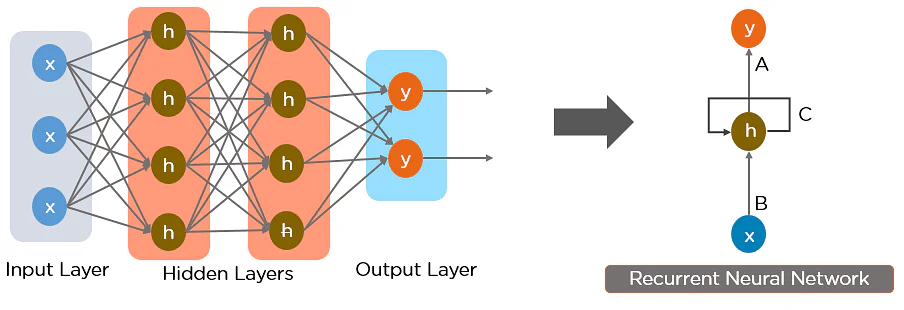
**AIDS Lab**

**EXPERIMENT NO. 10**

**Aim**: Implementation of supervised learning algorithm i.e. RNN.

**Theory**:

RNN works on the principle of saving the output of a particular layer and feeding this back to the input in order to predict the output of the layer. Below is how you can convert a Feed-Forward Neural Network into a Recurrent Neural Network:

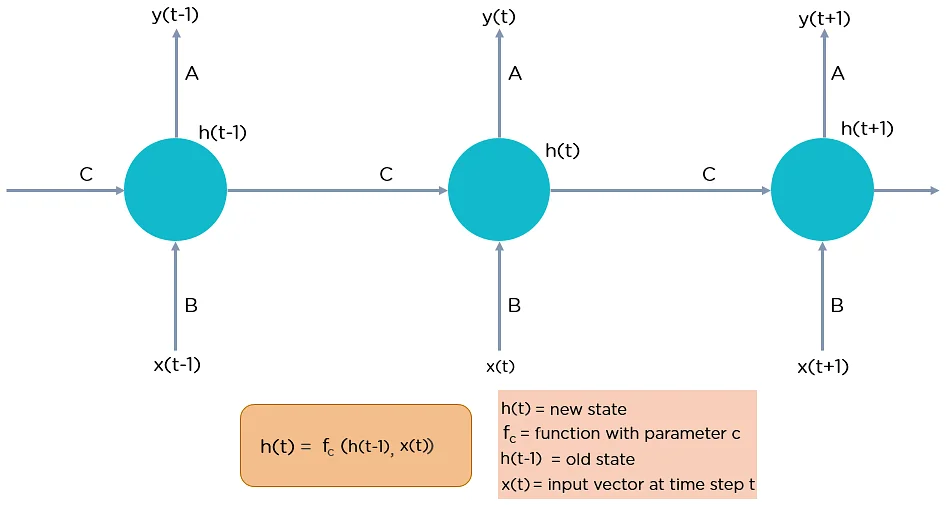


Here, “x” is the input layer, “h” is the hidden layer, and “y” is the output layer. A, B, and C are the network parameters used to improve the output of the model. At any given time t, the current input is a combination of input at x(t) and x(t-1). The output at any given time is fetched back to the network to improve on the output.

RNN were created because there were a few issues in the feed-forward neural network:

1. Cannot handle sequential data
2. Considers only the current input
3. Cannot memorize previous inputs

The solution to these issues is the RNN. An RNN can handle sequential data, accepting the current input data, and previously received inputs. RNNs can memorize previous inputs due to their internal memory. In Recurrent Neural networks, the information cycles through a loop to the middle hidden layer.



The input layer ‘x’ takes in the input to the neural network and processes it and passes it onto the middle layer.

The middle layer ‘h’ can consist of multiple hidden layers, each with its own activation functions and weights and biases. If you have a neural network where the various parameters of different hidden layers are not affected by the previous layer, ie: the neural network does not have memory, then you can use a recurrent neural network.

The Recurrent Neural Network will standardize the different activation functions and weights and biases so that each hidden layer has the same parameters. Then, instead of creating multiple hidden layers, it will create one and loop over it as many times as required.

