RNN from scratch

Dmitrii Vlasov

December 5, 2024

Abstract

This report demonstrates the implementation of a simple Recurrent Neural Network (RNN) for predicting future values in a sine wave sequence. The RNN is trained using backpropagation through time and evaluated on its ability to generalize to unseen data. Results highlight the RNN's potential for sequential data tasks and lay the groundwork for more complex architectures.

1 Introduction

Recurrent Neural Networks (RNNs) are a class of neural architectures designed specifically for sequential and temporal data tasks. Unlike standard feedforward networks, RNNs maintain a hidden state that evolves over time, allowing them to capture dependencies across multiple steps. This capability makes them well-suited for problems where the current input depends on a history of past inputs—such as time series forecasting, language modeling, and speech recognition.

In this report, we implement a simple vanilla RNN from scratch and apply it to the problem of predicting future values in a sine wave sequence. The sine wave's periodic and smooth nature provides a straightforward, illustrative example for understanding how RNNs process and learn from temporal patterns. By tackling this simple forecasting task, we gain insight into the RNN's internal workings, including how backpropagation through time (BPTT) computes gradients across multiple time steps, and the role of gradient clipping in stabilizing training.

While the model and dataset are intentionally uncomplicated, the methods described here serve as a foundation. They can be extended to handle more complex architectures, incorporate advanced optimizers, and address the challenges of long-term dependencies by employing variants such as LSTMs or GRUs. This foundational exercise thus provides a stepping stone toward applying RNN-based solutions to real-world sequential data problems.

2 Theory

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

2.1 Recurrent Neural Networks

At their core, RNNs have recurrent connections that transfer information from one time step to the next. For an input sequence $(x_1, x_2, \ldots, x_T)$, the hidden state at time t, denoted 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 recurrent connections.
- b_h: Bias vector for the hidden state.
- tanh: Nonlinear activation function.

The output y_t at each time step is computed as:

$$y_t = W_{hy} h_t + b_y,$$

where:

- W_{hy}: Weight matrix mapping the hidden state to the output.
- b_y: Bias vector for the output.

Here, y_t is the model's predicted output at time t. In our sine wave prediction scenario, y_t represents the model's prediction for the next input value x_{t+1}.

2.2 Loss Function

To train an RNN, we define a loss function that measures the difference between the model's predictions $y_1, y_2, \ldots, y_T$ and the target sequence $(x_2, x_3, \ldots, x_{T+1})$. We use the Mean Squared Error (MSE) loss:

$$\text{Loss} = \frac{1}{T} \sum_{t=1}^{T} (y_t - x_{t+1})^2.$$

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, extended to handle the temporal dimension (Backpropagation Through Time, BPTT).

Gradient of loss with respect to output:

$$\frac{\partial \text{Loss}}{\partial y_t} = \frac{2}{T}(y_t - x_{t+1})$$

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 exploding gradients, we clip the gradients:

$$g_{\text{clipped}} = \begin{cases} -\tau & \text{if } g < -\tau, \\ g & \text{if } -\tau \leq g \leq \tau, \\ \tau & \text{if } g > \tau. \end{cases}$$

2.4 Computational Complexity

Unrolling the RNN over T time steps increases computational complexity and memory usage. The complexity typically scales as:

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

where:

- T: Sequence length.
- n_h: Hidden state size.
- n_x: Input size.

3 Methodology

This section details the entire workflow of implementing and training a simple RNN to predict the next value in a sine wave sequence. We consider a sliding window over the sequence: given $(x_1, \ldots, x_T)$, we want to predict x_{T+1}. Note that our RNN produces outputs y_t where y_t attempts to predict x_{t+1}.

3.1 Data Preparation

We generate a sine wave:

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

We sample points from this function and form overlapping sequences of length $T = 10$. Each input sequence is $(x_1, x_2, \ldots, x_{10})$, and the target is x_{11}.

