Artificial Intelligence

Recurrent Neural Networks

Agenda

01 Issues with Feed Forward Network

02 Understanding Recurrent Neural Networks

03 Types of RNN

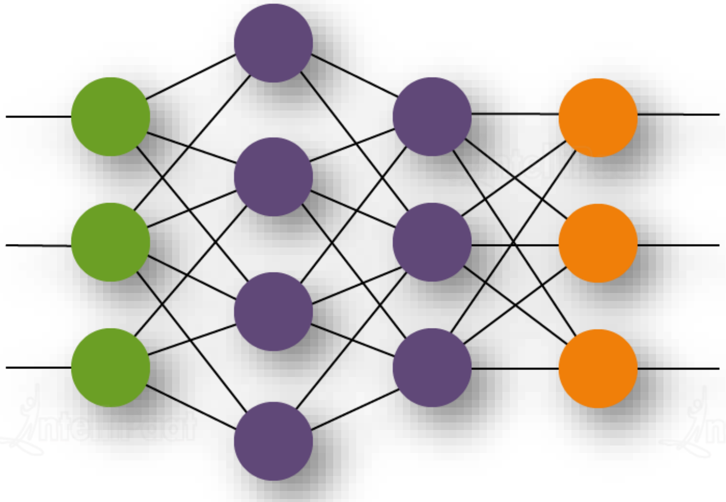
04 Issues with RNN

05 Vanishing Gradient Problem

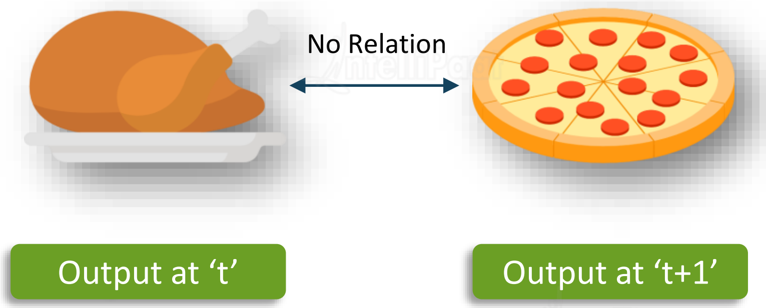
06 Long Short Term Networks

07 Demo on LSTM with Keras

Issues with Feed Forward Network

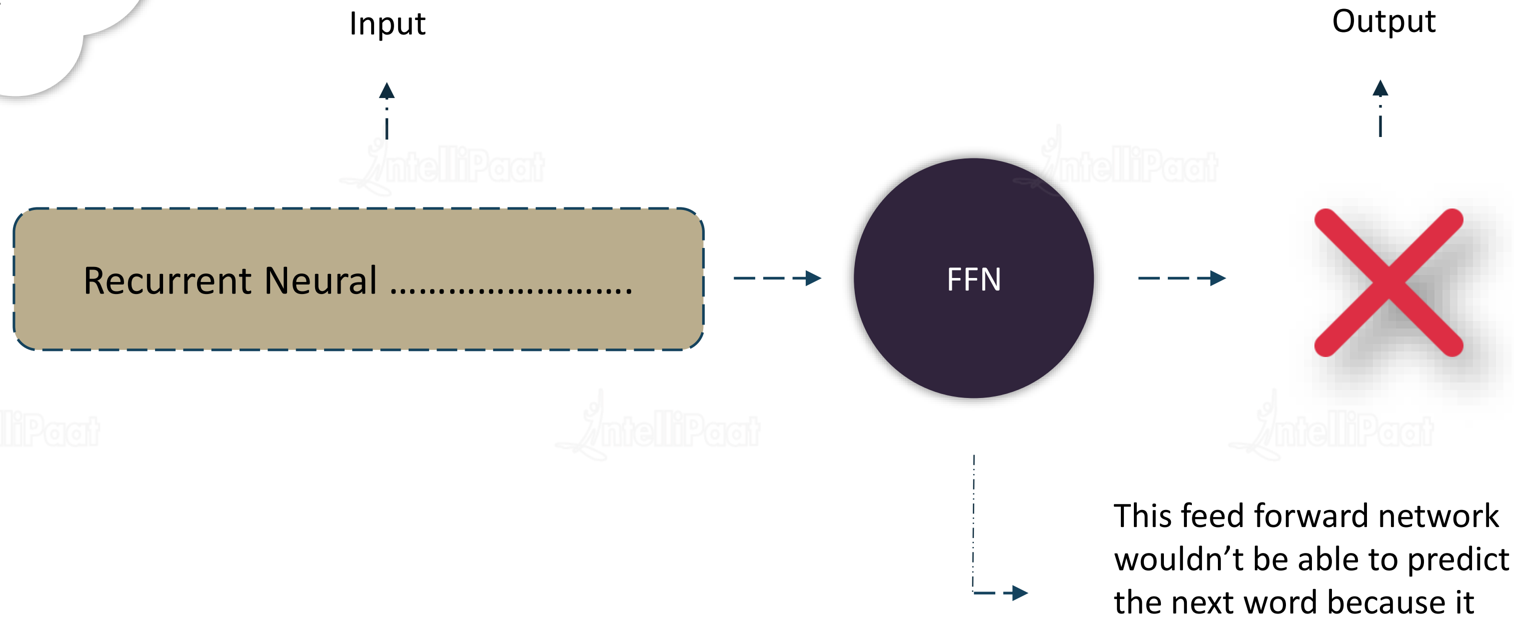


Feed Forward Network



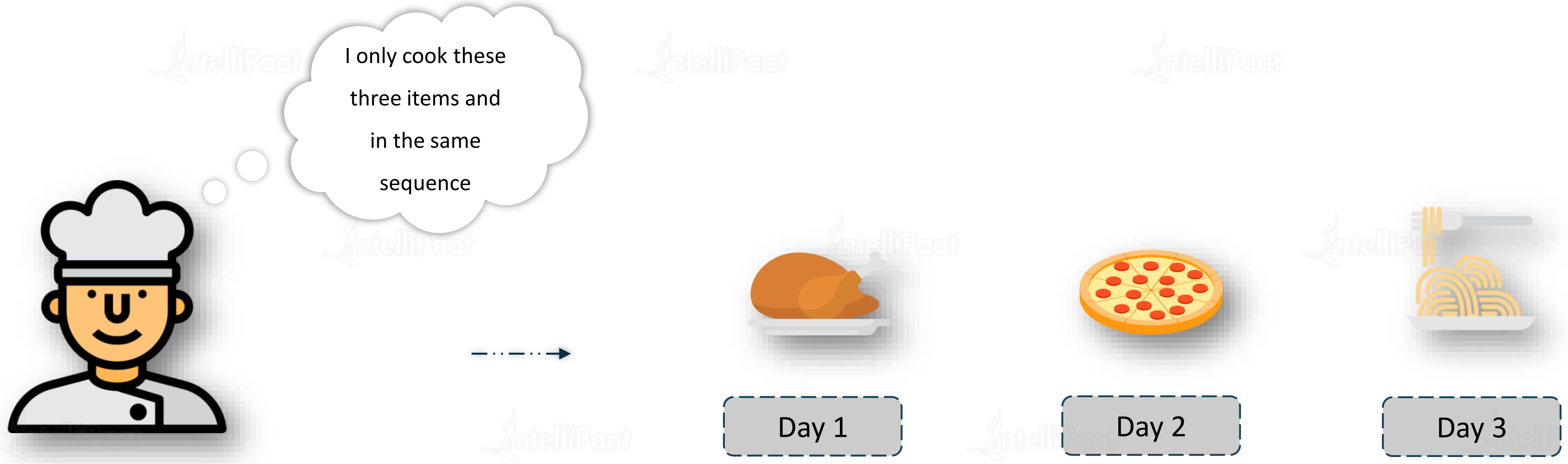
- Outputs are independent of each other
- Cannot handle sequential data
- Cannot memorize previous inputs

