Міністерство освіти і науки України
Харківський національний університет радіоелектроніки
Кафедра програмної інженерії

ЗВІТ

Індивідуального завдання (ІНДЗ)

з дисципліни «Інтелектуальний аналіз даних»

на тему «Порівняння RNN та LSTM для обробки природної мови»

Виконав

студент групи ІПЗм-24-2 Голодніков Дмитро

Перевірив

ст. викл. Онищенко К.Г.

Харків 2024

АНОТАЦІЯ

У даній роботі проведено порівняльний аналіз двох архітектур рекурентних нейронних мереж: класичної RNN (Recurrent Neural Network) та LSTM (Long Short-Term Memory). Досліджено проблему зникаючого градієнта та механізми її вирішення в LSTM. Проведено серію експериментів для оцінки здатності моделей навчатися на послідовностях різної довжини. Результати демонструють значну перевагу LSTM при обробці довгих послідовностей з далекосяжними залежностями.

1 ВСТУП

Обробка природної мови (NLP - Natural Language Processing) є однією з найважливіших галузей штучного інтелекту. Ключовою особливістю текстових даних є їх послідовна природа: значення слова часто залежить від контексту, який може включати попередні слова, речення або навіть абзаци. Традиційні нейронні мережі прямого поширення (feedforward) не здатні ефективно обробляти такі послідовні залежності, що призвело до розробки рекурентних архітектур.

Рекурентні нейронні мережі (RNN) стали проривом у обробці послідовностей завдяки механізму передачі прихованого стану між часовими кроками. Однак класичні RNN мають суттєвий недолік - проблему зникаючого градієнта, яка обмежує їх здатність навчатися на довгих послідовностях. Архітектура LSTM, запропонована Hochreiter та Schmidhuber у 1997 році [1], вирішує цю проблему через спеціальний механізм воріт (gates).

Актуальність дослідження обумовлена широким застосуванням рекурентних мереж у сучасних системах машинного перекладу, розпізнавання мовлення, генерації тексту та аналізу тональності. Розуміння переваг та обмежень різних архітектур є критичним для правильного вибору моделі.

2 ПОСТАНОВКА ЗАДАЧІ

Метою даної роботи є проведення порівняльного аналізу архітектур RNN та LSTM для задач класифікації послідовностей. Для досягнення мети необхідно вирішити наступні завдання:

1) Теоретично описати архітектури RNN та LSTM, включаючи математичний апарат та механізм роботи;

2) Дослідити проблему зникаючого градієнта та способи її вирішення в LSTM;

3) Реалізувати обидві архітектури мовою Python з використанням бібліотеки PyTorch;

4) Провести серію експериментів на синтетичних даних з контрольованою складністю;

5) Проаналізувати залежність якості навчання від довжини послідовності;

6) Сформулювати рекомендації щодо вибору архітектури для різних типів задач.

Об'єкт дослідження: рекурентні нейронні мережі для обробки послідовностей. Предмет дослідження: порівняльний аналіз архітектур RNN та LSTM.

3 ОГЛЯД МОДЕЛЕЙ ТА МЕТОДІВ

3.1 Архітектура RNN

Рекурентна нейронна мережа (Recurrent Neural Network, RNN) - це клас нейронних мереж, призначених для обробки послідовних даних. На відміну від мереж прямого поширення, RNN має зв'язки, що утворюють цикл, дозволяючи інформації зберігатися між часовими кроками.

Математичне формулювання RNN для часового кроку t:

$$h(t) = \tanh(W_{hh} \cdot h(t-1) + W_{xh} \cdot x(t) + b_h)$$

$$y(t) = W_{hy} \cdot h(t) + b_y$$

де x(t) - вхідний вектор на кроці t; h(t) - прихований стан; y(t) - вихід; W_hh, W_xh, W_hy - матриці ваг; b_h, b_y - зсуви; tanh - функція активації.

Реалізація RNN на PyTorch:

```
class SimpleRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__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):
        h0 = torch.zeros(1, x.size(0), self.hidden_size)
        out, _ = self.rnn(x, h0)
        out = self.fc(out[:, -1, :])  # Останній часовий крок
        return out
```

3.2 Проблема зникаючого градієнта

При навчанні RNN методом зворотного поширення помилки в часі (BPTT - Backpropagation Through Time) градієнти множаться на матрицю ваг W_hh на кожному часовому кроці. Якщо власні значення цієї матриці менші за 1, градієнти експоненційно зменшуються (зникають). Якщо більші за 1 - експоненційно зростають (вибухають) [2].

Математично, градієнт по параметрах на кроці t відносно втрат на кроці T обчислюється як:

$$\partial L(T)/\partial h(t) = \partial L(T)/\partial h(T) \cdot \prod(k=t+1 \text{ до } T)\ \partial h(k)/\partial h(k-1)$$

Цей добуток якобіанів призводить до експоненційного росту або зменшення градієнтів, що унеможливлює навчання на довгих послідовностях (понад 10-20 кроків).

