RNN from scratch

Dmitrii Vlasov

December 5, 2024

Abstract

This paper demonstrates the implementation of a simple Recurrent Neural Network (RNN) for predicting sine wave sequences. The RNN is trained using backpropagation and evaluated on its ability to generalize to unseen data. Results highlight the RNN's potential for sequential data tasks.

1 Introduction

Recurrent Neural Networks (RNNs) represent a class of neural networks specifically tailored for sequential data and temporal data. Unlike traditional feedworward neural networks like multilayer perceptrons (MLPs) , which assume that the data is independent, RNNs maintain an internal state that helps them process context of the data where each input depends on the previous ones. This makes RNNs suitable for tasks such as times series forecasting, natural language processing, and speach recognition.

2 Theory

Recurrent Neural Networks (RNNs) are specialized neural network architectures designed for processing sequential data. Unlike feedforward neural networks, which process inputs independently, RNNs incorporate temporal context by maintaining a hidden state that evolves over time. This section provides an overview of the theoretical foundation of basic RNNs, including their structure, learning process, and key challenges.

2.1 Recurrent Neural Networks

At their core, RNNs are defined by their recurrent connections, which allow the transfer of information from one time step to the next one. For an input sequence $x_1, x_2, \ldots, x_T$, the hidden state at time t is denoted as h_t, is computed as:

$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

where:

- x_t: Input at time step t,
- h_{t-1}: Hidden state from the previous time step,
- W_{xh}: Weight matrix mapping inputs to the hidden state,
- W_{hh}: Weight matrix for recurrence within the hidden layer,
- b_h: Bias vector for the hidden state,
- tanh: Activation function that introduces nonlinearity.

The output y_t at each time step is computed as:

$$\hat{y}_t = W_{hy}h_t + b_y$$

where:

- W_{hy}: Weight matrix mapping hidden state to the output,
- h_{t-1}: Hidden state from the previous time step,

This mechanism enables RNNs to encode information from all previous inputs in the hidden state, which captures the sequential dependencies.

2.2 Loss Function

To optimize a RNN, we define a loss fuinction that measures the difference between the model's pedictions $\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_T$ and the target sequence $y_1, y_2, \ldots, y_T$. In this paper we use the Mean Squared Error (MSE) loss:

$$\text{Loss} = \frac{1}{T}\sum_{t=1}^{T}(\hat{y}_t - y_t)^2$$

where T is the sequence length. The loss function serves to guide the training process by checking the prediction accuracy.

2.3 Backpropagation

Training RNNs involves optimizing the weight matrices W_{xh}, W_{hh}, W_{hy} and biases b_h, b_y to minimize the loss function.THis is achieved using backpropagation, which operates as follows:

1. Compute hidden states $h_1, h_2, \ldots, h_T$ and outputs $y_1, y_2, \ldots, y_T$
2. Calculate gradients for each step, propagating errors backwards through the sequence

Gradient of loss with respect to output:

$$\frac{\partial \text{Loss}}{\partial y_t} = \frac{2}{T}(y_t - t_t)$$

Gradient of loss with respect to hidden state:

$$\frac{\partial \text{Loss}}{\partial h_t} = \frac{\partial \text{Loss}}{\partial y_t}\frac{\partial y_t}{\partial h_t} + \frac{\partial \text{Loss}}{\partial h_{t+1}}\frac{\partial h_{t+1}}{\partial h_t}$$

Expanding:

$$\frac{\partial y_t}{\partial h_t} = W_{hy}^\top, \quad \frac{\partial h_{t+1}}{\partial h_t} = W_{hh}^\top \cdot (1 - h_{t+1}^2)$$

Substituting gives:

$$\frac{\partial \text{Loss}}{\partial h_t} = W_{hy}^\top \frac{\partial \text{Loss}}{\partial y_t} + W_{hh}^\top \left((1 - h_{t+1}^2)\frac{\partial \text{Loss}}{\partial h_{t+1}}\right)$$