Issues with Feed Forward Network

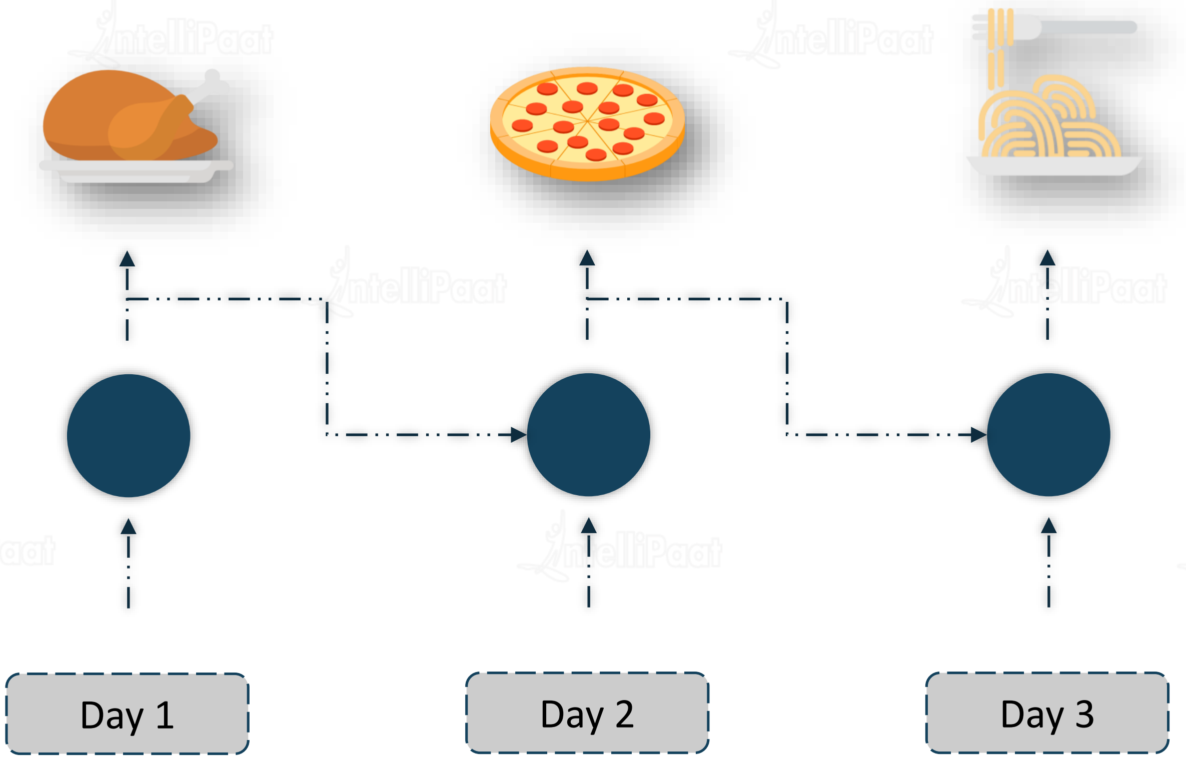


This feed forward network wouldn't be able to predict the next word because it cannot memorize the previous inputs

Solution with Recurrent Neural Network



Solution with Recurrent Neural Network

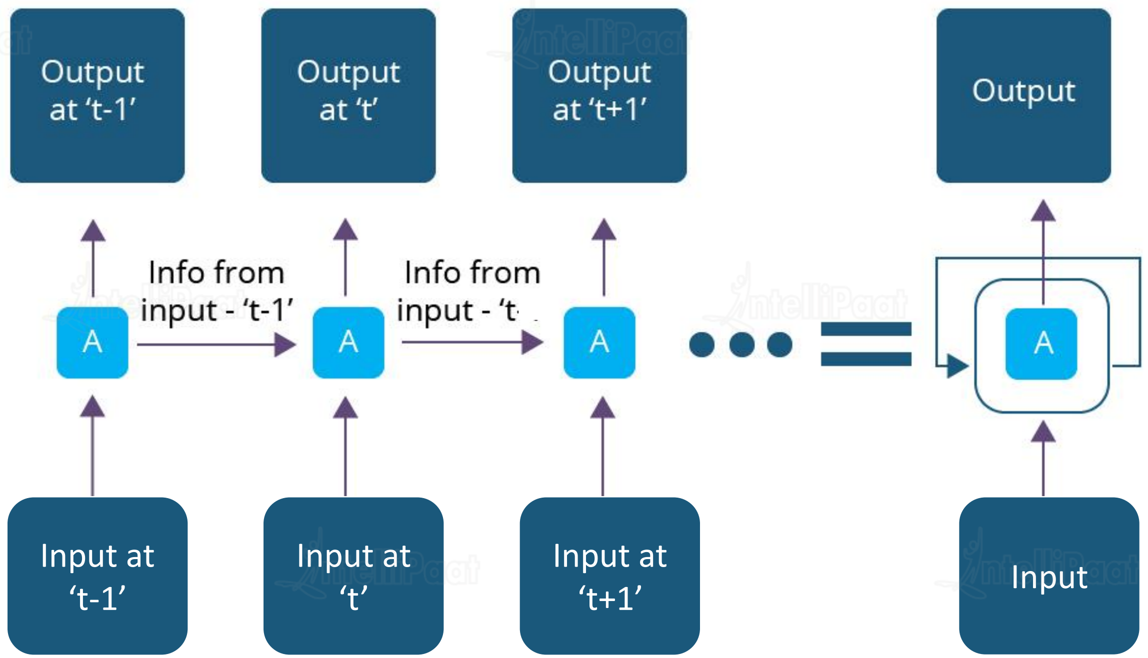


Recurrent Neural Network

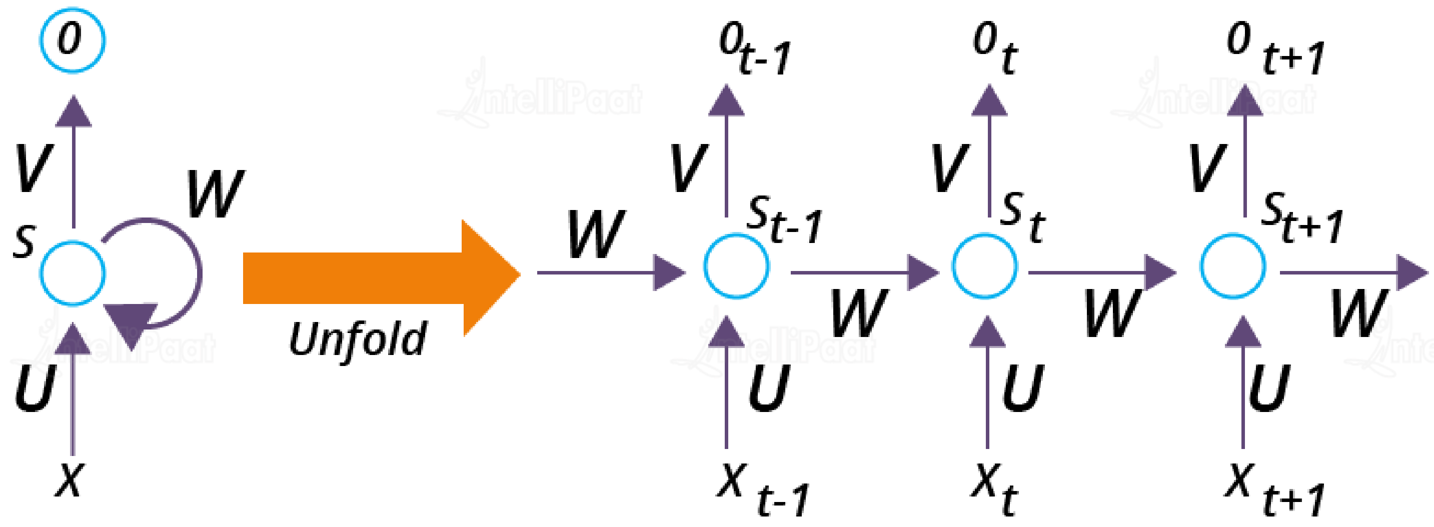
- Outputs are dependent on each other
- Can handle sequential data
- Can memorize previous inputs

