BSC~Microsoft Research Centre

DL: TimeSeries and RNN

Tassadaq Hussain
Professor Namal University
Director Centre for AI and Big Data

Collaborations:
Barcelona Supercomputing Center Barcelona, Spain
European Network on High Performance and Embedded Architecture and Compilation
Pakistan Supercomputing Center

UCERD
Gathering Intellectuals
www.ucerd.com

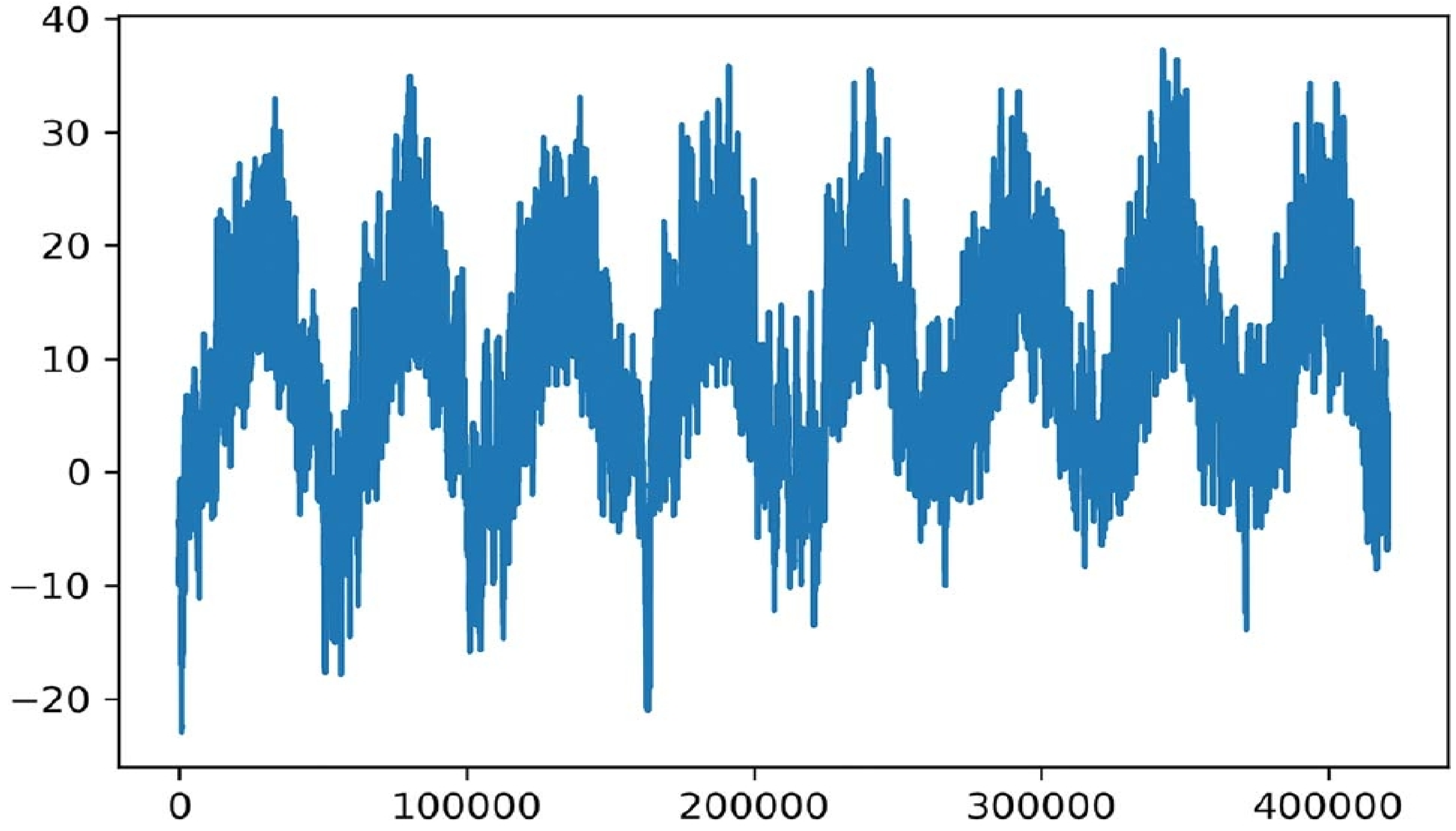
Understanding recurrent neural networks (RNNs)