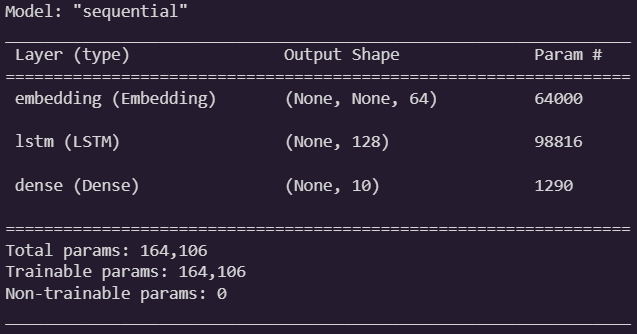
**Code and Output**:

Recurrent neural networks (RNN) are a class of neural networks that is powerful for modelling sequence data such as time series or natural language. Schematically, a RNN layer uses a for loop to iterate over the timesteps of a sequence, while maintaining an internal state that encodes information about the timesteps it has seen so far.

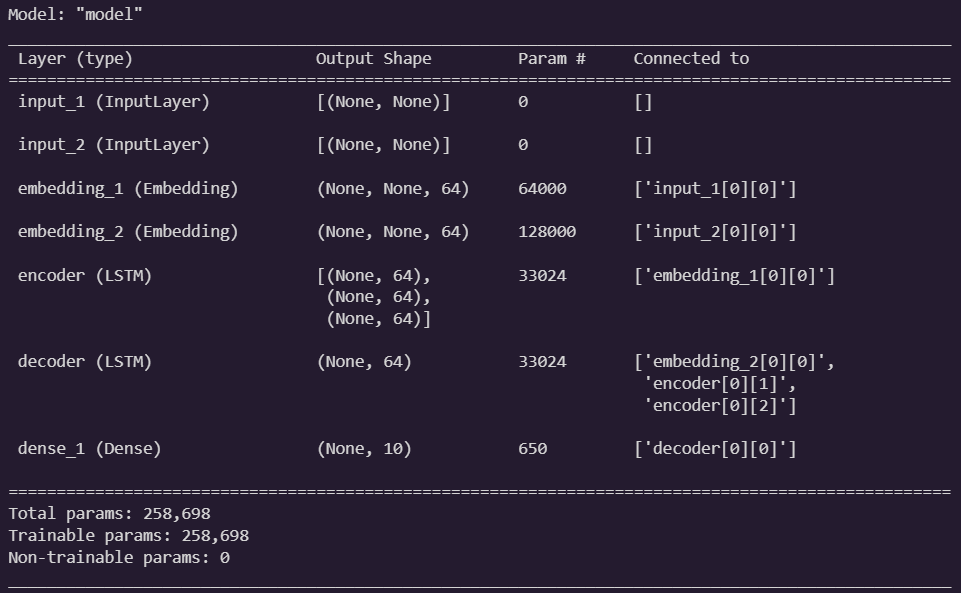
The Keras RNN API is designed with a focus on Ease of use as the built-in keras.layers.RNN, keras.layers.LSTM, keras.layers.GRU layers enable you to quickly build recurrent models without having to make difficult configuration choices. It also focuses on Ease of customization. You can also define your own RNN cell layer (the inner part of the for loop) with custom behaviour, and use it with the generic keras.layers.RNN layer (the for loop itself).

| import numpy as np import tensorflow as tf from tensorflow import keras from tensorflow.keras import layers |
| --- |

| model = keras.Sequential() *# Add an Embedding layer expecting input vocab of size 1000, and output embedding dimension of size 64.* model.add(layers.Embedding(input\_dim=1000, output\_dim=64)) *# Add a LSTM layer with 128 internal units.* model.add(layers.LSTM(128)) *# Add a Dense layer with 10 units.* model.add(layers.Dense(10)) model.summary() |
| --- |



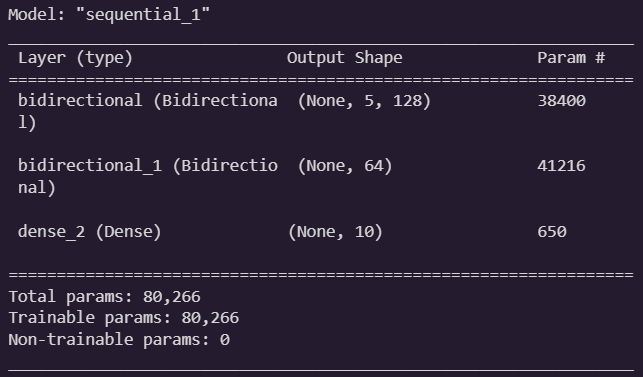
| encoder\_vocab = 1000 decoder\_vocab = 2000 encoder\_input = layers.Input(shape=(None,)) encoder\_embedded = layers.Embedding(input\_dim=encoder\_vocab, output\_dim=64)(  encoder\_input ) *# Return states in addition to output* output, state\_h, state\_c = layers.LSTM(64, return\_state=True, name="encoder")(  encoder\_embedded ) encoder\_state = [state\_h, state\_c] decoder\_input = layers.Input(shape=(None,)) decoder\_embedded = layers.Embedding(input\_dim=decoder\_vocab, output\_dim=64)(  decoder\_input ) *# Pass the 2 states to a new LSTM layer, as initial state* decoder\_output = layers.LSTM(64, name="decoder")(  decoder\_embedded, initial\_state=encoder\_state ) output = layers.Dense(10)(decoder\_output) model = keras.Model([encoder\_input, decoder\_input], output) model.summary() |
| --- |