Understanding Recurrent Neural Networks

- RNNs are called *recurrent* because they perform the same task for every element of a sequence, with the output being dependent 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



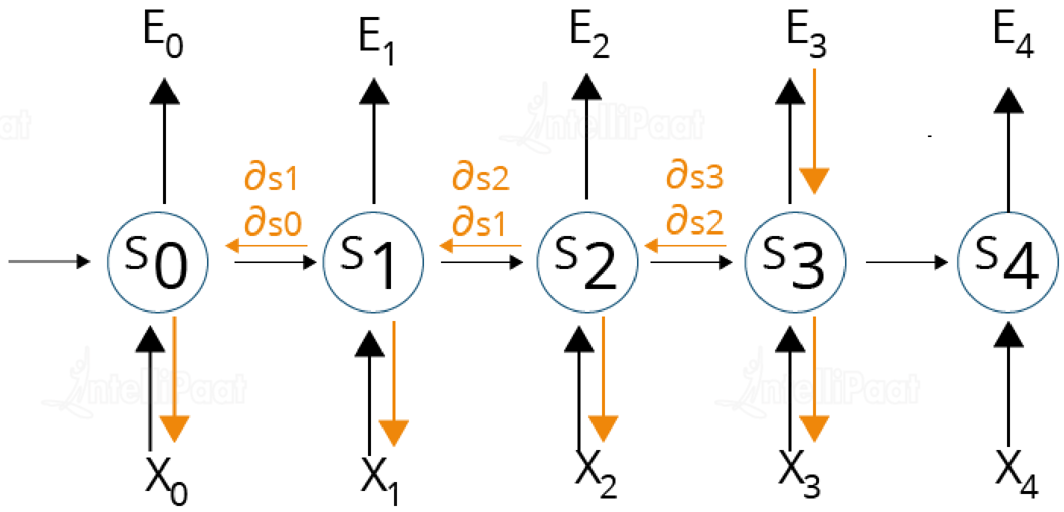
Understanding Recurrent Neural Networks



- X_t is the input at time step '*t*'
- S_t is the hidden state at time step '*t*'. It's the memory of the network. S_t is calculated based on the previous hidden state and the input at the current step: $s_t = f(Ux_t + Ws_{t-1})$. The function f is usually a non-linearity such as tanh or ReLu.
- O_t is the output at step '*t*'. $O_t = softmax(Vs_t)$

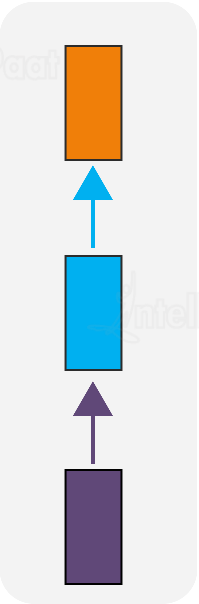
Back-Propagation through Time

- Backpropagation Through Time (BPTT) is used to update the weights in the recurrent neural network
- RNN typically predicts one output per each time step. Conceptually, Backpropagation through Time works by unrolling the network to get each of these individual time steps.
- Then, it calculates the error across each time step and adds up all of the individual errors to get the final accumulated error.
- Following which the network is rolled back up and the weights are updated



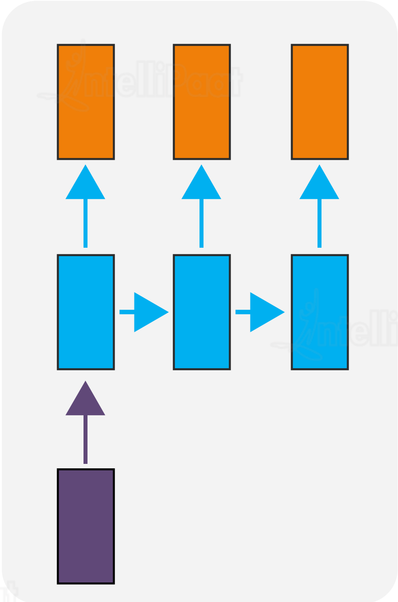
Types of RNN

One to one



single images (or words,...) are classified in single class (binary classification) i.e. is this a bird or not

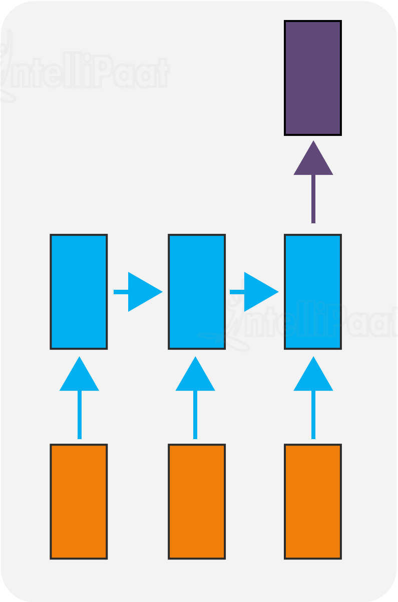
One to many



single images (or words,...) are classified in multiple classes

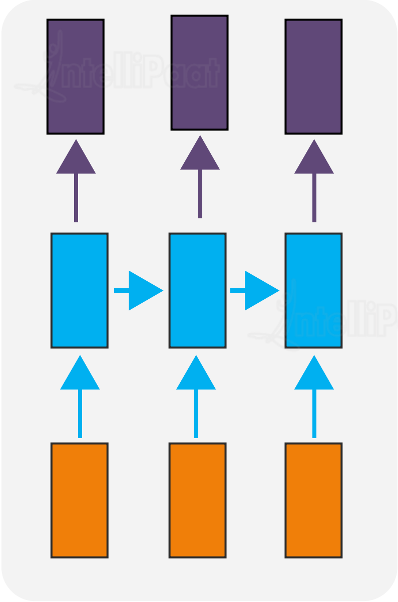
Types of RNN

Many to one



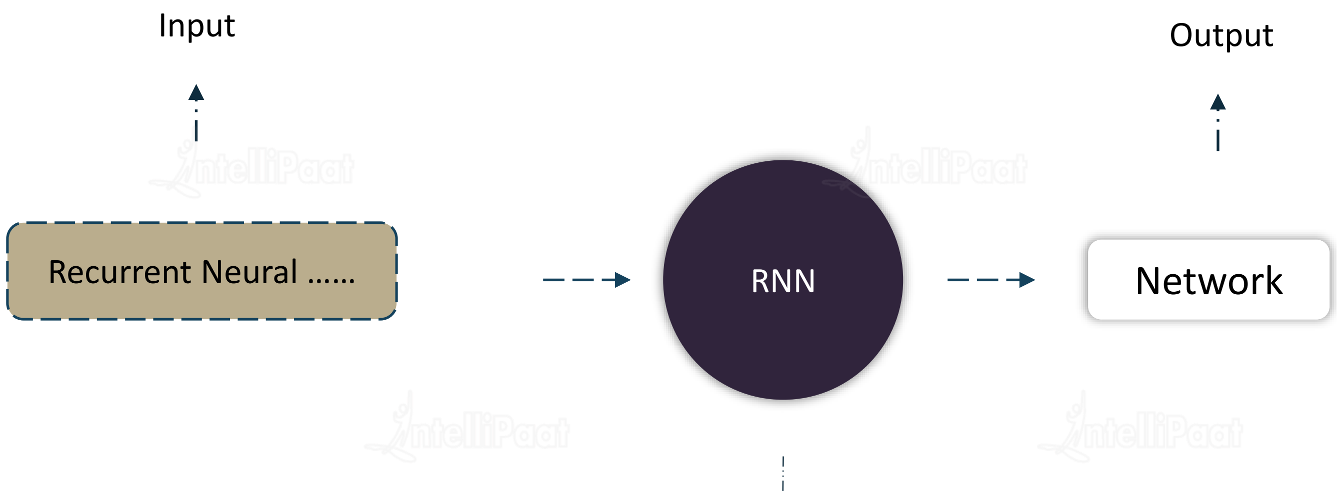
sequence of images (or words, ...) is classified in single class (binary classification of a sequence)

