import numpy as np

def relu(x): return np.maximum(0, x)

def relu_deriv(x): return np.where(x > 0, 1, 0)

def calculate_error(target, prediction): return np.mean((target - prediction) ** 2)

class RNNModel: def **init**(self, in_size, hid_size, out_size): self.hid_size = hid_size self.W_input = np.random.randn(hid_size, in_size) * 0.05 self.W_hidden = np.random.randn(hid_size, hid_size) * 0.05 self.W_output = np.random.randn(out_size, hid_size) * 0.05 self.b_hidden = np.zeros((hid_size, 1)) self.b_output = np.zeros((out_size, 1))

```
                                    def step_forward(self, x, h_prev):
                                                    x = x.reshape(-1, 1)
    h = relu(np.dot(self.W_input, x) + np.dot(self.W_hidden, h_prev)
                                                      + self.b_hidden)
                          y = np.dot(self.W_output, h) + self.b_output
                                                          return h, y

                                     def full_forward(self, sequence):
                                      h = np.zeros((self.hid_size, 1))
                                                  self.h_cache = [h]
                                        self.input_cache = sequence
                                                       outputs = []

                                                       for x in sequence:
                                          h, y = self.step_forward(x, h)
                                                 self.h_cache.append(h)
                                                      outputs.append(y)

                                                          return outputs

                               def compute_gradients(self, targets, outputs):
                                      d_W_input = np.zeros_like(self.W_input)
                                    d_W_hidden = np.zeros_like(self.W_hidden)
                                    d_W_output = np.zeros_like(self.W_output)
                                    d_b_hidden = np.zeros_like(self.b_hidden)
                                    d_b_output = np.zeros_like(self.b_output)

                                       dh_next = np.zeros((self.hid_size, 1))
                                                          total_error = 0
```

```
                    for t in reversed(range(len(self.input_cache))):
                                y_true = targets[t].reshape(-1, 1)
                                        y_pred = outputs[t]
                      total_error += calculate_error(y_true, y_pred)

                                    dy = y_pred - y_true
                     d_W_output += np.dot(dy, self.h_cache[t + 1].T)
                                        d_b_output += dy

                          dh = np.dot(self.W_output.T, dy) + dh_next
                       dh_raw = relu_deriv(self.h_cache[t + 1]) * dh

                              x = self.input_cache[t].reshape(-1, 1)
                                    d_W_input += np.dot(dh_raw, x.T)
                     d_W_hidden += np.dot(dh_raw, self.h_cache[t].T)
                                        d_b_hidden += dh_raw

                               dh_next = np.dot(self.W_hidden.T, dh_raw)

                return d_W_input, d_W_hidden, d_W_output, d_b_hidden,
                      d_b_output, total_error / len(self.input_cache)

                             def update_weights(self, gradients, lr):
        d_W_input, d_W_hidden, d_W_output, d_b_hidden, d_b_output, _ =
                                                          gradients
                                      self.W_input -= lr * d_W_input
                                    self.W_hidden -= lr * d_W_hidden
                                    self.W_output -= lr * d_W_output
                                    self.b_hidden -= lr * d_b_hidden
                                    self.b_output -= lr * d_b_output
```

def generate_sine_data(seq_length): x = np.linspace(0, 2 * np.pi, seq_length + 1) inputs = np.sin(x[:-1]).reshape(-1, 1) targets = np.sin(x[1:]).reshape(-1, 1) return [inputs[i] for i in range(seq_length)], [targets[i] for i in range(seq_length)]

def train_model(): seq_length = 50 inputs, targets = generate_sine_data(seq_length)

```
                    rnn = RNNModel(in_size=1, hid_size=16, out_size=1)

                                                    epochs = 3000
                                                       lr = 0.002

                                               for epoch in range(epochs):
```

```
                                outputs = rnn.full_forward(inputs)
                    gradients = rnn.compute_gradients(targets, outputs)
                                 rnn.update_weights(gradients, lr)
                                             loss = gradients[-1]
                                             if epoch % 300 == 0:
                            print(f"Cycle {epoch}, Error: {loss:.4f}")

                                    outputs = rnn.full_forward(inputs)
                                               print("\nPredictions:")
                                     for i in range(0, seq_length, 5):
          print(f"Input: {inputs[i][0]:.3f}, Predicted: {outputs[i][0,
                                0]:.4f}, Target: {targets[i][0]:.3f}")
```

if **name** == "**main**": train_model()