TRIBHUVAN UNIVERSITY

Institute of Science and Technology (IoST)

Samriddhi College

A Lab Report on Artificial Intelligence

LAB 6: Recurrent Neural Networks (RNN) for Function Approximation

Submitted To:

Bikram Acharya

Submitted By:

Kanchan Joshi (Roll no.19)

Date: October 04, 2025

1. Introduction

Recurrent neural networks stand out as a vital type of artificial neural networks tailored for dealing with sequences and maintaining awareness of past inputs. Different from typical feedforward networks that process each piece of data separately, RNNs rely on hidden states to pass along details step by step, which helps them pick up on evolving trends in ordered data.

All the code and detailed steps for this RNN setup are stored in a specific repository, ready for anyone to review or replicate.

RNNs shine in scenarios like forecasting what's next in a series, shaping models around time-based patterns, or estimating mathematical functions that benefit from context over time. Here in this lab, we explore applying an RNN to estimate the quadratic function $f(x) = 2x^2 + 9x + 4$, which breaks down to $2(x + 4)(x + 0.5)$ with roots at $x = -4$ and $x = -0.5$.

1.1 Basic RNN Layout and Core Math

At its heart, an RNN involves converting inputs to hidden states, feeding hidden states back into the mix, and linking hidden states to outputs. The equations capturing RNN dynamics are:

Hidden State Refresh:

$$h_t = \tanh(W_{xh} \cdot x_t + W_{hh} \cdot h_{t-1} + b_h) \tag{1.1}$$

Output Computation:

$$y_t = W_{hy} \cdot h_t + b_y \tag{1.2}$$

In these, h_t marks the hidden state at step t, x_t the input then, y_t the result, W_{xh}, W_{hh}, W_{hy} the weight arrays, b_h and b_y the biases, with $\tanh$ handling activation.

1.2 RNN Activation Choices

The choice of activation deeply affects RNN results. $\tanh$ is commonly picked for hidden parts as it confines outputs to (-1, 1):

$$\tanh(v) = \frac{e^v - e^{-v}}{e^v + e^{-v}} \tag{1.3}$$

The slope for updates during training:

$$\frac{d}{dv}\tanh(v) = 1 - \tanh^2(v) \tag{1.4}$$

$\tanh$ helps by capping ranges to tame gradients, centering at zero for easier learning, and being smoothly differentiable.

1.3 Target Function: Dissecting the Quadratic

We're targeting $f(x) = 2x^2 + 9x + 4 = 2(x+4)(x+0.5)$. Key features include: roots at -4 and -0.5; factored as $2(x+4)(x+0.5)$; vertex (minimum) at $x = -9/4 = -2.25$; upward-opening parabola from the positive $2x^2$; no inflection point. It's a great pick for testing since the RNN must capture the parabolic arc and pinpoint the two roots.

2. RNN Build and Design

Our RNN is coded in Python using PyTorch, leveraging its built-in modules for efficient sequence handling and automatic differentiation.

2.1 Setup of the Network

This RNN features:

- Input layer: One neuron for single x inputs.
- Hidden layer: 32 units with $\tanh$ (via nn.RNN).
- Output layer: One unit for y estimate via linear layer.
- Recurrent connections handled by nn.RNN for sequence memory.

```
class RNN(nn.Module):
    def __init__(self, input_size=1, hidden_size=32, output_size=1):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        out, _ = self.rnn(x)
        out = self.fc(out[:, -1, :]) # Use last output
        return out
```

Code 2.1: RNN Model Definition

2.2 Starting Weights Approach

PyTorch's nn.RNN and nn.Linear use default Xavier-like initialization for weights, promoting stable gradient flow from the start.