Many to many



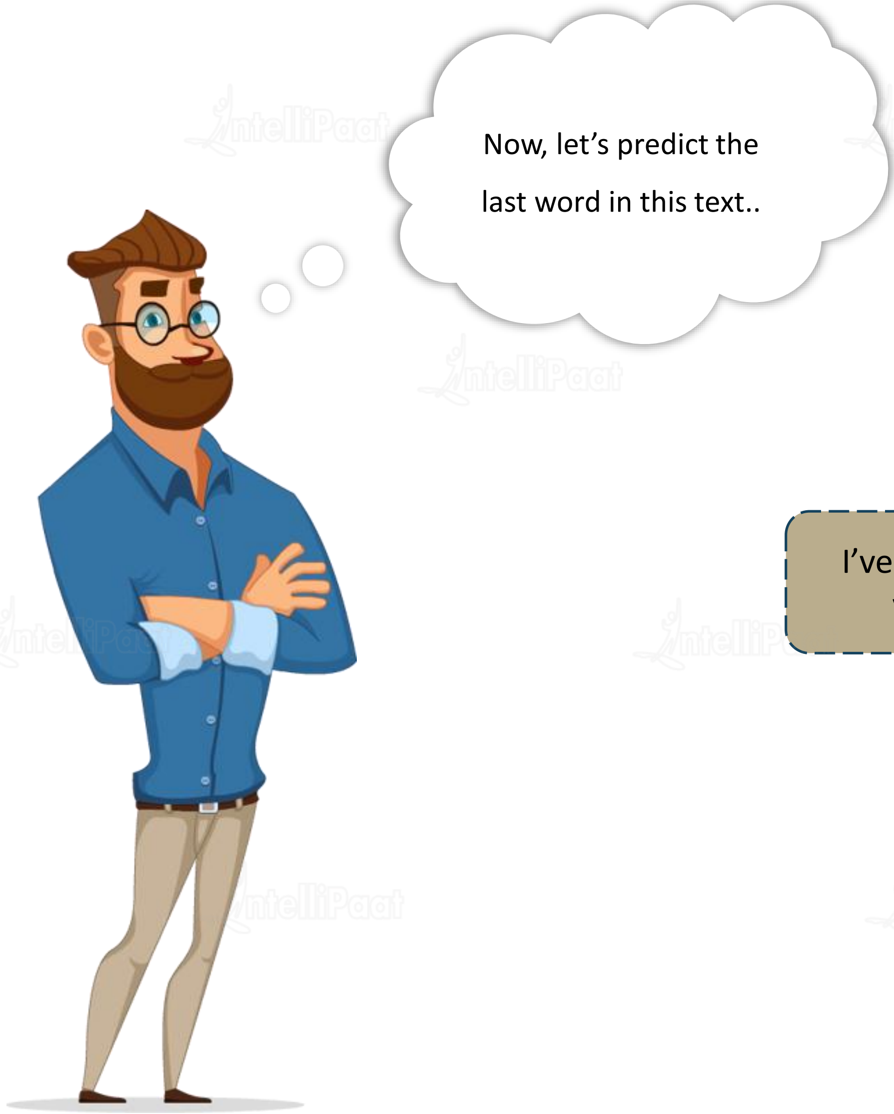
sequence of images (or words, ...) is classified in multiple classes

Issues with RNN



Here, the RNN does not need any further context. It can easily predict that the last word would be 'Network'

Issues with RNN



Input

I’ve been staying in Spain for the last 10 years. I can speak fluent …………..

RNN

Output

?

Regular RNN’s have difficulty in learning long range dependencies

Issues with RNN

I’ve been staying in Spain for the last 10 years. I can speak fluent ....

- In this case, the network needs the context of ‘Spain’ to predict the last word in this text, which is “Spanish”
- The gap between the word which we want to predict and the relevant information is very large and this is known as long term dependency

$$\partial E/\partial W = \partial E/\partial y3 * \partial y3/\partial h3 * \partial h3/\partial y2 * \partial y2/\partial h1 \ldots$$

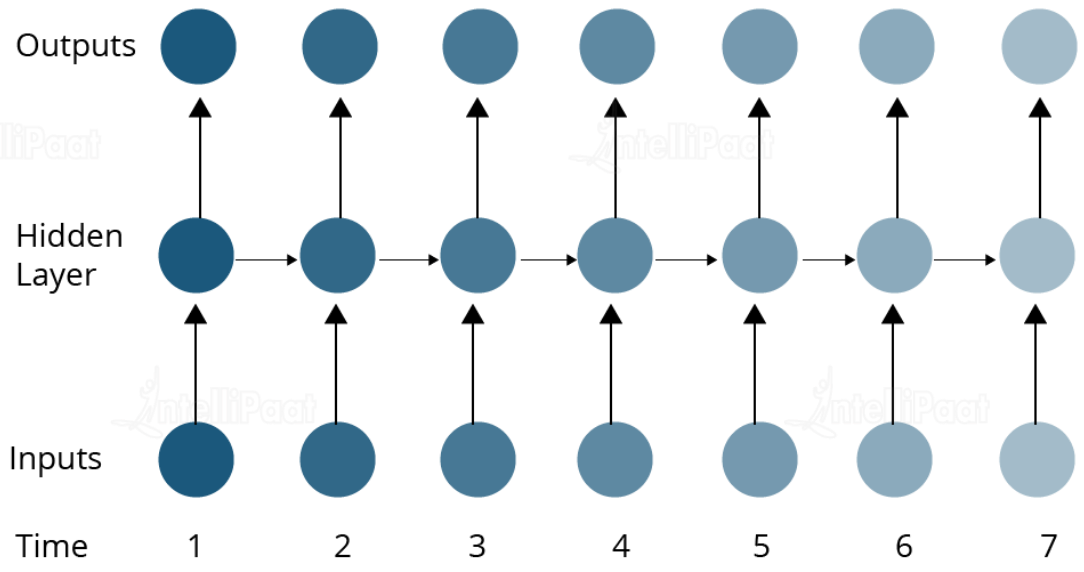
- There arises a long dependency while backpropagating the error

