history object stores the training history, including **training loss** and **validation loss** for each epoch, which can be used for further analysis or plotting training curves.

### **Why Use Early Stopping?**

* **Prevents Overfitting**: Without early stopping, a model can continue to improve on the training set but start performing worse on the validation set as it "memorizes" the training data instead of learning generalizable patterns.
* **Saves Time**: If the model's performance stops improving after a certain number of epochs, training can be stopped early, saving computational resources.

### **Summary of the Training Flow**

* The model will be trained for up to 100 epochs (with a batch size of 64) using the training data (X\_train and y\_train).
* The training will stop early if the validation loss doesn't improve for 20 epochs, and the model will **restore the best weights** from when the validation loss was lowest.
* After training, the history object will contain detailed metrics (like loss) for both the training and validation sets, which can be plotted to visualize the model’s performance.

Let me know if you want to explore how to visualize this training history or analyze the model's performance!

lag Feture

df = df\_no\_outliers.copy()

for lag in [1, 2, 3]:

df[f'total\_cases\_lag{lag}'] = df['total\_cases'].shift(lag)

df.dropna(inplace=True)

target\_col = 'total\_cases'

features = df.drop(columns=[target\_col]) explain code

This code is preparing your time series data for modeling by creating **lagged features** and setting up the **target column** and **features** for training. Let's go through it step by step.

### **1. Copying the Data:**

df = df\_no\_outliers.copy()

* **df\_no\_outliers.copy()**:  
  + This creates a copy of the df\_no\_outliers DataFrame, likely after you’ve already removed any **outliers** from the original data.
  + Using .copy() ensures that you don’t modify the original DataFrame by mistake while working with the new df DataFrame.

### **2. Creating Lagged Features:**

for lag in [1, 2, 3]:

df[f'total\_cases\_lag{lag}'] = df['total\_cases'].shift(lag)

* **Lagged Features**:  
  + This loop creates lagged versions of the total\_cases column, which is common in time series forecasting.
  + **Lagging** means shifting the total\_cases column by lag steps (time periods). The purpose of lagging is to provide the model with information about previous time steps, which helps it learn temporal patterns.

#### **What happens in the loop?**

* For **lag = 1**:  
  + The new feature column total\_cases\_lag1 is created, which is the value of total\_cases shifted by 1 time step.
  + Example: For time\_step = 1, the lagged value will be total\_cases[0] for total\_cases\_lag1[1].
* For **lag = 2**:  
  + The new feature column total\_cases\_lag2 is created, which is the value of total\_cases shifted by 2 time steps.
* For **lag = 3**:  
  + Similarly, total\_cases\_lag3 is created.

#### **The shift(lag) function:**

* .shift(lag) moves the data forward by lag positions, filling the missing values at the start with NaN.
* For example, if the original data looks like [10, 20, 30, 40, 50]:  
  + shift(1) would result in [NaN, 10, 20, 30, 40].
  + shift(2) would result in [NaN, NaN, 10, 20, 30].
  + shift(3) would result in [NaN, NaN, NaN, 10, 20].

### **3. Dropping Rows with Missing Values:**

df.dropna(inplace=True)

* After creating lagged features, some rows will have NaN values due to the shifting operation (e.g., the first few rows will have missing values for the lagged columns).
* **df.dropna(inplace=True)** removes all rows that have any NaN values. This is done to ensure that the data is complete for modeling. The **inplace=True** argument means that the changes will be made directly to the df DataFrame, without needing to assign it to a new variable.

### **4. Setting Target and Features:**

target\_col = 'total\_cases'

features = df.drop(columns=[target\_col])

* **Setting the Target**:  
  + target\_col = 'total\_cases' defines the **target column** (the variable you're trying to predict) as total\_cases.
* **Defining the Features**:  
  + features = df.drop(columns=[target\_col]) removes the **target column** (total\_cases) from the DataFrame, leaving only the **feature columns** (which will include the lagged versions of total\_cases).
  + The resulting features DataFrame contains the predictor variables, which include:  
    - The original total\_cases\_lag1, total\_cases\_lag2, total\_cases\_lag3, and potentially other features in the dataset.

