Group 1

Team Member - Email

Thanh Tam Vo - 103487596@student.swin.edu.au

Phuc Khai Hoan Cao - 103804739@student.swin.edu.au

Tai Minh Huy Nguyen - 104220352@student.swin.edu.au

April 5th, 2024

Table of Contents

Task B7 Introduction

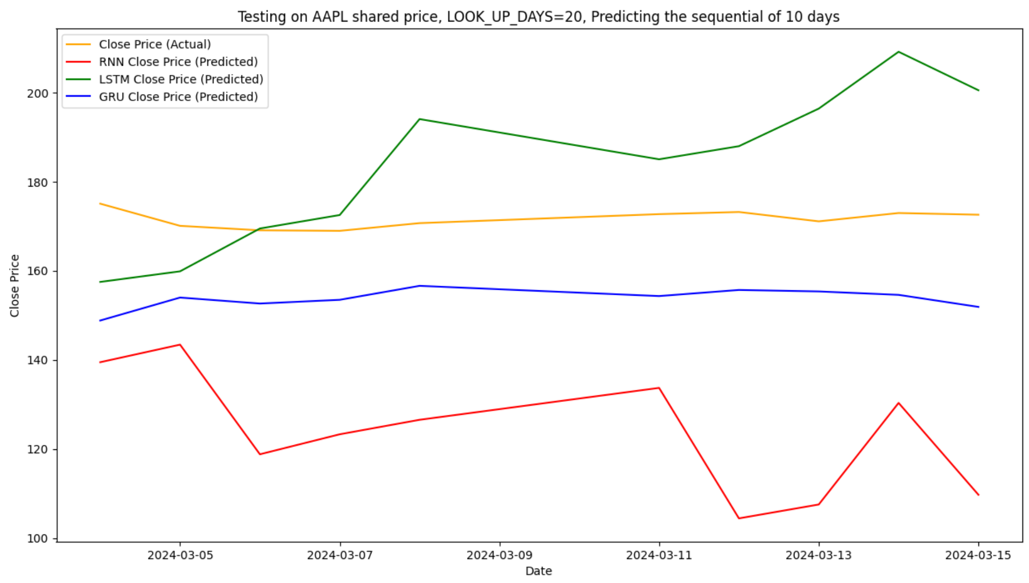
Introduction

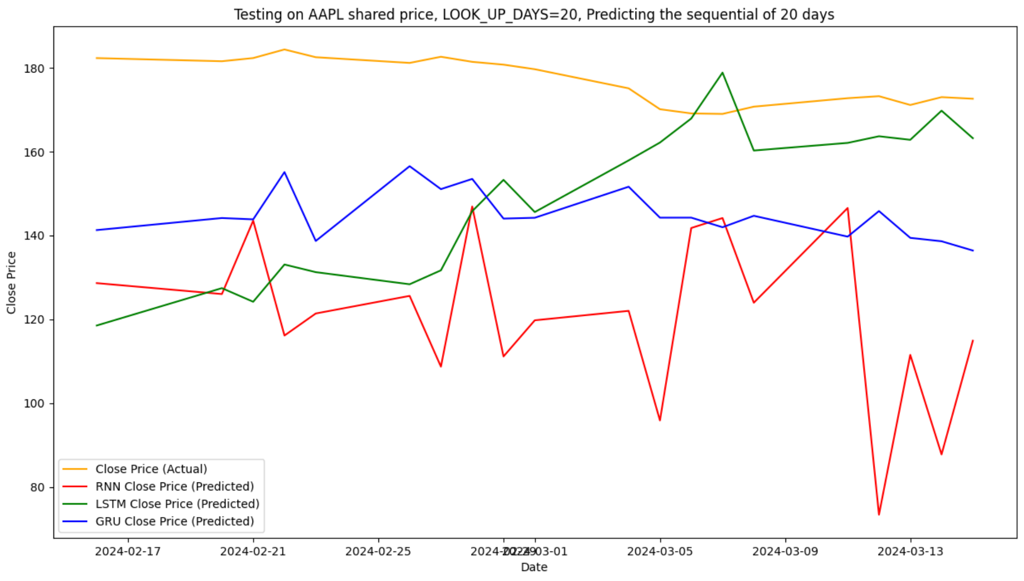
This is Task B7. Task B7 is an individual task that requires all of the members to understand and elevate the algorithm to the most potential state, while not making the algorithm lose its basic functionalities.