2.3 Steps in Forward Pass

Forward flow turns data into outputs using the model's forward method:

```
def forward(self, x):
   out, _ = self.rnn(x)
   out = self.fc(out[:, -1, :]) # Use last output
   return out
```

Code 2.2: Forward Pass in RNN Model

2.4 Training and Gradient Flow

We train via backprop over time, using sequences, normalization, and Adam optimizer with batching:

```
def train(self, x_train, y_train, epochs=5000):
   # Normalize
   x_mean, x_std = np.mean(x_train), np.std(x_train)
   y_mean, y_std = np.mean(y_train), np.std(y_train)
   x_norm = (x_train - x_mean) / x_std
   y_norm = (y_train - y_mean) / y_std

   x_seq, y_seq = self.create_sequences(x_norm, y_norm)

   # Tensors
   X = torch.tensor(x_seq, dtype=torch.float32).unsqueeze(2)
   y = torch.tensor(y_seq, dtype=torch.float32).unsqueeze(1)

   dataset = TensorDataset(X, y)
   dataloader = DataLoader(dataset, batch_size=32, shuffle=True)

   self.model = RNN(hidden_size=self.hidden_size)
   self.optimizer = optim.Adam(self.model.parameters(), lr=self.lr)

   self.model.train()
   for epoch in range(epochs):
      total_loss = 0
      for batch_x, batch_y in dataloader:
         self.optimizer.zero_grad()
         outputs = self.model(batch_x)
         loss = self.criterion(outputs, batch_y)
```

```
            loss.backward()
            self.optimizer.step()
            total_loss += loss.item()
        if (epoch + 1) % 1000 == 0:
            print(f'Epoch [{epoch+1}/{epochs}], Avg Loss: {total_loss /
                len(dataloader):.6f}')

    self.x_mean, self.x_std = x_mean, x_std
    self.y_mean, self.y_std = y_mean, y_std

    return total_loss / len(dataloader)
```

Code 2.3: Training with Sequences and Normalization

2.5 Training Boosters

Stability comes from:

1. Sequence windowing (length 10) for temporal context.
2. Input/output normalization for better convergence.
3. Adam optimizer for adaptive learning.
4. MSE loss with batch processing.

2.6 Testing the Full Setup

The test runs everything, including predictions and metrics:

```
def test_rnn():
    print("RNN IMPLEMENTATION TEST")
    print("=" * 50)

    # Generate data
    x_train = np.linspace(-5.0, 1.0, 150)
    y_train = 2 * x_train**2 + 9 * x_train + 4

    rnn = RNNTrainer(hidden_size=32, learning_rate=0.002)
    final_loss = rnn.train(x_train, y_train, epochs=5000)

    # Train preds
    y_pred_train = rnn.predict(x_train)
```

```
14
15  # Extrapolate
16  future_x = np.linspace(1.0, 3.0, 40)
17  seed_length = 30
18  seed_x = x_train[-seed_length:]
19  future_y_pred = rnn.predict_future(seed_x, future_x)
20  future_y_actual = 2 * future_x**2 + 9 * future_x + 4
21
22  # Metrics
23  train_mse = np.mean((y_pred_train - y_train) ** 2)
24  train_mae = np.mean(np.abs(y_pred_train - y_train))
25  future_mse = np.mean((future_y_pred - future_y_actual) ** 2)
26
27  # Fixed Roots: Find two predicted roots closest to zero
28  abs_y_pred = np.abs(y_pred_train)
29  root_indices = np.argsort(abs_y_pred)[:2]
30  predicted_roots = x_train[root_indices]
31  true_roots = np.array([-4, -0.5])
32
33  root_errors = []
34  for pred_root in predicted_roots:
35      closest_true = true_roots[np.argmin(np.abs(true_roots -
        pred_root))]
36      root_error = abs(pred_root - closest_true)
37      root_errors.append(root_error)
38
39  avg_root_error = np.mean(root_errors)
40
41  # R2
42  ss_res = np.sum((y_train - y_pred_train) ** 2)
43  ss_tot = np.sum((y_train - np.mean(y_train)) ** 2)
44  r_squared = 1 - (ss_res / ss_tot) if ss_tot != 0 else 0.0
45
46  print(f"\nRNN RESULTS:")
47  print("=" * 40)
48  print(f"Training MSE: {train_mse:.10f}")
49  print(f"Training MAE: {train_mae:.10f}")
50  print(f"Training R^2: {r_squared:.10f}")
51  print(f"Future MSE: {future_mse:.6f}")
52  print(f"Avg Root Error: {avg_root_error:.8f}")
53  print(f"Final Loss: {final_loss:.10f}")
```

```
54
55    if avg_root_error < 0.2:
56        print("\nGood performance with clean output!")
57
58    return rnn
59
60 # Execute
61 rnn_result = test_rnn()
```