Gradients with respect to parameters
For W_hy:

$$\frac{\partial \text{Loss}}{\partial W_{hy}} = \sum_{t=1}^{T}\frac{\partial \text{Loss}}{\partial y_t}h_t^\top$$

For b_y:

$$\frac{\partial \text{Loss}}{\partial b_y} = \sum_{t=1}^{T}\frac{\partial \text{Loss}}{\partial y_t}$$

For W_{xh}:

$$\frac{\partial \text{Loss}}{\partial W_{xh}} = \sum_{t=1}^{T}\left((1 - h_t^2) \odot \frac{\partial \text{Loss}}{\partial h_t}\right)x_t^\top$$

For W_{hh}:

$$\frac{\partial \text{Loss}}{\partial W_{hh}} = \sum_{t=1}^{T} \left((1 - h_t^2) \odot \frac{\partial \text{Loss}}{\partial h_t} \right) h_{t-1}^\top$$

For b_h:

$$\frac{\partial \text{Loss}}{\partial b_h} = \sum_{t=1}^{T} (1 - h_t^2) \odot \frac{\partial \text{Loss}}{\partial h_t}$$

To address the problem of exploding gradients, the computed gradients are clipped to a predetermined range:

$$G \to \begin{cases} \lambda \frac{G}{\|G\|} & \text{if } \|G\| > \lambda, \\ G & \text{otherwise.} \end{cases}$$

2.4 Computational Complexity

Unrolling the RNN in time introduces addition complexity which is proportional to the sequence length T. Additionally, calculating the gradients for all time stamps involves storing intermediate states, which increases memory requirements.

Thus, the complexity is:

$$\mathcal{O}(T \cdot n_h^2 + T \cdot n_h \cdot n_x),$$

where:

- T is a sequence length,
- n_h is the hidden state size,
- n_x is the input size.

3 Methodology

3.1 Data Generation

To evaluate the RNN, we generated a sine wave dataset. A sine wave is a simple periodic function with clear temporal dependencies, making it an ideal candidate for sequence prediction tasks. The sine wave data was generated using:

$$y = \sin(x), \quad x \in [0, 100].$$

We divided the data into overlapping sequences of length $T = 10$ (sequence size). For each sequence, the task was to predict the next value in the sequence. The dataset was split into:

- Training set: 80% of the data.
- Test set: 20% of the data.

3.2 RNN Implementation

The RNN was implemented from scratch in Python using NumPy. The architecture consisted of:

- Input size: 1, as each time step is a single scalar value.
- Hidden size: 16, which determines the number of hidden units.
- Output size: 1, as the model predicts a single scalar value.

The forward pass computes the hidden states and outputs using the equations described in Section 2. The backward pass calculates gradients using backpropagation through time.

3.3 Training Procedure

The RNN was trained using the Mean Squared Error (MSE) loss function. The Adam optimizer was not used to maintain simplicity; instead, gradient descent with a fixed learning rate of $\eta = 0.005$ was employed. The training process included:

- 2000 epochs.
- Gradient clipping to address exploding gradients.
- Monitoring the loss at regular intervals.

3.4 Evaluation

The trained RNN was evaluated on the test set. Predictions were compared with true values, and performance was visualized using line plots.

4 Results

The RNN successfully learned the temporal dependencies of the sine wave.