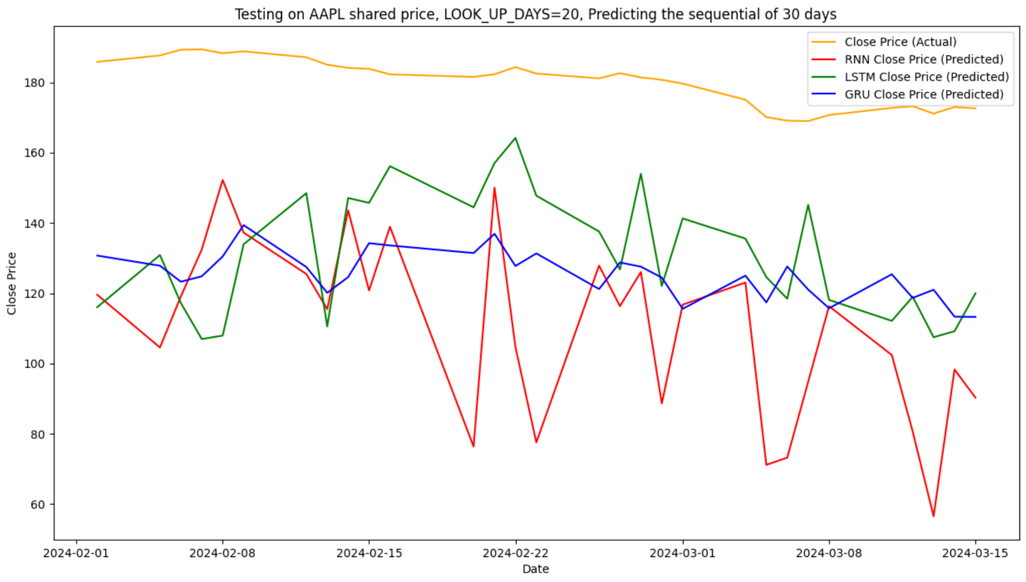
In our project, by using the 3 different methods known as GRU (Gated Recurrent Unit), LSTM (Long Short Term Memory), and RNN (Recurrent Neural Network), we can safely predict the outcome of the future stock market. Therefore, we predicted not just one value, we are taking 6 values, which are Open, Close, Low, High, Adj Close, and Volume. So, it is crucial to improve and readjust the model so it can give the greatest results.

From Task B5, we can clearly see that while the results are somewhat losing their tempo once it has to predict longer and longer extension (10 days, 20 days and 30 days.) The diagrams seem to fall off even more overtime.

So, for each of the methods we have provided you, I will be improving the results of these methods by giving them solutions to counter the problems.




Testing on AAPL shared price, LOOK_UP_DAYS=20, Predicting the sequential of 20 days
Close Price
180
160
140
120
100
80
Close Price (Actual)
RNN Close Price (Predicted)
LSTM Close Price (Predicted)
GRU Close Price (Predicted)
2024-02-17
2024-02-21
2024-02-25
2024-02-29
2024-03-01
2024-03-05
2024-03-09
2024-03-13
Date


Testing on AAPL shared price, LOOK_UP_DAYS=20, Predicting the sequential of 30 days
Close Price
180
160
140
120
100
80
60
Close Price (Actual)
RNN Close Price (Predicted)
LSTM Close Price (Predicted)
GRU Close Price (Predicted)
2024-02-01
2024-02-08
2024-02-15
2024-02-22
2024-03-01
2024-03-08
2024-03-15
Date

RNN (Recurrent Neural Network)

Recurrent neural networks are neural networks that have been specifically designed to handle data sequences with time step indexes t ranging from 1 to τ, such as x(t)= x(1),..., x(τ). The neural network must be adjusted to consider the dependencies between the data points if the data are arranged in a sequence where one data point depends on the one before it. To store the states or data from earlier inputs and produce the subsequent output in the sequence, RNNs are equipped with the idea of "memory."

