```
#%% md
# Neural Computing Coursework
Objectives:
1. Compare the two RNN algorithms: RNN and LSTM
2. Performance comparison
3. Contrast with their advantage and disadvantages


Dataset: 10 years of S&P 500 index data
NECO Methods:
1. Recurrent Neural Network
2. LSTM
#%% md
## All imports and settings
#%%
import torch
from torch import nn
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score, mean_squared_error
from skorch import NeuralNetRegressor
from statsmodels.tsa.seasonal import seasonal_decompose

plt.rcParams['figure.dpi'] = 200
plt.rcParams['figure.figsize'] = (10,5)
#%% md
## Use of computational devices
The following block will set the devicd according to what is available. The priority has been set as **cuda** > **metal for Apple Silicon
#%%
if torch.cuda.is_available():
  device = torch.device('cuda')
# elif torch.backends.mps.is_available():
#   device = torch.device('mps')
else:
  device = torch.device('cpu')

print("Using device: {}".format(device))
#%% md
Read the data from csv file and set the index column to **Date**
#%%
stock_data = pd.read_csv('sp500_index.csv', index_col='Date', parse_dates=True)
print(stock_data.head())
#%%
close = stock_data[['S&P500']]
print(close.head())
#%% md
## First look of the Time Series Data
- The first graph shows the data plotting across time. The interval here is by day.
- The second grpah here showed the value after applying the method of differencing.
#%%
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = (16,9)
plt.rcParams['figure.dpi'] = 100
stock_data.plot(y='S&P500')
plt.grid('on')
plt.title("Time Series Data of S&P500 index from 2013 to 2023")
#%%
fig, ax = plt.subplots(5, 5)
# plt.figure(figsize=(10, 10))
plt.tight_layout()
col = 0
row = 0
while row<25:
    pd.plotting.lag_plot(stock_data['S&P500'], lag=row+1, ax=ax[col][row%5])
    ax[col][row%5].set_title("Lag {}".format(row+1))
    row += 1
    if row%5 == 0:
        col += 1

plt.subplots_adjust(top=5, bottom=3, right=1.2)

#%%
from statsmodels.tsa.seasonal import seasonal_decompose

series = stock_data
decomposition_result = seasonal_decompose(series,period=252, model='multiplicative')
decomposition_result.plot()
#%%
x = stock_data[['S&P500']].values
time = np.linspace(0, len(x), len(x)).reshape(-1, 1)

trend = LinearRegression()
trend.fit(time, x)
x_trend = trend.predict(time)
stock_data.plot(kind='line', y='S&P500')
# plt.plot(x_trend)

plt.legend(['S&P500', 'Trend Component'])
plt.annotate('R2-Score: {}'.format(r2_score(x, x_trend)), xy=(10, 4750))
plt.annotate('RMSE: {}'.format(np.sqrt(mean_squared_error(x_trend, x))), xy=(10, 4650))
#%%
close_data = close['S&P500'].values.astype('float32')
#%%
stock_data.describe().transpose()
#%% md
## Dataset preparation
#%% md
The range of the time series differs more than 3000 considering the maxima and minima. Therefore, it is better to have the dataset normali
#%%
```

```python
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature_range=(-1, 1))
stock_data['S&P500'] = scaler.fit_transform(stock_data['S&P500'].values.reshape(-1, 1))
stock_data.describe().transpose()
#%%
def create_dataset(seq, window_size):
    x = []
    y = []
    for i in range(len(seq) - window_size):
        # windows are the training features
        window = seq[i : i+ window_size]
        # The RNN is many-to-many architecture, therefore the output size
        label = seq[i + window_size: i + window_size + 1]
        x.append(window)
        y.append(label)

    # Convert to numpy array for easier indexing
    x = np.array(x)
    y = np.array(y)

    # Split data into training and testing set
    train_size = int(0.8*len(seq))
    x_train, x_test = x[:train_size], x[train_size:]
    y_train, y_test = y[:train_size], y[train_size:]

    # Convert to tensor
    x_train = torch.from_numpy(x_train).type(torch.Tensor).to(device)
    x_test = torch.from_numpy(x_test).type(torch.Tensor).to(device)
    y_train = torch.from_numpy(y_train).type(torch.Tensor).to(device)
