Experiment No 12

Aim : Implementation of Simple RNN

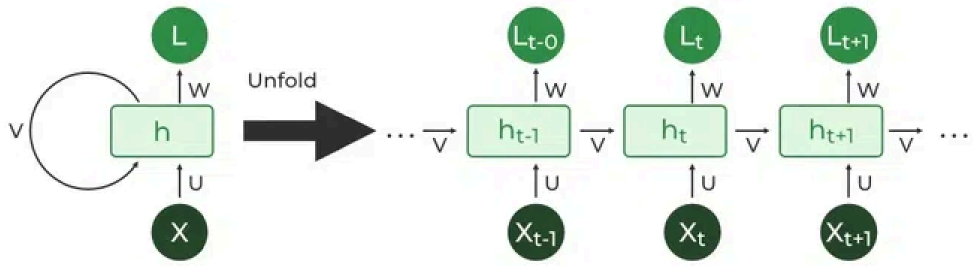
What is Recurrent Neural Network (RNN)?

Recurrent Neural Network(RNN) is a type of Neural Network where the output from the previous step is fed as input to the current step. In traditional neural networks, all the inputs and outputs are independent of each other. Still, in cases when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words. Thus RNN came into existence, which solved this issue with the help of a Hidden Layer. The main and most important feature of RNN is its Hidden state, which remembers some information about a sequence. The state is also referred to as Memory State since it remembers the previous input to the network. It uses the same parameters for each input as it performs the same task on all the inputs or hidden layers to produce the output. This reduces the complexity of parameters, unlike other neural networks.

Key Concepts of How RNNs Work

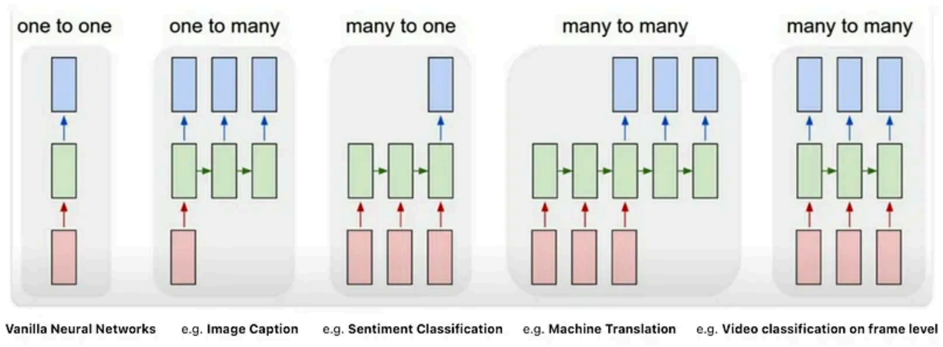
1. **Hidden States**: RNNs maintain a *hidden state* (memory) that captures information from previous steps in the sequence. At each time step, the hidden state is updated based on the current input and the previous hidden state, allowing the RNN to remember past information.
2. **Recurrent Loop**: In a standard neural network, each input only flows in one direction (input to output). In an RNN, however, there is a loop that passes the hidden state from one time step to the next. This looping mechanism makes it "recurrent."
3. **Sequential Processing**: When a sequence (e.g., a sentence or a stock price series) is fed into an RNN, each element of the sequence is processed one step at a time. At each time step, the RNN takes the current input and the previous hidden state to compute a new hidden state. This process continues until the end of the sequence.
4. **Shared Weights**: The weights in an RNN are shared across each time step, which helps it generalize over different lengths of sequences. This makes RNNs efficient for processing long sequences since they don't need different weights for each input in the sequence.

Recurrent Neuron and RNN Unfolding



The fundamental processing unit in a Recurrent Neural Network (RNN) is a Recurrent Unit, which is not explicitly called a “Recurrent Neuron.” This unit has the unique ability to maintain a hidden state, allowing the network to capture sequential dependencies by remembering previous inputs while processing. Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) versions improve the RNN’s ability to handle long-term dependencies.