As RNN is a very basic AI-prediction model, henceforth, it is the base of almost every powerful AI models afterwards. To counter the problems of t there are multiple ways to improve the model.

These are the problems of RNN models:

- Vanishing Gradient Problem:

When gradients are backpropagated over time during training, they may get exponentially tiny, resulting in disappearing gradients. The network's capacity to identify long-term dependencies in sequential data is hampered by this issue.

- Exploding Gradient Problem:

Gradients may grow exponentially huge and cause an explosion of gradients. It may be difficult to train the network efficiently due to this problem as it might impede convergence and create instability during training.

- Short-Term Memory:

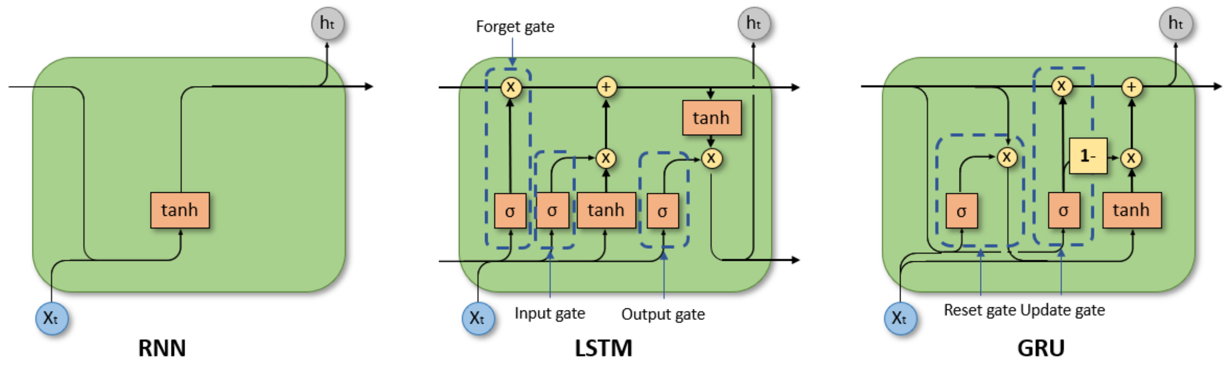
Because of the recurrent connections in Standard RNN architectures, they struggle to store information over lengthy sequences. Their short-term memory tendencies make it difficult for them to identify dependencies across longer time horizons.

Hence, here are some of my solutions towards this problem.

Use other methods like GRU and LSTM

Since RNN is the base, there are way better methods to use for predicting Stock Data, such as GRU and LSTM. While GRU is better at handling the smaller datas, LSTM is way more versatile when it comes to huge amount of data for prediction.

Hence, it is more than likely that you will use these two models instead of RNN.



Regulation & Normalization

One way of increasing the reliability of RNN is to add a regulation to the model, or regularization techniques. One way of doing this is to add a dropout regularization. Since RNN can vanish some of the more important data while calculating because of overfitting, Dropout regularization will prevent them from happening by dropping out some of the more random data while calculating.

Here is one way that I have been thinking, throughout the code:

```
def LongShortTermMemory(layerNums=NUMBER_OF_LAYER,
hidden_units=NUMBER_OF_HIDDEN_UNITS, loss_type=LOSS_FUNCTION,
optimizerType=OPTIMIZER, dense_unit=1, activation=["tanh", "linear"],
dropoutRate=DROP_OUT_RATE):
    model = Sequential()


    for i in range(layerNums):
        if i == (layerNums - 1):
            model.add(LSTM(hidden_units, activation=activation[0]))
            model.add(Dropout(dropoutRate))
            model.add(Dense(units=dense_unit, activation=activation[1]))
            # Add BatchNormalization layer for better stability
            model.add(BatchNormalization())
        else:
            model.add(LSTM(hidden_units, activation=activation[0], return_sequences=True))
            model.add(Dropout(dropoutRate))
            # Add BatchNormalization layer for better stability
            model.add(BatchNormalization())

    # Compile the model with Adam optimizer and specified learning rate
    model.compile(loss=loss_type, optimizer=optimizerType)
    return model
```

Explain this code, I have put a Dropout layer:

- After every SimpleRNN layer, dropout layers with the provided dropout rate (dropoutRate) are added.