Applying RNNs to a temperature-forecasting example

Advanced RNN usage patterns

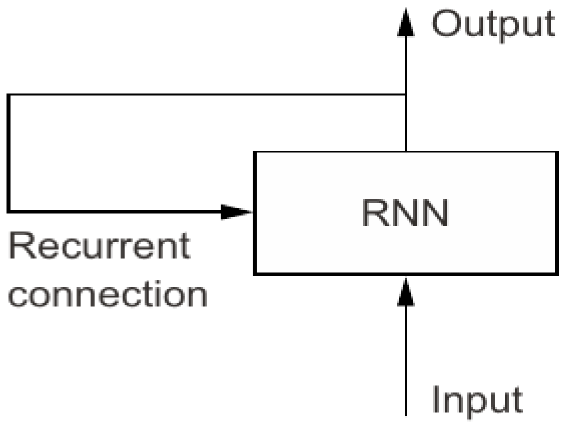
UCERD

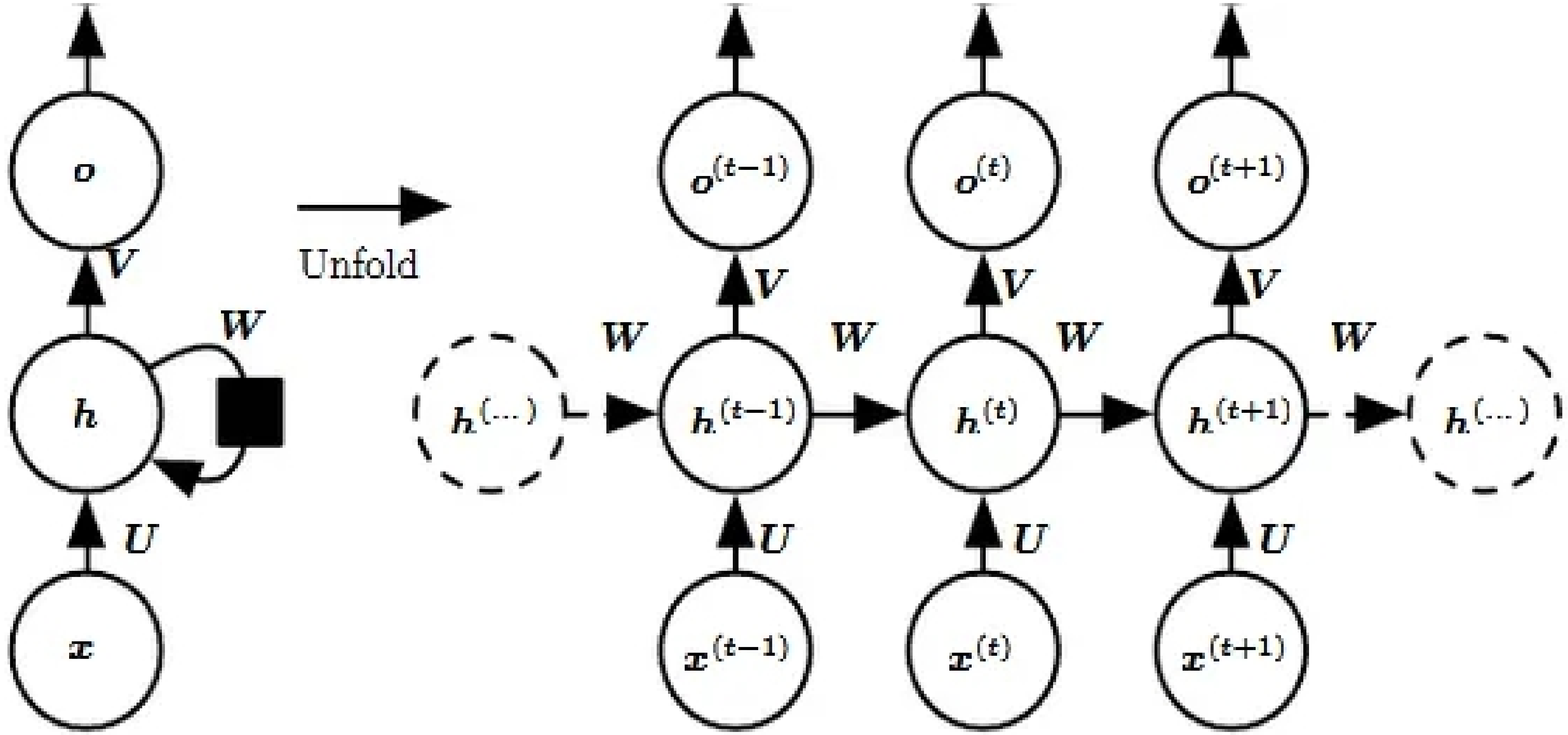
Temperature Sensor Data



What is RNN

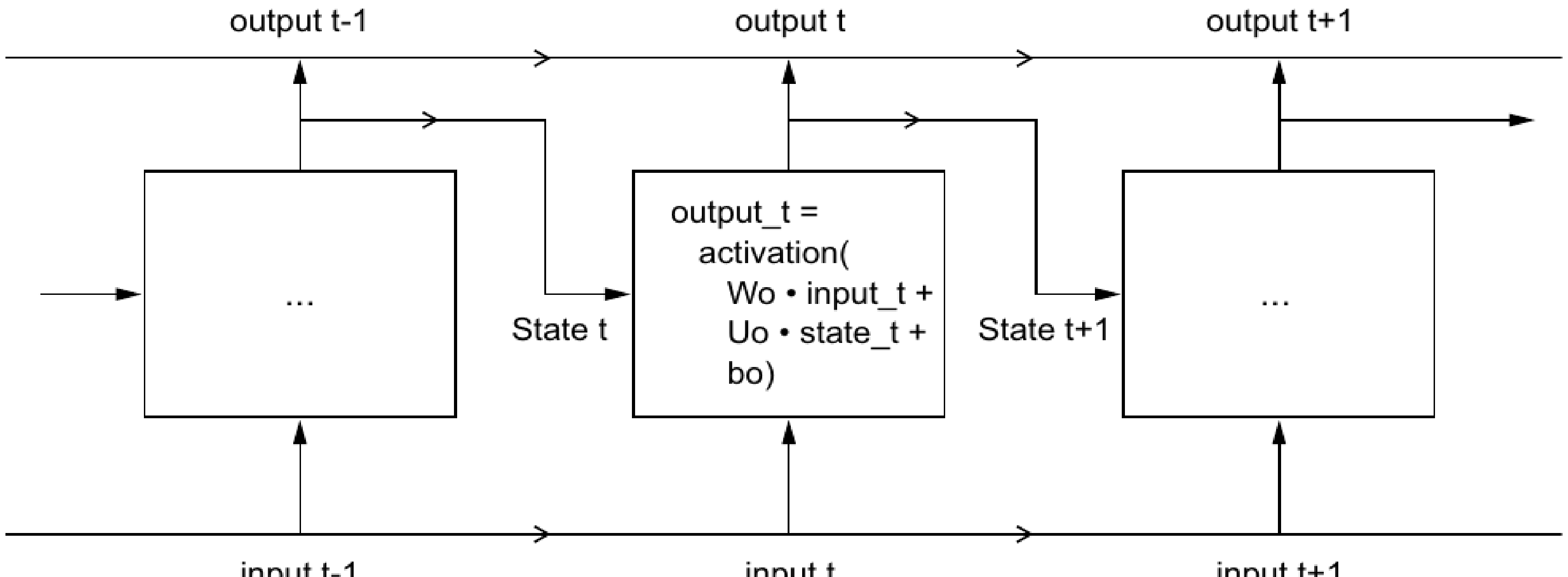
- A recurrent neural network is a neural network that is specialized for processing a sequence of data x(t)= x(1), . . . , x(τ) with the time step index T ranging from 1 to T.
- Applications such as: speech and language, NLP problem
- RNNs are called recurrent because they perform the same task for every element of a sequence, with the output being depended on the previous computations.
- Another way to think about RNNs is that they have a “memory” which captures information about what has been calculated so far.