| paragraph1 = np.random.random((20, 10, 50)).astype(np.float32) paragraph2 = np.random.random((20, 10, 50)).astype(np.float32) paragraph3 = np.random.random((20, 10, 50)).astype(np.float32) lstm\_layer = layers.LSTM(64, stateful=True) output = lstm\_layer(paragraph1) output = lstm\_layer(paragraph2) output = lstm\_layer(paragraph3) *# reset\_states() will reset the cached state to the original initial\_state.* *# If no initial\_state was provided, zero-states will be used by default.* lstm\_layer.reset\_states() |
| --- |

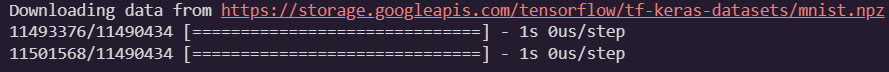
| paragraph1 = np.random.random((20, 10, 50)).astype(np.float32) paragraph2 = np.random.random((20, 10, 50)).astype(np.float32) paragraph3 = np.random.random((20, 10, 50)).astype(np.float32) lstm\_layer = layers.LSTM(64, stateful=True) output = lstm\_layer(paragraph1) output = lstm\_layer(paragraph2) existing\_state = lstm\_layer.states new\_lstm\_layer = layers.LSTM(64) new\_output = new\_lstm\_layer(paragraph3, initial\_state=existing\_state) |
| --- |

| model = keras.Sequential() model.add(  layers.Bidirectional(layers.LSTM(64, return\_sequences=True), input\_shape=(5, 10)) ) model.add(layers.Bidirectional(layers.LSTM(32))) model.add(layers.Dense(10)) model.summary() |
| --- |



| batch\_size = 64 *# Each MNIST image batch is a tensor of shape (batch\_size, 28, 28).* *# Each input sequence will be of size (28, 28) (height is treated like time).* input\_dim = 28 units = 64 output\_size = 10 *# labels are from 0 to 9* *# Build the RNN model* def build\_model(allow\_cudnn\_kernel=True):  *# CuDNN is only available at the layer level, and not at the cell level.*  *# This means `LSTM(units)` will use the CuDNN kernel,*  *# while RNN(LSTMCell(units)) will run on non-CuDNN kernels.*  if allow\_cudnn\_kernel:  *# The LSTM layer with default options uses CuDNN.*  lstm\_layer = keras.layers.LSTM(units, input\_shape=(None, input\_dim))  else:  *# Wrapping a LSTMCell in a RNN layer will not use CuDNN.*  lstm\_layer = keras.layers.RNN(  keras.layers.LSTMCell(units), input\_shape=(None, input\_dim)  )  model = keras.models.Sequential(  [  lstm\_layer,  keras.layers.BatchNormalization(),  keras.layers.Dense(output\_size),  ]  )  return model |
| --- |

| mnist = keras.datasets.mnist (x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data() x\_train, x\_test = x\_train / 255.0, x\_test / 255.0 sample, sample\_label = x\_train[0], y\_train[0] |
| --- |



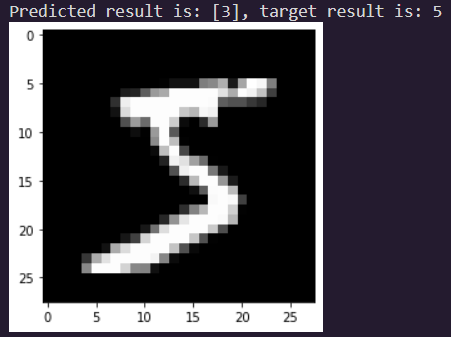
| model = build\_model(allow\_cudnn\_kernel=True) model.compile(  loss=keras.losses.SparseCategoricalCrossentropy(from\_logits=True),  optimizer="sgd",  metrics=["accuracy"], ) model.fit(  x\_train, y\_train, validation\_data=(x\_test, y\_test), batch\_size=batch\_size, epochs=1 ) |
| --- |