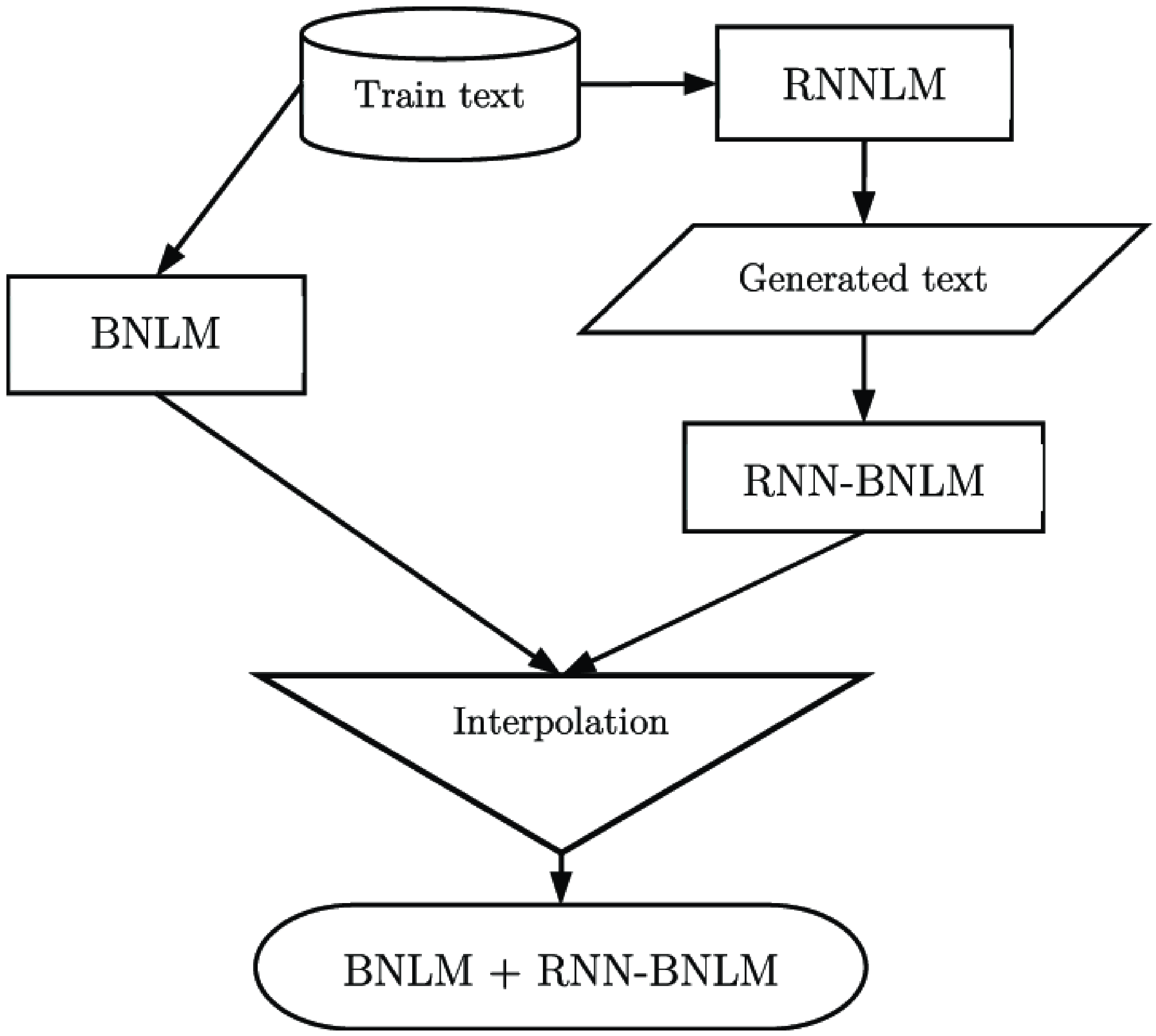
- One hyperparameter that regulates the percentage of units dropped during training is the dropout rate. This parameter can be changed in accordance with the effectiveness of your model and the level of regularization required.