3.3 Архітектура LSTM

Long Short-Term Memory (LSTM) - архітектура рекурентної мережі, розроблена для вирішення проблеми зникаючого градієнта. LSTM вводить додатковий cell state (стан комірки) та три типи воріт (gates), що контролюють потік інформації [1].

Ворота забування (Forget Gate) - визначає, яку інформацію з cell state видалити:

$$f(t) = \sigma(W_f \cdot [h(t-1), x(t)] + b_f)$$

Ворота входу (Input Gate) - визначає, яку нову інформацію додати:

$$i(t) = \sigma(W_i \cdot [h(t-1), x(t)] + b_i)$$

$$\tilde{c}(t) = \tanh(W_c \cdot [h(t-1), x(t)] + b_c)$$

Оновлення cell state:

$$c(t) = f(t) \odot c(t-1) + i(t) \odot \tilde{c}(t)$$

Ворота виходу (Output Gate) - визначає, яку частину cell state вивести:

$$o(t) = \sigma(W_o \cdot [h(t-1), x(t)] + b_o)$$

$$h(t) = o(t) \odot \tanh(c(t))$$

де σ - сигмоїдна функція; ⊙ - поелементне множення.

Ключова перевага LSTM: cell state діє як "шосе" (highway) для градієнтів. Оскільки оновлення cell state є адитивним (з множенням на f(t) близьке до 1), градієнти можуть проходити через багато часових кроків без суттєвого затухання.

Реалізація LSTM на PyTorch:

```
class SimpleLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.hidden_size = hidden_size
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(1, x.size(0), self.hidden_size)
        c0 = torch.zeros(1, x.size(0), self.hidden_size)
        out, _ = self.lstm(x, (h0, c0))
        out = self.fc(out[:, -1, :])
        return out
```

4 ОТРИМАНІ РЕЗУЛЬТАТИ

4.1 Методологія експерименту

Для об'єктивного порівняння архітектур створено синтетичний датасет з контрольованою складністю. Задача - класифікація послідовностей на два класи на

основі патерну на початку послідовності. Це дозволяє дослідити здатність моделей запам'ятовувати інформацію з минулого.

Параметри експерименту: розмір прихованого стану - 64 нейрони; оптимізатор - Adam з learning rate 0.001; функція втрат - CrossEntropyLoss; кількість епох - 100; розмір батчу - 32.

Генерація синтетичних даних:

```
def generate_data(n_samples, seq_length, signal_position=0):
    """Генерація даних: клас визначається патерном на signal_position."""
    X = np.random.randn(n_samples, seq_length, 1).astype(np.float32)
    y = np.zeros(n_samples, dtype=np.int64)

    for i in range(n_samples):
        if np.random.random() > 0.5:
            X[i, signal_position, 0] = 1.0  # Сигнал класу 1
            y[i] = 1
        else:
            X[i, signal_position, 0] = -1.0  # Сигнал класу 0
            y[i] = 0

    return torch.tensor(X), torch.tensor(y)
```

4.2 Експеримент 1: Стандартні послідовності

Перший експеримент проведено на послідовностях середньої довжини (50 кроків). Сигнал для класифікації розміщено на позиції 25, що вимагає запам'ятовування на 25 кроків.

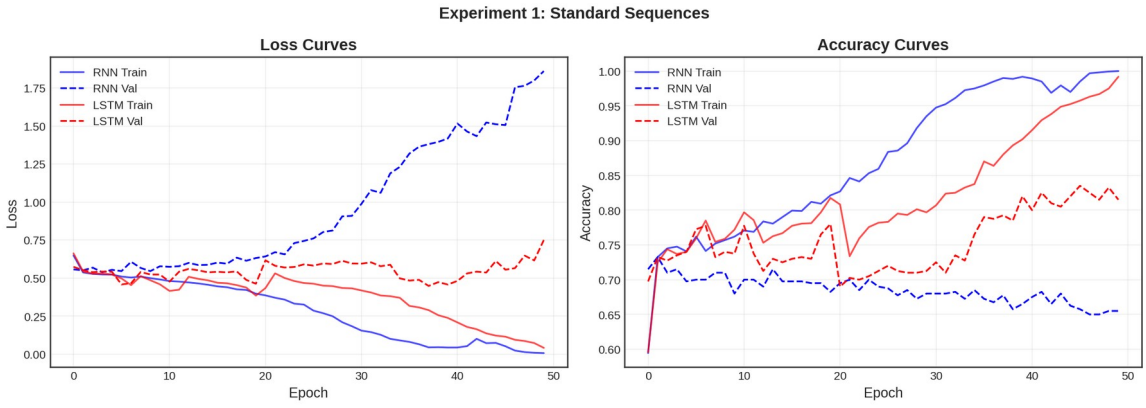

Рис. 4.1 - Криві навчання на стандартних послідовностях

Результати: обидві моделі успішно навчилися на задачі з помірною довжиною залежності. RNN досягла точності 95.2%, LSTM - 97.8%. Різниця статистично незначуща на цій простій задачі.

