**HƯỚNG DẪN CÀI ĐẶT**

1. Chuẩn bị dữ liệu và trực quan hóa dữ liệu:
   1. Chuẩn hóa dữ liệu:

|  |
| --- |
| #Chuan hoa du lieu      def normalize\_data(self, X\_value, y\_value, scaler = 'Min-Max'):          #chuan hoa min-max          if scaler == 'Min-Max':              X\_scaler = MinMaxScaler()              y\_scaler = MinMaxScaler()          # Chuan hoa zero-mean          elif scaler == 'Zero-Mean':              X\_scaler = StandardScaler()              y\_scaler = StandardScaler()          # du lieu goc          else:              return X\_value.to\_numpy(), y\_value.to\_numpy()          X\_scaler.fit(X\_value)          y\_scaler.fit(y\_value)            dump(X\_scaler, open('./static/X\_scaler.pkl', 'wb'))          dump(y\_scaler, open('./static/y\_scaler.pkl', 'wb'))          X\_scale\_dataset = X\_scaler.fit\_transform(X\_value)          y\_scale\_dataset = y\_scaler.fit\_transform(y\_value)          return X\_scale\_dataset, y\_scale\_dataset |

* Chuẩn hóa dữ liệu với 2 dạng chuẩn hóa được hỗ trợ từ viện của **sklearn** hoặc trả về dữ liệu gốc:
* Chuẩn hóa Min-Max
* Chuẩn hóa Zero-Mean
* **X\_scaler.fit(X\_value)** và **y\_scaler.fit(y\_value)** được sử dụng để tính toán các tham số chuẩn hóa cho dữ liệu đầu vào **X\_value** và **y\_value**. Từ đó, **dump(X\_scaler, open('./static/X\_scaler.pkl', 'wb'))** và **dump(y\_scaler, open('./static/y\_scaler.pkl', 'wb'))** được sử dụng để lưu trữ các tham số đã chuẩn hóa vào các tệp **X\_scaler.pkl** và **y\_scaler.pkl**. Cuối cùng, **X\_scaler.fit\_transform(X\_value)** và **y\_scaler.fit\_transform(y\_value)** được sử dụng để chuẩn hóa dữ liệu đầu vào **X\_value** và **y\_value** và lưu trữ kết quả vào **X\_scale\_dataset** và **y\_scale\_datase**t tương ứng.
  1. Chia tập dữ liệu x-y:

|  |
| --- |
| def get\_X\_y(self, X\_data, y\_data, n\_steps\_in, n\_steps\_out):          X = list()          y = list()          length = len(X\_data)          # chia dữ liệu vào các mẫu dữ liệu có độ dài nhất định cho việc huấn luyện mô hình          for i in range(0, length, 1):              X\_value = X\_data[i: i + n\_steps\_in][:, :]              y\_value = y\_data[i + n\_steps\_in: i + (n\_steps\_in + n\_steps\_out)][:, 0]              if len(X\_value) == n\_steps\_in and len(y\_value) == n\_steps\_out:                  X.append(X\_value)                  y.append(y\_value)          return np.array(X), np.array(y) |

* Hàm lấy dữ liệu đầu vào **X\_data** và **y\_data**, cùng với số ngày thời gian đầu vào **n\_steps\_in** và số ngày thời gian đầu ra **n\_steps\_ou**t. Hàm này trả về hai mảng numpy **X** và **y**, chứa các mẫu dữ liệu được chuẩn bị để huấn luyện mô hình dự đoán chuỗi thời gian.
* **X\_value** và **y\_value** được sử dụng để lấy các giá trị đầu vào và đầu ra cho mỗi mẫu dữ liệu. Sau đó, **X\_value** và **y\_value** được kiểm tra để đảm bảo rằng chúng có độ dài phù hợp với **n\_steps\_in** và **n\_steps\_out**. Nếu độ dài của **X\_value** và **y\_value** phù hợp, chúng được thêm vào các mảng **X** và **y** tương ứng.
  1. Chia tập dữ liệu train/validation/test:

|  |
| --- |
| #Chia tap du lieu train/validation/test      def split\_train\_test(self, data, train\_ratio = 0.7, valid\_ratio = 0.2):          # tính Độ dài của tập train/valid/test  total\_size = len(data)          train\_size = int(total\_size \* train\_ratio)          valid\_size = int(total\_size \* valid\_ratio)          test\_size = total\_size - train\_size - valid\_size          # Chia tap du lieu dua tren do dai cua tung tap          data\_train, data\_valid, data\_test = (          data[:train\_size, :],          data[train\_size:train\_size + valid\_size, :],          data[train\_size + valid\_size: train\_size + valid\_size + test\_size, :])          return data\_train, data\_valid, data\_test |

* Sau khi tính tổng độ dài **total\_size** = len(data), kích thước của tập dữ liệu **train/ validation/ test** được tính bằng tích của tỉ lệ từng tập với tổng độ dài **total\_size.**
* Sau khi tính kích thước của từng tập ta thực hiện chia dữ liệu cho từng tập tương ứng với kích thước của chúng. Kết quả, ta thu được 3 tập dữ liệu tương ứng **data\_train, data\_valid, data\_test**.
  1. Chia giá trị Index (ngày) cho tập train/validation/test:

|  |
| --- |
| def predict\_index(self, dataset, X\_train, X\_valid, n\_steps\_in, n\_steps\_out):          #Gia tri index train          train\_predict\_index = dataset.iloc[n\_steps\_in: X\_train.shape[0] + n\_steps\_in + n\_steps\_out - 1, :].index          #Gia tri index valid          valid\_start\_index = X\_train.shape[0] + n\_steps\_in + n\_steps\_out # Define the starting index for the validation set          valid\_end\_index = valid\_start\_index + X\_valid.shape[0] -1  # Define the ending index for the validation set          valid\_predict\_index = dataset.iloc[valid\_start\_index:valid\_end\_index, :].index            #Gia tri index test          test\_start\_index = valid\_end\_index  # Define the starting index for the test set          test\_predict\_index = dataset.iloc[test\_start\_index:, :].index          return train\_predict\_index, valid\_predict\_index, test\_predict\_index |

* Sau khi thực hiện chia dữ liệu cho 3 tập Train/ Validation/ Test, ta tiếp tục thực hiện chia các giá trị **Index**(ngày) tương ứng cho 3 tập trên. Hàm này trả về giá trị **train\_predict\_index, valid\_predict\_index, test\_predict\_index,** được sử dụng để lấy các chỉ mục dự đoán cho mỗi tập dữ liệu và chúng được tính toán dựa trên kích thước của **X\_train, X\_valid, n\_steps\_in, n\_steps\_out.**
  1. Làm sạch dữ liệu và hoàn thành việc chuẩn bị dữ liệu:

|  |
| --- |
| def CleanData(self, data, n\_steps\_in = 10, n\_steps\_out = 1, feature = 'Price',\                     train\_ratio = 0.7, valid\_ratio = 0.2, scaler = 'Min-Max'):            # loai bo cac ki tu va dau tren cac bien du lieu          data = data.dropna()          column\_names = tuple(data.drop(data.columns[0], axis=1).columns.values)          for column in column\_names:              data[column] = data[column].fillna('0').astype(str).str.replace(',', '')\                  .str.replace('K', 'e3').str.replace('M', 'e6')\                      .str.replace('%', 'e-2').map(lambda x: pd.eval(x) if x != 'nan' else np.nan).astype(float)          # Dinh dang cot dau tien theo dang ngay gio va sap xep du lieu          data[data.columns[0]] = pd.to\_datetime(data[data.columns[0]])          data = data.sort\_values(by=data.columns[0])          data.set\_index(data.columns[0], inplace=True)          # sao luu data truoc khi chia tap du lieu          data\_old = data          X\_value = data[[feature]]          y\_value = data[[feature]]          # Chuan hoa du lieu          X\_scale\_dataset, y\_scale\_dataset = self.normalize\_data(X\_value, y\_value, scaler)          # Chia tap X-y          X, y = self.get\_X\_y(X\_scale\_dataset, y\_scale\_dataset, n\_steps\_in, n\_steps\_out)          # Chia tap train/validation/test          X\_train, X\_valid, X\_test, = self.split\_train\_test(X, train\_ratio, valid\_ratio)          y\_train, y\_valid, y\_test, = self.split\_train\_test(y, train\_ratio, valid\_ratio)          # Chia index(ngay) train/validation/test          index\_train, index\_valid, index\_test, = self.predict\_index(data, X\_train, X\_valid, n\_steps\_in, n\_steps\_out)            return data, data\_old, X\_train, X\_test, X\_valid, y\_train, y\_test, \              y\_valid, index\_train, index\_test, index\_valid |

* Lấy tên các cột có trong file input, trừ cột đầu tiên (cột đầu tiên là cột thời gian của dòng dữ liệu). Tiếp theo, chúng ta bắt đầu biến đổi các cột dữ liệu cho đồng nhất. Ví dụ: Ở đây, các cột có xuất hiện giá trị “K” sẽ tương ứng e^3, chẳng hạn như giá trị 3.56K sẽ tương đương với 3.56 \* e^3.
* Tiếp theo, ta chuẩn hóa cột đầu tiên (cột thời gian của tập dữ liệu) trong data đầu vào theo dạng DateTime và tiến hành sort theo cột ngày.
* Sau đó, ta tiến hành chuẩn hóa dữ liệu, đồng thời chia tập dữ liệu thành các tập train, validation, test và các giá trị index tương ứng với các tập trên.
  1. Trực quan hóa dữ liệu:

|  |
| --- |
| def OneLine(data, selected\_column\_name):          trace = go.Scatter(x=data.data\_old.index ,y=data.data\_old[selected\_column\_name], mode='lines')          layout = go.Layout(              title='Biểu đồ trực quan hóa',              xaxis=dict(title='Ngày'),              yaxis=dict(title=''),              hovermode='closest'          )          fig = go.Figure(data=[trace], layout=layout)          fig.update\_layout()  # Kích thước tùy chỉnh 800x400          return fig |

* Sau khi thực hiện xử lý dữ liệu, cuối cùng ta thực hiện trực quan hóa dữ liệu lên biểu đồ với cột mà người dùng chọn.

1. Xây dựng mô hình và các hàm liên quan:
   1. Xây dựng mô hình đơn:
      1. Mô hình CNN:

|  |
| --- |
| def CNN\_Model(input\_dim=10, output\_dim=1, units = 32, learning\_rate = 0.0001, activation = 'relu'):      model = Sequential()      # Thêm lớp Convolutional 1D đầu tiên      model.add(Conv1D(units, input\_shape=(input\_dim, 1), \                       kernel\_size=3, strides=1, padding='same', activation=activation))      model.add(Conv1D(units, kernel\_size=3, strides=1, padding='same', activation=activation))      model.add(MaxPooling1D(pool\_size=2,strides=2, padding='same'))      # Hoàn thiện mô hình      model.add(Flatten())      model.add(Dense(256, activation=activation))      model.add(Dense(units=output\_dim))      # Thiết lập cấu hình cho mô hình để sẵn sàng cho quá trình huấn luyện.      model.compile(optimizer=Adam(learning\_rate=learning\_rate), loss='mse')      return model |

* Khởi tạo một mô hình tuần tự **Sequential()**. Sau đó, ta tạo lớp đầu tiên của mô hình với lớp 1D Convolution (**Conv1D**) với số filters = số lượng filter (số filter sẽ được set lại sau quá trình Optimize), **input\_shape = (input\_dim, 1)** với **input\_dim** là số điểm dữ liệu đầu vào (**n\_step\_in**) do người dùng chọn, kích thước **kernel** là 3, với bước nhảy **strides** =1. Và **padding** = ‘same’ sẽ tự động thêm các giá trị đệm vào đầu vào sao cho đầu ra có kích thước bằng với đầu vào. Và cuối cùng là hàm kích hoạt sẽ được thêm người dùng chọn (**ReLU, LeakyReLU, tanh**).
* Ở lớp tiếp theo, ta thêm một lớp **MaxPooling1D**, với kích thước là 2 và bước nhảy = 2, lớp này giúp ta có thể gộp các phần tử và loại bỏ bớt thông tin không cần thiết giúp quá trình học được tốt hơn. Sau hàm này, dữ liệu sau các lớp tích chập sẽ được giảm đi một nửa.
* Tiếp theo, ta xây dựng một mô hình mạng nơ-ron với các lớp **Flatten** và **Dense**. Lớp **Flatten** được sử dụng để chuyển đổi đầu vào thành một vector 1 chiều. Lớp **Dense** đầu tiên được sử dụng để thêm một lớp **Fully Connection** với 32 đơn vị đầu ra. Cuối cùng, lớp **Dense** được sử dụng để thêm một **Fully Connection** với output\_dim (số output đầu ra do người dùng chỉ định) đơn vị đầu ra và hàm kích hoạt được chỉ định.
* Cuối cùng là ta tiến hành **Compile** để cấu hình quá trình học của mô hình, sử dụng hàm tối ưu **Adam** với số **learning\_rate** được chỉ định và hàm mất mát mse.
  + 1. Mô hình LSTM:

|  |
| --- |
| def LSTM\_Model(input\_dim=10, output\_dim=1, units =32, learning\_rate=0.0001, activation = 'relu') -> tf.keras.models.Model:      model = Sequential()      model.add(LSTM(units=units, return\_sequences=True, input\_shape=(input\_dim, 1), activation=activation))      model.add(LSTM(units=units, activation=activation))      model.add(Dense(32, activation=activation))      model.add(Dense(units=output\_dim))      model.compile(optimizer=Adam(learning\_rate=learning\_rate), loss='mse')      return model |

* Tương tự như mô hình CNN, ta thực hiện xây dựng mô hình LSTM.
* Mô hình được xây dựng theo thứ tự sau:
  + **model = Sequential():** Khởi tạo một mô hình tuần tự.
  + **model.add(LSTM(units=units, return\_sequences=True, input\_shape=(input\_dim, 1), activation=activation)):** Thêm một layer **LSTM** vào mô hình với **units** là số lượng **units** trong layer này, **return\_sequences=True** cho phép layer trả về output cho mỗi sequence input, **input\_shape=(input\_dim, 1)** định nghĩa kích thước đầu vào của layer là **(số ngày đầu vào, 1)**, và **activation=activation** là hàm kích hoạt được sử dụng.
  + **model.add(LSTM(units=units, activation=activation)):** Thêm một layer LSTM không trả về sequences. Điều này chỉ có ý nghĩa nếu layer trước đó đã trả về sequences. Các tham số khác giống như layer trước.
  + **model.add(Dense(32, activation=activation)):** Thêm một layer Dense với 32 units và hàm kích hoạt activation.
  + **model.add(Dense(units=output\_dim)):** Thêm một layer Dense với số units bằng output\_dim, không sử dụng hàm kích hoạt.
  + **model.compile(optimizer=Adam(learning\_rate=learning\_rate), loss='mse'):** Compile mô hình với thuật toán tối ưu hóa Adam và hàm mất mát là 'mse' (mean squared error).
    1. Mô hình RNN:

|  |
| --- |
| def RNN\_Model(input\_dim=10, output\_dim=1, units =32, learning\_rate=0.0001, activation = 'relu') -> tf.keras.models.Model:      model = Sequential()      model.add(SimpleRNN(units=units, return\_sequences=True, \                          input\_shape=(input\_dim, 1), activation=activation))      model.add(SimpleRNN(units=units, activation=activation))      model.add(Dense(32, activation=activation))      model.add(Dense(units=output\_dim))      model.compile(optimizer=Adam(learning\_rate=learning\_rate), loss='mse')      return model |