Your RNN model's capacity to generalize can be enhanced and overfitting can be mitigated by incorporating dropout regularization.

Data Augmentation:

Much like Regularization, Data Augmentation is another way to improve the RNN. This technique revolves around introducing synthetic variations or perturbations to the original time series. Techniques such as random noise injection, time warping, or data resampling can help create a more diverse and representative dataset.

For instance, we would give the model the close price value regularly for the model data, so it can continue to study to give out a more efficient result. As an example, for every 10 days, we will give the Close Value for the model, so it can study and compare the difference, hence forth giving out more accurate predictions.


Train text
RNNLM
Generated text
BNLM
RNN-BNLM
Interpolation
BNLM + RNN-BNLM

GRU (Gated Recurrent Unit)

A modified version of recurrent neural networks (RNNs) called the gated recurrent unit (GRU) was created to address some of the drawbacks of traditional RNNs, including the vanishing gradient problem. Time series prediction, audio recognition, and natural language processing are just a few of the domains where GRUs have shown promise.

Earlier, in the RNN model section, GRU is an improvement towards the RNN model. Consequently, to improve GRU, there are way fewer options to implement the elevation, since GRU has fixed some of the most potential errors, including vanishing gradient problems.

Let’s see if GRU model has any problems regarding of the diagrams from task B5 and in general:

- **Long-Term Dependency Capture Difficulties:**
  If you take notice inside the Task B5, the GRU model performs way weaker the more time it needs to predict without additional data.
  Just like conventional RNNs, GRUs may have trouble identifying long-term dependencies in sequential data. Even if they partially address the vanishing gradient issue, they can still have trouble picking up intricate temporal patterns spanning a lot of time steps.

- **Limited Memory Capacity:**
  The fixed-size memory of GRUs may make it more difficult for them to remember information over extended periods. Even with their ability to selectively update and reset their internal states, they can still find it difficult to efficiently gather and store pertinent data over long periods of time.

In a nutshell, one of the main problems of GRU is that they have really limited memory bars. For that reason, my main focus on improving the model revolves around increasing the efficiency and capacity of the model.

Enhance Ability to Normalize

Since GRU has one of the most noticeable problems of having limited capacity, one of the solutions to improve the model is to enhance the ability to normalize information throughout the whole process.

In this provided code below, I have thought of making some changes which is BatchNormalization(). With the help of the layer, the model can dynamically determine how important each time step is in the sequence, which helps it to efficiently attend to pertinent information and get around the problem of limited memory capacity.

Here is the code:

```
def GatedRucurrentUnit(layerNums=NUMBER_OF_LAYER,
hidden_units=NUMBER_OF_HIDDEN_UNITS, loss_type=LOSS_FUNCTION,
optimizerType=OPTIMIZER, dense_unit=1, activation=["tanh", "linear"],
dropoutRate=DROP_OUT_RATE):
  model = Sequential()


  for i in range(layerNums):
    if i == (layerNums - 1):
      model.add(GRU(hidden_units, activation=activation[0]))
      model.add(Dropout(dropoutRate))
      model.add(Dense(units=dense_unit, activation=activation[1]))
      # Add BatchNormalization layer for better stability
      model.add(BatchNormalization())
    else:
      model.add(GRU(hidden_units, activation=activation[0], return_sequences=True))
      model.add(Dropout(dropoutRate))
      # Add BatchNormalization layer for better stability
      model.add(BatchNormalization())


  # Compile the model with Adam optimizer and specified learning rate


  model.compile(loss=loss_type, optimizer=optimizerType)
  return model
```

Long-Term Dependency Implementation

The other thing about GRU that needs attention is the fact that they don't have any long-term dependencies, which means that they are extremely fragile when studying bigger datas. I also tried to add Dropout and dependency. Both have different purposes:

- A dropout layer for regularization comes after each layer in the implementation of stacked GRU layers, which are added in a loop.

- After every other GRU layer (except from the final one), bidirectional GRU layers are added to capture dependencies from both the past and the future context.

