National University of Singapore
College of Design & Engineering - ECE

EE4400 Data Engineering and Deep Learning
Tutorial 3 - LSTM Solutions

Q1. LSTMs solve the problem using a unique additive gradient structure that includes direct access to the forget gate's activations, enabling the network to encourage desired behaviour from the error gradient using frequent gates updates on every time step of the learning process.

Q2. The key idea is that the network can learn what to store in the long-term state $\mathbf{c}_t$, what to throw away, and what to read from it:

- use forget gate to drop, then add what input gate selected
- the transformed long term state is combined with the output to form short term state $\mathbf{h}_t$ (equal to output $\mathbf{y}_t$).

Q3. Refer to the Jupyter notebook with results and explanations.

In [242]:

```
# Tutorial 3 Question 3

# import libraries

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN
from tensorflow.keras.layers import LSTM
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.layers import Dense

import numpy as np
from numpy import array
import os
import pandas as pd
import matplotlib.pyplot as plt
```

In [243]:

```
# define input sequence

raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]

# choose a number of time steps
seq_len = 3

# choose number of future time steps to predict
n_steps = 2
```

In [244]:

```
def split_sequence(sequence, seq_len, n_steps):
    X, y = list(), list()
    for i in range(len(sequence)):
        # find the end of this pattern
        end_ix = i + seq_len
        # check if we are beyond the sequence
        if end_ix + n_steps > len(sequence):
            break
        # gather input and output parts of the pattern
        seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:end_ix+n_steps]
        X.append(seq_x)
        y.append(seq_y)
    return array(X), array(y)
```

In [245]:

```
# split into sequences

seq_X, seq_Y = split_sequence(raw_seq, seq_len, n_steps)

# summarize the data
for i in range(len(seq_X)):
    print(seq_X[i], seq_Y[i])
```

```
[10 20 30] [40 50]
[20 30 40] [50 60]
[30 40 50] [60 70]
[40 50 60] [70 80]
[50 60 70] [80 90]
```

In [246]:

```
# split the sequence generated into final training X and Y sequence to be fed to model

X = seq_X.reshape((seq_X.shape[0], seq_X.shape[1], 1))
Y = seq_Y

print(X.shape, Y.shape)
```

```
(5, 3, 1) (5, 2)
```

In [247]:

```
#RNN
model = Sequential()
model.add(SimpleRNN(50, activation='sigmoid', input_shape=(seq_len, 1)))
model.add(Dense(n_steps))
model.compile(optimizer='adam', loss='mse')

print(model.summary())

# model training

history = model.fit(X, Y, epochs= 1000, verbose=0)
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 simple_rnn_1 (SimpleRNN)    (None, 50)                2600

 dense_1 (Dense)             (None, 2)                 102

=================================================================
Total params: 2,702
Trainable params: 2,702
Non-trainable params: 0
_________________________________________________________________
None
```

In [248]:

```
# Test Prediction

X_test = array([70, 80, 90])
X_test = X_test.reshape((1, seq_len, 1))
Y_test = model.predict(X_test)
print(Y_test)
```

```
1/1 [==============================] - 0s 131ms/step
[[39.35645 43.52418]]
```

In [249]:

```
# Batch Normalization

model = Sequential()
model.add(BatchNormalization(input_shape=(seq_len, 1)))
model.add(SimpleRNN(50, activation='sigmoid'))
# note: using ReLU instead of sigmoid improves RNN even without BN
model.add(BatchNormalization())
model.add(Dense(n_steps))
model.compile(optimizer='adam', loss='mse')

print(model.summary())
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 batch_normalization_1 (Batc  (None, 3, 1)             4
 hNormalization)

 simple_rnn_2 (SimpleRNN)    (None, 50)                2600

 batch_normalization_2 (Batc  (None, 50)               200
 hNormalization)

 dense_2 (Dense)             (None, 2)                 102

=================================================================
Total params: 2,906
Trainable params: 2,804
Non-trainable params: 102
_________________________________________________________________
None
```

For Simple RNN layer:

Number of parameters is given by the formula:

(num_features + num_units)* num_units + num_units

num_units = equals the number of units in the RNN

num_features = equals the number features of your input

Now you have two things happening in your RNN.

First you have the recurrent loop, where the state is fed recurrently into the model to generate the next step. Weights for the recurrent step are:

recurrent_weights = num_units*num_units

The secondly you have new input of your sequence at each step.

input_weights = num_features*num_units

So now we have the weights, whats missing are the biases - for every unit one bias:

biases = num_units*1

So finally we have the formula:

(num_features + num_units)* num_units + num_units = (50 + 1) * 50 + 50 = 2600

For Batch Normalization Layer:

Batch Normalization computes 4 parameters (beta, gamma, mean, std deviation) for each feature. So no. parameters = 4 * 50 = 200

Note that mean and std deviation are not trainable parameters as they are calculated.

(Work out the first BN layer yourself.)

For Output layer:

**output_channel_number * (input_channel_number + 1) = 2 * (50 + 1) = 102

In [250]:
```
# model training

history1 = model.fit(X, Y, epochs= 1000, verbose=0)
```

In [251]:
```
# Test Prediction

X_test = array([70, 80, 90])
X_test = X_test.reshape((1, seq_len, 1))
Y_test = model.predict(X_test)
print(Y_test)
```

```
1/1 [==============================] - 0s 161ms/step
[[ 99.656166 104.80522 ]]
```

In [252]:
```
#LSTM
model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(seq_len, 1)))
model.add(Dense(n_steps))
model.compile(optimizer='adam', loss='mse')

print(model.summary())
```