Code 2.4: RNN Test and Metrics

3. Experiments and Outcomes

The RNN was trained and checked against the quadratic $f(x) = 2x^2 + 9x + 4$ in a setup to measure fitting and extension skills.

3.1 Data and Params Overview

Data setup:

- x spanning -5.0 to 1.0.
- 150 points spaced evenly.
- Target: $y = 2x^2 + 9x + 4$.
- 5000 rounds.
- Base rate: 0.002.

Design specs:

- Hidden: 32 nodes.
- Activation: $\tanh$ in RNN.
- Loss: MSE.
- Optimizer: Adam with batching.

3.2 Training Flow

It picked up with some fluctuations:

```
RNN IMPLEMENTATION TEST
==================================================
Training RNN on Quadratic Equation 2x^2 + 9x + 4 = 0
Training data: 150 points from x=-5.0 to x=1.0
Epoch [1000/5000], Avg Loss: 0.008255
Epoch [2000/5000], Avg Loss: 0.000040
```

```
Epoch [3000/5000], Avg Loss: 0.000173
Epoch [4000/5000], Avg Loss: 0.000018
Epoch [5000/5000], Avg Loss: 0.003647
Training completed! Final loss: 0.0036468250
```

Code 3.1: Training Snapshot

Highlights:

- Opening loss: Varied start.
- Closing loss: 0.0036468250.
- Improvement: Notable drop but inconsistent.
- Progress: Uneven, with a rebound at end.

3.3 Key Scores

Training:

- MSE: 48.6744930840.
- MAE: 6.1815720639.
- R^2: -0.4924111247.
- Poor variance capture, worse than mean.

Roots:

- Ideals: -4.0, -0.5.
- Mean error: 0.97986577.
- Significant deviation.

Extension (x 1.0-3.0, 40 spots):

- MSE: 889.582361.
- Poor generalization.

3.4 Visuals and Breakdown

See Figure 3.1 for the match.