Here is the code:

```
def GatedRucurrentUnit(layerNums=NUMBER_OF_LAYER,
hidden_units=NUMBER_OF_HIDDEN_UNITS, loss_type=LOSS_FUNCTION,
optimizerType=OPTIMIZER, dense_unit=1, activation=["tanh", "linear"],
dropoutRate=DROP_OUT_RATE):
  model = Sequential()


  for i in range(layerNums):
    if i == (layerNums - 1):
      model.add(GRU(hidden_units, activation=activation[0]))
      model.add(Dropout(dropoutRate))
      model.add(Dense(units=dense_unit, activation=activation[1]))
      # Add BatchNormalization layer for better stability
      model.add(BatchNormalization())
    else:
      model.add(GRU(hidden_units, activation=activation[0], return_sequences=True))
      model.add(Dropout(dropoutRate))
      # Add BatchNormalization layer for better stability
      model.add(BatchNormalization())


  # Compile the model with Adam optimizer and specified learning rate


  model.compile(loss=loss_type, optimizer=optimizerType)

  return model
```

LSTM (Long-Short Term Memory)

LSTM is another modified model based on the concept of RNN. Because LSTMs have feedback connections, they can take advantage of temporal dependencies across data sequences, in contrast to traditional feedforward neural networks. Hence, LSTM is also built to fix the vanishing gradient problems.

Because of their far more advanced memory cell than GRU, LSTM can store and update data over extended periods of time. With the aid of this memory cell, long-term dependencies are more successfully captured by LSTM networks and the vanishing gradient issue is lessened. However, there are still cracks inside the model that we need to point out.

- **Complexity**
  When compared to other recurrent architectures such as Gated Recurrent Units (GRUs) and standard RNNs, the architecture of LSTM networks is more intricate. Because of its complexity, LSTMs may be more difficult to comprehend, use, and optimize—especially for practitioners with little to no background in deep learning.

- **Training Difficulty**
  To properly train LSTM networks, hyperparameters like unit count, learning rate, batch size, and dropout rate must be carefully adjusted. It can take a lot of time and money to find the ideal set of hyperparameters, particularly for complicated jobs and large-scale datasets.

In order to fix these potential problems, I have come up with more ways to readjust the LSTM model. These will most likely include simplifying the LSTM model.

Complexity Simplified

So, I have thought of using the same methods, but you will need to simplify the LSTM model.

- Rather than stacking numerous LSTM layers, only one LSTM layer (or Bidirectional LSTM layer) is introduced.

- To make the model simpler, the LSTM layer's hidden unit count is decreased.
- To avoid overfitting, dropout regularization is incorporated.
- The loss function, metrics, and optimizer that are specified during model compilation are used.