### **Summary of the Process:**

1. **Copy the Data**: A copy of the data without outliers is made.
2. **Create Lagged Features**:  
   * Lagged features (total\_cases\_lag1, total\_cases\_lag2, total\_cases\_lag3) are added to the dataset.
   * Each lag represents the value of the target variable (total\_cases) from previous time steps.
3. **Drop Missing Values**: Rows with any NaN values (created by shifting) are removed.
4. **Define Target and Features**:  
   * The target\_col is set to 'total\_cases', which is what you're predicting.
   * The features DataFrame contains all the other variables, including the lagged features, which will be used as inputs for training.

### **Why Create Lagged Features?**

* Time series data often depends on previous values. By including lagged versions of total\_cases, you're allowing the model to learn from past observations (e.g., the number of cases 1, 2, or 3 weeks ago) in order to make future predictions.

Let me know if you’d like help with further steps like splitting the data for training or building the model!

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Great questions! Let's address them clearly one by one:

## **✅ Why do we use lag features in time series?**

### **🔁 What is a *lag feature*?**

A **lag feature** is simply a value from a **previous time step**. For example:

* total\_cases\_lag1 is the number of dengue cases **one week ago**.
* total\_cases\_lag2 is from **two weeks ago**, and so on.

### **🧠 Why use lag features?**

Because in time series data, **the future is usually influenced by the past**.

For example:

If dengue cases were rising in the past few weeks, they're likely to remain high or increase in the next week too.

So, by including previous values (lags), the model can **learn temporal patterns**, such as:

* Seasonality (e.g., cases increase during the rainy season).
* Trends (e.g., a gradual rise or fall in cases).
* Short-term fluctuations.

Without lag features, the model has no memory of what happened in previous time steps—it would treat each week as independent, which breaks the logic of time series forecasting.

## **🔢 Why use lag 1, 2, and 3 only?**

### **📏 Short answer:**

You chose **lag 1, 2, and 3** as a **window size of 3 weeks** to give the model some recent history. This is a **manual design choice**.

But you could choose more lags, such as 4, 6, 12, or even 52 (one year), depending on:

* **The frequency of your data** (weekly, monthly, daily).
* **Domain knowledge** (e.g., how long outbreaks typically last).
* **Performance** (you can test different lag lengths and pick what gives best accuracy).

### **💡 Tip:**

Try different lags (e.g., [1, 2, 3, 4, 5, 6]) and use cross-validation or a validation set to find which combination gives the best predictions.

## **📦 Where are lag features stored and used after creation?**

### **After creating lag features:**

for lag in [1, 2, 3]:

df[f'total\_cases\_lag{lag}'] = df['total\_cases'].shift(lag)

They become **new columns** in your df DataFrame:

total\_cases\_lag1 | total\_cases\_lag2 | total\_cases\_lag3

-------------------------------------------------------

20 | 18 | 22

### **Then:**

features = df.drop(columns=[target\_col])

* These lag features become part of your **input features** (features) used to **train the model**.

Your target (y) is still total\_cases, and the model will try to **learn a mapping from lag features → future total\_cases**.

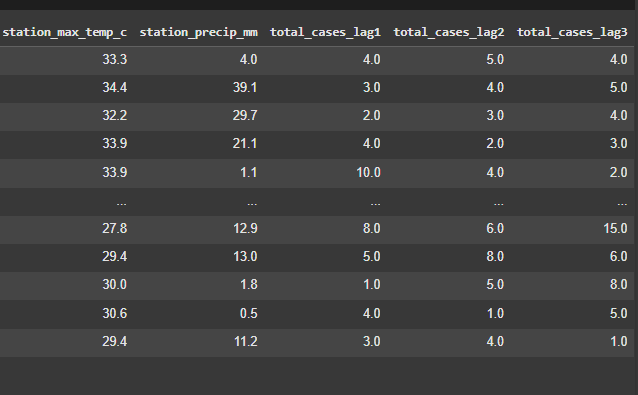
### **Final flow:**

1. **Create lag features** → df
2. **Drop missing values** → clean df
3. **Separate features and target** → features, target\_col
4. **Scale** (if needed)
5. **Train/test split**
6. **Train model** using lagged features as input
7. **Predict future cases**

## **🔚 Summary**

| **Concept** | **Purpose** |
| --- | --- |
| Lag features | Provide the model with past data to help predict future values |
| Lags 1, 2, 3 | Capture short-term memory of the system (last 3 time steps) |
| Stored in | New columns in your DataFrame |
| Used in | features for model input, while total\_cases is your prediction target |

If you're interested, I can help you automate lag selection, test different lag combinations, or plot feature importance.