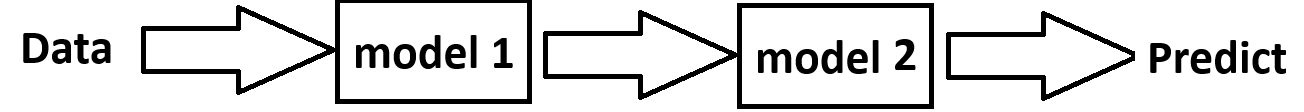
* Mô hình được xây dựng theo thứ tự sau:
  + **model = Sequential()**: Khởi tạo một mô hình tuần tự.
  + **model.add(SimpleRNN(units=units, return\_sequences = True , input\_shape = (input\_dim, 1), activation=activation))**: Thêm một layer **SimpleRNN** vào mô hình với **units** là số lượng units trong layer này, **return\_sequences=True** cho phép layer trả về output cho mỗi sequence input, **input\_shape=(input\_dim, 1)** định nghĩa kích thước đầu vào của layer là (input\_dim, 1), và **activation=activation** là hàm kích hoạt được sử dụng.
  + **model.add(SimpleRNN(units=units, activation=activation))**: Thêm một layer SimpleRNN không trả về sequences. Điều này chỉ có ý nghĩa nếu layer trước đó đã trả về sequences. Các tham số khác giống như layer trước.
  + **model.add(Dense(32, activation=activation))**: Thêm một layer Dense với **32 units** và hàm kích hoạt activation.
  + **model.add(Dense(units=output\_dim))**: Thêm một layer Dense với số units bằng output\_dim, không sử dụng hàm kích hoạt.
  + **model.compile(optimizer=Adam(learning\_rate=learning\_rate), loss='mse')**: Compile mô hình với thuật toán tối ưu hóa Adam và hàm mất mát là 'mse' (mean squared error).
    1. Mô hình GRU:

|  |
| --- |
| def GRU\_Model(input\_dim=10, output\_dim=1, units =32, learning\_rate=0.0001, activation = 'relu') -> tf.keras.models.Model:      model = Sequential()      model.add(GRU(units=units, return\_sequences=True, input\_shape=(input\_dim, 1), activation=activation))      model.add(GRU(units=units, activation=activation))      model.add(Dense(32, activation=activation))      model.add(Dense(units=output\_dim))      model.compile(optimizer=Adam(learning\_rate=learning\_rate), loss='mse')      return model |

* Tương tự mô hình LSTM và RNN.
  + 1. Huấn luyện mô hình:

|  |
| --- |
| model\_training = eda.train\_model (m1 , epochs = best\_params['epochs'] , batch\_size=best\_params['batch\_size'])  def train\_model(self, model , epochs, batch\_size):          model.fit(self.X\_train, self.y\_train, epochs=epochs, \                    batch\_size=batch\_size, validation\_data=(self.X\_valid, self.y\_valid), verbose=2, shuffle=False)          return model |

* Thực hiện huấn luyện mô hình với tập dữ liệu train và validation. Trong đó batch\_size và epochs là các siêu tham số thu được sau quá trình Optimizer.
  1. Xây dựng mô hình tuần tự:



* Với ý tưởng lai ghép các mô hình đơn và xây dựng các mô hình tuần tự. Trong đó, các lớp trong mô hình là các mô hình khác nhau. Việc kết hợp các loại mạng nơ-ron khác nhau trong một mô hình học sâu giúp tăng cường khả năng xử lý và hiểu dữ liệu phức tạp hơn.
* Việc xây dựng mô hình như trên, 2 mô hình có thể hỗ trợ nhau một cách đồng thời để xử lý các thông tin hạn chế của mô hình còn lại.
  + 1. Mô hình CNN – LSTM:

|  |
| --- |
| def CNN\_LSTM\_Model(input\_dim=10, output\_dim=1, units = 32, learning\_rate = 0.0001, activation = 'relu') -> tf.keras.models.Model:      model = Sequential()      #CNN      model.add(Conv1D(units, input\_shape=(input\_dim, 1), \                       kernel\_size=3, strides=1, padding='same', activation=activation))      model.add(Conv1D(units, kernel\_size=3, strides=1, padding='same', activation=activation))      model.add(MaxPooling1D(pool\_size=2,strides=2, padding='same'))      model.add(Flatten())      #tạo ra một tensor mới có hình dạng (None, out, units/filter)      model.add(RepeatVector(output\_dim))      #LSTM      model.add(LSTM(units=units, return\_sequences=True, activation=activation))      model.add(LSTM(units=units, activation=activation))      # fully connection      model.add(Dense(32, activation=activation))      model.add(Dense(units=output\_dim))      model.compile(optimizer=Adam(learning\_rate=learning\_rate), loss='mse')      return model |

* Trong mô hình trên, ta khởi tạo một mô hình tuần tự **Sequential()**.
* Sau đó ta thực hiện xây dựng mô hình CNN đơn giản với lớp đầu tiên là lớp tích chập 1D (**Conv1D**) với số lớp filter = số units (sẽ được chọn lại sau quá trình tối ưu hóa). Với **input\_shape=(input\_dim, 1),** với **input\_dim** là số điểm dữ liệu đầu vào (**n\_step\_in**) do người dùng chọn, kích thước **kernel** = 3, với bước nhảy **strides** =1. Và **padding** = ‘same’ sẽ tự động thêm các giá trị đệm vào đầu vào sao cho đầu ra có kích thước bằng với đầu vào. Và cuối cùng là hàm kích hoạt sẽ được thêm người dùng chọn (**ReLU, LeakyReLU, tanh**). Tương tự, ta xây dụng lớp tích chập thứ 2 với các thông số tương tự.
* Ở lớp tiếp theo, ta thêm một lớp **MaxPooling1D**, với kích thước là 2 và bước nhảy = 2, lớp này giúp ta có thể gộp các phần tử và loại bỏ bớt thông tin không cần thiết giúp quá trình học được tốt hơn. Sau hàm này, dữ liệu sau các lớp tích chập sẽ được giảm đi một nửa. Cuối cùng, ta thêm một lớp **Flatten()**, để biến đổi dữ liệu thành dạng vector 1 chiều.
* Sau khi xây dựng lớp đầu tiên là CNN, và kết quả đầu ra ở dạng vector 1 chiều. Ta thêm một lớp **RepeatVector()**, để biến đổi vector đầu ra ở mô hình đầu tiên phù hợp với đầu vào của mô hình thứ 2.
* Sau khi biến đổi dạng của mô hình thứ nhất, ta tiếp tục xây dựng mô hình thứ 2 với lớp đầu tiên là **LSTM(units=units, return\_sequences=True, activation = activation))**, với số lượng units = số units sau khi tối ưu hóa,  **return\_sequences = True** cho phép layer trả về output cho mỗi sequence input **và activation=activation** là hàm kích hoạt được sử dụng.
* **model.add(LSTM(units=units, activation=activation))**: Thêm một layer LSTM không trả về sequences. Điều này chỉ có ý nghĩa nếu layer trước đó đã trả về sequences. Các tham số khác giống như layer trước.
* **model.add(Dense(32, activation=activation))**: Thêm một layer Dense với 32 units và hàm kích hoạt activation.
* **model.add(Dense(units=output\_dim))**: Thêm một layer Dense với số units bằng output\_dim, không sử dụng hàm kích hoạt.
* **model.compile(optimizer=Adam(learning\_rate=learning\_rate), loss='mse')**: Compile mô hình với thuật toán tối ưu hóa Adam và hàm mất mát là 'mse' (mean squared error).
  + 1. Mô hình LSTM – CNN:

|  |
| --- |
| def LSTM\_CNN\_Model(input\_dim=10, output\_dim=1, units = 32, learning\_rate = 0.0001, activation = 'relu') -> tf.keras.models.Model:      model = Sequential()      #LSTM      model.add(LSTM(units=units, return\_sequences=True, input\_shape=(input\_dim, 1), activation=activation))      model.add(LSTM(units=units, activation=activation))        #tạo ra một tensor mới có hình dạng (None, out, units/filter)      model.add(RepeatVector(output\_dim))      #CNN      model.add(Conv1D(units, kernel\_size=3, strides=1, padding='same', activation=activation))      model.add(MaxPooling1D(pool\_size=2,strides=2, padding='same'))      model.add(Flatten())      # fully connection      model.add(Dense(32, activation=activation))      model.add(Dense(units=output\_dim))      model.compile(optimizer=Adam(learning\_rate=learning\_rate), loss='mse')      return model |

* Ngược lại với mô hình CNN - LSTM, mô hình LSTM – CNN xây dựng với mô hình đầu tiên là mô hình LSTM.
* Mô hình được xây dựng như sau:
  + **model = Sequential()**: Khai báo mô hình tuần tự/
    - **model.add(LSTM(units=units, return\_sequences=True, input\_shape = (input\_dim, 1) , activation=activation))**: Thêm một layer **LSTM** vào mô hình với **units** là số lượng **units** trong layer này, **return\_sequences=True** cho phép layer trả về output cho mỗi sequence input, **input\_shape=(input\_dim, 1)** định nghĩa kích thước đầu vào của layer là **(số ngày đầu vào, 1)**, và **activation=activation** là hàm kích hoạt được sử dụng.
    - **model.add(LSTM(units=units, activation=activation)):** Thêm một layer LSTM không trả về sequences. Điều này chỉ có ý nghĩa nếu layer trước đó đã trả về sequences. Các tham số khác giống như layer trước.
    - **model.add(RepeatVector(output\_dim))**: lớp này biến đổi dạng tensor phù hợp dữ liệu đầu vào ở mô hình 2 (None, out, units/filter).
    - **model.add(Conv1D(units, kernel\_size=3, strides=1, padding='same', activation=activation))**: Xây dựng lớp đầu tiên của mô hình thứ 2. Mô hình CNN, với lớp tích chập 1D (**Conv1D**) với số lớp filter= units, kích thước kernel = 3, bước nhảy strides = 1. Và **padding** = ‘same’ sẽ tự động thêm các giá trị đệm vào đầu vào sao cho đầu ra có kích thước bằng với đầu vào. Và cuối cùng là hàm kích hoạt sẽ được thêm người dùng chọn (**ReLU, LeakyReLU, tanh**).
    - **model.add(MaxPooling1D(pool\_size=2,strides=2, padding='same'))**: Thêm một hàm gộp với kích thước = 2 và bước nhảy = 2.
    - **model.add(Flatten())**: cuối cùng ta biến đổi dữ liệu đầu ra thành dạng vector 1 chiều.
    - **model.add(Dense(32, activation=activation)):** Thêm một layer Dense với 32 units và hàm kích hoạt activation.
    - **model.add(Dense(units=output\_dim)):** Thêm một layer Dense với số units bằng output\_dim, không sử dụng hàm kích hoạt.
    - **model.compile(optimizer=Adam(learning\_rate=learning\_rate), loss='mse'):** Compile mô hình với thuật toán tối ưu hóa Adam và hàm mất mát là 'mse' (mean squared error).
    1. Mô hình CNN – RNN:
* Với ý tưởng xây dựng mô hình CNN – LSTM và LSTM – CNN, ta xây dựng các mô hình tuần tự còn lại tương tự.

|  |
| --- |
| def CNN\_RNN\_Model(input\_dim=10, output\_dim=1, units = 32, learning\_rate = 0.0001, activation = 'relu') -> tf.keras.models.Model:      model = Sequential()      #CNN      model.add(Conv1D(units, input\_shape=(input\_dim, 1), \                       kernel\_size=3, strides=1, padding='same', activation=activation))      model.add(Conv1D(units, kernel\_size=3, strides=1, padding='same', activation=activation))      model.add(MaxPooling1D(pool\_size=2,strides=2, padding='same'))      model.add(Flatten())      #tạo ra một tensor mới có hình dạng (None, out, units/filter)      model.add(RepeatVector(output\_dim))      #RNN      model.add(SimpleRNN(units=units, return\_sequences=True, activation=activation))      model.add(SimpleRNN(units=units, activation=activation))      # fully connection      model.add(Dense(32, activation=activation))      model.add(Dense(units=output\_dim))      model.compile(optimizer=Adam(learning\_rate=learning\_rate), loss='mse')      return model |

* + 1. Mô hình RNN – CNN:

|  |
| --- |
| def RNN\_CNN\_Model(input\_dim=10, output\_dim=1, units = 32, learning\_rate = 0.0001, activation = 'relu') -> tf.keras.models.Model:      model = Sequential()      #RNN      model.add(SimpleRNN(units=units, return\_sequences=True,\                      input\_shape=(input\_dim, 1), activation=activation))      model.add(SimpleRNN(units=units, activation=activation))      #tạo ra một tensor mới có hình dạng (None, out, units/filter)      model.add(RepeatVector(output\_dim))      #CNN      model.add(Conv1D(units, kernel\_size=3, strides=1, padding='same', activation=activation))      model.add(MaxPooling1D(pool\_size=2,strides=2, padding='same'))      model.add(Flatten())      # fully connection      model.add(Dense(32, activation=activation))      model.add(Dense(units=output\_dim))      model.compile(optimizer=Adam(learning\_rate=learning\_rate), loss='mse')      return model |

* + 1. Mô hình CNN – GRU:

|  |
| --- |
| def CNN\_GRU\_Model(input\_dim=10, output\_dim=1, units = 32, learning\_rate = 0.0001, activation = 'relu') -> tf.keras.models.Model:      model = Sequential()      #CNN      model.add(Conv1D(units, input\_shape=(input\_dim, 1), \                       kernel\_size=3, strides=1, padding='same', activation=activation))      model.add(Conv1D(units, kernel\_size=3, strides=1, padding='same', activation=activation))      model.add(MaxPooling1D(pool\_size=2,strides=2, padding='same'))      model.add(Flatten())      #tạo ra một tensor mới có hình dạng (None, out, units/filter)      model.add(RepeatVector(output\_dim))      #GRU      model.add(GRU(units=units, return\_sequences=True, activation=activation))      model.add(GRU(units=units, activation=activation))      # fully connection      model.add(Dense(32, activation=activation))      model.add(Dense(units=output\_dim))      model.compile(optimizer=Adam(learning\_rate=learning\_rate), loss='mse')      return model |

