DEEP LEARNING

ASSIGNMENT -2

During the training of a deep RNN, you encounter the vanishing gradient problem. How would you mitigate this issue?

The vanishing gradient problem during the training of deep RNNs (Recurrent Neural Networks). The vanishing gradient problem occurs when the gradients (used to update the model's weights during training) become very small as they propagate back through time, making it difficult for the model to learn effectively, especially in earlier layers. This issue is common in **deep RNNs** like Simple RNN, but it can also affect other types of recurrent networks like **LSTMs** and **GRUs**, albeit to a lesser extent.

The code provided is designed to:

1. Build and train different types of RNN models (Simple RNN, LSTM, GRU).
2. Visualize how gradients propagate through the layers of these models.
3. Highlight how each type of RNN handles the vanishing gradient problem.

Code Explanation:

1.Imports and Data Generation:

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, LSTM, Dense, GRU
import numpy as np
import matplotlib.pyplot as plt
```

Here, necessary libraries are imported:

- **TensorFlow**: For building and training neural networks.
- **NumPy**: For generating random data.
- **Matplotlib**: For plotting the gradient values.

The function generate data simulates time-series data for a binary classification task.

```
def generate_data(seq_len, n_samples=1000):
X = np.random.randn(n_samples, seq_len, 1)  # Random sequences
y = (np.mean(X, axis=1) > 0).astype(int)      # Binary classification based on mean
return X, y
```

2. Model Building:

```
def build_rnn_model(rnn_type='SimpleRNN'):
model = Sequential()
if rnn_type == 'SimpleRNN':
model.add(SimpleRNN(10,  activation='tanh',  input_shape=(seq_len, 1)))
elif rnn_type == 'LSTM':
model.add(LSTM(10, activation='tanh', input_shape=(seq_len, 1)))
elif rnn_type == 'GRU':
model.add(GRU(10, activation='tanh', input_shape=(seq_len, 1)))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='adam',          loss='binary_crossentropy', metrics=['accuracy'])
return model
```

- This function builds a sequential model based on the type of RNN you specify (Simple RNN, LSTM, or GRU).
- Each RNN has 10 hidden units, followed by a dense layer with a sigmoid activation for binary classification.
- The model is compiled using the Adam optimizer and the binary cross-entropy loss function, which is standard for binary classification tasks.

3.Training and Gradient Visualization:

```
def train_and_plot_gradients(model, rnn_type):
history = model.fit(X, y, epochs=5, batch_size=32, verbose=1)
@tf.function
def get_gradients():
with tf.GradientTape() as tape:
y_pred = model(X[:1], training=True)
loss = tf.keras.losses.binary_crossentropy(y[:1], y_pred)
return tape.gradient(loss, model.trainable_weights)
gradients = get_gradients()
gradients_norms = [tf.norm(g).numpy() for g in gradients if g is not None]
plt.figure(figsize=(8, 4))
plt.bar(range(len(gradients_norms)), gradients_norms, color='blue', alpha=0.7)
plt.xlabel('Layer Index')
plt.ylabel('Gradient Norm')
plt.title(f'Gradient Norms for {rnn_type}')
plt.show()
```

- Model Training: The model.fit() method trains the model on the generated data X and y for 5 epochs with a batch size of 32.
- Gradient Calculation: The tf.GradientTape is used to compute the gradients of the loss with respect to the model's trainable weights after running a forward pass on the first batch (X[:1]).

- Gradient Visualization: The norms of the computed gradients are then plotted to visualize how much each layer contributes to the weight updates. Large gradients indicate significant updates, while very small values (near zero) can signal the vanishing gradient problem.

4. Testing with Different RNN Types:

1.First, the Simple RNN model is trained, and its gradient flow is visualized:

model_rnn = build_rnn_model('SimpleRNN')

train_and_plot_gradients(model_rnn, 'SimpleRNN'

2.Next, the **LSTM** model is trained and analyzed in the same way:

model_lstm = build_rnn_model('LSTM')

train_and_plot_gradients(model_lstm, 'LSTM')

3.Finally, the **GRU** model is evaluated:

model_gru = build_rnn_model('GRU')

train_and_plot_gradients(model_gru, 'GRU')

Vanishing Gradient Problem:

When training RNNs, especially with Simple RNN, the gradients can shrink exponentially as they are propagated back through the network. This means that earlier layers learn very slowly, as the weights in those layers get updated minimally, leading to **slow convergence** or a model that is unable to learn long-range dependencies.

How the Code Addresses the Vanishing Gradient Problem:

1.By comparing the gradient flow for **Simple RNN**, **LSTM**, and **GRU**, you can observe that:

2.**Simple RNN** is more prone to the vanishing gradient problem.

3.**LSTM** and **GRU** architectures are designed with mechanisms (gates) to retain important information and avoid vanishing gradients, making them more robust for longer sequences.

CODING:

```
# Install TensorFlow if needed
!pip install tensorflow matplotlib --quiet

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, LSTM, Dense, GRU
import numpy as np
import matplotlib.pyplot as plt

# Simulated data
def generate_data(seq_len, n_samples=1000):
    X = np.random.randn(n_samples, seq_len, 1)
    y = (np.mean(X, axis=1) > 0).astype(int)  # Binary classification
    return X, y

seq_len = 50
X, y = generate_data(seq_len)
```

```
# Model definition
def build_rnn_model(rnn_type='SimpleRNN'):
    model = Sequential()
    if rnn_type == 'SimpleRNN':
        model.add(SimpleRNN(10, activation='tanh',
input_shape=(seq_len, 1)))
    elif rnn_type == 'LSTM':
        model.add(LSTM(10, activation='tanh', input_shape=(seq_len,
1)))
    elif rnn_type == 'GRU':
        model.add(GRU(10, activation='tanh', input_shape=(seq_len,
1)))

    model.add(Dense(1, activation='sigmoid'))
    model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
    return model

# Train and visualize gradient flow
def train_and_plot_gradients(model, rnn_type):
    history = model.fit(X, y, epochs=5, batch_size=32, verbose=1)

    # Visualize gradient flow
    @tf.function
    def get_gradients():
        with tf.GradientTape() as tape:
```

```
        y_pred = model(X[:1], training=True)
        loss = tf.keras.losses.binary_crossentropy(y[:1], y_pred)
    return tape.gradient(loss, model.trainable_weights)

gradients = get_gradients()
gradients_norms = [tf.norm(g).numpy() for g in gradients if g is not None]

plt.figure(figsize=(8, 4))
plt.bar(range(len(gradients_norms)), gradients_norms, color='blue', alpha=0.7)
plt.xlabel('Layer Index')
plt.ylabel('Gradient Norm')
plt.title(f'Gradient Norms for {rnn_type}')
plt.show()

