Task 5 Report

NACA HITCHMAN - 103072170

Multistep problem

```
#   multi step prediction function
def create_sequences_ms(data, seq_length, n_steps_ahead):
    xs = []
    ys = []
    for i in range(len(data)-(seq_length+n_steps_ahead)+1):
        x = data[i:(i+seq_length)]
        y = data[(i+seq_length):(i+seq_length+n_steps_ahead)]
        xs.append(x)
        ys.append(y)
    return np.array(xs), np.array(ys)
```

My function create_sequences_ms helps to solve the multistep prediction problem. This function allows predictions to be made for a sequence of closing prices of n_steps_ahead days into the future. The create_sequences_ms function takes three parameters: data, seq_length, and n_steps_ahead. It creates two arrays, xs and ys, which store the input and output sequences for training the machine learning model. The function iterates over the data and appends subsequences of length seq_length to xs, and then extracts another subsequence of length n_steps_ahead from the data for the y sequence, starting from the index immediately following the end of the x subsequence. This subsequence represents the target closing prices to be predicted.

```
# Predict the next k days
real_data = [data['X_test'][-1, :, :]]
real_data = np.array(real_data)
print(real_data.shape)
real_data = np.reshape(real_data, (real_data.shape[0], real_data.shape[1],
n_features))

# Predict the next k days
prediction = model.predict(real_data)  # shape: (1, k, n_features)
prediction =
data["column_scaler"][prediction_column].inverse_transform(prediction[:, :,
closing_price_index])  # shape: (1, k)

# Loop over the prediction and print each day's predicted price
for i, price in enumerate(prediction[0]):
    print(f"Prediction for day {i+1}: {price}")
```

I also have adjusted the code for actually outputting next k days' stock prices. It starts by preparing the input data (real_data) using the last available data from the test set. The shape of real_data is adjusted to match the expected input shape of the model. Then, the model predicts the next k days' stock prices using the prepared input data. The predicted prices are transformed back to their original scale using a column scaler. Finally, a loop is used to print each day's predicted price.

Multivariate problem

```
def create_sequences_mv(data, seq_length):
    xs = []
    ys = []
    for i in range(len(data)-(seq_length+1)):
        x = data[i:(i+seq_length)]
        y = data[i+seq_length]
        xs.append(x)
        ys.append(y)
    return np.array(xs), np.array(ys)
```

My function above helps to solves the simple multivariate prediction problem. It creates sequences considering other features of the same stock (including opening price, highest price, lowest price, closing price, adjusted closing price, and trading volume) as input for predicting the closing price of the company for a specified day in the future. The more important work comes from passing the data to it, as all the features of the data must be passed, and not just the Close column. Creation of the model must also be adjusted to account for this, including passing the feature count, and reshaping the model output to include the feature count.

```
closing_price_index = FEATURE_COLUMNS.index(prediction_column)

# Get the actual prices
actual_prices =
data["column_scaler"][prediction_column].inverse_transform(data["y_test"][:, -
1, closing_price_index].reshape(-1,1)).ravel()
# Predict the prices
predicted_prices = model.predict(data['X_test'])
predicted_close_prices = predicted_prices[:, -1, closing_price_index].reshape(-
1, 1)
predicted_close_prices =
data["column_scaler"][prediction_column].inverse_transform(predicted_close_pric
es).ravel()
```

As can be seen here, as the model prediction now outputs data with a shape that includes the feature count, we need to get the index of the column we want (which is ‘Close’ in this case) and use that to only get the data for that part of the prediction output.

Combination

Combining the two is as simple as just including the necessary code for both systems as described above. The only code that needs to be adjusted to fit both systems is in the model creation, where the dense layer and output shape need to be adjusted.

```
# Apply a TimeDistributed Dense layer to each time step independently
model.add(TimeDistributed(Dense(n_features)))
# Select the last n_outputdays time steps from the sequence
model.add(Lambda(lambda x: x[:, -n_outputdays:, :]))
model.compile(optimizer=optimizer, loss=loss)

return model
```

As can be seen here, when the dense layer is added, we are also using Time Distributed to apply it to each time step, assisting with both the multi step and multi variate factors. The Lamba layer then must be added to ensure the model output is the correct shape.

Results

Firstly, for the multi-step, we can the prediction successfully displays for the specified number of days in the future.

```
N_STEPS = 5;
```

```
1/1 [==============================] - 0s 178ms/step
Prediction for day 1: 193.30259704589844
Prediction for day 2: 196.73583984375
Prediction for day 3: 196.71815490722656
Prediction for day 4: 201.86024475097656
Prediction for day 5: 207.55703735351562
```

Moving on to multi-variate, the left is the result of my v0.4 code, and the right is the result of my current code. Neither are especially accurate. It may mean a more complex model is needed that would take longer to train to properly see results. Regardless, the code does fulfill the specifications of the task.