    y_test = torch.from_numpy(y_test).type(torch.Tensor).to(device)

    x_train = x_train.unsqueeze(2)
    x_test = x_test.unsqueeze(2)
    # return processed data
    return x_train, x_test, y_train, y_test

lookback = 100
training_size = int(len(stock_data)*0.8)
raw_data = stock_data['S&P500'].values
train_data = raw_data[:training_size]
test_data = raw_data[training_size:]

X_train, X_test, y_train, y_test = create_dataset(train_data, lookback)
# For the X_train and X_test to be used for LSTM training the size should be [1763, 12, 1] and [744, 12, 1]
print(X_train.size(), X_test.size())
# For the y_train and y_test, it should be [1763, 1, 1] and [744, 1, 1]
print(y_train.size(), y_test.size())
#%%
def create_testing_data(test_data, window_size):
    x = []
    y = []
    for i in range(len(test_data) - window_size):
        # windows are the training features
        window = test_data[i : i+ window_size]
        # The RNN is many-to-many architecture, therefore the output size
        label = test_data[i + window_size: i + window_size + 1]
        x.append(window)
        y.append(label)

    # Convert to numpy array for easier indexing
    x = np.array(x)
    y = np.array(y)
    x = torch.from_numpy(x).type(torch.Tensor)
    y = torch.from_numpy(y).type(torch.Tensor)
    x = x.unsqueeze(2)

    return x, y

testing_features, testing_target = create_testing_data(test_data, 100)
print(testing_features, testing_features)
#%% md
## Model Definition
#%% md
### RNN
#%%
class RNN(nn.Module):
    def __init__(self, hidden_dim=10, num_layers=1):
        super(RNN, self).__init__()
        self.input_dim = 1
        self.output_dim = 1
        self.num_layers = num_layers
        self.hidden_dim= hidden_dim
        self.rnn = nn.RNN(self.input_dim, hidden_dim, self.num_layers,batch_first=True)
        self.linear = nn.Linear(hidden_dim, self.output_dim)

    def forward(self, x):
        # x (batch_size, seq_length, input_size)
        # hidden (n_layers, batch_size, hidden_dim)
        # r_out (batch_size, time_step, hidden_size)
        x.to(device)
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_dim).requires_grad_().to(device)
        # get RNN outputs
        out, hn = self.rnn(x, h0)
        # get final output
        output = self.linear(out[:, -1, :])
        return output
#%% md
### LSTM
#%%
class LSTM(nn.Module):
    def __init__(self,hidden_dim=10, num_layers=1):
        super(LSTM, self).__init__()
        self.input_dim = 1
```

```python
        self.output_dim = 1
        self.hidden_dim = hidden_dim
        self.num_layers = num_layers
        self.lstm = nn.LSTM(self.input_dim, hidden_dim, num_layers, batch_first=True)
        self.linear = nn.Linear(hidden_dim, self.output_dim)

    def forward(self, x):
        # Initialize hidden state with zeros
        x.to(device)
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_dim).requires_grad_().to(device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_dim).requires_grad_().to(device)
        out, (hn, cn) = self.lstm(x, (h0, c0))
        out = self.linear(out[:, -1, :])
        return out
#%% md
## Model Training
#%%
def train_model(model, optimizer):
    num_epochs = 2000
    training_loss = []
    testing_loss = []
    loss_func = nn.MSELoss()

    # Generate data for training
    X_train, X_test, y_train, y_test = create_dataset(train_data, lookback)
    # Move all the stuff to computation device
    model.to(device)
    X_train.to(device)
    y_train.to(device)
    X_test.to(device)
    y_test.to(device)
    # (n_layers, batch_size, hidden_dim)

    for epoch in range(1, num_epochs + 1):
        model.train()
        pred= model(X_train)
        train_loss = loss_func(pred, y_train)
        optimizer.zero_grad()
        train_loss.backward(retain_graph=True)
        optimizer.step()
        training_loss.append(train_loss.item())

        if epoch % 100 == 0:
            model.eval()
            y_pred = model(X_test)
            test_loss = loss_func(y_pred, y_test)