* + 1. Mô hình GRU – CNN:

|  |
| --- |
| def GRU\_CNN\_Model(input\_dim=10, output\_dim=1, units = 32, learning\_rate = 0.0001, activation = 'relu') -> tf.keras.models.Model:      model = Sequential()      #GRU      model.add(GRU(units=units, return\_sequences=True,\                      input\_shape=(input\_dim, 1), activation=activation))      model.add(GRU(units=units, activation=activation))      #tạo ra một tensor mới có hình dạng (None, out, units/filter)      model.add(RepeatVector(output\_dim))      #CNN      model.add(Conv1D(units, kernel\_size=3, strides=1, padding='same', activation=activation))      model.add(MaxPooling1D(pool\_size=2,strides=2, padding='same'))      model.add(Flatten())      # fully connection      model.add(Dense(32, activation=activation))      model.add(Dense(units=output\_dim))      model.compile(optimizer=Adam(learning\_rate=learning\_rate), loss='mse')      return model |

* + 1. Mô hình LSTM – RNN:

|  |
| --- |
| def LSTM\_RNN\_Model(input\_dim=10, output\_dim=1, units = 32, learning\_rate = 0.0001, activation = 'relu') -> tf.keras.models.Model:      model = Sequential()      #LSTM      model.add(LSTM(units=units, return\_sequences=True, input\_shape=(input\_dim, 1), activation=activation))      model.add(LSTM(units=units, activation=activation))        #tạo ra một tensor mới có hình dạng (None, out, units/filter)      model.add(RepeatVector(output\_dim))      #RNN      model.add(SimpleRNN(units=units, return\_sequences=True, activation=activation))      model.add(SimpleRNN(units=units, activation=activation))      # fully connection      model.add(Dense(32, activation=activation))      model.add(Dense(units=output\_dim))      model.compile(optimizer=Adam(learning\_rate=learning\_rate), loss='mse')      return model |

* + 1. Mô hình RNN – LSTM:

|  |
| --- |
| def RNN\_LSTM\_Model(input\_dim=10, output\_dim=1, units = 32, learning\_rate = 0.0001, activation = 'relu') -> tf.keras.models.Model:      model = Sequential()      #RNN      model.add(SimpleRNN(units=units, return\_sequences=True, input\_shape=(input\_dim, 1), activation=activation))      model.add(SimpleRNN(units=units, activation=activation))        #tạo ra một tensor mới có hình dạng (None, out, units/filter)      model.add(RepeatVector(output\_dim))      #LSTM      model.add(LSTM(units=units, return\_sequences=True, activation=activation))      model.add(LSTM(units=units, activation=activation))      # fully connection      model.add(Dense(32, activation=activation))      model.add(Dense(units=output\_dim))      model.compile(optimizer=Adam(learning\_rate=learning\_rate), loss='mse')      return model |

* + 1. Mô hình LSTM – GRU:

|  |
| --- |
| def LSTM\_GRU\_Model(input\_dim=10, output\_dim=1, units = 32, learning\_rate = 0.0001, activation = 'relu') -> tf.keras.models.Model:      model = Sequential()      #LSTM      model.add(LSTM(units=units, return\_sequences=True, input\_shape=(input\_dim, 1), activation=activation))      model.add(LSTM(units=units, activation=activation))        #tạo ra một tensor mới có hình dạng (None, out, units/filter)      model.add(RepeatVector(output\_dim))      #GRU      model.add(GRU(units=units, return\_sequences=True, activation=activation))      model.add(GRU(units=units, activation=activation))      # fully connection      model.add(Dense(32, activation=activation))      model.add(Dense(units=output\_dim))      model.compile(optimizer=Adam(learning\_rate=learning\_rate), loss='mse')      return model |

* + 1. Mô hình GRU – LSTM:

|  |
| --- |
| def GRU\_LSTM\_Model(input\_dim=10, output\_dim=1, units = 32, learning\_rate = 0.0001, activation = 'relu') -> tf.keras.models.Model:      model = Sequential()      #GRU      model.add(GRU(units=units, return\_sequences=True, input\_shape=(input\_dim, 1), activation=activation))      model.add(GRU(units=units, activation=activation))        #tạo ra một tensor mới có hình dạng (None, out, units/filter)      model.add(RepeatVector(output\_dim))      #LSTM      model.add(LSTM(units=units, return\_sequences=True, activation=activation))      model.add(LSTM(units=units, activation=activation))      # fully connection      model.add(Dense(32, activation=activation))      model.add(Dense(units=output\_dim))      model.compile(optimizer=Adam(learning\_rate=learning\_rate), loss='mse')      return model |

* + 1. Mô hình RNN – GRU:

|  |
| --- |
| def RNN\_GRU\_Model(input\_dim=10, output\_dim=1, units = 32, learning\_rate = 0.0001, activation = 'relu') -> tf.keras.models.Model:      model = Sequential()      #RNN      model.add(SimpleRNN(units=units, return\_sequences=True, input\_shape=(input\_dim, 1), activation=activation))      model.add(SimpleRNN(units=units, activation=activation))        #tạo ra một tensor mới có hình dạng (None, out, units/filter)      model.add(RepeatVector(output\_dim))      #GRU      model.add(GRU(units=units, return\_sequences=True, activation=activation))      model.add(GRU(units=units, activation=activation))      # fully connection      model.add(Dense(32, activation=activation))      model.add(Dense(units=output\_dim))      model.compile(optimizer=Adam(learning\_rate=learning\_rate), loss='mse')      return model |

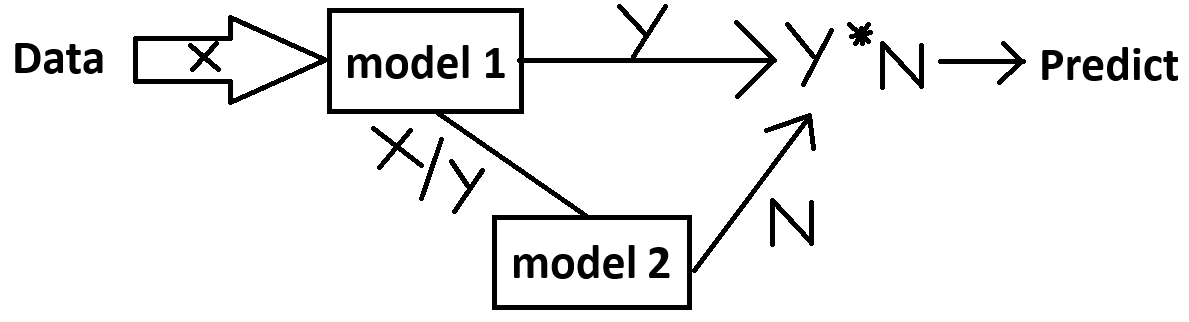
* + 1. Mô hình GRU – RNN:

|  |
| --- |
| def GRU\_RNN\_Model(input\_dim=10, output\_dim=1, units = 32, learning\_rate = 0.0001, activation = 'relu') -> tf.keras.models.Model:      model = Sequential()      #GRU      model.add(GRU(units=units, return\_sequences=True, input\_shape=(input\_dim, 1), activation=activation))      model.add(GRU(units=units, activation=activation))        #tạo ra một tensor mới có hình dạng (None, out, units/filter)      model.add(RepeatVector(output\_dim))      #RNN      model.add(SimpleRNN(units=units, return\_sequences=True, activation=activation))      model.add(SimpleRNN(units=units, activation=activation))      # fully connection      model.add(Dense(32, activation=activation))      model.add(Dense(units=output\_dim))      model.compile(optimizer=Adam(learning\_rate=learning\_rate), loss='mse')      return model |