# Test with SimpleRNN
model_rnn = build_rnn_model('SimpleRNN')
train_and_plot_gradients(model_rnn, 'SimpleRNN')

# Test with LSTM
model_lstm = build_rnn_model('LSTM')
train_and_plot_gradients(model_lstm, 'LSTM')
```

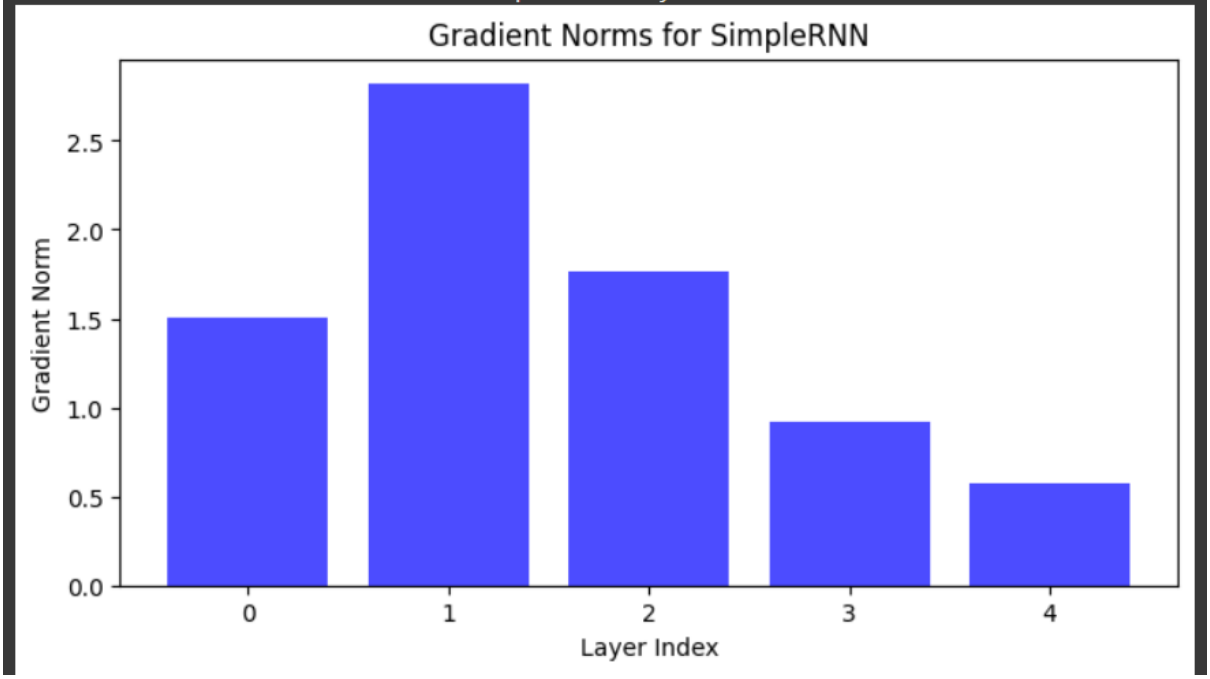
*# **Test with GRU***

model_gru = build_rnn_model('GRU')

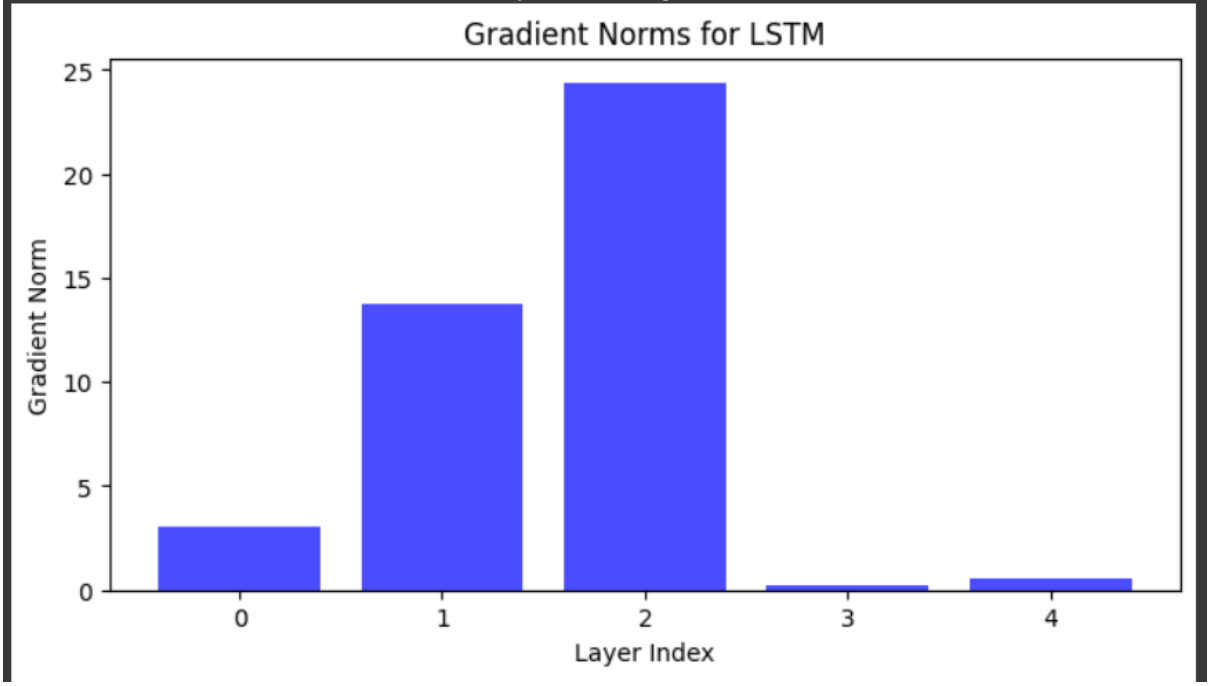
train_and_plot_gradients(model_gru, 'GRU')

OUTPUT:

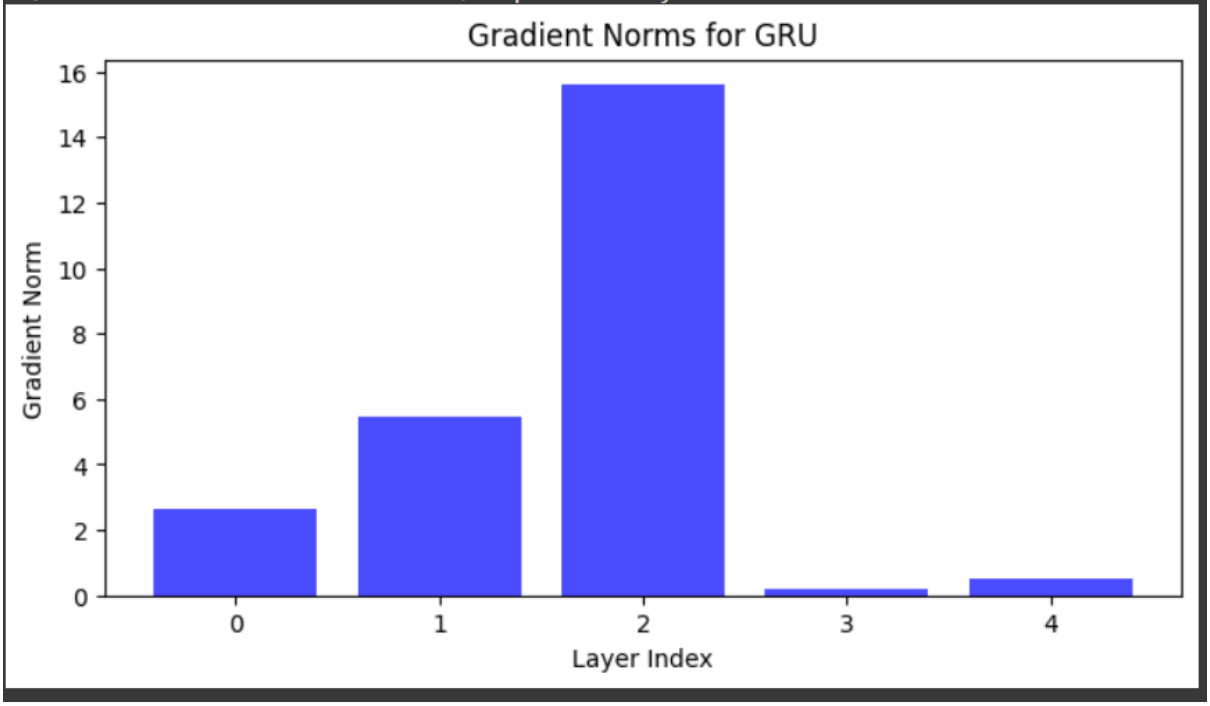
```
Epoch 1/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 5s 15ms/step - accuracy: 0.5255 - loss: 0.7114
Epoch 2/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.5741 - loss: 0.6796
Epoch 3/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 1s 16ms/step - accuracy: 0.7176 - loss: 0.5810
Epoch 4/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.8069 - loss: 0.4955
Epoch 5/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.7991 - loss: 0.4942
```



```
Epoch 1/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 3s 20ms/step - accuracy: 0.6195 - loss: 0.6746
Epoch 2/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.6807 - loss: 0.6486
Epoch 3/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.8219 - loss: 0.4863
Epoch 4/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step - accuracy: 0.8794 - loss: 0.4035
Epoch 5/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - accuracy: 0.9079 - loss: 0.3287
```



```
Epoch 1/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 3s 19ms/step - accuracy: 0.5971 - loss: 0.6735
Epoch 2/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.6142 - loss: 0.6664
Epoch 3/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 1s 17ms/step - accuracy: 0.5836 - loss: 0.6619
Epoch 4/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.6601 - loss: 0.6339
Epoch 5/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 1s 17ms/step - accuracy: 0.8497 - loss: 0.3969
```



CONCLUSION:

This code serves to demonstrate the **vanishing gradient problem** in RNNs, especially in deep and long-sequence networks. The key learning here is that using advanced RNN architectures like **LSTM** and **GRU** can help mitigate this issue. By visualizing gradients, we can directly observe the extent of the problem and see how different architectures handle it.