            testing_loss.append(test_loss.item())
            print("Epoch {}, Train Loss: {}, Test Loss: {}".format(epoch, train_loss.item(), test_loss.item()))

    training_history = {
        "train_loss": training_loss,
        "test_loss": testing_loss
    }
    return model, training_history
#%% md
### RNN model Training
#%%
%%time
rnn_model_1 = RNN(hidden_dim=10, num_layers=1)
rnn_optimizer = torch.optim.Adam(rnn_model_1.parameters(), lr=0.01)
rnn_trained, rnn_history = train_model(rnn_model_1, rnn_optimizer)
#%%
plt.plot(rnn_history['train_loss'])
plt.title('RNN baseline training Loss')
#%% md
### LSTM model training
#%%
%%time
lstm_model_1 = LSTM(hidden_dim=10, num_layers=1)
lstm_optimizer = torch.optim.Adam(lstm_model_1.parameters(), lr=0.01)
lstm_trained, lstm_history = train_model(lstm_model_1, lstm_optimizer)
#%%
plt.plot(rnn_history['test_loss'])
plt.title('RNN baseline testing loss')
#%% md
## Results Comparison
#%%
def visualize_prediction(lstm=lstm_trained, rnn=rnn_trained):
    lstm_pred = lstm(testing_features).cpu().detach().numpy()
    rnn_pred = rnn(testing_features).cpu().detach().numpy()
    plt.figure(figsize=(10, 5), dpi=200)
    plt.grid('on')
    plt.plot(lstm_pred)
    plt.plot(rnn_pred)
    plt.plot(testing_target.cpu().detach().numpy())
    plt.legend(['LSTM', 'RNN', 'S&P500'])
    plt.title("Prediction Comparison")
visualize_prediction(lstm_model_1, rnn_model_1)
#%%
plt.figure(figsize=(10, 5), dpi=150)
plt.plot(lstm_history['train_loss'])
plt.plot(rnn_history['train_loss'])
plt.legend(['LSTM', 'RNN'])
plt.ylabel('Training Loss')
plt.xlabel('Epochs')
plt.title('Training Loss')
#%%
plt.figure(figsize=(10, 5), dpi=150)
epochs = [i*100 for i in range(1, 21)]
plt.plot(epochs, lstm_history['test_loss'])
plt.plot(epochs, rnn_history['test_loss'])
plt.ylabel('Validation Loss')
plt.xlabel('epochs')
plt.grid('on')
```

```python
plt.legend(['LSTM', 'RNN'])
plt.title("Loss on the Validation Set")

#%%
lstm_pred = lstm_trained(testing_features).cpu().detach().numpy()
rnn_pred = rnn_trained(testing_features).cpu().detach().numpy()
print("R2_score of LSTM: {}".format(r2_score(lstm_pred, testing_target.cpu().detach().numpy())))
print("R2_score of RNN: {}".format(r2_score(rnn_pred, testing_target.cpu().detach().numpy())))
#%%
mse_loss = nn.MSELoss()
print("LSTM Loss, MSE: {}, RMSE: {}".format(mse_loss(lstm_trained(testing_features), testing_target).detach().numpy(), np.sqrt(mse_loss(ls
print("RNN Loss, MSE: {}, RMSE: {}".format(mse_loss(rnn_trained(testing_features), testing_target).detach().numpy(), np.sqrt(mse_loss(rnn_
#%% md
## Model Optimization
#%%
from skorch import NeuralNetRegressor
from sklearn.model_selection import GridSearchCV
from skorch.callbacks import EarlyStopping

early_stopping = EarlyStopping(patience=10)
lstm_optimized = NeuralNetRegressor(
    LSTM,
    criterion=nn.MSELoss,
    optimizer=torch.optim.Adam,
    verbose=0,
    device=device,
    callbacks=[early_stopping]
)
rnn_optimized = NeuralNetRegressor(
    RNN,
    criterion=nn.MSELoss,
    optimizer=torch.optim.Adam,
    verbose=0,
    device=device,
    callbacks=[early_stopping]
)
X_train, X_test, y_train, y_test = create_dataset(raw_data, lookback)

# For the X_train and X_test to be used for LSTM training the size should be [1763, 12, 1] and [744, 12, 1]

param_grid = {
    'module__num_layers': [1,2,3],
    'module__hidden_dim': [10, 20, 50],
    'lr': [0.1, 0.01, 0.001]
}
lstm_grid = GridSearchCV(
    estimator=lstm_optimized,
    param_grid=param_grid,
    n_jobs=1,
    cv=3,
    scoring='neg_mean_squared_error'
)
rnn_grid = GridSearchCV(
    estimator=rnn_optimized,
    param_grid=param_grid,
    n_jobs=1,
    cv=3,
    scoring='neg_mean_squared_error'
)
lstm_grid_result = lstm_grid.fit(X_train, y_train)
rnn_grid_result  = rnn_grid.fit(X_train, y_train)
#%%
result = pd.DataFrame(lstm_grid_result.cv_results_)
result.sort_values(by='rank_test_score').head(10)
#%%
%%time