* + 1. Huấn luyện mô hình:

|  |
| --- |
| model\_training = eda.train\_model (m1 , epochs = best\_params['epochs'] , batch\_size=best\_params['batch\_size'])  def train\_model(self, model , epochs, batch\_size):          model.fit(self.X\_train, self.y\_train, epochs=epochs, \                    batch\_size=batch\_size, validation\_data=(self.X\_valid, self.y\_valid), verbose=2, shuffle=False)          return model |

* Thực hiện huấn luyện mô hình với tập dữ liệu train và validation. Trong đó batch\_size và epochs là các siêu tham số thu được sau quá trình Optimizer.
  1. Xây dựng mô hình tuần tự nhân:



* Sau khi xây dựng mô hình tuần tự, các mô hình lần lượt xử lý dữ liệu và đưa ra dự đoán. Ta tiếp tục xây dựng mô hình tuần tự nhân, với ý tưởng ở sơ đồ trên, khi dữ liệu đầu vào X qua mô hình 1 và đưa ra dự đoán Y, ta thực hiện tính lỗi X/Y để làm dữ liệu đầu vào mô hình 2 để đưa ra lỗi dự đoán N. Sau đó ta thực hiện nhân Y và N để đưa ra dự đoán cuối cùng.
* Ý tưởng trên, ta có thể giảm thiểu được lỗi của mô hình. Thay vì mô hình 2 tiếp tục xử lý thông tin từ mô hình 1, thì ở mô hình tuần tự nhân, mô hình 2 tính lỗi dự đoán của mô hình 1 và khi nhân lại với kết quả dự đoán ở mô hình 1, kết quả dự đoán sẽ gần hơn với kết quả thực tế.
  + 1. Lựa chọn mô hình:

|  |
| --- |
| match mod:     case 'CNN':  m1 = CNN\_Model(input\_dim=input\_dim, output\_dim=output\_dim, units = best\_params['units'], learning\_rate = best\_params['learning\_rate'], activation= activation)  case 'RNN':  m1 = RNN\_Model(input\_dim=input\_dim, output\_dim=output\_dim, units = best\_params['units'], learning\_rate = best\_params['learning\_rate'], activation= activation)  case 'GRU':  m1 = GRU\_Model(input\_dim=input\_dim, output\_dim=output\_dim, units = best\_params['units'], learning\_rate = best\_params['learning\_rate'], activation= activation)  case 'LSTM':  m1 = LSTM\_Model(input\_dim=input\_dim, output\_dim=output\_dim, units = best\_params['units'], learning\_rate = best\_params['learning\_rate'], activation= activation)  match mod1:  case 'CNN':  m2 = CNN\_Model(input\_dim=input\_dim, output\_dim=output\_dim, units = best\_params['units'], learning\_rate = best\_params['learning\_rate'], activation= activation)  case 'RNN':  m2 = RNN\_Model(input\_dim=input\_dim, output\_dim=output\_dim, units = best\_params['units'], learning\_rate = best\_params['learning\_rate'], activation= activation)  case 'GRU':  m2 = GRU\_Model(input\_dim=input\_dim, output\_dim=output\_dim, units = best\_params['units'], learning\_rate = best\_params['learning\_rate'], activation= activation)  case 'LSTM':  m2 = LSTM\_Model(input\_dim=input\_dim, output\_dim=output\_dim, units = best\_params['units'], learning\_rate = best\_params['learning\_rate'], activation= activation) |

* Khi lựa chọn mô hình sử dụng là mô hình tuần tự nhân, hệ thống yêu cầu người dùng lựa chọn mô hình sử dụng cho cả mô hình 1 và mô hình 2.
* Các tham số đầu vào **input\_dim, output\_dim** và **activation** được người dùng chọn khi Optimze/ Train. Các tham số **units**, **learning\_rate** được điều chỉnh sau quá trình **Optimizer**.
  + 1. Huấn luyện mô hình:

|  |
| --- |
| model\_training = eda.train\_model(m1,epochs=best\_params['epochs'], batch\_size=best\_params['batch\_size'])  predict\_train, actual\_train, index\_train, predict\_scale\_1\_train, actua\_scale\_train = eda.TestingModel(model\_training)  model\_training\_2 = eda.train\_model\_seq(m2, (actual\_train/predict\_train),epochs=best\_params['epochs'], batch\_size=best\_params['batch\_size'])  def train\_model(self, model , epochs, batch\_size):          model.fit(self.X\_train, self.y\_train, epochs=epochs, \  batch\_size=batch\_size, validation\_data=(self.X\_valid, self.y\_valid), verbose=2, shuffle=False)          return model    def train\_model\_seq(self, model, predict\_1, epochs, batch\_size):          model.fit(predict\_1, predict\_1, epochs=epochs, \                    batch\_size=batch\_size, verbose=2, shuffle=False)          return model |

* Tương tự mô hình đơn và mô hình tuần tự. Tuy nhiên, dữ liệu trong quá trình huấn luyện mô hình 2 sẽ được thay bằng **actual\_train/ predict\_train.**
  1. Xây dựng mô hình tuần tự cộng:

**A diagram of a model

Description automatically generated**

* Với dữ liệu thực tế X, tuần tự cộng được xây dựng với mô hình 1 đưa ra 1 dự đoán Y, đầu vào của mô hình 2 là lỗi dự báo (X-Y) và đưa ra kết quả dự đoán 2 là N. Kết quả dự đoán cuối cùng được tính bằng Y + N.
  + 1. Lựa chọn mô hình:

|  |
| --- |
| match mod:     case 'CNN':  m1 = CNN\_Model(input\_dim=input\_dim, output\_dim=output\_dim, units = best\_params['units'], learning\_rate = best\_params['learning\_rate'], activation= activation)  case 'RNN':  m1 = RNN\_Model(input\_dim=input\_dim, output\_dim=output\_dim, units = best\_params['units'], learning\_rate = best\_params['learning\_rate'], activation= activation)  case 'GRU':  m1 = GRU\_Model(input\_dim=input\_dim, output\_dim=output\_dim, units = best\_params['units'], learning\_rate = best\_params['learning\_rate'], activation= activation)  case 'LSTM':  m1 = LSTM\_Model(input\_dim=input\_dim, output\_dim=output\_dim, units = best\_params['units'], learning\_rate = best\_params['learning\_rate'], activation= activation)  match mod1:  case 'CNN':  m2 = CNN\_Model(input\_dim=input\_dim, output\_dim=output\_dim, units = best\_params['units'], learning\_rate = best\_params['learning\_rate'], activation= activation)  case 'RNN':  m2 = RNN\_Model(input\_dim=input\_dim, output\_dim=output\_dim, units = best\_params['units'], learning\_rate = best\_params['learning\_rate'], activation= activation)  case 'GRU':  m2 = GRU\_Model(input\_dim=input\_dim, output\_dim=output\_dim, units = best\_params['units'], learning\_rate = best\_params['learning\_rate'], activation= activation)  case 'LSTM':  m2 = LSTM\_Model(input\_dim=input\_dim, output\_dim=output\_dim, units = best\_params['units'], learning\_rate = best\_params['learning\_rate'], activation= activation) |