Types of RNN



Limitations of Basic RNNs

- **Vanishing and Exploding Gradient:** In long sequences, gradients can become too small or too large during backpropagation, making it hard to train the model effectively. This is known as the vanishing/exploding gradient problem.
- **Short-Term Memory:** Basic RNNs struggle to remember information from earlier in the sequence for longer sequences.

Implementation

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense
```

```
text = "This is ABC a software training institute"
chars = sorted(list(set(text)))
char_to_index = {char: i for i, char in enumerate(chars)}
index_to_char = {i: char for i, char in enumerate(chars)}
```

```
seq_length = 3
sequences = []
labels = []

for i in range(len(text) - seq_length):
    seq = text[i:i+seq_length]
    label = text[i+seq_length]
    sequences.append([char_to_index[char] for char in seq])
    labels.append(char_to_index[label])
```

```
X = np.array(sequences)
y = np.array(labels)

X_one_hot = tf.one_hot(X, len(chars))
y_one_hot = tf.one_hot(y, len(chars))
```

```
model = Sequential()
model.add(SimpleRNN(50, input_shape=(seq_length, len(chars)), activation='relu'))
model.add(Dense(len(chars), activation='softmax'))
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/rnn/rnn.py:204: UserWarning: Do
using Sequential models, prefer using an `Input(shape)` object as the first layer in the
  super().__init__(**kwargs)
```

```
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

```
model.fit(X_one_hot, y_one_hot, epochs=100)
```

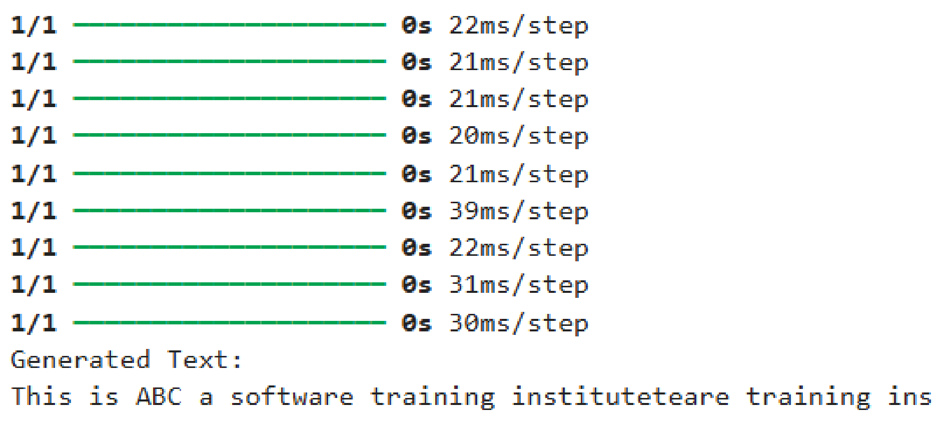
```
Epoch 93/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.9720 - loss: 0.1891
Epoch 94/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.9720 - loss: 0.1865
Epoch 95/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.9720 - loss: 0.1926
Epoch 96/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9720 - loss: 0.1888
Epoch 97/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9720 - loss: 0.1858
```

```
start_seq = "This is A"
generated_text = start_seq

for i in range(50):
    x = np.array([[char_to_index[char] for char in generated_text[-seq_length:]]])
    x_one_hot = tf.one_hot(x, len(chars))
    prediction = model.predict(x_one_hot)
    next_index = np.argmax(prediction)
    next_char = index_to_char[next_index]
    generated_text += next_char

print("Generated Text:")
print(generated_text)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 377ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step
```



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step
Generated Text:
This is ABC a software training instituteteare training ins
```

Conclusion

Successfully implementing RNNs allows for effective handling of sequential data, making them ideal for tasks like time series prediction and language processing. However, basic RNNs can struggle with longer sequences due to issues like vanishing gradients and limited memory. To address these challenges in practical applications, using LSTM or GRU layers is recommended, as they provide enhanced memory capabilities and improve model stability and accuracy.