lstm_best_result = result.sort_values(by='rank_test_score').head(1)

lstm_learning_rate = float(lstm_best_result['param_lr'].values)
lstm_hidden_dim = int(lstm_best_result['param_module__hidden_dim'].values)
lstm_num_layers = int(lstm_best_result['param_module__num_layers'].values)
print("Best Parameters, learning rate = {}, hidden_dim = {}, num_layers = {}".format(lstm_learning_rate, lstm_hidden_dim, lstm_num_layers)

X_train, X_test, y_train, y_test = create_dataset(train_data, lookback)
best_lstm = LSTM(hidden_dim=lstm_hidden_dim, num_layers=lstm_num_layers)
best_lstm.to(device)
best_lstm_optimizer = torch.optim.Adam(best_lstm.parameters(), lr=lstm_learning_rate)
best_trained_lstm, best_lstm_train_history = train_model(best_lstm, best_lstm_optimizer)
#%%
plt.figure(figsize=(10, 5), dpi=200)
plt.plot(best_lstm_train_history['test_loss'])
plt.title('Test Loss')
#%%
best_lstm_pred = best_trained_lstm(testing_features)
plt.plot(testing_target.detach().numpy())
plt.plot(best_lstm_pred.detach().numpy())
plt.legend(['Testing set', 'Best LSTM'])
plt.title('Prediction of Optimized LSTM')
#%%
rnn_result = pd.DataFrame(rnn_grid_result.cv_results_)
rnn_result.sort_values(by='rank_test_score', inplace=True)
rnn_result
#%%
%%time
rnn_best_result = rnn_result.head(1)

rnn_learning_rate = float(rnn_best_result['param_lr'].values)
rnn_hidden_dim = int(rnn_best_result['param_module__hidden_dim'].values)
rnn_num_layers = int(rnn_best_result['param_module__num_layers'].values)
print("Best Parameters, learning rate = {}, hidden_dim = {}, num_layers = {}".format(rnn_learning_rate, rnn_hidden_dim, rnn_num_layers))
X_train, X_test, y_train, y_test = create_dataset(train_data, lookback)
best_rnn = RNN(hidden_dim=rnn_hidden_dim, num_layers=rnn_num_layers)
best_rnn_optimizer = torch.optim.Adam(best_rnn.parameters(), lr=rnn_learning_rate)
best_rnn.to(device)
```

```
best_trained_rnn, best_rnn_train_history = train_model(best_rnn, best_rnn_optimizer)
#%%
plt.plot(range(1, 21), best_rnn_train_history['test_loss'])
plt.plot(range(1, 21), best_lstm_train_history['test_loss'])
plt.legend(['RNN', 'LSTM'])
plt.title('Testing loss of final model')
#%%
best_rnn_pred = best_trained_rnn(testing_features)
#%%
visualize_prediction(best_trained_lstm, best_trained_rnn)
#%%
print("R2_score of LSTM: {}".format(r2_score(best_trained_lstm(testing_features).detach().numpy(), testing_target.cpu().detach().numpy()))
print("R2_score of RNN: {}".format(r2_score(best_trained_rnn(testing_features).detach().numpy(), testing_target.cpu().detach().numpy())))
#%%
mse_loss = nn.MSELoss()
print("LSTM Loss, MSE: {}, RMSE: {}".format(mse_loss(best_trained_lstm(testing_features), testing_target).detach().numpy(), np.sqrt(mse_lo
print("RNN Loss, MSE: {}, RMSE: {}".format(mse_loss(best_trained_rnn(testing_features), testing_target).detach().numpy(), np.sqrt(mse_loss
#%%
all_data_features, all_data_target = create_testing_data(raw_data, 100)
best_lstm_pred = best_trained_lstm(all_data_features).detach().numpy()
best_rnn_pred = best_trained_rnn(all_data_features).detach().numpy()
base_rnn_pred = rnn_model_1(all_data_features).detach().numpy()
base_lstm_pred = lstm_model_1(all_data_features).detach().numpy()

plt.figure(figsize=(16, 9), dpi=200)
plt.plot(stock_data.index[100:], all_data_target.detach().numpy())
plt.plot(stock_data.index[100:], best_lstm_pred)
plt.plot(stock_data.index[100:], best_rnn_pred)
plt.plot(stock_data.index[100:], base_rnn_pred)
plt.plot(stock_data.index[100:], base_lstm_pred)
plt.axvline(stock_data.index[int(0.8*len(raw_data)+100):int(0.8*len(raw_data))+1+100], color='r')
plt.legend(['Orginal Data', 'Optimized LSTM', 'Optimized RNN', 'Base RNN', 'Base LSTM'])
#%% md
## Save Model for testing
#%%
torch.save(best_trained_rnn.state_dict(), 'best_rnn.pth')
torch.save(best_trained_lstm.state_dict(), 'best_lstm.pth')
#%% md
Test if the files can be loaded back
#%%
test_rnn = RNN(hidden_dim=50, num_layers=2)
test_rnn.load_state_dict(torch.load('best_rnn.pth'))
test_rnn(X_test)
#%%
```