Vanishing Gradient Problem

- Now, if there is a really long dependency, there’s a good probability that one of the gradients might approach zero and this would lead to all the gradients rushing to zero exponentially fast due to multiplication

$$\partial E/\partial W=0$$

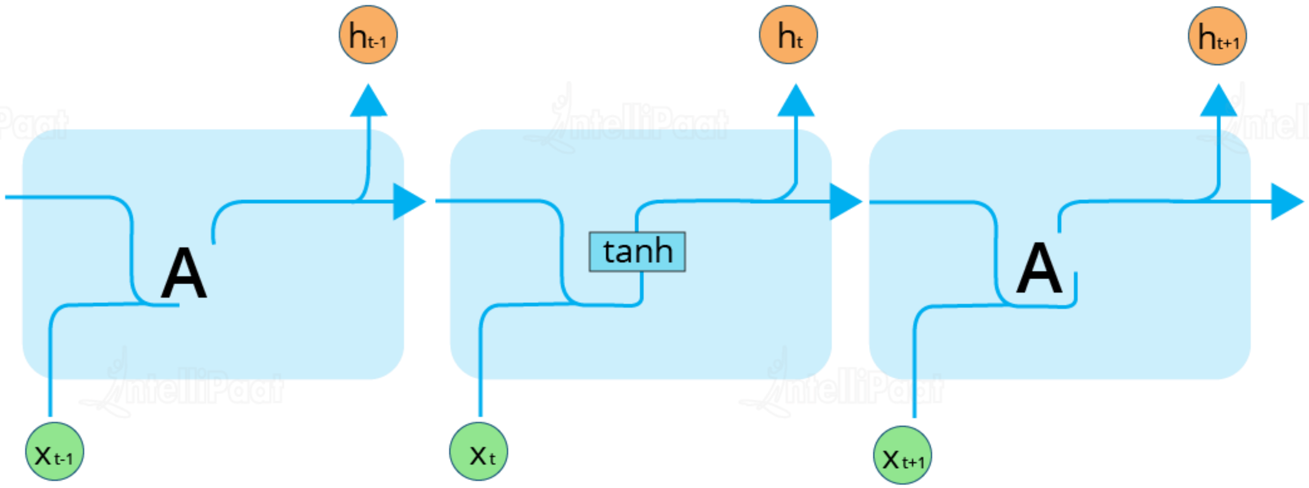
- Such states would no longer help the network to learn anything. This is known as vanishing gradient problem



Long Short Term Networks

Long Short Term Networks are special kind of RNNs which are explicitly designed to avoid the long-term dependency problem

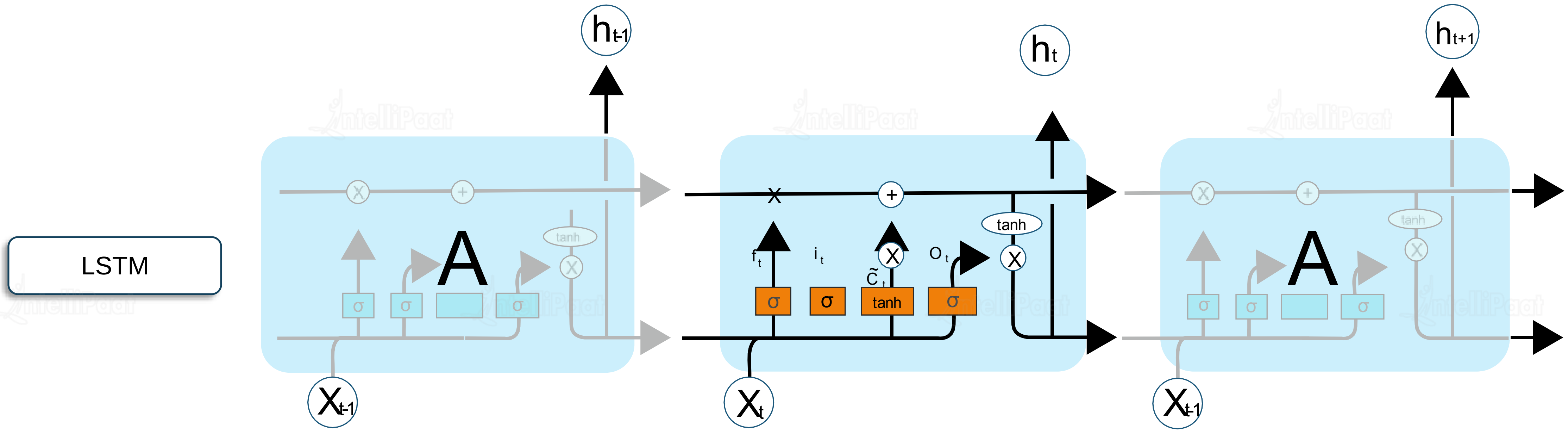
Standard RNN



All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer

Long Short Term Networks

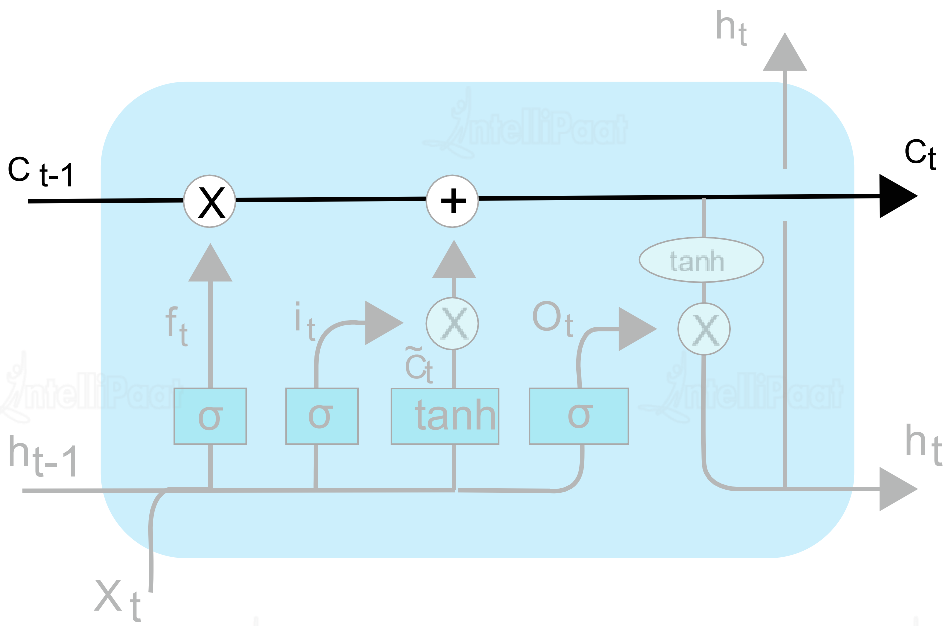
Long Short Term Networks are special kind of RNNs which are explicitly designed to avoid the long-term dependency problem



LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way

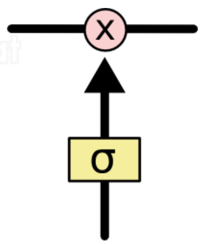
Core Idea behind LSTMs

The key to LSTMs is the cell state. The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It’s very easy for information to just flow along it unchanged



The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates

Core Idea behind LSTMs



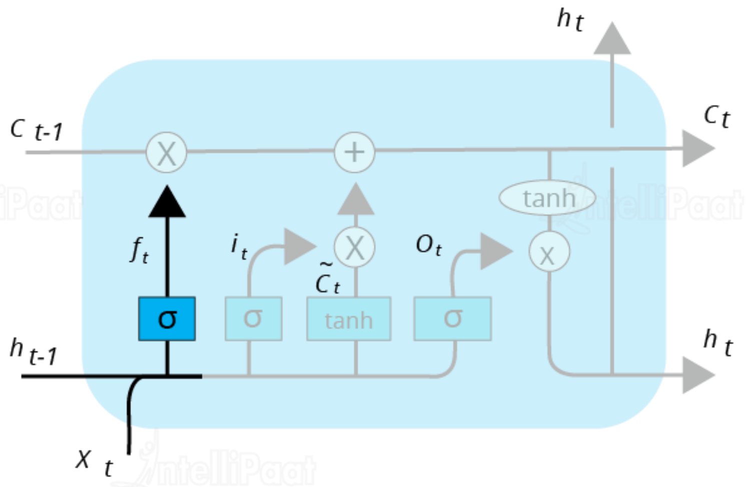
Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation

The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means “let nothing through,” while a value of one means “let everything through!”