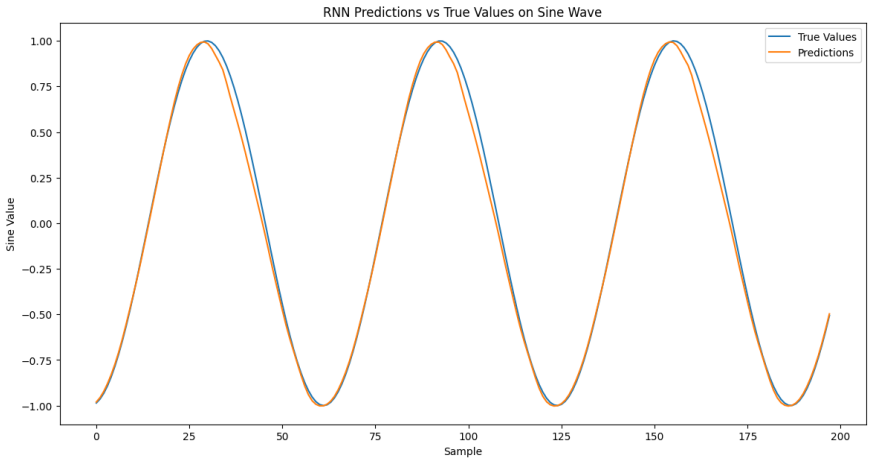

Figure 1: RNN predictions vs true values on test set

5 Discussion

5.1 Strengths

- The RNN effectively captures temporal dependencies in sequential data, as demonstrated by the close alignment of predictions with true values.
- Gradient clipping successfully mitigates exploding gradients, ensuring stable training.

5.2 Limitations

- The vanilla RNN struggles with long-term dependencies due to the vanishing gradient problem.
- The training process is computationally expensive, as gradients must be propagated back through the entire sequence.
- The model's simplicity limits its ability to handle more complex datasets.

5.3 Future Work

To address the limitations, future work could explore:

- Using Long Short-Term Memory (LSTM) or Gated Recurrent Unit (GRU) architectures to better handle long-term dependencies.
- Employing advanced optimization techniques like the Adam optimizer for faster convergence.
- Testing the model on real-world time series datasets to evaluate its practical utility.

6 Conclusion

This paper presented the implementation of a simple Recurrent Neural Network (RNN) for sine wave prediction. The RNN demonstrated its ability to capture temporal dependencies in sequential data, achieving accurate predictions on the test set. While the vanilla RNN successfully learned the sine wave's periodicity, its limitations highlight the need for more advanced architectures like LSTMs and GRUs for handling complex, long-term dependencies. This work underscores the importance of RNNs in sequence modeling and lays the foundation for further exploration of recurrent architectures.

