Reproducibility Study: Supervised Reinforcement Learning with Recurrent Neural Network for Dynamic Treatment Recommendation

Zachary Hicks
Joseph Lee

December 2, 2024

Abstract

This report presents our reproduction study of the paper "Supervised Reinforcement Learning with Recurrent Neural Network for Dynamic Treatment Recommendation" (Wang et al., 2018). We reimplemented the core architecture using PyTorch, introducing modern deep learning practices while maintaining the paper's fundamental approach. Our implementation achieved comparable results with improved code modularity and training stability. The implementation code is available at: https://github.com/zachhicks27/projectDraft.git The presentation video can be found at: https://drive.google.com/file/d/1jEtchZUYWkMWTC2UNNHpPYanV8lFAOku/view?usp=sharing

1 Introduction

The original paper presented SRL-RNN (Supervised Reinforcement Learning with Recurrent Neural Network), combining supervised learning and reinforcement learning for dynamic treatment recommendation. The key innovation was the fusion of two complementary signals:

- Indicator signal: representing doctor prescriptions (supervised learning)
- Evaluation signal: representing patient outcomes (reinforcement learning)

This integration aimed to leverage both medical expertise and outcome optimization, demonstrating a 4.4% reduction in estimated mortality while maintaining high prescription accuracy.

2 Scope of Reproducibility

We focused on reproducing the following key aspects:

2.1 Core Hypotheses

1. The combination of supervised and reinforcement learning can improve treatment recommendations
2. The model can effectively balance between matching physician decisions and optimizing patient outcomes
3. The architecture can handle complex relationships between multiple medications, diseases, and patient characteristics

2.2 Implementation Focus

Our reproduction effort concentrated on:

- Reimplementing the core architecture using PyTorch
- Enhancing the attention mechanism
- Improving data processing pipeline
- Adding proper sequence masking

3 Methodology

3.1 Model Architecture

We maintained the core components while modernizing the implementation:

```
class SRL_DTR:
    def __init__(self, config):
        self.actor = ActorNetwork(
            state_size=config.state_dim,
            action_size=config.med_size,
            batch_size=config.batch_size,
            tau=config.tau,
            learning_rate=config.lra,
            epsilon=config.epsilon,
            device=self.device
        )

        self.critic = CriticNetwork(
            state_size=config.state_dim,
            action_size=config.med_size,
            batch_size=config.batch_size,
            device=self.device
        )
```

Listing 1: Core Model Architecture

3.2 Key Improvements

3.2.1 Attention Mechanism

We enhanced the attention mechanism to better handle variable-length sequences:

```
def apply_attention(self, sequence, lengths, attention_layer):
    batch_size, seq_len, hidden_size = sequence.shape
    mask = torch.arange(seq_len, device=device)[None, :] < lengths[:, None]
    mask = mask.float().unsqueeze(-1)

    attention_scores = attention_layer(sequence)
    attention_scores = attention_scores.masked_fill(~mask.bool(), float('-inf'))
    attention_weights = F.softmax(attention_scores, dim=1)

    context = (attention_weights * sequence).sum(dim=1)
    return context
```

Listing 2: Enhanced Attention Mechanism

3.3 Dataset Description

We used the MIMIC-III dataset with the following preprocessing steps:

```
class MIMICDataset(Dataset):
    def __init__(self, data_path, split='train'):
        # Load core tables
        self.demographics = pd.read_csv(f'{split}_demographics.csv')
        self.timeseries = pd.read_csv(f'{split}_timeseries.csv')
        self.diagnoses = pd.read_csv(f'{split}_diagnoses.csv')
        self.prescriptions = pd.read_csv(f'{split}_prescriptions.csv')

        # Define feature columns
        self.lab_cols = ['dbp', 'fio2', 'gcs', 'sbp', 'hr',
                         'rr', 'spo2', 'temp']
        self.demo_cols = ['age', 'gender', 'weight', 'height']
```

Listing 3: Dataset Processing

Key preprocessing steps included:

- Standardization of medication codes using ATC mapping
- Handling of missing values using KNN imputation
- Sequence padding and masking for variable-length data
- Proper train/validation/test splitting

3.4 Computational Implementation

3.4.1 Software Framework

- Primary Framework: PyTorch 1.9.0
- Additional Libraries: pandas, numpy, scikit-learn
- Development Environment: Python 3.8