The dataset is split into:

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

The training set is used to learn the parameters, while the test set evaluates generalization.

3.2 Forward Pass Through the RNN

For the sequence $(x_1, x_2, \ldots, x_{10})$:

1. **Initialization:** Start with:
$$h_0 = \mathbf{0}.$$

2. **Step-by-step Processing:** At the first time step:
$$h_1 = \tanh(W_{xh}x_1 + W_{hh}h_0 + b_h), \quad y_1 = W_{hy}h_1 + b_y.$$
Here, y_1 is the prediction for x_2.
Next:
$$h_2 = \tanh(W_{xh}x_2 + W_{hh}h_1 + b_h), \quad y_2 = W_{hy}h_2 + b_y.$$
y_2 predicts x_3, and so forth until $t = 10$:
$$h_{10} = \tanh(W_{xh}x_{10} + W_{hh}h_9 + b_h), \quad y_{10} = W_{hy}h_{10} + b_y.$$
y_{10} predicts the next value x_{11}.

3. **Predicting the Next Value:** After observing $(x_1, \ldots, x_{10})$, the final output y_{10} is our forecast for x_{11}.

3.3 Loss Calculation

We measure performance by comparing the predicted y_{10} with the true x_{11} using the MSE loss:
$$\text{Loss} = (y_{10} - x_{11})^2.$$

3.4 Backpropagation Through Time (BPTT)

After the forward pass and loss calculation, we must update the RNN's parameters so that future predictions better match the true values. In a standard feedforward neural network, this is done by backpropagating the error through the network's layers. For an RNN, the process is similar, but we must account for the fact that the hidden state at each time step depends on all previous time steps. This procedure is known as Backpropagation Through Time (BPTT).

The main idea of BPTT is to "unroll" the RNN over the sequence length and then apply backpropagation as if it were a deep feedforward network with shared parameters across time steps.

1. **Unrolling the Network in Time:** Consider the RNN as it processed the sequence $(x_1, x_2, \ldots, x_{10})$ during the forward pass. Conceptually, we can think of the RNN as having created a chain of computations:
$$(x_1 \to h_1 \to y_1), \quad (x_2 \to h_2 \to y_2), \quad \ldots, \quad (x_{10} \to h_{10} \to y_{10}).$$
Each "link" in this chain uses the same parameters $(W_{xh}, W_{hh}, W_{hy}, b_h, b_y)$, but processes a different input and produces a different hidden state and output. By imagining this sequence as a series of layers aligned in time, we form a computational graph that shows explicitly how the final loss depends on all previous inputs, hidden states, and outputs.

2. **Computing the Loss Gradient at the Final Time Step:** After the forward pass, we have the loss, for example:
$$\text{Loss} = (y_{10} - x_{11})^2.$$
We begin by computing the gradient of the loss with respect to the final output y_{10}:
$$\frac{\partial \text{Loss}}{\partial y_{10}} = 2(y_{10} - x_{11}).$$
This gives us a starting point for backpropagation. Since y_{10} depends on h_{10}, and h_{10} depends on x_{10} and h_9, we can continue applying the chain rule to move backwards through time.

3. **Moving One Step Back in Time:** From:

$$y_{10} = W_{hy}h_{10} + b_y,$$

we compute how the loss changes with respect to h_{10}:

$$\frac{\partial \text{Loss}}{\partial h_{10}} = \frac{\partial \text{Loss}}{\partial y_{10}} W_{hy}^{\top}.$$

Next, we consider the transition from h_9 to h_{10}:

$$h_{10} = \tanh(W_{xh}x_{10} + W_{hh}h_9 + b_h).$$

Applying the chain rule, we find how $\frac{\partial \text{Loss}}{\partial h_{10}}$ affects h_9, W_{xh}, W_{hh}, and b_h. This involves using the derivative of tanh, which is $1 - h_{10}^2$, to backpropagate through the nonlinearity:

$$\frac{\partial \text{Loss}}{\partial h_9} = \left((1 - h_{10}^2)\frac{\partial \text{Loss}}{\partial h_{10}}\right) W_{hh}^{\top}.$$