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 50)                10400

 dense_3 (Dense)             (None, 2)                 102

=================================================================
Total params: 10,502
Trainable params: 10,502
Non-trainable params: 0
_________________________________________________________________
None
```

For LSTM layer:

Number of parameters is given by the formula:

4 * ((num_features + num_units)* num_units + num_units)

4 - The factor 4 is multiplied for LSTM because LSTM computes 4 types of weights per layer (W_forget, W_input, W_output, W_cell)

num_units = equals the number of units in the LSTM

num_features = equals the number features of your input

recurrent_weights = num_units*num_units

input_weights = num_features*num_units

So now we have the weights, whats missing are the biases - for every unit one bias:

biases = num_units*1

So finally we have the formula:

4 * ((num_features + num_units)* num_units + num_units) = 4* ((50 + 1) * 50 + 50) = 10400

For Output layer:

**output_channel_number * (input_channel_number + 1) = 2 * (50 + 1) = 102

In [253]:
```
# model training

history2 = model.fit(X, Y, epochs= 1000, verbose=0)
```

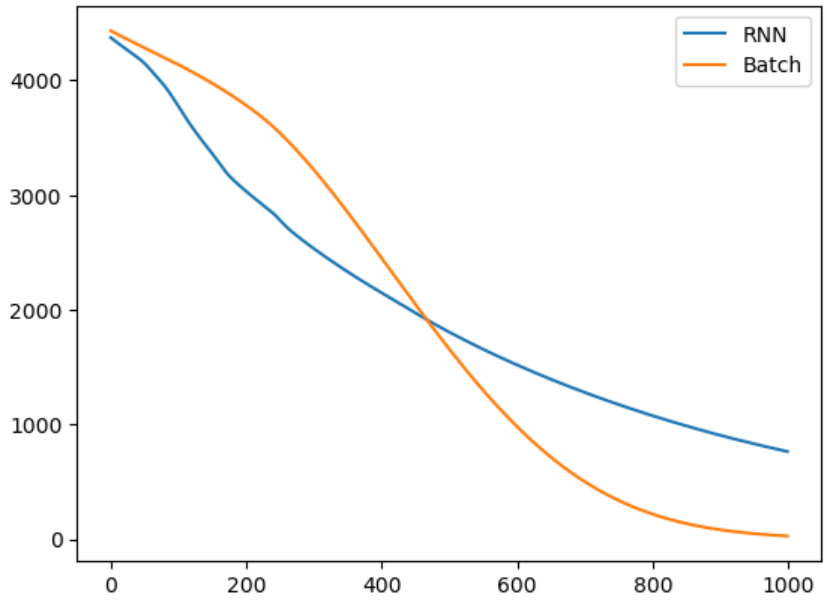
In [254]:
```
# Test Prediction

X_test = array([70, 80, 90])
X_test = X_test.reshape((1, seq_len, 1))
Y_test = model.predict(X_test)
print(Y_test)
```

```
1/1 [==============================] - 0s 158ms/step
[[101.01683  111.027054]]
```

In [255]:
```
# plot training loss
plt.plot(history.history["loss"])
plt.plot(history1.history["loss"])
plt.legend(("RNN", "Batch"))
```

Out[255]: <matplotlib.legend.Legend at 0x18ad701dbb0>



From the plot above we observe the loss function of the model without and with batch normalization.

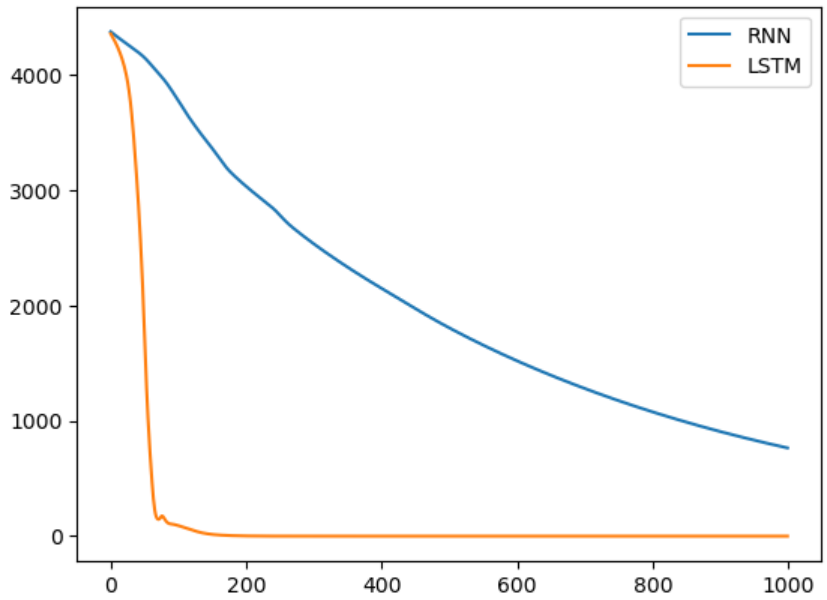
Batch normalization (BN) does not work well with RNN across time steps because the operation changes the characteristics of the sequential data presented in an RNN. The repeated application of the gain and offset in BN during the recurrent procedure may also lead to exploding or vanishing gradients.

However, batch normalization between RNN layers seems to be helpful when the activation function in the RNN is sigmoid.

Layer normalization can work well in an RNN as it captures statistics across inputs at a single time step.

In [256]:
```
1 plt.plot(history.history["loss"])
2 plt.plot(history2.history["loss"])
3 plt.legend(("RNN", "LSTM"))
```

Out[256]: <matplotlib.legend.Legend at 0x18ac5af2ca0>



From the plot above we can note that:

The performance of LSTM is better than RNN. To truly observe the benefit of LSTM, we can predict much further into the future. In our example, we only predict 2 time steps into the future. LSTM models are capable of predicting much further into the future than RNN with reasonable accuracy.

In [257]:
```
1 keras.backend.clear_session()
```