3.4.2 Hardware

- CPU: Intel Core i7
- RAM: 32GB
- GPU: NVIDIA RTX 3080

3.4.3 Hyperparameters

```
class Config:
    def __init__(self):
        self.batch_size = 30
        self.gamma = 0.99
        self.tau = 0.001
        self.lra = 0.001  # Actor learning rate
        self.lrc = 0.005  # Critic learning rate
        self.epsilon = 0.5  # SL-RL balance
        self.time_stamp = 50
```

Listing 4: Configuration

4 Original Paper Methodology

4.1 Overview

The original paper presented a novel approach combining supervised learning (SL) and reinforcement learning (RL) for dynamic treatment recommendation. Their key innovation was the integration of two complementary signals:

- Indicator Signal: Doctor prescriptions (supervised component)
- Evaluation Signal: Patient outcomes (reinforcement component)

4.2 Core Components

The authors structured their solution around three main components:

4.2.1 1. Off-Policy Actor-Critic Framework

The authors used an actor-critic architecture where:

- Actor: Recommends medications based on patient state
- Critic: Evaluates the quality of recommendations
- Off-policy learning: Enables learning from historical data

4.2.2 2. Combined Learning Objective

They formulated a joint objective function:

$$J(\theta) = (1 - \epsilon)J_{RL}(\theta) + \epsilon(-J_{SL}(\theta)) \tag{1}$$

where:

- $J_{RL}(\theta)$: Reinforcement learning objective maximizing patient outcomes
- $J_{SL}(\theta)$: Supervised learning objective matching doctor decisions
- ϵ: Weighting parameter balancing the two objectives

4.2.3 3. RNN for Temporal Dependencies

The authors used RNN to handle the partially-observed Markov Decision Process (POMDP) nature of the problem, capturing temporal dependencies in patient histories.

5 Our Implementation

5.1 Data Preprocessing Pipeline

Our preprocessing implementation made several key improvements to the original approach:

```python
def load_and_preprocess_mimic(base_path):
    """Enhanced preprocessing with ATC mapping."""
    # Load core tables
    patients = pd.read_csv(f'{base_path}/PATIENTS.csv')
    admissions = pd.read_csv(f'{base_path}/ADMISSIONS.csv')
    diagnoses = pd.read_csv(f'{base_path}/DIAGNOSES_ICD.csv')
    prescriptions = pd.read_csv(f'{base_path}/PRESCRIPTIONS.csv')

    # Process each component
    diagnoses_processed = process_diagnoses(diagnoses)
    prescriptions_processed = map_to_atc_codes(
        prescriptions,
        atc_csv_path='atc_classes.csv'
    )
    timeseries = process_timeseries_enhanced(
        chartevents,
        labevents,
        outputevents
    )
```

Listing 5: Core Preprocessing Functions

Key preprocessing improvements included:

1. **Medication Standardization**: We implemented ATC code mapping:
   - Standardized drug names using regex patterns
   - Mapped to level-3 ATC codes
   - Created common drug name mappings for exact matches
2. **Time Series Processing**: Enhanced handling of temporal data:
   - Combined multiple data sources (chart events, lab events, output events)
   - Implemented 24-hour unit conversion
   - Added proper sequence masking
3. **Data Validation**: Added comprehensive validation steps:

```python
    def add_data_quality_checks(demographics, timeseries,
                                diagnoses, medications):
        # Check coverage of physiological parameters
        param_coverage = timeseries.groupby('itemid').agg({
            'hadm_id': 'nunique',
            'valuenum': ['count', 'mean', 'std']
        })

        # Check temporal distribution
        temporal_dist = timeseries.groupby('hadm_id').agg({
            'charttime': ['min', 'max', 'count'],
            'itemid': 'nunique'
        })

```

Listing 6: Data Validation

5.2 Training Implementation

Our training process introduced several improvements:

```
class SRL_DTR:
    def train_step(self, batch):
        # Forward pass through actor
        actions = self.actor(
            states['labs'],
            states['diagnoses'],
            states['demographics'],
            seq_lengths
        )

        # Calculate losses
        rl_loss = -q_values.mean()
        sl_loss = F.binary_cross_entropy(
            actions.clone(),
            flattened_actions
        )
        actor_loss = (1 - self.config.epsilon) * rl_loss + \
                     self.config.epsilon * sl_loss
```

Listing 7: Training Implementation

Key training improvements:

1. **Enhanced Learning Process**:
    - Implemented gradient clipping for stability
    - Added proper sequence masking for variable-length data
    - Improved memory efficiency with batch processing
2. **Monitoring and Evaluation**:
    - Added comprehensive metrics tracking
    - Implemented validation during training
    - Added early stopping based on validation metrics
3. **Model Architecture**:
    - Enhanced attention mechanisms for both lab values and diagnoses
    - Improved temporal processing with bidirectional LSTM
    - Added dropout for better regularization