Similarly, we accumulate gradients for W_{xh}, W_{hh}, b_h using:

$$\frac{\partial \text{Loss}}{\partial W_{xh}}, \quad \frac{\partial \text{Loss}}{\partial W_{hh}}, \quad \frac{\partial \text{Loss}}{\partial b_h}.$$

At this point, we have handled the last time step. The gradients we've computed now become the "incoming gradients" for the previous time step.

4. **Repeating for All Time Steps:** We now move one step further back in time to $t = 9$. We use $\frac{\partial \text{Loss}}{\partial h_9}$ computed above as the starting point. Since h_9 influences y_9, h_8, and parameters, we apply the chain rule again:

$$\frac{\partial \text{Loss}}{\partial y_9} = \frac{\partial \text{Loss}}{\partial h_9} W_{hy}, \quad \frac{\partial \text{Loss}}{\partial h_8} = \left((1 - h_9^2)\frac{\partial \text{Loss}}{\partial h_9}\right) W_{hh}^{\top},$$

and so forth, accumulating gradients for all parameters at time step 9.

We continue this backward traversal through the sequence:

$$t = 8, 7, 6, \ldots, 1.$$

At each step t, we:

(a) Compute $\frac{\partial \text{Loss}}{\partial y_t}$ from $\frac{\partial \text{Loss}}{\partial h_t}$.

(b) Use $\frac{\partial \text{Loss}}{\partial h_t}$ to find how the loss changes with respect to $x_t, h_{t-1}, W_{xh}, W_{hh}, b_h$.

(c) Accumulate these gradients into the total gradients for each parameter.

By the time we reach $t = 1$, we have integrated contributions from all time steps into $\frac{\partial \text{Loss}}{\partial W_{xh}}$, $\frac{\partial \text{Loss}}{\partial W_{hh}}$, $\frac{\partial \text{Loss}}{\partial W_{hy}}$, $\frac{\partial \text{Loss}}{\partial b_h}$, $\frac{\partial \text{Loss}}{\partial b_y}$.

5. **Gradient Clipping (If Needed):** If any gradient exceeds a predefined threshold τ, we clip it. This prevents excessively large updates that could destabilize training. For instance, if $\frac{\partial \text{Loss}}{\partial W_{xh}}$ is very large, we rescale it so that it lies within $[-\tau, \tau]$.

$$g_{\text{clipped}} = \begin{cases} -\tau & \text{if } g < -\tau, \\ g & \text{if } -\tau \leq g \leq \tau, \\ \tau & \text{if } g > \tau. \end{cases}$$

6. **Parameter Updates:** Once we have computed and possibly clipped all gradients, we update the parameters using gradient descent:

$$W_{xh} \leftarrow W_{xh} - \eta \frac{\partial \text{Loss}}{\partial W_{xh}}, \quad W_{hh} \leftarrow W_{hh} - \eta \frac{\partial \text{Loss}}{\partial W_{hh}}, \quad \dots$$

This completes one round of backpropagation through time.

By repeating this cycle—forward pass, loss calculation, backward pass (BPTT), and parameter update—across many epochs, the RNN gradually refines its parameters. Over time, the network becomes better at producing outputs y_t that predict the next values x_{t+1} in the sine wave sequence, improving its forecasting accuracy.

3.5 Iterative Training and Evaluation

Training a Recurrent Neural Network (RNN) is not a one-step process; rather, it is an iterative procedure that involves repeatedly presenting training examples, updating model parameters, and gradually improving performance. This improvement is tracked over multiple epochs, where an epoch is a single pass through the entire training dataset.