Optuna

import optuna

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import GRU, Dropout, Dense

from tensorflow.keras.callbacks import EarlyStopping

from sklearn.metrics import mean\_squared\_error

def create\_gru\_model(input\_shape, trial):

model = Sequential()

n\_layers = trial.suggest\_int('n\_layers', 1, 3)

for i in range(n\_layers):

units = trial.suggest\_int(f'units\_l{i}', 64, 256, step=32)

dropout = trial.suggest\_float(f'dropout\_l{i}', 0.1, 0.5)

return\_seq = i < n\_layers - 1

if i == 0:

model.add(GRU(units, return\_sequences=return\_seq, input\_shape=input\_shape))

else:

model.add(GRU(units, return\_sequences=return\_seq))

model.add(Dropout(dropout))

model.add(Dense(1))

optimizer = trial.suggest\_categorical('optimizer', ['adam', 'nadam', 'rmsprop'])

model.compile(optimizer=optimizer, loss='mse')

return model

def objective(trial):

model = create\_gru\_model((X\_train.shape[1], X\_train.shape[2]), trial)

early\_stop = EarlyStopping(monitor='val\_loss', patience=15, restore\_best\_weights=True)

model.fit(

X\_train, y\_train,

validation\_split=0.2,

epochs=100,

batch\_size=trial.suggest\_categorical('batch\_size', [32, 64, 128]),

callbacks=[early\_stop],

verbose=0

)

y\_pred\_val = model.predict(X\_test)

val\_mse = mean\_squared\_error(y\_test, y\_pred\_val)

return val\_mse

You're using **Optuna**, a powerful hyperparameter optimization library, to **automatically search for the best configuration** of a GRU-based neural network for time series forecasting.

Let’s break down and **explain each part** of the code.

## **🔧 Part 1: create\_gru\_model(...) — Build a dynamic GRU model**

This function dynamically builds a GRU model based on trial parameters suggested by Optuna.

def create\_gru\_model(input\_shape, trial):

* input\_shape: Shape of your input data (e.g., (sequence\_length, num\_features)).
* trial: An Optuna trial object, which suggests hyperparameters.

### **🔁 Loop through layers:**

n\_layers = trial.suggest\_int('n\_layers', 1, 3)

* You’re choosing **1 to 3 GRU layers**. Each trial may try a different number of layers.

#### **For each layer:**

units = trial.suggest\_int(f'units\_l{i}', 64, 256, step=32)

dropout = trial.suggest\_float(f'dropout\_l{i}', 0.1, 0.5)

return\_seq = i < n\_layers - 1

* **units**: Number of GRU units (neurons) in that layer — between 64 and 256.
* **dropout**: Dropout rate to avoid overfitting — between 0.1 and 0.5.
* **return\_seq**: Only return sequences if it's **not** the last GRU layer. This is required for stacking multiple GRU layers.

#### **Build the model layer by layer:**

if i == 0:

model.add(GRU(units, return\_sequences=return\_seq, input\_shape=input\_shape))

else:

model.add(GRU(units, return\_sequences=return\_seq))

model.add(Dropout(dropout))

* First layer includes the input\_shape.
* You add a Dropout layer after each GRU to reduce overfitting.

### **🧮 Output layer and optimizer:**

model.add(Dense(1))

optimizer = trial.suggest\_categorical('optimizer', ['adam', 'nadam', 'rmsprop'])

model.compile(optimizer=optimizer, loss='mse')

* The Dense(1) layer outputs **one prediction** (dengue cases).
* Optuna picks from 3 optimizers: adam, nadam, or rmsprop.
* **Loss = Mean Squared Error (MSE)** since this is a regression task.

