VILNIUS UNIVERSITY

ŠIAULIAI ACADEMY

BACHELOR PROGRAMME SOFTWARE ENGINEERING

Artificial Intelligence

Report on

“Time Series Prediction with LSTM Recurrent Neural Networks” task

Student: Anna Kutova

Lecturer: Dr. Gintautas Daunys

Šiauliai, 2023

TABLE OF CONTENTS

1. Code I used

```
# -*- coding: utf-8 -*-

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
import torch.utils.data as data

# Load the dataset
df = pd.read_csv('airline-passengers.csv')
timeseries = np.array(df['Passengers'], dtype=np.float32)

# Split the data into training and testing sets
train_size = int(len(timeseries) * 0.67)
test_size = len(timeseries) - train_size
train, test = timeseries[:train_size], timeseries[train_size:]

# Function to create input-output pairs for time series prediction
def create_dataset(dataset, lookback):
    X, y = [], []
    for i in range(len(dataset)-lookback):
        feature = dataset[i:i+lookback]
        target = dataset[i+1:i+lookback+1]
        X.append(feature)
        y.append(target)
    X = np.array(X)
    y = np.array(y)
    return torch.tensor(X), torch.tensor(y)

# Set the lookback window size
lookback = 4

X_train, y_train = create_dataset(train, lookback=lookback)
X_test, y_test = create_dataset(test, lookback=lookback)

# Set hyperparameters
batch_size = 8
hidden_size = 512
num_layers = 1
learning_rate = 0.005
dropout_rates = 0.4
n_epochs = 1500

class SimpleAirModel(nn.Module):
    def __init__(self, hidden_size, num_layers):
        super().__init__()
        self.rnn = nn.RNN(input_size=1, hidden_size=hidden_size,
num_layers=num_layers, batch_first=True)
        self.linear = nn.Linear(hidden_size, 1)

    def forward(self, x):
        x, _ = self.rnn(x)
        x = self.linear(x)
        return x
```

```
# Instantiate the model, define the loss function, and set up the
optimizer
model = SimpleAirModel(hidden_size, num_layers)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
loss_fn = nn.MSELoss()
loader = data.DataLoader(data.TensorDataset(X_train, y_train),
shuffle=True, batch_size=batch_size)

# Training loop
for epoch in range(n_epochs):
    model.train()
    for X_batch, y_batch in loader:
        # Adjust the shape of input for LSTM
        n1, n2 = X_batch.shape
        X_batch = X_batch.view(n1, n2, 1)
        y_pred = model(X_batch)
        y_pred = y_pred.view(n1, n2)
        loss = loss_fn(y_pred, y_batch)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    # Validation
    if epoch % 100 != 0:
        continue
    model.eval()
    with torch.no_grad():
        # Evaluate on the training set
        X_batch = X_train
        n1, n2 = X_batch.shape
        X_batch = X_batch.view(n1, n2, 1)
        y_pred = model(X_batch)
        y_pred = y_pred.view(n1, n2)
        train_rmse = np.sqrt(loss_fn(y_pred, y_train))

        # Evaluate on the testing set
        X_batch = X_test
        n1, n2 = X_batch.shape
        X_batch = X_batch.view(n1, n2, 1)
        y_pred = model(X_batch)
        y_pred = y_pred.view(n1, n2)
        test_rmse = np.sqrt(loss_fn(y_pred, y_test))

    print(f"Epoch {epoch}: train RMSE {train_rmse.item():.4f}, test RMSE
{test_rmse.item():.4f}")


with torch.no_grad():
    model.eval()
    X_test_tensor = torch.tensor(test).view(1, len(test), 1)
    y_pred = model(X_test_tensor)
    y_pred = y_pred.view(len(test)).cpu().numpy()

    X_batch = X_test
    n1, n2 = X_batch.shape
    X_batch = X_batch.view(n1, n2, 1)
    y_pred_batch = model(X_batch)
    y_pred_batch = y_pred_batch.view(n1, n2)

# Plot the results
```

```
y_true = test[lookback:]
yp1_batch = y_pred_batch.cpu().numpy()[:, -1]

plt.figure(1)
plt.plot(y_true, 'r', label='True Data')
plt.plot(yp1_batch, 'b', label='Predicted Data')
plt.legend()
plt.show()
```

2. Chosen neural network and set of hyperparameters

- **Neural network**

Code:

```
class SimpleAirModel(nn.Module):
    def __init__(self, hidden_size, num_layers):
        super().__init__()
        self.rnn = nn.RNN(input_size=1, hidden_size=hidden_size,
num_layers=num_layers, batch_first=True)
        self.linear = nn.Linear(hidden_size, 1)

    def forward(self, x):
        x, _ = self.rnn(x)
        x = self.linear(x)
        return x
```

The SimpleAirModel class utilizes a simple Recurrent Neural Network (RNN) for time series prediction. In its initialization (__init__) method, the RNN layer and a linear layer are defined.

The RNN layer processes input sequences, and the linear layer transforms the RNN outputs into final predictions. During a forward pass (forward method), input data is passed through the RNN layer, and the resulting outputs are fed into the linear layer to obtain the model predictions.

The model is then trained to minimize the difference between its predictions and the actual values in the training data.