1. **Training Procedure:** Each training iteration consists of:
   (a) *Forward Pass:* For each training sequence $(x_1, x_2, \dots, x_{10})$, the RNN computes a series of hidden states $(h_1, h_2, \dots, h_{10})$ and outputs $(y_1, y_2, \dots, y_{10})$. The final output y_{10} is compared to the target x_{11} to compute the loss.
   (b) *Backward Pass (BPTT):* Using the loss computed in the forward pass, we perform backpropagation through time. Gradients are calculated and parameters $(W_{xh}, W_{hh}, W_{hy}, b_h, b_y)$ are updated to reduce the loss.
2. **Epochs:** One epoch involves going through all the training sequences. After each epoch, the parameters should be slightly better at capturing the underlying patterns in the data. Initially, the predictions y_t may be poor approximations of x_{t+1}, but as we accumulate many parameter updates over numerous epochs, the RNN refines its internal representations and improves its predictions.
3. **Tracking Training Loss:** During training, we monitor the loss over epochs. As the RNN's parameters adapt to the data, the loss should decrease, indicating that the predictions y_t are getting closer to the targets x_{t+1}.
4. **Generalization and the Test Set:** While reduced training loss is a good sign, it is crucial to ensure that the model does not simply memorize the training sequences. To assess this, we use a separate test set containing sequences the model has never seen during training. After completing the training phase, we:
   (a) Run the RNN on the test set input sequences.
   (b) Compute its predictions for the next values.
   (c) Compare these predictions to the true next values and measure the loss or error.

   If the model's performance on the test set is close to its performance on the training set, it indicates that the model has *generalized* well and learned the underlying structure of the sine wave, rather than just memorizing the training examples.
5. **Fine-Tuning and Additional Techniques:** If the test error is significantly higher than the training error (indicating poor generalization), one might:
   - Adjust hyperparameters such as the learning rate η or the hidden state size.
   - Implement regularization techniques to prevent overfitting.

- Use more advanced RNN variants such as LSTMs or GRUs that can handle longer-term dependencies and reduce the risk of vanishing or exploding gradients.

In essence, the iterative training and evaluation process allows the RNN to progressively refine its capacity to forecast future values of the sine wave. By continuously comparing predictions to true targets, adjusting parameters, and validating on unseen data, the model moves from random guessing towards increasingly accurate predictions that capture the temporal patterns in the data.

4 Results

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

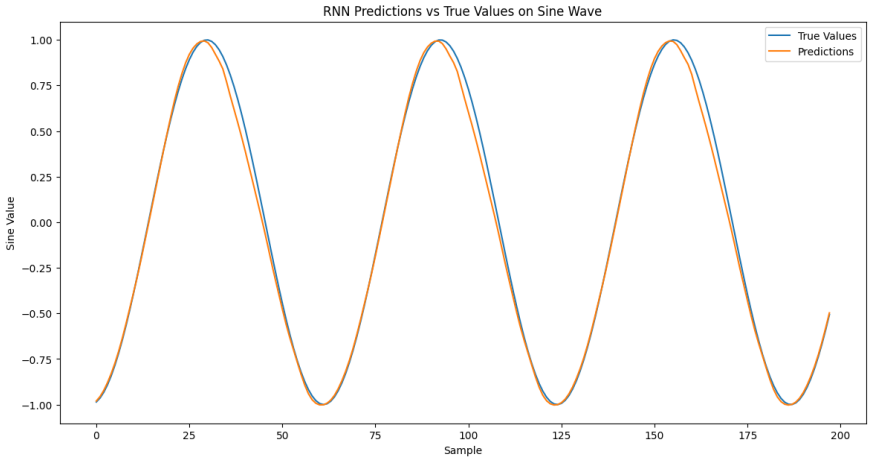

Figure 1: RNN predictions vs true values on test set

5 Discussion

5.1 Strengths

- The RNN captures temporal dependencies, producing predictions (y_t) that closely follow the sine wave.
- Gradient clipping helps maintain stable training and mitigate exploding gradients.