A Python Code

```python
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(12)

class RNN:
    def __init__(self, input_size, hidden_size, output_size, learning_rate=0.001):
        self.hidden_size = hidden_size
        self.learning_rate = learning_rate

        self.W_xh = np.random.randn(hidden_size, input_size) * 0.01  # Input to hidden
        self.W_hh = np.random.randn(hidden_size, hidden_size) * 0.01  # Hidden to
            hidden
        self.W_hy = np.random.randn(output_size, hidden_size) * 0.01  # Hidden to
            output

        self.b_h = np.zeros((hidden_size, 1))
        self.b_y = np.zeros((output_size, 1))

    def forward(self, inputs):
        h_prev = np.zeros((self.hidden_size, 1))
        hidden_states = []
        outputs = []

        for x in inputs:
            h_current = np.tanh(np.dot(self.W_xh, x) + np.dot(self.W_hh, h_prev) + self
                .b_h)
            hidden_states.append(h_current)

            y = np.dot(self.W_hy, h_current) + self.b_y
            outputs.append(y)

            h_prev = h_current

        return outputs, hidden_states

    def compute_loss(self, outputs, targets):
        loss = 0
        for y, t in zip(outputs, targets):
            loss += np.sum((y - t) ** 2)
        return loss / len(outputs)
```

```
38
39      def backward(self, inputs, targets, outputs, hidden_states):
40          dW_xh = np.zeros_like(self.W_xh)
41          dW_hh = np.zeros_like(self.W_hh)
42          dW_hy = np.zeros_like(self.W_hy)
43          db_h = np.zeros_like(self.b_h)
44          db_y = np.zeros_like(self.b_y)
45
46          dh_next = np.zeros((self.hidden_size, 1))
47
48          for t in reversed(range(len(inputs))):
49              dy = outputs[t] - targets[t]
50              dW_hy += np.dot(dy, hidden_states[t].T)
51              db_y += dy
52
53              dh = np.dot(self.W_hy.T, dy) + dh_next
54              dh_raw = (1 - hidden_states[t] ** 2) * dh
55
56              db_h += dh_raw
57              dW_xh += np.dot(dh_raw, inputs[t].T)
58              if t > 0:
59                  dW_hh += np.dot(dh_raw, hidden_states[t-1].T)
60              else:
61                  dW_hh += np.dot(dh_raw, np.zeros_like(hidden_states[t]).T)
62
63              dh_next = np.dot(self.W_hh.T, dh_raw)
64
65          # Clip gradients to prevent exploding gradients
66          for d in [dW_xh, dW_hh, dW_hy, db_h, db_y]:
67              np.clip(d, -1, 1, out=d)
68
69          self.W_xh -= self.learning_rate * dW_xh
70          self.W_hh -= self.learning_rate * dW_hh
71          self.W_hy -= self.learning_rate * dW_hy
72          self.b_h -= self.learning_rate * db_h
73          self.b_y -= self.learning_rate * db_y
74
75      def train(self, inputs, targets, epochs=1000):
76          for epoch in range(epochs):
77              total_loss = 0
78              for seq, target in zip(inputs, targets):
79                  outputs, hidden_states = self.forward(seq)
80                  loss = self.compute_loss(outputs, [target]*len(seq))
81                  self.backward(seq, [target]*len(seq), outputs, hidden_states)
82                  total_loss += loss
83              avg_loss = total_loss / len(inputs)
84
85              if (epoch + 1) % (epochs // 10) == 0 or epoch == 0:
86                  print(f'Epoch {epoch+1}/{epochs}, Loss: {avg_loss:.6f}')
87
88      def predict(self, inputs):
89          outputs, _ = self.forward(inputs)
90          return outputs
91
92  # Generate a sine wave
93  def generate_sine_wave(seq_length, total_samples):
94      x = np.linspace(0, 100, total_samples)
95      y = np.sin(x)
96
97      X = []
98      Y = []
99      for i in range(total_samples - seq_length):
100         seq_in = y[i:i+seq_length]
101         seq_out = y[i+seq_length]
102         X.append([np.array([[val]]) for val in seq_in])
103         Y.append(np.array([[seq_out]]))
104     return X, Y
105
```

```
SEQ_LENGTH = 10
TOTAL_SAMPLES = 1000

X, Y = generate_sine_wave(SEQ_LENGTH, TOTAL_SAMPLES)

split = int(0.8 * len(X))
X_train, Y_train = X[:split], Y[:split]
X_test, Y_test = X[split:], Y[split:]

print(f'Training samples: {len(X_train)}, Testing samples: {len(X_test)}')

INPUT_SIZE = 1
HIDDEN_SIZE = 16
OUTPUT_SIZE = 1
LEARNING_RATE = 0.005
EPOCHS = 2000

rnn = RNN(INPUT_SIZE, HIDDEN_SIZE, OUTPUT_SIZE, learning_rate=LEARNING_RATE)

print("Starting training...")
rnn.train(X_train, Y_train, epochs=EPOCHS)
print("Training completed.")

predictions = []
for seq in X_test:
    output = rnn.predict(seq)[-1]
    predictions.append(output.item())

true_values = [y.item() for y in Y_test]

plt.figure(figsize=(14, 7))
plt.plot(true_values, label='True Values')
plt.plot(predictions, label='Predictions')
plt.title('RNN Predictions vs True Values on Sine Wave')
plt.xlabel('Sample')
plt.ylabel('Sine Value')
plt.legend()
plt.show()
```

Listing 1: RNN Implementation for Sine Wave Prediction