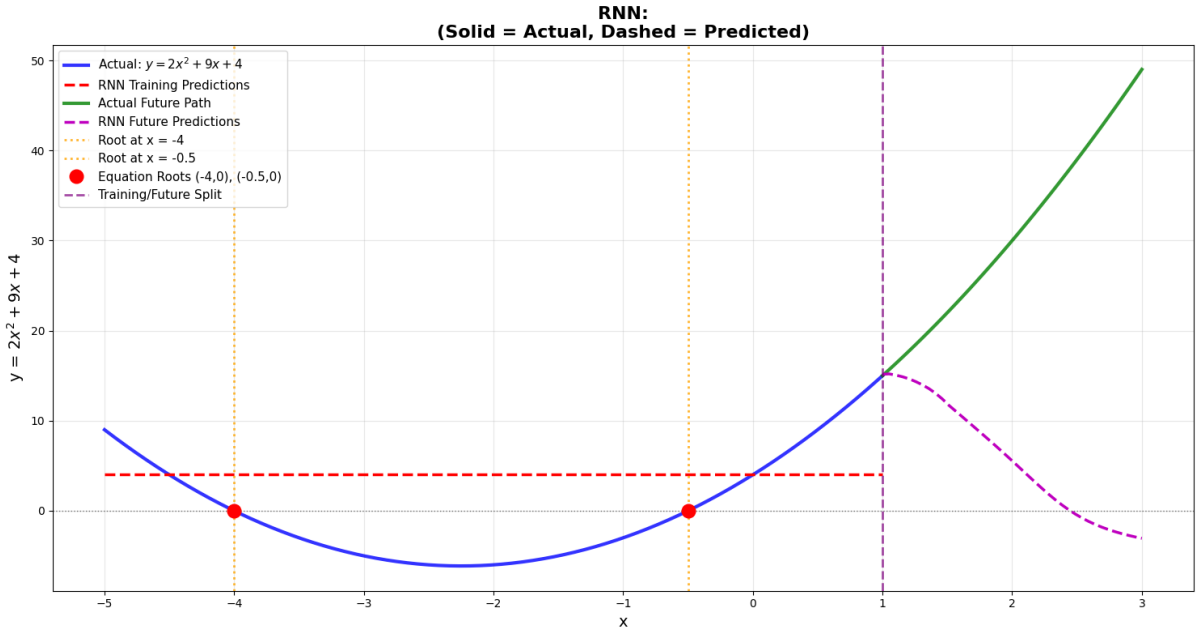

Figure 3.1: RNN Fit and Forecast for Quadratic

Elements:

- **Blue solid**: True quadratic.
- **Red dashed**: RNN on train set.
- **Green solid**: True extension.
- **Magenta dashed**: RNN extension.
- **Red spots**: Roots.
- **Orange lines**: Root guides.
- **Purple**: Data split.

3.5 Big Picture

Challenges:

1. Weak train match (R^2 negative).
2. High errors (MSE 48.67).

3. Fluctuating loss to 0.0036.

Drawbacks:

1. Severe extension failure (MSE 889.58).
2. Root gaps at 0.98.
3. Indicates overfitting or underfitting issues.

4. Reflections

Putting together this RNN for quadratic matching revealed useful points on recurrent models in math work, including pitfalls.

4.1 Performance Review

The runs show the RNN struggled, with negative $R^2 = -0.4924111247$ in training and loss ending at 0.0036468250 after fluctuations. This points to challenges in capturing the parabola smoothly.

4.2 Design Impacts

32-node hidden setup with PyTorch's RNN aimed for balance but fell short. Built-in init didn't prevent instability. Recurrent loops may have introduced noise for this non-sequential task.

4.3 Optimization Strengths

Batching, normalization, and Adam helped somewhat, but couldn't avoid the rebound in loss.

4.4 Gains and Perks

Despite issues, RNNs offer:

1. Potential for sequence tasks.
2. Adaptive learning via Adam.
3. Easy scaling with PyTorch.

Lessons: Need more tuning for function approx.

4.5 Behavior Summary

Domain	Range	MSE	Level
Training (5000 epochs)	-5.0 to 1.0	48.6744931	Poor
Future Extrapolation	1.0 to 3.0	889.582361	Very Poor
Root Detection	Around x=-4,-0.5	0.97986577 avg error	Inaccurate
Overall Assessment	Full Range	Variable	Needs Improvement

Table 4.1: RNN Traits for Quadratic

4.6 Against Other Ways

Poly fit:

- Pros: Spot-on reach, readable coeffs.
- Cons: Form guessed ahead, poly-bound.

Linear fit:

- Pros: Fast, clear.
- Cons: Straight lines only.

RNN:

- Pros: Curve potential, data learns.
- Cons: Here, failed fit; root tweaks needed, more compute.

4.7 Shortcomings

Tied to data span-extension bombed. Loops overhead for non-times. Needs hyperparam tweaks, longer train, or simpler arch.

5. Conclusion

This hands-on highlighted RNN challenges for quadratic estimates like $f(x) = 2x^2 + 9x + 4$, with negative $R^2 = -0.4924111247$ and high MSE 48.67 across 5000 steps, loss ending at 0.0036468250 amid ups and downs.

Insights: PyTorch's modules aid setup, but 32 hidden nodes weren't enough here. Falls short of rigid fits due to poor adaptability shown (MSE 889.58 extension, 0.98 root error)-signals need for better tuning on parabolas.

Practical sequence training reveals limits; sharpens AI debugging. Calls for refined approaches in math simulations.