6 Results

6.1 Quantitative Results

Our implementation achieved the following metrics:

- Precision: 0.962
- Recall: 1.000

- F1 Score: 0.980

Confusion Matrix Results:

- True Positives: 25
- True Negatives: 1508
- False Positives: 1
- False Negatives: 0

6.2 Comparison with Original Paper

Our implementation achieved comparable results while introducing several improvements:

- Better training stability through gradient clipping
- Enhanced attention mechanism for sequence handling
- Improved code modularity and maintainability
- More robust data preprocessing pipeline

7 Discussion

7.1 Reproducibility Assessment

The paper proved to be largely reproducible, with some areas requiring modern architectural updates.

7.2 What Was Easy

- Understanding the core architecture
- Implementing the basic actor-critic framework
- Processing the MIMIC-III dataset structure

7.3 What Was Difficult

- Stabilizing the training process
- Handling variable-length sequences effectively
- Balancing supervised and reinforcement learning signals
- Managing memory with large sequence batches

7.4 Suggestions for Improvement

Based on our implementation experience, we suggest:

1. Adding transformer-based architectures for sequence processing
2. Implementing uncertainty estimation in recommendations
3. Enhancing the evaluation metrics for clinical relevance
4. Improving the data preprocessing pipeline documentation

8 Conclusion

Our reproduction study successfully implemented the core concepts of the original paper while introducing modern deep learning practices. The results demonstrate the validity of combining supervised and reinforcement learning for treatment recommendation, with our implementation achieving strong performance metrics.

9 References

1. Wang, L., Zhang, W., He, X., & Zha, H. (2018). Supervised Reinforcement Learning with Recurrent Neural Network for Dynamic Treatment Recommendation. KDD '18.
2. Johnson, A. E., et al. (2016). MIMIC-III, a freely accessible critical care database. Scientific data, 3(1), 1-9.

.1 Implementation Details

.1.1 Network Architecture

Our PyTorch implementation used the following architecture specifications:

- Actor Network:
  - Lab Processing: Bidirectional LSTM with hidden size 180
  - Disease Processing: Embedding layer (size 40) with attention
  - Demographics: Dense layer (size 40) with PReLU activation
  - Output: Sigmoid activation for medication probabilities
- Critic Network:
  - Shared architecture with actor for state processing
  - Additional action processing branch
  - Combined features processed through dense layers
  - Output: Single Q-value estimation

.1.2 Data Processing Pipeline

The data processing implemented several key steps:

1. ATC Code Mapping:
    - Standardized 1,000 most frequent medications
    - Mapped to level-3 ATC codes using regex patterns
    - Achieved 85.4% coverage of all medication records
2. Time Series Processing:
    - Processed 9 key vital signs including blood pressure, heart rate, etc.
    - Resampled to 24-hour intervals
    - Applied KNN imputation for missing values
3. Diagnosis Processing:
    - Selected top 2,000 ICD-9 codes by frequency
    - Covered 95.3% of all diagnosis records
    - Created efficient embedding lookups

.1.3 Training Procedure

Our training implementation introduced several improvements:

1. Batch Processing:
    - Batch size: 30
    - Gradient clipping at 1.0
    - Learning rates: 0.001 (actor), 0.005 (critic)
2. Loss Calculation:
    - RL Loss: Negative Q-value mean
    - SL Loss: Binary cross-entropy with doctor decisions
    - Combined with epsilon=0.5 weighting
3. Training Schedule:
    - 100 epochs
    - Validation every 10 epochs
    - Early stopping based on Jaccard similarity

.1.4 Evaluation Metrics

We implemented comprehensive evaluation metrics:

1. Accuracy Metrics:
    - Jaccard similarity with doctor decisions
    - Precision: 0.962
    - Recall: 1.000
    - F1 Score: 0.980
2. Confusion Matrix Results:
    - True Positives: 25
    - True Negatives: 1508
    - False Positives: 1
    - False Negatives: 0

.1.5 Technical Challenges

Key challenges and solutions included:

1. Memory Management:
    - Challenge: Large sequence processing
    - Solution: Efficient batch processing and proper device management
2. Training Stability:
    - Challenge: Balancing RL and SL objectives
    - Solution: Gradient clipping and careful epsilon tuning
3. Data Processing:
    - Challenge: Inconsistent drug names
    - Solution: Robust ATC mapping with regex standardization