| noncudnn\_model = build\_model(allow\_cudnn\_kernel=False) noncudnn\_model.set\_weights(model.get\_weights()) noncudnn\_model.compile(  loss=keras.losses.SparseCategoricalCrossentropy(from\_logits=True),  optimizer="sgd",  metrics=["accuracy"], ) noncudnn\_model.fit(  x\_train, y\_train, validation\_data=(x\_test, y\_test), batch\_size=batch\_size, epochs=1 ) |
| --- |



| import matplotlib.pyplot as plt with tf.device("CPU:0"):  cpu\_model = build\_model(allow\_cudnn\_kernel=True)  cpu\_model.set\_weights(model.get\_weights())  result = tf.argmax(cpu\_model.predict\_on\_batch(tf.expand\_dims(sample, 0)), axis=1)  print(  "Predicted result is: %s, target result is: %s" % (result.numpy(), sample\_label)  )  plt.imshow(sample, cmap=plt.get\_cmap("gray")) |
| --- |



| class NestedCell(keras.layers.Layer):  def \_\_init\_\_(self, unit\_1, unit\_2, unit\_3, \*\*kwargs):  self.unit\_1 = unit\_1  self.unit\_2 = unit\_2  self.unit\_3 = unit\_3  self.state\_size = [tf.TensorShape([unit\_1]), tf.TensorShape([unit\_2, unit\_3])]  self.output\_size = [tf.TensorShape([unit\_1]), tf.TensorShape([unit\_2, unit\_3])]  super(NestedCell, self).\_\_init\_\_(\*\*kwargs)   def build(self, input\_shapes):  *# Expect input\_shape to contain 2 items, [(batch, i1), (batch, i2, i3)]*  i1 = input\_shapes[0][1]  i2 = input\_shapes[1][1]  i3 = input\_shapes[1][2]  self.kernel\_1 = self.add\_weight(  shape=(i1, self.unit\_1), initializer="uniform", name="kernel\_1"  )  self.kernel\_2\_3 = self.add\_weight(  shape=(i2, i3, self.unit\_2, self.unit\_3),  initializer="uniform",  name="kernel\_2\_3",  )   def call(self, inputs, states):  *# Inputs should be in [(batch, input\_1), (batch, input\_2, input\_3)]*  *# State should be in shape [(batch, unit\_1), (batch, unit\_2, unit\_3)]*  input\_1, input\_2 = tf.nest.flatten(inputs)  s1, s2 = states  output\_1 = tf.matmul(input\_1, self.kernel\_1)  output\_2\_3 = tf.einsum("bij,ijkl->bkl", input\_2, self.kernel\_2\_3)  state\_1 = s1 + output\_1  state\_2\_3 = s2 + output\_2\_3  output = (output\_1, output\_2\_3)  new\_states = (state\_1, state\_2\_3)  return output, new\_states   def get\_config(self):  return {"unit\_1": self.unit\_1, "unit\_2": unit\_2, "unit\_3": self.unit\_3} |
| --- |

| unit\_1 = 10 unit\_2 = 20 unit\_3 = 30 i1 = 32 i2 = 64 i3 = 32 batch\_size = 64 num\_batches = 10 timestep = 50 cell = NestedCell(unit\_1, unit\_2, unit\_3) rnn = keras.layers.RNN(cell) input\_1 = keras.Input((None, i1)) input\_2 = keras.Input((None, i2, i3)) outputs = rnn((input\_1, input\_2)) model = keras.models.Model([input\_1, input\_2], outputs) model.compile(optimizer="adam", loss="mse", metrics=["accuracy"]) |
| --- |

| input\_1\_data = np.random.random((batch\_size \* num\_batches, timestep, i1)) input\_2\_data = np.random.random((batch\_size \* num\_batches, timestep, i2, i3)) target\_1\_data = np.random.random((batch\_size \* num\_batches, unit\_1)) target\_2\_data = np.random.random((batch\_size \* num\_batches, unit\_2, unit\_3)) input\_data = [input\_1\_data, input\_2\_data] target\_data = [target\_1\_data, target\_2\_data] model.fit(input\_data, target\_data, batch\_size=batch\_size) |
| --- |



**Conclusion**:

Thus we studied an overview of the Recurrent Neural Network model and how to implement a Deep Learning Application using RNN.