Week 4 Attempt Summary

This week we were tasked with moving from our v2 code base to the v3 code base

Before the report begins, In the previous versions of the code base, the csv file was downloaded to the same directory as the code base was. In the new version of the code base, the downloaded data, testing results and the logs are saved separately in there corresponding folder.

In this Week we take our code base to a more optimal level where we don't construct the deep learning model manually by adding layers of the network, instead we create a function that would take some inputs as 'number_of_layers' , "layer_name (LSTM, RNN and GRU)" and etc..., to create a deep learning model for us.

```
def create_model(n_steps, n_features, loss='huber', units=256,
cell=GRU, n_layers=2, dropout=0.4, optimizer='adam',
bidirectional=False):
    model = Sequential()
    for i in range(n_layers):
        if i == 0:
            # First layer
            if bidirectional:
                model.add(tf.keras.layers.Bidirectional(cell(units,
return_sequences=True), input_shape=(n_steps, n_features)))
            else:
                model.add(cell(units, return_sequences=True,
input_shape=(n_steps, n_features)))
        elif i == n_layers - 1:
            # Last layer
            if bidirectional:
                model.add(tf.keras.layers.Bidirectional(cell(units,
return_sequences=False)))
            else:
                model.add(cell(units, return_sequences=False))
```

```
        else:
            # Hidden layers
            if bidirectional:
                model.add(tf.keras.layers.Bidirectional(cell(units,
return_sequences=True)))
            else:
                model.add(cell(units, return_sequences=True))
        model.add(Dropout(dropout))

    model.add(Dense(1, activation='linear'))
    model.compile(loss=loss, optimizer=optimizer)
    print("Model created and compiled.")
    return model
```

The create model function will now create a simple neural network using the Tensor Flow library including the following features:

- **The function takes an input for a "number of layers" variable that can be changed in the code.**
- **The Function will allow to use LSTM and GRU as cell types**
- **The function uses bidirectional layers**
- **The Use of Hyper parameters**

Sequence Creation

Then function "create_sequences" prepares the time series data for training by creating input and output pairs.

The flow of the function looks like this

- **1st the function takes in 2 parameters, data and n_step**
  - **Data – The time series data**
  - **n_step – the number of time steps to use for each input sentence.**
- **It creates 2 list elements, named "X" and "Y", to store the input sequence and corresponding output values.**

- **It Iterates through the data, creating overlapping sequences of length 'n_step' for inputs for "X" and the value after each sequence for output "Y"**
- **Finally refactoring it as a Numpy Array to be the return of the function.**

Function definition

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

Iterative Prediction

This function is used to create future predictions beyond the training data, in this specific case the number of future steps is equal to the variable "**LOOK_UP_STEPS**" which is 15. (This can be seen further into the code where the function is instantiated)

The function takes in 4 arguments

- The name of the model – *model*
- The dataset – *data*
- The number of time steps used in training – *n_step*
- The required number of future steps to be predicted – *Future_steps*

The Function will enter into a for loop for a specific number of further steps

- It uses the model to predict the next values based on the current sequence.
- Then the scaling will be applied to inverse transform the prediction data to get the actual price.

- Then it will continue to append the sequence with the new predicted values while removing the old values

Function definition

```
def iterative_predict(model, data, n_steps, future_steps):
    """
    Predict prices iteratively one step at a time, using previous
predictions as inputs.
    """
    # Retrieve the last 'n_steps' from the test set (ensure the
correct column is used)
    last_sequence = data['test']['Adj Close'].values[-n_steps:]

    # Reshape to match the model's expected input shape: (1, N_STEPS,
1)
    last_sequence = last_sequence.reshape((1, n_steps, 1))

    predictions = []

    for step in range(future_steps):
        # Predict the next step
        prediction = model.predict(last_sequence)

        # Reshape the prediction to match the shape needed for
appending (1, 1)
        prediction = prediction.reshape((1, 1, 1))

        # If scaling, apply inverse transform using the 'Adj Close'
scaler only
        if SCALE:
            prediction = data['column_scaler']['Adj
Close'].inverse_transform(prediction.reshape(-1, 1)).reshape(1, 1, 1)

        # Store the predicted price
        predictions.append(prediction[0][0][0])
```

```
        # Update the sequence with the new prediction for the next
step
        # Remove the first element and append the prediction at the
end
        last_sequence = np.append(last_sequence[:, 1:, :], prediction,
axis=1)

    return predictions
```

Getting the final Data Frame (df)

This function is responsible for constructing the last and final data frame that includes the true data and the predicted data.

It prepares the test data by extracting the ‘Adj Close’ value

Then it reshapes the test data to fit the model expected input shape (batch_size and n_step)

It does the prediction from the trained model

Then an inverse transformation is done on both true and predicted data to get the actual stock prices

Finally returning it to the function.

```
def get_final_df(model, data, n_steps, look_ahead=LOOKUP_STEP):

    # Prepare the test data
    X_test = data["test"]['Adj Close'].values
    X_test = np.array([X_test[i:i + n_steps] for i in
range(len(X_test) - n_steps)])

    # Reshape X_test to be in the form (batch_size, n_steps,
n_features)
    X_test = X_test.reshape((X_test.shape[0], n_steps, 1))

    y_test = data["test"]['Adj Close'].values[n_steps:]  # True future
prices
```

```
    # Perform prediction on the test set
    y_pred = model.predict(X_test)

    # If scaling is enabled, inverse transform the predicted and true
values
    if SCALE:
        y_test = data["column_scaler"]["Adj
Close"].inverse_transform(np.expand_dims(y_test, axis=0)).flatten()
        y_pred = data["column_scaler"]["Adj
Close"].inverse_transform(y_pred).flatten()

    # Create a copy of the test DataFrame
    test_df = data["test_df"].copy()

    # Ensure the test DataFrame has enough rows to match the predicted
values
    test_df = test_df.iloc[-len(y_test):].copy()

    # Add predicted future prices and true future prices to the
DataFrame
    test_df[f"adjclose_{look_ahead}"] = y_pred
    test_df[f"true_adjclose_{look_ahead}"] = y_test

    return test_df
```