o
V
W
h
U
x
Unfold
$o^{(t-1)}$
$o^{(t)}$
$o^{(t+1)}$
V
W
$h^{(\dots)}$
$h^{(t-1)}$
$h^{(t)}$
$h^{(t+1)}$
$h^{(\dots)}$
U
$x^{(t-1)}$
$x^{(t)}$
$x^{(t+1)}$

- In summary, an RNN is a for loop that reuses quantities computed during the previous iteration of the loop, nothing more.
- There are many different RNNs fitting this definition that you can be build



```
num_features = 14
inputs = keras.Input(shape=(None, num_features))
outputs = layers.SimpleRNN(16)(inputs)
```

- **Stacking More Layers**

```
inputs = keras.Input(shape=(steps, num_features))
x = layers.SimpleRNN(16, return_sequences=True)(inputs)
x = layers.SimpleRNN(16, return_sequences=True)(x)
outputs = layers.SimpleRNN(16)(x)
```

- Input: x(t) is taken as the input to the network at time step t. For example, x1,could be a one-hot vector corresponding to a word of a sentence.
- Hidden state: h(t) represents a hidden state at time t and acts as “memory” of the network. h(t) is calculated based on the current input and the previous time step’s hidden state: $h(t) = f(U\,x(t) + W\,h(t-1))$. The function f is taken to be a non-linear transformation such as tanh, ReLU.
- Weights: The RNN has input to hidden connections parameterized by a weight matrix U, hidden-to-hidden recurrent connections parameterized by a weight matrix W, and hidden-to-output connections parameterized by a weight matrix V and all these weights (U,V,W) are shared across time.
- Output: o(t) illustrates the output of the network. In the figure I just put an arrow after o(t) which is also often subjected to non-linearity, especially when the network contains further layers downstream.

- Forward Pass
-
- The figure does not specify the choice of activation function for the hidden units. Before we proceed we make few assumptions: 1) we assume the hyperbolic tangent activation function for hidden layer. 2) We assume that the output is discrete, as if the RNN is used to predict words or characters. A natural way to represent discrete variables is to regard the output o as giving the un-normalized log probabilities of each possible value of the discrete variable. We can then apply the softmax operation as a post-processing step to obtain a vector ŷof normalized probabilities over the output.
-
- The RNN forward pass can thus be represented by below set of equations.
-
- This is an example of a recurrent network that maps an input sequence to an output sequence of the same length. The total loss for a given sequence of x values paired with a sequence of y values would then be just the sum of the losses over all the time steps. We assume that the outputs o(t)are used as the argument to the softmax function to obtain the vector ŷ of probabilities over the output. We also assume that the loss L is the negative log-likelihood of the true target y(t)given the input so far.
- Backward Pass
-
- The gradient computation involves performing a forward propagation pass moving left to right through the graph shown above followed by a backward propagation pass moving right to left through the graph. The runtime is O(τ) and cannot be reduced by parallelization because the forward propagation graph is inherently sequential; each time step may be computed only after the previous one. States computed in the forward pass must be stored until they are reused during the backward pass, so the memory cost is also O(τ). The back-propagation algorithm applied to the unrolled graph with O(τ) cost is called back-propagation through time (BPTT). Because the parameters are shared by all time steps in the network, the gradient at each output depends not only on the calculations of the current time step, but also the previous time steps.
-

UCERD

- Computing Gradients
- Given our loss function L, we need to calculate the gradients for our three weight matrices U, V, W, and bias terms b, c and update them with a learning rate α. Similar to normal back-propagation, the gradient gives us a sense of how the loss is changing with respect to each weight parameter. We update the weights W to minimize loss with the following equation:

- The same is to be done for the other weights U, V, b, c as well.