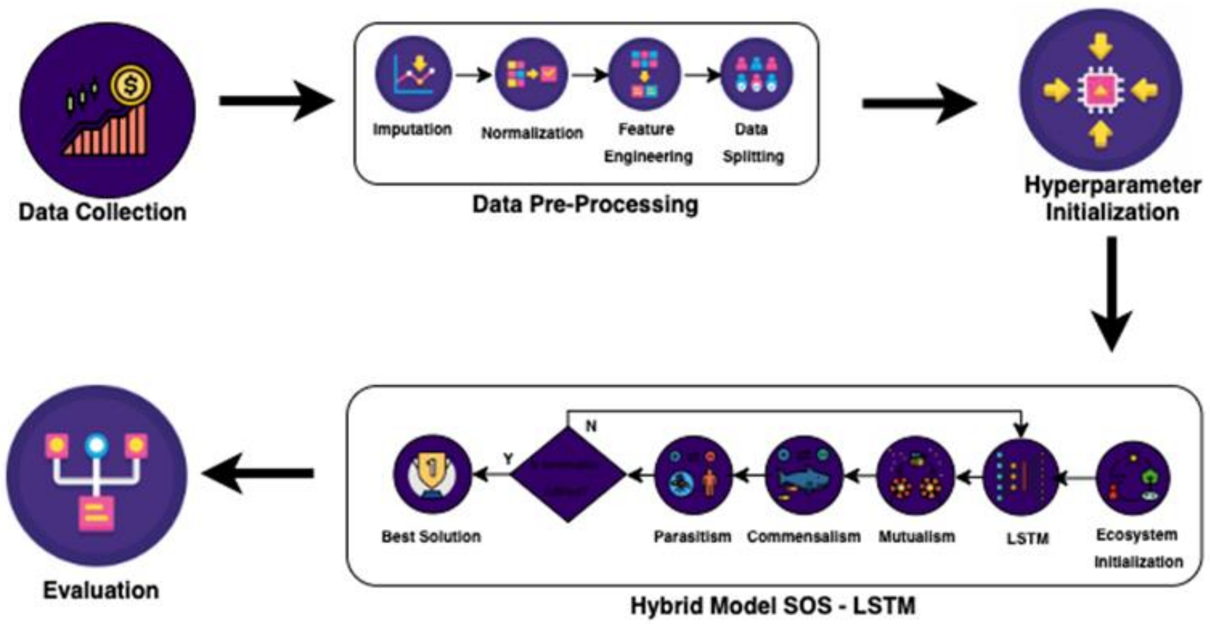
Here is the code that I have tried to implement.

```
def LongShortTermMemory(layerNums=NUMBER_OF_LAYER,
hidden_units=NUMBER_OF_HIDDEN_UNITS, loss_type=LOSS_FUNCTION,
optimizerType=OPTIMIZER, dense_unit=1, activation=["tanh", "linear"],
dropoutRate=DROP_OUT_RATE):
  model = Sequential()


  for i in range(layerNums):
    if i == (layerNums - 1):
      model.add(LSTM(hidden_units, activation=activation[0]))
      model.add(Dropout(dropoutRate))
      model.add(Dense(units=dense_unit, activation=activation[1]))
      # Add BatchNormalization layer for better stability
      model.add(BatchNormalization())
    else:
      model.add(LSTM(hidden_units, activation=activation[0], return_sequences=True))
      model.add(Dropout(dropoutRate))
      # Add BatchNormalization layer for better stability
      model.add(BatchNormalization())


  # Compile the model with Adam optimizer and specified learning rate
```

```
    model.compile(loss=loss_type, optimizer=optimizerType)

    return model
```

Hyperparameter Optimization

Hyperparameter optimization is another technique that you can use to further optimize your LSTM model. It entails figuring out the ideal set of values for the parameters—like the number of layers, units, epochs, learning rate, or activation function—that govern the model's behavior and performance.



In reality, I can automatically find the ideal set of hyperparameters using methods like grid search or random search and apply hyperparameter optimization inside your LSTM code. Furthermore, for more effective hyperparameter tweaking, I can employ more sophisticated techniques like genetic algorithms or Bayesian optimization.

Here is the code:

```
from sklearn.model_selection import GridSearchCV

from keras.wrappers.scikit_learn import KerasRegressor


def create_lstm_model(hidden_units=64, activation='tanh', dropout_rate=0.2,
optimizer='adam'):
    model = Sequential()

    model.add(LSTM(hidden_units, activation=activation, dropout=dropout_rate))

    model.add(Dense(1))

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

    return model


lstm_regressor = KerasRegressor(build_fn=create_lstm_model, verbose=0)


param_grid = {

    'hidden_units': [32, 64, 128],

    'activation': ['relu', 'tanh'],

    'dropout_rate': [0.2, 0.3, 0.4],

    'optimizer': ['adam', 'rmsprop']

}

grid_search = GridSearchCV(estimator=lstm_regressor, param_grid=param_grid, cv=3,
scoring='neg_mean_squared_error')

grid_result = grid_search.fit(X_train, y_train)


print("Best parameters found: ", grid_result.best_params_)


best_model = grid_result.best_estimator_

best_model.fit(X_train, y_train)
```

More ways to research and implement for Data Stock

While GRU performs better in forecasting the stock price than LSTM, there are still numerous ways to even elevate these data.

Hybrid Architecture:

Since GRU is more bounded to predict the stock market data, we will pick GRU as the base of the code.