## **🎯 Part 2: objective(trial) — What Optuna tries to minimize**

This function defines what Optuna will evaluate and try to minimize: the **validation MSE**.

model = create\_gru\_model((X\_train.shape[1], X\_train.shape[2]), trial)

* Create a GRU model using the trial’s parameters.

### **⏹️ Early stopping:**

early\_stop = EarlyStopping(monitor='val\_loss', patience=15, restore\_best\_weights=True)

* Stop training early if validation loss doesn’t improve for 15 epochs.
* Restore the best weights from the best epoch.

### **📈 Train the model:**

model.fit(

X\_train, y\_train,

validation\_split=0.2,

epochs=100,

batch\_size=trial.suggest\_categorical('batch\_size', [32, 64, 128]),

callbacks=[early\_stop],

verbose=0

)

* Model trains for up to 100 epochs.
* Uses a validation split of 20% from the training data.
* Batch size is also selected by Optuna (32, 64, or 128).
* verbose=0 silences the training output.

### **🔍 Predict and evaluate:**

y\_pred\_val = model.predict(X\_test)

val\_mse = mean\_squared\_error(y\_test, y\_pred\_val)

return val\_mse

* After training, the model predicts on **X\_test**.
* Calculates **MSE between predictions and actual y\_test values**.
* This MSE is what **Optuna tries to minimize** across trials.

## **🔁 Usage in Optimization**

To actually run this tuning:

study = optuna.create\_study(direction='minimize')

study.optimize(objective, n\_trials=50)

* Tries **50 different combinations** of hyperparameters.
* Picks the one that gives **lowest MSE on validation set**.

## **✅ Summary**

| **Component** | **Purpose** |
| --- | --- |
| create\_gru\_model | Dynamically creates a GRU model with hyperparameters suggested by Optuna |
| objective | Trains the model, predicts on test set, and returns validation MSE |
| EarlyStopping | Stops training early if validation loss doesn’t improve |
| Optuna | Tries many hyperparameter combinations to minimize prediction error |

Let me know if you'd like to visualize the tuning results or extract the best model!

Run Optuna Tuning

study = optuna.create\_study(direction='minimize')

study.optimize(objective, n\_trials=30)

print("Best trial:", study.best\_trial)

print("Best hyperparameters:", study.best\_trial.params)

You're now running the **hyperparameter optimization** process using Optuna. Let's walk through each line to explain exactly what it does:

## **✅ Code Breakdown**

study = optuna.create\_study(direction='minimize')

### **🎯 create\_study(direction='minimize')**

* This creates a new **optimization session**, called a study.
* The goal is to **minimize** the objective function you defined earlier.
* In your case, the **objective function returns MSE (mean squared error)**, which you want to **minimize** — lower error = better predictions.

study.optimize(objective, n\_trials=30)

### **🔁 study.optimize(...)**

* This runs the **optimization loop**.
* Optuna will:  
  1. Try **30 different combinations** of hyperparameters (n\_trials=30).
  2. For each trial:  
     + Call the objective(trial) function.
     + Build and train a GRU model with the trial’s hyperparameters.
     + Evaluate it on validation/test data using **mean squared error (MSE)**.
     + Store the results.

print("Best trial:", study.best\_trial)

### **🏆 study.best\_trial**

* After all 30 trials, this shows the **best-performing trial**, i.e., the one with the **lowest MSE**.
* It includes:  
  + The trial number (e.g., Trial #17)
  + The MSE score it achieved
  + All hyperparameters used

print("Best hyperparameters:", study.best\_trial.params)

### **⚙️ study.best\_trial.params**

* This prints only the **dictionary of hyperparameters** that led to the best result.
* Example output:

{

'n\_layers': 2,

'units\_l0': 128,

'dropout\_l0': 0.3,

'units\_l1': 96,

'dropout\_l1': 0.2,

'optimizer': 'adam',

'batch\_size': 64

}

These are the values you could **use to retrain the model** later on the full training set for final predictions.