Working of LSTMs

Step 1

The first step in our LSTM is to decide what information we're going to throw away from the cell state. This decision is made by a sigmoid layer called the "forget gate layer"

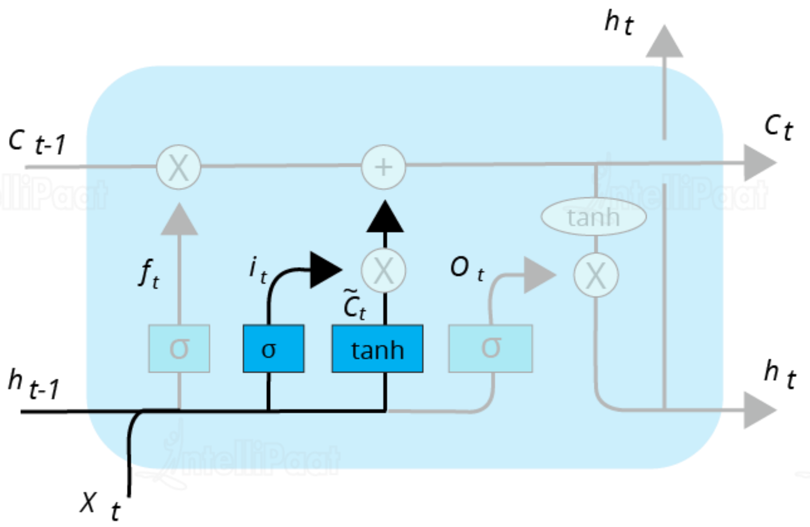


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Working of LSTMs

Step 2

The next step is to decide what new information we're going to store in the cell state. This has two parts. First, a sigmoid layer called the "input gate layer" decides which values we'll update. Next, a tanh layer creates a vector of new candidate values, that could be added to the state



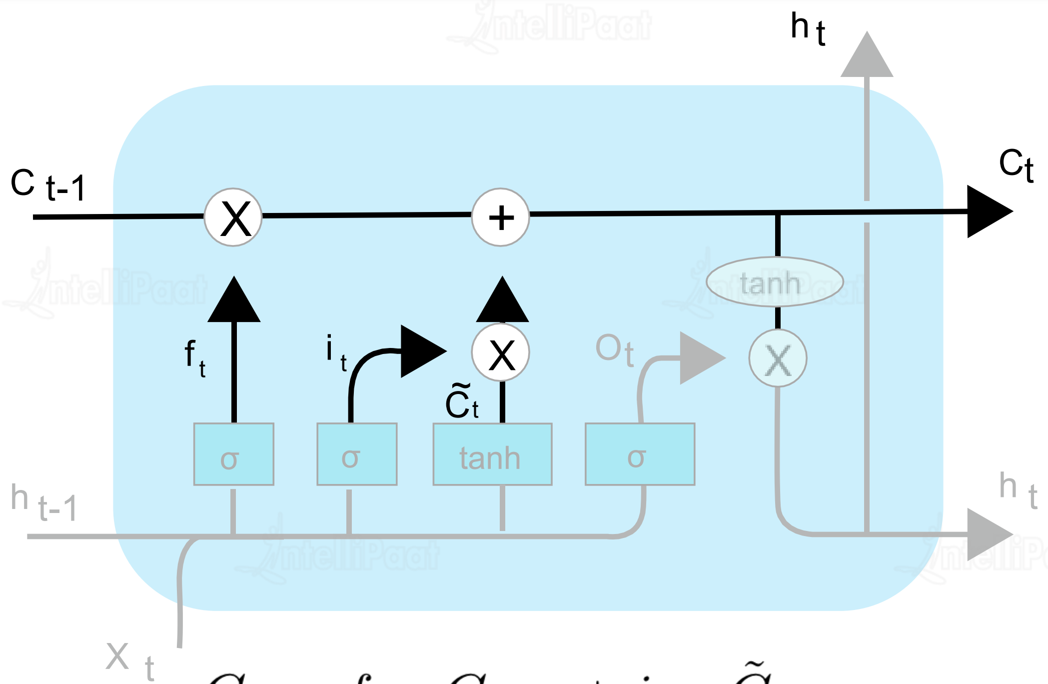
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Working of LSTMs

Step 3

Then we have to update the old cell state, C_{t-1}, into new cell state C_t. So, we multiply the old state (C_{t-1}) by f_t, forgetting the things we decided to forget earlier. Then we add (i_t * $C\sim_t$). This is the new candidate values, scaled by how much we decided to update each state value

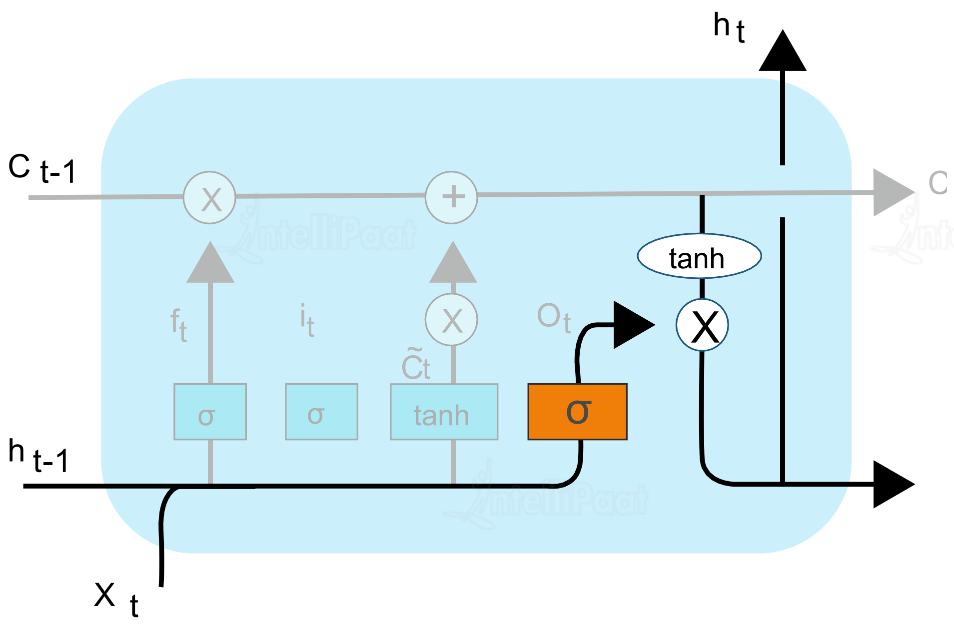


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Working of LSTMs

Step 4

Finally, we'll run a sigmoid layer which decides what part of the cell state we're going to output. Then, we put the cell state through tanh and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to



$$o_t = \sigma \left(W_o \left[h_{t-1}, x_t \right] + b_o \right)$$

$$h_t = o_t * \tanh \left(C_t \right)$$

Implementing a Simple RNN

Loading the required packages:

1

```
In [1]: from keras.models import Sequential
        from keras.layers import Dense
        from keras.layers import LSTM
        from sklearn.model_selection import train_test_split
        import numpy as np
        import matplotlib.pyplot as plt
```

Preparing the input data:

2

```
In [2]: #Data preparation
        Data = [[[i+j] for i in range(5)] for j in range(100)]
        Data[:5]
```

```
Out[2]: [[[0], [1], [2], [3], [4]],
         [[1], [2], [3], [4], [5]],
         [[2], [3], [4], [5], [6]],
         [[3], [4], [5], [6], [7]],
         [[4], [5], [6], [7], [8]]]
```