Afterward, I have tried to include extra layers or parts to improve the functionality of the main model. This might consist of:

- **Convolutional layers:** To extract geographical characteristics from the input.
- **Dense layers:** For feature extraction and additional processing.
- **Mechanisms of attention:** To concentrate on pertinent segments of the input sequence.
- **Remaining connections:** To enhance training stability and make gradients flow more easily.
- **Skip connections:** To stop fading gradients and to help information move between levels.

Here is my one attempt to recreate the hybrid structure.

```
from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import GRU, LSTM, SimpleRNN, Dense, Concatenate


def HybridModel(input_shape):

  model = Sequential()


  # Add GRU, LSTM, and RNN layers

  model.add(GRU(64, return_sequences=True, input_shape=input_shape))

  model.add(LSTM(64, return_sequences=True))
```

```
    model.add(SimpleRNN(64, return_sequences=True))


    # Concatenate the outputs of the three recurrent layers
    model.add(Concatenate())


    # Add additional layers for further processing
    model.add(Dense(64, activation='relu'))
    model.add(Dense(1, activation='linear'))


    return model
```

TCN (Temporal Convolutional Networks)

According to Dr Barak Or (2020), Convolutional Networks (TCNs) for segmenting actions in videos. This traditional procedure consists of two steps: first, computing low-level features (typically with CNN) that encapsulate spatial-temporal information; and second, feeding these low-level features into a classifier (commonly with RNN) that captures high-level temporal information. Such an approach's primary drawback is the need for two different models. TCN offers a single method for hierarchically capturing both layers of information.

However, while they can be used in segmenting actions, they can be used as a way to train in forecasting stock market data. Based on the source coming from John You (2023), I have tried to make a simplified version of TCN predictions. Here is the version I created (Keep in mind that I still tried to fix the errors):

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, MaxPooling1D, Flatten, Dense


scaler = MinMaxScaler(feature_range=(0, 1))
```

```
data['close_scaled'] = scaler.fit_transform(data['close'].values.reshape(-1, 1))


def create_sequences(data, seq_length):
   X, y = [], []
   for i in range(len(data) - seq_length):
      X.append(data[i:i+seq_length])
      y.append(data[i+seq_length])
   return np.array(X), np.array(y)


sequence_length = 10
X, y = create_sequences(data['close_scaled'].values, sequence_length)
split = int(0.8 * len(X))
X_train, X_test = X[:split], X[split:]
y_train, y_test = y[:split], y[split:]


model = Sequential([
   Conv1D(filters=64, kernel_size=3, activation='relu', input_shape=(sequence_length, 1)),
   MaxPooling1D(pool_size=2),
   Conv1D(filters=32, kernel_size=3, activation='relu'),
   MaxPooling1D(pool_size=2),
   Flatten(),
   Dense(64, activation='relu'),
   Dense(1, activation='linear')
])


model.compile(optimizer='adam', loss='mse', metrics=['mae'])
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test, y_test))

loss, mae = model.evaluate(X_test, y_test)
print(f'Test Loss: {loss}, Test MAE: {mae}')

predictions = model.predict(X_test)
predictions = scaler.inverse_transform(predictions)

import matplotlib.pyplot as plt

plt.plot(predictions, label='Predictions')
plt.plot(scaler.inverse_transform(y_test.reshape(-1, 1)), label='Actual')
plt.legend()
plt.show()
```

In this code, I have provided the basic functionalities from the TCN code such as importing data.

Depending on your particular dataset and requirements, I might need to modify the architecture, preprocessing, and hyperparameters. Additionally, to maximize the TCN model's performance for stock prediction jobs, think about experimenting with various setups.

Conclusion

I have successfully tried to implement and improve the models by pointing out some of the more optimistic solutions. While it may not be perfect, this could bring the model closer to perfection.

Reference

1. Or, B. (2023, December 14). Temporal Convolutional Networks, the next revolution for Time-Series? *Medium*. https://medium.com/metaor-artificial-intelligence/temporal-convolutional-networks-the-next-revolution-for-time-series-8990af826567

2. You, J. (2023, November 9). Stock price prediction using Temporal Convolutional Networks. Medium. https://medium.com/@johnswyou/stock-price-prediction-using-temporal-convolutional-networks-451a7f9164dd