## **🔚 Summary**

| **Line** | **Purpose** |
| --- | --- |
| create\_study(direction='minimize') | Set up an optimization session for minimizing MSE |
| study.optimize(...) | Run 30 trials of model training + evaluation |
| study.best\_trial | Show the best-performing trial (lowest error) |
| study.best\_trial.params | Show the best hyperparameter settings |

Let me know if you'd like to plot optimization history, or retrain the final model using the best hyperparameters!

Train Final Model with Best Params

best\_params = study.best\_trial.params

def train\_best\_gru(params):

model = Sequential()

for i in range(params['n\_layers']):

units = params[f'units\_l{i}']

dropout = params[f'dropout\_l{i}']

return\_seq = i < params['n\_layers'] - 1

if i == 0:

model.add(GRU(units, return\_sequences=return\_seq, input\_shape=(X\_train.shape[1], X\_train.shape[2])))

else:

model.add(GRU(units, return\_sequences=return\_seq))

model.add(Dropout(dropout))

model.add(Dense(1))

model.compile(optimizer=params['optimizer'], loss='mse')

early\_stop = EarlyStopping(monitor='val\_loss', patience=15, restore\_best\_weights=True)

model.fit(X\_train, y\_train,

validation\_split=0.2,

epochs=100,

batch\_size=params['batch\_size'],

callbacks=[early\_stop],

verbose=1)

return model

final\_model = train\_best\_gru(best\_params)

Great — this code takes the **best hyperparameters found by Optuna** and **re-trains a final GRU model** using those parameters. Here's a full explanation of how it works, step by step:

## **🏗️ Step 1: Extract Best Hyperparameters**

best\_params = study.best\_trial.params

* You extract the **best set of parameters** (found during tuning) from your study object.
* This will look like:

{

'n\_layers': 2,

'units\_l0': 128,

'dropout\_l0': 0.3,

'units\_l1': 96,

'dropout\_l1': 0.2,

'optimizer': 'adam',

'batch\_size': 64

}

## **🔁 Step 2: Define and Train the Final Model**

def train\_best\_gru(params):

model = Sequential()

...

This function builds and trains a GRU model using the **best parameters** from Optuna.

### **🔧 Build the Model**

for i in range(params['n\_layers']):

units = params[f'units\_l{i}']

dropout = params[f'dropout\_l{i}']

return\_seq = i < params['n\_layers'] - 1

* Loop over the number of GRU layers (1–3).
* Extract units and dropout for each layer from the params dictionary.
* return\_seq is True **except for the last GRU layer**, which must output a single vector instead of a sequence.

if i == 0:

model.add(GRU(units, return\_sequences=return\_seq, input\_shape=(X\_train.shape[1], X\_train.shape[2])))

else:

model.add(GRU(units, return\_sequences=return\_seq))

model.add(Dropout(dropout))

* The first GRU layer includes input\_shape.
* Each GRU layer is followed by a Dropout layer to reduce overfitting.

### **🧮 Add Final Output Layer**

model.add(Dense(1))

* A Dense(1) layer makes the final prediction: one value per sample (total dengue cases).

### **⚙️ Compile the Model**

model.compile(optimizer=params['optimizer'], loss='mse')

* Use the best optimizer (e.g., 'adam').
* Loss is **mean squared error** for regression.

## **⏹️ Step 3: Train with EarlyStopping**

early\_stop = EarlyStopping(monitor='val\_loss', patience=15, restore\_best\_weights=True)

* Stops training early if validation loss doesn’t improve for 15 epochs.

model.fit(

X\_train, y\_train,

validation\_split=0.2,

epochs=100,

batch\_size=params['batch\_size'],

callbacks=[early\_stop],

verbose=1

)

* Trains the model for up to 100 epochs.
* Uses the **best batch size** found by Optuna (32, 64, or 128).
* Validation split is 20% of training data.
* verbose=1 shows training progress.