Creating 100 vectors with 5 consecutive numbers

Implementing a Simple RNN

Preparing the output data:

3

```
In [2]: Target = [(i+5) for i in range(100)]
        Target[:5]
```

```
Out[2]: [5, 6, 7, 8, 9]
```

Converting the data & target into numpy arrays:

4

```
In [5]: data = np.array(Data,dtype=float)
        target = np.array(Target,dtype=float)
```

Having a glance at the shape:

5

```
In [6]: data.shape, target.shape
```

```
Out[6]: ((100, 5, 1), (100,))
```

Implementing a Simple RNN

Dividing the data into train & test sets:

6

```
In [10]: #Dividing data into train & test
         x_train, x_test, y_train, y_test = train_test_split(data,target,test_size=0.2,random_state=4)
```

Creating a sequential model:

7

```
In [11]: #RNN
         model = Sequential()
```

Adding the LSTM layer with the output and input shape:

8

```
In [12]: model.add(LSTM((1),batch_input_shape=(None,5,1),return_sequences=False))
```

Implementing a Simple RNN

Compiling the model with 'Adam' optimizer:

9

```
In [13]: model.compile(loss='mean_absolute_error',optimizer='adam',metrics=['accuracy'])
```

Having a glance at the model summary:

10

```
In [14]: model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 1)                 12
=================================================================
Total params: 12
Trainable params: 12
Non-trainable params: 0
_________________________________________________________________
```

Implementing a Simple RNN

Fitting a model on the train set:

11

```
In [15]: history = model.fit(x_train,y_train,epochs=50,validation_data=(x_test,y_test))
```

```
Train on 80 samples, validate on 20 samples
Epoch 1/50
80/80 [==============================] - 1s 8ms/step - loss: 56.6895 - acc: 0.0000e+00 - val_loss: 45.7512 - val_acc: 0.0000e
+00
Epoch 2/50
80/80 [==============================] - 0s 263us/step - loss: 56.6895 - acc: 0.0000e+00 - val_loss: 45.7511 - val_acc: 0.000
0e+00
Epoch 3/50
80/80 [==============================] - 0s 300us/step - loss: 56.6894 - acc: 0.0000e+00 - val_loss: 45.7511 - val_acc: 0.000
0e+00
Epoch 4/50
80/80 [==============================] - 0s 275us/step - loss: 56.6894 - acc: 0.0000e+00 - val_loss: 45.7511 - val_acc: 0.000
0e+00
```

Implementing a Simple RNN

Predicting the values on the test set:

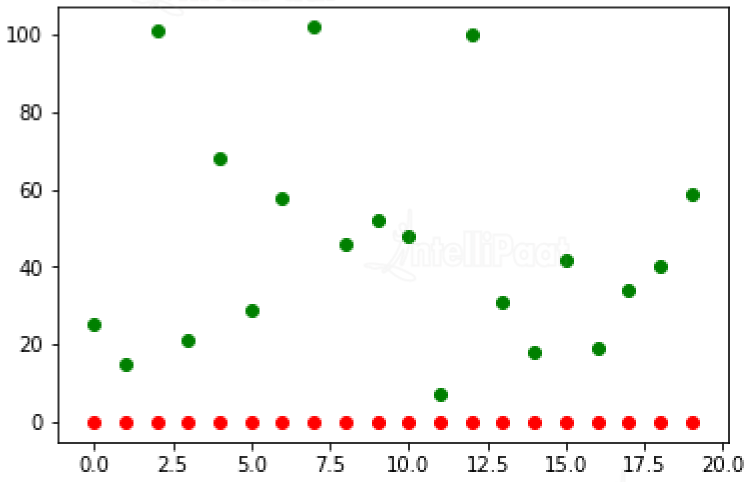
12

```
In [ ]: results = model.predict(x_test)
```

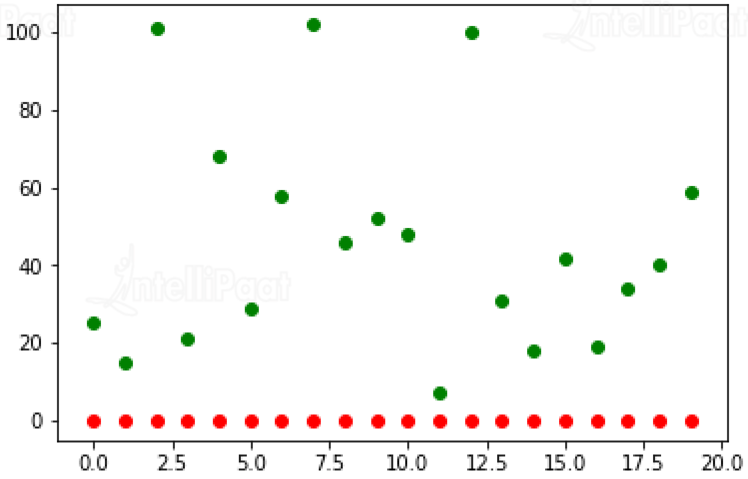
Making a scatter plot for actual values and predicted values:

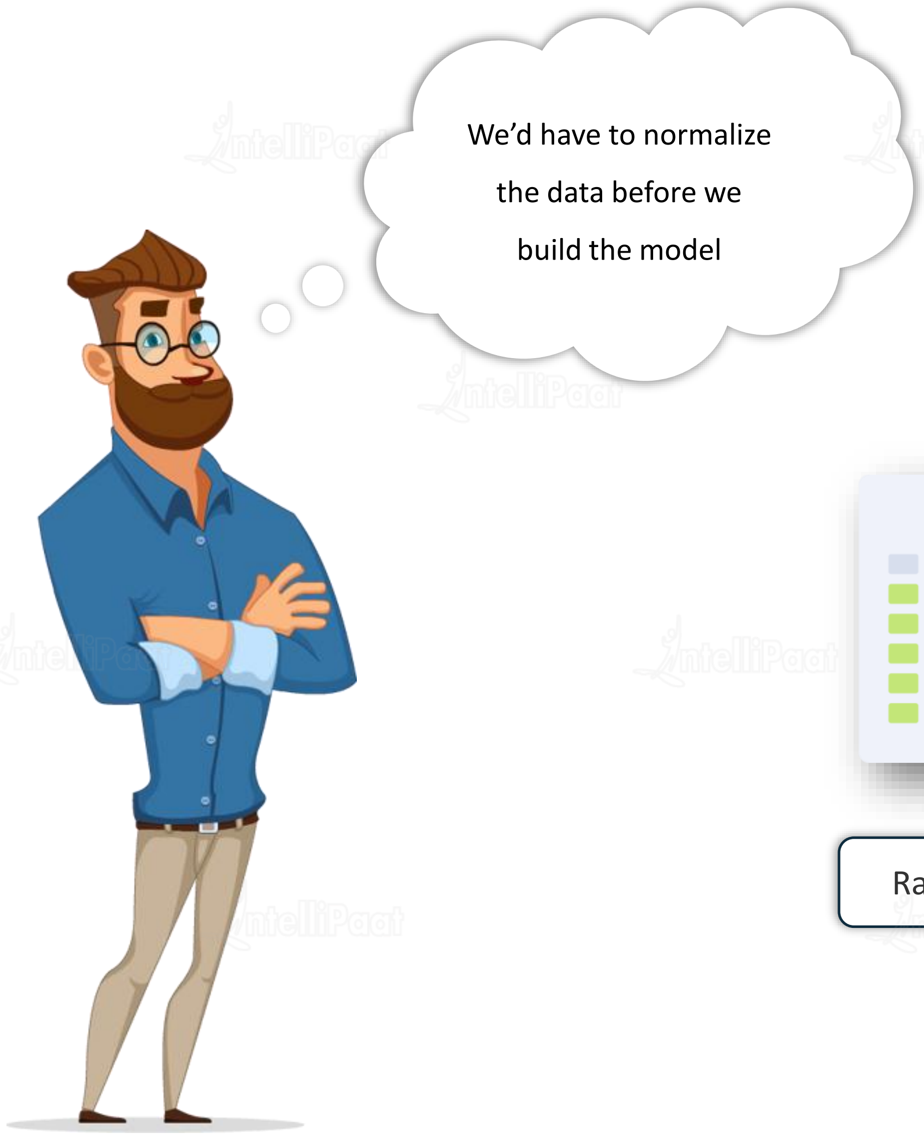
13

```
In [20]: plt.scatter(range(20),results,c='r')
         plt.scatter(range(20),y_test,c='g')
         plt.show()
```