5.2 Limitations

- Vanilla RNNs struggle with long-term dependencies due to vanishing gradients.
- Training can be computationally expensive for long sequences.
- The model's simplicity limits its performance on more complex datasets.

5.3 Future Work

- Implementing LSTM or GRU units can better handle long-term dependencies.
- Using advanced optimizers (e.g., Adam) can speed convergence.
- Applying the model to real-world time series provides a more robust test.

6 Conclusion

This report presented the implementation of a simple RNN for predicting future sine wave values. The RNN effectively learned the sequence's temporal patterns, producing accurate short-term forecasts ($y_t \approx x_{t+1}$). While the basic RNN architecture and training method are sufficient for simple periodic data, more advanced architectures are needed for handling complex, long-term dependencies. This foundational exercise provides insight into the inner workings of RNNs and sets the stage for future exploration of more sophisticated recurrent models.

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
        self.W_hh = np.random.randn(hidden_size, hidden_size) * 0.01
        self.W_hy = np.random.randn(output_size, hidden_size) * 0.01

        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)

    def backward(self, inputs, targets, outputs, hidden_states):
        dW_xh = np.zeros_like(self.W_xh)
        dW_hh = np.zeros_like(self.W_hh)
        dW_hy = np.zeros_like(self.W_hy)
        db_h = np.zeros_like(self.b_h)
        db_y = np.zeros_like(self.b_y)

        dh_next = np.zeros((self.hidden_size, 1))

        for t in reversed(range(len(inputs))):
            dy = outputs[t] - targets[t]
            dW_hy += np.dot(dy, hidden_states[t].T)
            db_y += dy

```

```
            dh = np.dot(self.W_hy.T, dy) + dh_next
            dh_raw = (1 - hidden_states[t] ** 2) * dh

            db_h += dh_raw
            dW_xh += np.dot(dh_raw, inputs[t].T)

            if t > 0:
                dW_hh += np.dot(dh_raw, hidden_states[t-1].T)
            else:
                dW_hh += np.dot(dh_raw, np.zeros_like(hidden_states[t]).T)

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

        # Clip gradients
        for d in [dW_xh, dW_hh, dW_hy, db_h, db_y]:
            np.clip(d, -1, 1, out=d)

        self.W_xh -= self.learning_rate * dW_xh
        self.W_hh -= self.learning_rate * dW_hh
        self.W_hy -= self.learning_rate * dW_hy
        self.b_h -= self.learning_rate * db_h
        self.b_y -= self.learning_rate * db_y

    def train(self, inputs, targets, epochs=1000):
        for epoch in range(epochs):
            total_loss = 0
            for seq, target in zip(inputs, targets):
                outputs, hidden_states = self.forward(seq)
                loss = self.compute_loss(outputs, [target]*len(seq))
                self.backward(seq, [target]*len(seq), outputs, hidden_states)
                total_loss += loss
            avg_loss = total_loss / len(inputs)

            if (epoch + 1) % (epochs // 10) == 0 or epoch == 0:
                print(f'Epoch {epoch+1}/{epochs}, Loss: {avg_loss:.6f}')

    def predict(self, inputs):
        outputs, _ = self.forward(inputs)
        return outputs

def generate_sine_wave(seq_length, total_samples):
    t = np.linspace(0, 100, total_samples)
    x = np.sin(t)

    X = []
    Y = []
    for i in range(total_samples - seq_length):
        seq_in = x[i:i+seq_length]
        seq_out = x[i+seq_length]
        X.append([np.array([[val]]) for val in seq_in])
        Y.append(np.array([[seq_out]]))
    return X, Y

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 (x)')
plt.plot(predictions, label='Predictions (y)')
plt.title('RNN Predictions (y) vs True Values (x) on Sine Wave')
plt.xlabel('Sample')
plt.ylabel('Sine Value')
plt.legend()
plt.show()
```

Listing 1: RNN Implementation for Sine Wave Prediction