## **✅ Step 4: Return the Trained Model**

return model

And then:

final\_model = train\_best\_gru(best\_params)

* You train the final model using the **best architecture + hyperparameters**.
* This final\_model is now ready for:  
  + **Evaluation** on the test set
  + **Prediction** on new unseen data
  + **Saving** for future use

## **🔚 Summary**

| **Line** | **Purpose** |
| --- | --- |
| best\_params = ... | Extract best hyperparameters |
| train\_best\_gru(...) | Build and train a GRU model using best params |
| model.fit(...) | Train the model with early stopping |
| final\_model = ... | Store the trained final model |

Would you like to now evaluate final\_model on the test set and visualize predictions?

Cell 11: Evaluate Performance (MSE, RMSE)

# Predict and inverse transform

y\_pred\_scaled = final\_model.predict(X\_test)

y\_test\_inv = target\_scaler.inverse\_transform(y\_test)

y\_pred\_inv = target\_scaler.inverse\_transform(y\_pred\_scaled)

# Metrics

mse = mean\_squared\_error(y\_test\_inv, y\_pred\_inv)

rmse = np.sqrt(mse)

print(f"Test MSE: {mse:.2f}")

print(f"Test RMSE: {rmse:.2f}")

You're now **evaluating the performance** of your final trained model on the test set, using proper scaling and metrics. Let's go step by step to explain what's happening:

## **✅ Step 1: Predict on Test Data (Scaled)**

y\_pred\_scaled = final\_model.predict(X\_test)

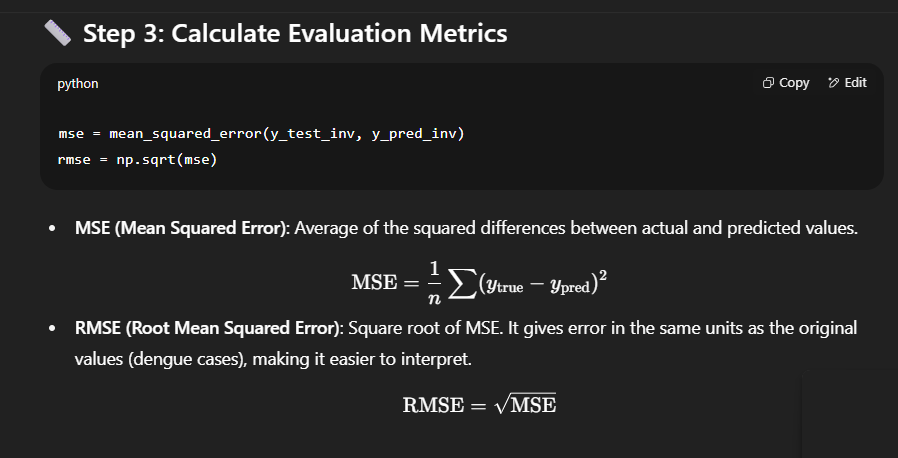
* Your model outputs **predictions in scaled form** (because you trained it on normalized target values using MinMaxScaler).
* y\_pred\_scaled shape is likely (n\_samples, 1).

## **🔄 Step 2: Inverse Transform the Predictions and Targets**

y\_test\_inv = target\_scaler.inverse\_transform(y\_test)

y\_pred\_inv = target\_scaler.inverse\_transform(y\_pred\_scaled)

* Since both y\_test and y\_pred\_scaled are in the **scaled (0–1)** range, you **convert them back** to their original scale (actual case numbers).
* This is important so that the error metrics are in **real-world units** (i.e., number of dengue cases).



## **📢 Step 4: Print the Results**

print(f"Test MSE: {mse:.2f}")

print(f"Test RMSE: {rmse:.2f}")

* Displays the model's prediction error on **unseen test data**.
* Lower values indicate better performance.

## **🧪 Example Output (what you might see)**

Test MSE: 45.27

Test RMSE: 6.72

This means the model's predictions are off by about **6.7 cases** on average (per test sample), which is pretty good if your target values are in a reasonable range (e.g., 10–100+ cases/week).

## **✅ Summary**

| **Step** | **Action** | **Why It Matters** |
| --- | --- | --- |
| Predict | model.predict(X\_test) | Get scaled predictions |
| Inverse transform | Back to original scale | So results are meaningful |
| Metrics | MSE, RMSE | Evaluate performance |
| Print | Show results | Understand model accuracy |

Would you like help with plotting predictions vs actual values for better visualization?