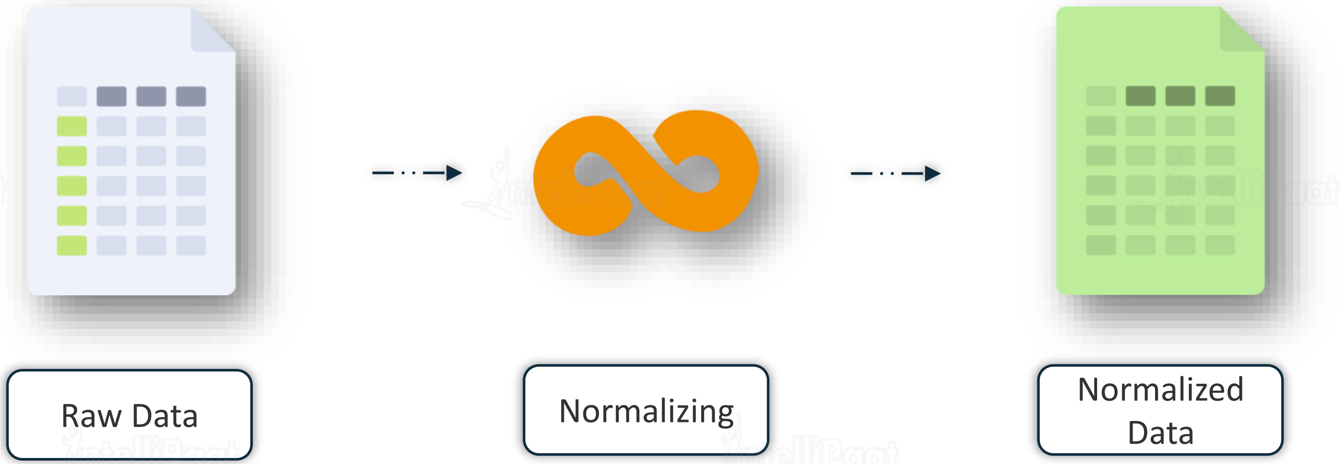


We see that the model fails miserably and none of the predictions are correct





We’d have to normalize
the data before we
build the model



Raw Data
Normalizing
Normalized
Data

Implementing a Simple RNN

Normalizing the input data:

14

```
In [22]: Data = [[[(i+j)/100] for i in range(5)] for j in range(100)]
         Data[:5]
```

```
Out[22]: [[[0.0], [0.01], [0.02], [0.03], [0.04]],
          [[0.01], [0.02], [0.03], [0.04], [0.05]],
          [[0.02], [0.03], [0.04], [0.05], [0.06]],
          [[0.03], [0.04], [0.05], [0.06], [0.07]],
          [[0.04], [0.05], [0.06], [0.07], [0.08]]]
```

Normalizing the output data:

15

```
In [23]: Target = [(i+5)/100 for i in range(100)]
         Target[:5]
```

```
Out[23]: [0.05, 0.06, 0.07, 0.08, 0.09]
```

Fitting the model with normalized values and number of epochs to be 500:

16

```
history = model.fit(x_train,y_train,epochs=500,validation_data=(x_test,y_test))
```

Implementing a Simple RNN

Predicting the values on test set:

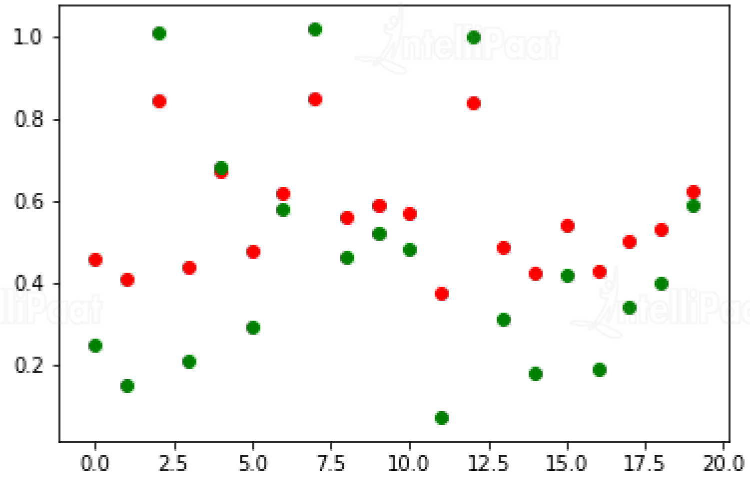
17

```
In [50]: results = model.predict(x_test)
```

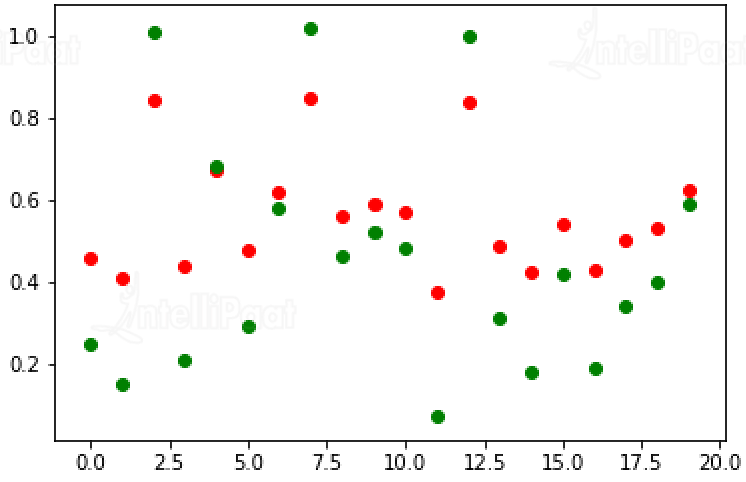
Making a scatter plot for actual values & predicted values:

18

```
In [51]: plt.scatter(range(20),results,c='r')
         plt.scatter(range(20),y_test,c='g')
         plt.show()
```




We see that the loss has reduced after normalizing the data and increasing the epochs


0.2
0.4
0.6
0.8
1.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

Quiz

Quiz 1

Gated Recurrent units can help prevent vanishing gradient problem in RNN.

A	True
B	False

Answer 1

Gated Recurrent units can help prevent vanishing gradient problem in RNN.

A True

B False

Quiz 2

How many types of RNN exist?

A	4
B	2
C	3
D	None of these

Answer 2

How many types of RNN exist?

A	4
B	2
C	3
D	None of these

Quiz 3

How many gates are there in LSTM?

A	1
B	2
C	3
D	4

Answer 3

How many gates are there in LSTM?

A	1
B	2
C	3
D	4

India: +91-7847955955

US: 1-800-216-8930 (TOLL FREE)

sales@intellipaat.com

24/7 Chat with Our Course Advisor