* Tương tự tuần tự nhân.
  + 1. Huấn luyện mô hình

|  |
| --- |
| model\_training = eda.train\_model(m1,epochs=best\_params['epochs'], batch\_size=best\_params['batch\_size'])  predict\_train, actual\_train, index\_train, predict\_scale\_1\_train, actua\_scale\_train = eda.TestingModel(model\_training)  model\_training\_2 = eda.train\_model\_seq(m2, (actual\_train-predict\_train),epochs=best\_params['epochs'], batch\_size=best\_params['batch\_size']) |

* Khác với tuần tự nhân, mô hình 2 được huấn luyện với dữ liệu đầu vào là **actual\_train - predict\_train.**
* Các tham số đầu vào **input\_dim, output\_dim** và **activation** được người dùng chọn khi Optimze/ Train. Các tham số **units**, **learning\_rate** được điều chỉnh sau quá trình **Optimizer**.
  1. Xây dựng mô hình song song:

A diagram of a model

Description automatically generated

* Dữ liệu thực tế X là đầu vào của mô hình 1 và mô hình 2, và cho ta 2 dự đoán là A và B.
* Kết quả được tính bằng công thức:

=

Predict = \* A + (1 - ) \* B

* + 1. Lựa chọn mô hình

|  |
| --- |
| checkpoint = torch.load("./model/"+ mod +".pth")  model\_training = checkpoint["model"]  unit\_training = checkpoint["units"]  epochs\_traning = checkpoint['epochs']  batch\_size\_training = checkpoint['batch\_size']  LR\_training = checkpoint['learning\_rate']  time\_traing = checkpoint['time\_train']  checkpoint\_2 = torch.load("./model/"+ mod1 +".pth")  model\_training\_2 = checkpoint\_2["model"]  unit\_training\_2 = checkpoint\_2["units"]  epochs\_traning\_2 = checkpoint\_2['epochs']  batch\_size\_training\_2 = checkpoint\_2['batch\_size']  LR\_training\_2 = checkpoint\_2['learning\_rate']  time\_traing\_2 = checkpoint\_2['time\_train'] |

* Tương tự tuần tự nhân và tuần tự cộng. Sau khi lựa chọn mô hình sử dụng, thực hiện load các mô hình đơn từ các mục .pth tương ứng.
  + 1. Xây dựng hàm tính toán

|  |
| --- |
| def calculate\_omega(y, y1, y2):      numerator = np.sum((y2 - y1) \* (y - y1))      denominator = np.sum((y2 - y1) \*\* 2)      omega = numerator / denominator if denominator != 0 else 0      # Ensure w\_parallel is between 0 and 1      omega = max(0, min(omega, 1))      return omega  predict\_1, actual\_1, index\_1, predict\_scale\_1, actua\_scale\_1 = eda.TestingModel(model\_train)  predict\_2, actual\_2, index\_2, predict\_scale\_2, actua\_scale\_2 = eda.TestingModel(model\_train\_2)  actual, index, actua\_scale = actual\_1, index\_1, actua\_scale\_1  omega = calculate\_omega(actual, predict\_1, predict\_2)  predict = omega \* predict\_1 + (1 - omega) \* predict\_2  predict\_scale = omega \* predict\_scale\_1 + (1 - omega) \* predict\_scale\_2 |

* Thực hiện tính toán Omega và tính kết quả cuối cùng.
  1. Xây dựng các hàm tối ưu hóa:
* Bài toán được xây dựng với 3 phương pháp tối ưu hóa các tham số là RandomSearchCV, GridSearchCV, BayesSearchCV.
* RandomSearchCV: Khai báo các tham số cần truyền cho quá trình tối ưu. Sau đó thực hiện quá trình chọn ngẫu nhiên các tham số để thực hiện tối ưu và chọn ra bộ tham số có mức độ **neg\_mean\_squared\_error** cao nhất.

|  |
| --- |
| param\_ran = {                                  'units': [16, 32, 64, 128, 256],                                  'epochs': range(1, 101),                                  'batch\_size': [16, 32, 64, 128, 256],                                  'learning\_rate': [0.0001, 0.001]                                  }                              opti\_search = RandomizedSearchCV(m, param\_distributions=param\_ran, cv=3, n\_iter=10, n\_jobs=-1, scoring='neg\_mean\_squared\_error')                              opti\_search.fit(eda.X\_valid, eda.y\_valid) |

* GridSearchCV: Khai báo các tham số cần truyền cho quá trình tối ưu. Sau đó thực hiện quá trình tối ưu các tham số và cuối cùng chọn ra bộ tham số có mức độ **neg\_mean\_squared\_error** cao nhất.

|  |
| --- |
| param\_grid = {                                  'units': [32, 64, 128, 256],                                  'epochs': [20, 40, 60, 80, 100],                                  'batch\_size': [16, 32, 64, 128],                                  'learning\_rate': [0.0001, 0.001]                                  }                              opti\_search = GridSearchCV(m, param\_grid=param\_grid, n\_jobs=-1, cv=3 ,scoring='neg\_mean\_squared\_error', error\_score='raise')                                opti\_search.fit(eda.X\_valid, eda.y\_valid) |

* + - * BayesSearchCV:

|  |
| --- |
| param\_bay = {                                  'units': Categorical([16, 32, 64, 128, 256]),                                  'epochs': Integer(1, 100),                                  'batch\_size': Categorical([16, 32, 64, 128, 256]),                                  'learning\_rate': Categorical([0.0001, 0.001])                                  }                              opti\_search =BayesSearchCV(m, search\_spaces=param\_bay, cv=3, n\_iter=10, n\_jobs=-1, scoring='neg\_mean\_squared\_error') |

**HƯỚNG DẪN SỬ DỤNG**

* Trong đề tài này, tất cả các thư mục được lưu trong cùng một file lớn có tên “KLTN\_CNN\_TimeSeries”. Trong đó, bao gồm các thư mục con như sau:
  + Data: File chứa các thư mục dữ liệu (1\_weather.csv, 3\_Gold\_Price.csv, 8\_DauTho Historical Data.csv, 9\_monthly-beer-production.csv).
  + Model: Chứa tệp Optimize\_Model.pth và các file mô hình. Đây là tệp để lưu các tham số của mô hình sau khi optimize và các mô hình sau khi huấn luyện.
  + Source: Chứa source code chính của chương trình .
  + Static: Chứa 2 file X\_scaler.pkl và y\_scaler.pkl. Đây là 2 file lưu trữ dữ liệu sau khi scale của X và y.
  + Output: thư mục chứa dữ liệu được xuất ra.
* Các bước thực nghiệm như sau:
  + Bước 1: Mở đến giao diện chính của đề tài
    - Mở Visual Code và thực hiện chọn đường dẫn đến folder “KLTN\_CNN\_TimeSeries”:

A screenshot of a computer

Description automatically generated

Hình 1: Mở Visual Code và mở thư mục “KLTN\_CNN\_TimeSeries”

* + - Tại thanh công cụ, ta chọn **Terminal** 🡪 **New Terminal**, để thực hiện mở một Terminal mới. Sau đó, ta nhập lệnh sau vào Terminal:

**streamlit run .\source\main.py**

**A screen shot of a computer

Description automatically generated**

Hình 2: Chạy lệnh streamlit run .\source\main.py

* + - Truy cập đến địa chỉ [**http://localhost:8501**](http://localhost:8501) để vào chương trình

**A screenshot of a computer

Description automatically generated**

Hình 3: Giao diện chính của chương trình

* + Bước 2: Thực hiện Optimize Model:
    - Ta thực hiện chọn tập dữ liệu hoặc kéo thả tập dữ liệu vào nơi upload huấn luyện. Ví dụ: Khi ta kéo thả tập dữ liệu **DauTho Historical Data.csv** từ thư mục **Data** vào file upload, chương trình sẽ tự động hiển thị dữ liệu dưới dạng bảng, trực quan dữ liệu dưới dạng biểu đồ và ta cũng có thể thay đổi cột thống kê:
      * Dữ liệu dưới dạng bảng:

A screenshot of a computer screen

Description automatically generated

Hình.4: HIển thị tập dữ liệu đã chọn

* + - * Trực quan hóa dữ liệu:

A graph with blue lines and white text

Description automatically generated

Hình.5: Trực quan hóa tập dữ liệu đã chọn

* + - Tiếp theo, ta thực hiện chọn mô hình sử dụng và các thông số để thực hiện Optimize như: Mô hình sử dụng, số ngày input, số ngày output, tỷ lệ tập train, validation, test (100%-train%-validation%), hàm kích hoạt, phương pháp chuẩn hóa dữ liệu. Cuối cùng, ta thực hiện Optimize bằng cách bấm vào nút **Optimize Model**

**A screenshot of a video player

Description automatically generated**

Hình 6: Chọn các thông số và mô hình sử dụng

* + - Sau khi Optimize hoàn thành, ta thu được các tham số tốt nhất của mô hình:

**A screenshot of a black and white screen

Description automatically generated**

Hình 7: Các siêu tham số sau khi bấm Optimize Model

* + - Ngoài việc thực hiện Optimizer để nhận được các thông số huấn luyện thì ta có thể thực hiện chọn các thông số thủ công bằng cách click vào checkbox “**Nhập các tham số huấn luyện tự chọn:”**  và thực hiện chọn các thông số.
* Bước 3: Thực hiện train mô hình:
  + Ta thực hiện train model với các siêu tham số vừa Optimize bằng cách click vào nút **Train model** ở góc trái màn hình. Kết quả thu được là thời gian train model.

A screen shot of a computer

Description automatically generated

Hình 8: Các siêu tham số được sử dụng để train và thời gian train

A screenshot of a computer

Description automatically generated

Hình 9: Bảng trọng số các lớp của mô hình vừa huấn luyện

* + Ngoài việc train mô hình, ta có thể thực hiện Re-Train mô hình với những tham số đã dùng để huấn luyện mô hình trước đó mà không cần nhập hay Optimizer lại.
* Bước 4: Thực hiện kiểm định (Test mô hình):
  + Ta thực hiện chọn tập dữ liệu vừa huấn luyện để thực hiện test mô hình. Tương tự như Train, ta thực hiện chọn hoặc kéo thả file nơi upload và chương trình sẽ tự động hiển thị dữ liệu dưới dạng bảng, trực quan dữ liệu dưới dạng biểu đồ:
    - Dữ liệu dạng bảng

A screenshot of a computer screen

Description automatically generated

Hình 10: Hiển thị tập dữ liệu tiến hành test mô hình

* + - Trực quan hóa dữ liệu:

A graph with blue lines and white text

Description automatically generated

Hình 11: Trực quan hóa tập dữ liệu sắp tiến hành testing

* + Để thực hiện kiểm tra mô hình, ta thực hiện chọn cột cần dự đoán và bấm vào nút **Test model** ở góc trái màn hình:

**A screenshot of a computer

Description automatically generated**

Hình 12: Lựa chọn 1 cột trong tập dữ liệu testing để tiến hành dự đoán

* + Kết quả thu được bao gồm: Các thông số đã Train model, bảng thông tin dự đoán – thực tế, bảng thông số lỗi của mô hình và một biểu đồ trực quan hóa dữ liệu dự đoán – thực tế**.**

A screenshot of a computer

Description automatically generated

Hình 13: Kết quả sau khi test mô hình gồm: Các siêu tham số đã dùng để train, Kết quả dự đoán và thực tế, và các thông số lỗi

A graph on a screen

Description automatically generated

Hình 14: Biểu đồ so sánh kết quả dự đoán và thực tế

* + Ở đây, ta cũng có thể thực hiện xuất kết quả sau khi test mô hình bằng cách nhấp vào nút “**Xuất kết quả**” ở cuối màn hình.

**A screenshot of a table

Description automatically generated**

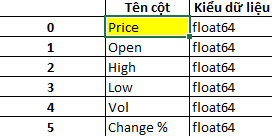
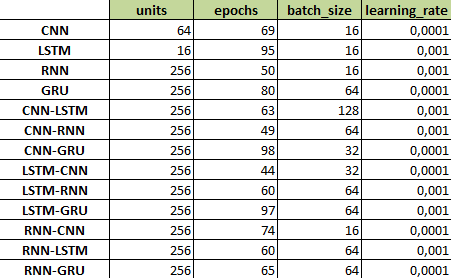
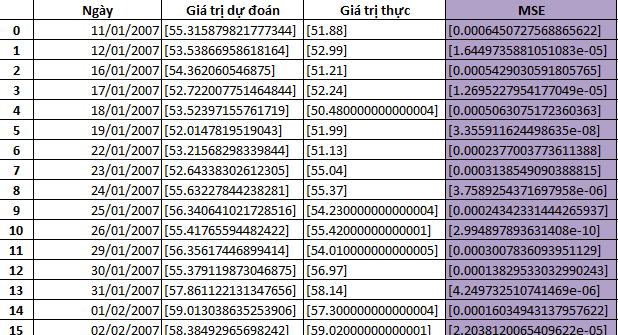
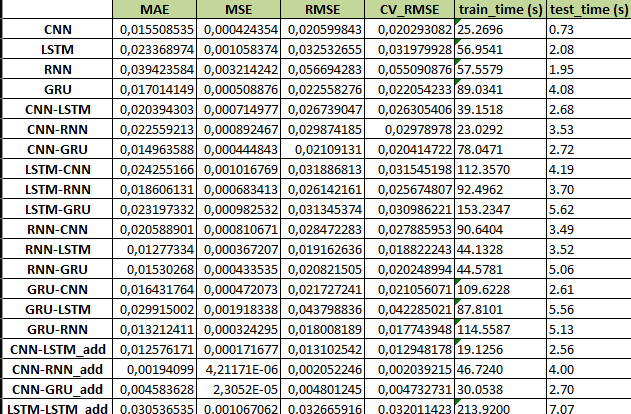
Hình 15: Hình ảnh minh họa kết quả xuất 1 mô hình

* Ngoài ra, ta còn có thể xuất tất cả các mô hình trên cột mà ta test, bằng cách click vào nút “**Xuất tất cả model train trên cột muốn test**” ở phía trên nút “**Test model**”. Tuy nhiên, để xuất tất cả các mô hình, chương trình sẽ yêu cầu ta huấn luyện đầy đủ 46 mô hình yêu cầu.

A screenshot of a computer

Description automatically generated

*Hình 16: Các mô hình yêu cầu để xuất tất cả các mô hình cùng lúc*



*Hình 17: Hình ảnh minh họa kết quả xuất tất cả mô hình*