Name: **B. SAI CHARAN**

Roll No: **2203A51L72**

Batch No: **21CSBTB12**

ASSIGNMENT – 5

Question:

Implement Hidden Markov Models, Vanishing Gradients and exploding gradient

Answer:

Neural network models are trained by the optimization algorithm of gradient descent. The input training data helps these models learn, and the loss function gauges how accurate the prediction performance is for each iteration when parameters get updated. As training goes, the goal is to reduce the loss function/prediction error by adjusting the parameters iteratively. Specifically, the gradient descent algorithm has a forward step and a backward step, which lets it do this.

Hidden Markov Models (HMMs):

Hidden Markov Models are commonly used for sequence data and can be implemented in Python using the hmmlearn library. Here's an example:

```
from hmmlearn import hmm
import numpy as np

# Define the number of states and observations
n_states = 3
n_observations = 4

# Create a GaussianHMM model
model = hmm.GaussianHMM(n_components=n_states, covariance_type="diag", n_iter=100)

# Generate sample data
observations = np.array([[0.2], [0.5], [0.9], [1.1]])

# Fit the model
model.fit(observations)

# Predict states
hidden_states = model.predict(observations)
print("Hidden states:", hidden_states)
```

Vanishing Gradient:

The vanishing gradient problem happens in deep networks and in RNNs with long sequences. A few methods to handle this include:

- Using LSTM/GRU: Long Short-Term Memory (LSTM) units and Gated Recurrent Units (GRUs) are specifically designed to overcome this.
- ReLU Activation: ReLU and its variants (like Leaky ReLU) can help reduce vanishing gradients by not saturating at zero.
- Gradient Clipping: Clip gradients to prevent small gradients from vanishing too quickly.

```
import torch
import torch.nn as nn
import torch.optim as optim

# Define an LSTM model
class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers=1):
        super(LSTMModel, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        # LSTM Layer
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)

        # Fully connected layer for output
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        # Initialize hidden and cell states with zeros
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size)

        # Forward propagate LSTM
        out, _ = self.lstm(x, (h0, c0))  # out: tensor of shape (batch_size, seq_length, hidden_size)

        # Decode the hidden state of the last time step
```

```python
        return out

# Model, Loss, and Optimizer
input_size = 10     # Input feature dimension
hidden_size = 50    # LSTM hidden layer dimension
output_size = 1     # Output dimension
num_layers = 2      # Number of stacked LSTM layers
model = LSTMModel(input_size, hidden_size, output_size, num_layers)

# Sample data
data = torch.randn(32, 5, input_size)  # Batch of 32 sequences, each of length 5
target = torch.randn(32, output_size)

# Loss and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
num_epochs = 100
for epoch in range(num_epochs):
    model.train()

    # Forward pass
    output = model(data)
    loss = criterion(output, target)

    # Backward pass and optimization
    optimizer.zero_grad()
    loss.backward()

    # Clip gradients to prevent exploding gradients
    torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=5.0)

    optimizer.step()

    if epoch % 10 == 0:
        print(f'Epoch [{epoch}/{num_epochs}], Loss: {loss.item():.4f}')
```

```
Epoch [0/100], Loss: 0.6967
Epoch [10/100], Loss: 0.6644
Epoch [20/100], Loss: 0.5998
Epoch [30/100], Loss: 0.5010
Epoch [40/100], Loss: 0.3786
Epoch [50/100], Loss: 0.2567
Epoch [60/100], Loss: 0.1485
Epoch [70/100], Loss: 0.0497
Epoch [80/100], Loss: 0.0159
Epoch [90/100], Loss: 0.0020
```

3. Exploding Gradient:

Exploding gradients can be tackled with gradient clipping. In PyTorch, for example:

```
import torch
import torch.nn as nn
import torch.optim as optim

# Sample model and optimizer
model = nn.LSTM(input_size=10, hidden_size=20, num_layers=2)
optimizer = optim.Adam(model.parameters())

# Forward pass and loss calculation
data = torch.randn(5, 3, 10)  # (seq_length, batch_size, input_size)
target = torch.randn(5, 3, 20)  # (seq_length, batch_size, hidden_size)
output, _ = model(data)
loss = nn.MSELoss()(output, target)

# Backpropagation
optimizer.zero_grad()
loss.backward()

# Gradient clipping
nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
optimizer.step()
```