- Let us now compute the gradients by BPTT for the RNN equations above. The nodes of our computational graph include the parameters U, V, W, b and c as well as the sequence of nodes indexed by t for x (t), h(t), o(t) and L(t). For each node n we need to compute the gradient ∇nL recursively, based on the gradient computed at nodes that follow it in the graph.

- Gradient with respect to output o(t) is calculated assuming the o(t) are used as the argument to the softmax function to obtain the vector ŷ of probabilities over the output. We also assume that the loss is the negative log-likelihood of the true target y(t).

```
from tensorflow import keras
from tensorflow.keras import layers
import os
import numpy as np
fname = os.path.join("jena_climate_2009_2016.csv")
with open(fname) as f:
    data = f.read()
lines = data.split("\n")
header = lines[0].split(",")
lines = lines[1:]
temperature = np.zeros((len(lines),))
raw_data = np.zeros((len(lines), len(header) - 1))
sequence_length = 120
sampling_rate = 6
sequence_length = 120
delay = sampling_rate * (sequence_length + 24 - 1)
batch_size = 256
num_train_samples=210225
num_val_samples=105112
num_test_samples=105114
```

Dataset Preparation

```
train_dataset =
keras.utils.timeseries_dataset_from
_array(

raw_data[:-delay],

targets=temperature[delay:],

sampling_rate=sampling_rate,

sequence_length=sequence_length
, shuffle=True,
batch_size=batch_size,
start_index=0,
end_index=num_train_samples)
```

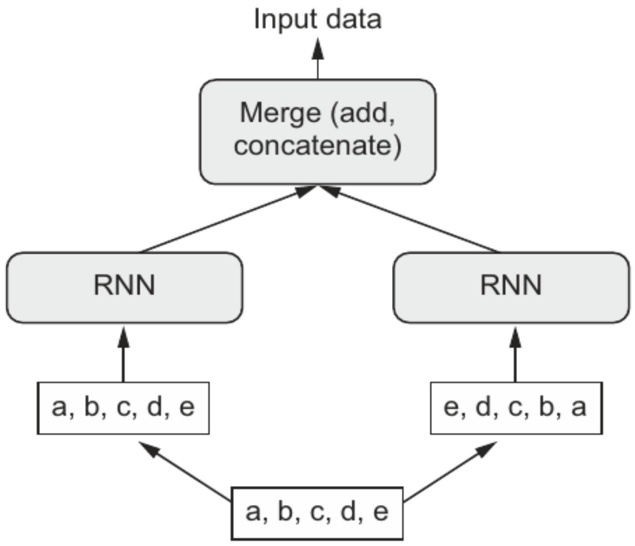
```
val_dataset =
keras.utils.timeseries_dataset_from
_array(

raw_data[:-delay],

targets=temperature[delay:],

sampling_rate=sampling_rate,

sequence_length=sequence_length,

shuffle=True,

batch_size=batch_size,

start_index=num_train_samples,

end_index=num_train_samples +
num_val_samples)
```

```
test_dataset =
keras.utils.timeseries_dataset_from
_array(

raw_data[:-delay],

targets=temperature[delay:],

sampling_rate=sampling_rate,

sequence_length=sequence_length,

shuffle=True,

batch_size=batch_size,

start_index=num_train_samples +
num_val_samples)
```

UCERD

Bidirectional RNN

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Bidirectional(layers.LSTM(16))(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
epochs=1,
validation_data=val_dataset)
```


Input data
Merge (add, concatenate)
RNN
RNN
a, b, c, d, e
e, d, c, b, a
a, b, c, d, e