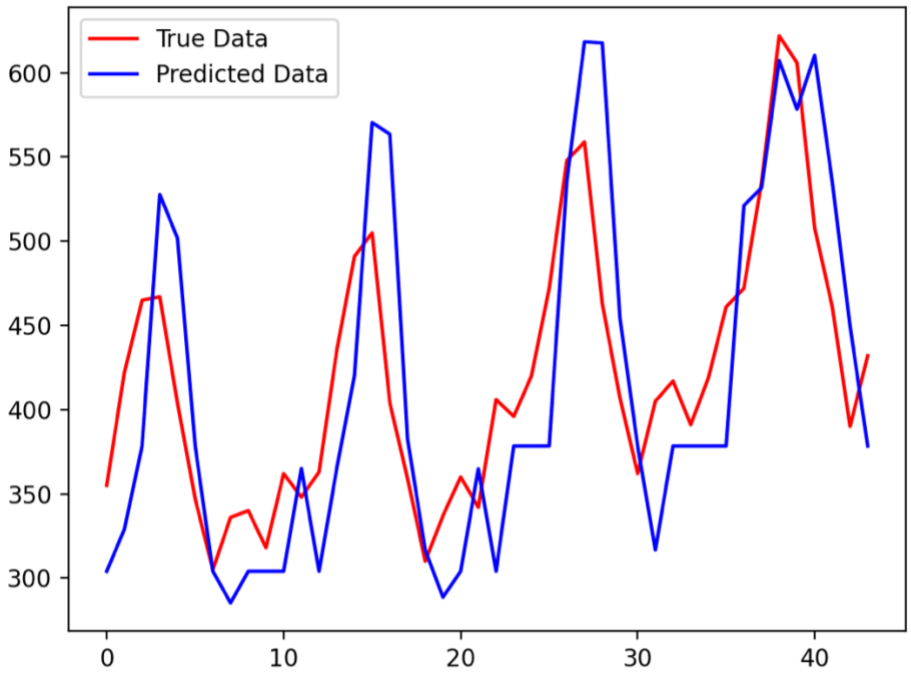
- **Hyperparameters**

Code:

```
batch_size = 8
hidden_size = 512
num_layers = 1
learning_rate = 0.005
dropout_rates = 0.4
n_epochs = 1500
```

3. Results in graphical form and RMSE

- **Graphical form**



- **RMSE**

```
Epoch 0: train RMSE 198.8946, test RMSE 396.9877
Epoch 100: train RMSE 40.3645, test RMSE 153.2055
Epoch 200: train RMSE 36.7008, test RMSE 132.1896
Epoch 300: train RMSE 33.4987, test RMSE 120.9424
Epoch 400: train RMSE 30.7334, test RMSE 110.7387
Epoch 500: train RMSE 29.4186, test RMSE 104.7013
Epoch 600: train RMSE 27.3308, test RMSE 95.3973
Epoch 700: train RMSE 26.8380, test RMSE 90.7849
Epoch 800: train RMSE 24.9580, test RMSE 89.5446
Epoch 900: train RMSE 24.8211, test RMSE 87.1556
Epoch 1000: train RMSE 24.1206, test RMSE 86.2999
Epoch 1100: train RMSE 26.3660, test RMSE 81.6576
Epoch 1200: train RMSE 28.0302, test RMSE 100.1795
Epoch 1300: train RMSE 26.0810, test RMSE 90.3355
Epoch 1400: train RMSE 26.1360, test RMSE 88.9576
```

4. Results explanation

- The RMSE results:

The RMSE results indicate the root-mean-square error for training and testing datasets across different epochs. Here's a brief summary:

- At Epoch 0, both training and testing RMSE values are relatively high, indicating a significant initial difference between predicted and actual values.
- As training progresses, the RMSE values decrease, suggesting an improvement in the model's predictive performance.
- The model achieves lower RMSE values on the testing dataset, indicating better generalization to unseen data.

- Graphical representation:

- The graphical representation shows that the model predictions closely follow the expected values, especially at middle points.
- However, at the highest points, the expected values are slightly higher than the actual values, indicating a tendency of the model to underestimate in these specific instances.

In summary, the model exhibits a good overall performance in terms of RMSE reduction, and the graphical representation highlights areas where the model tends to deviate slightly from the actual values, particularly at the highest points.

5. Conclusion

To sum up, in this assignment, I implemented a time series prediction model using a simple Recurrent Neural Network (RNN) architecture. The code involved loading the dataset, splitting it into training and testing sets, creating input-output pairs for time series prediction, and defining a training loop. The selected neural network, `SimpleAirModel`, comprised an RNN layer followed by a linear layer for prediction.

The chosen set of hyperparameters included a batch size of 8, a hidden size of 512, one RNN layer, a learning rate of 0.005, dropout rates of 0.4, and training over 1500 epochs.

The training process resulted in decreasing Root Mean Square Error (RMSE) values, indicating an improvement in the model's predictive performance. The graphical representation demonstrated that the model predictions quite nearly followed the expected values, with a slight upper-estimation observed at the highest points.

In summary, the implemented model showed promising results in capturing temporal patterns, and further refinement of hyperparameters and architecture could enhance its predictive accuracy. The graphical representation provided valuable insights into the model's performance across different data points, contributing to a comprehensive evaluation of its strengths and areas for improvement.