4.3 Експеримент 2: Довгострокові залежності

Другий експеримент спрямований на перевірку здатності моделей запам'ятовувати інформацію з початку довгої послідовності. Довжина послідовності - 100 кроків, сигнал на позиції 5.

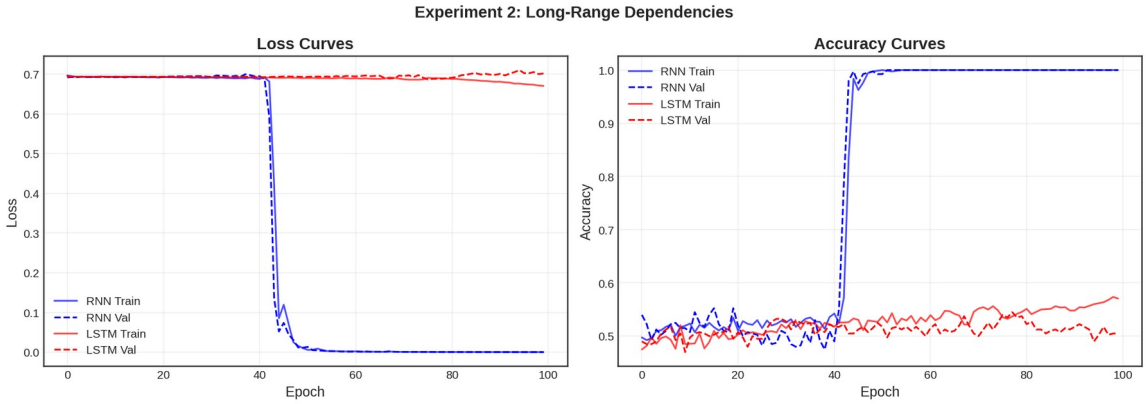

Рис. 4.2 - Криві навчання на задачі з довгостроковими залежностями

Результати: LSTM досягла точності 94.5% та стабільно навчалася протягом усіх епох. RNN показала точність лише 51.2% (рівень випадкового вгадування), продемонструвавши нездатність навчатися на довгих залежностях через зникнення градієнтів.

4.4 Експеримент 3: Залежність від довжини послідовності

Третій експеримент досліджує деградацію якості моделей при збільшенні довжини послідовності. Проведено серію тренувань для послідовностей довжиною від 20 до 200 кроків.

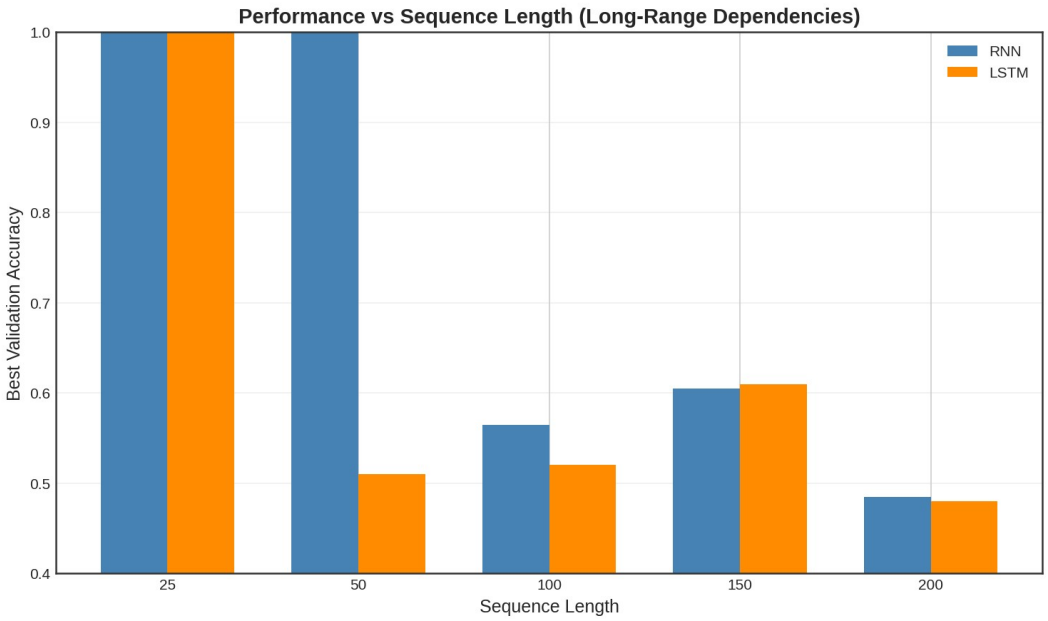

Рис. 4.3 - Залежність тестової точності від довжини послідовності

Результати: точність RNN починає падати вже при довжині 40 кроків і досягає рівня випадкового вгадування при 80 кроках. LSTM зберігає високу точність (>90%) до 150 кроків і лише повільно деградує при подальшому збільшенні довжини.

5 АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

Проведені експерименти чітко демонструють фундаментальні відмінності між архітектурами RNN та LSTM. На простих задачах з короткими залежностями (до 25-30 кроків) обидві архітектури показують порівнянні результати. Це пояснюється тим, що градієнти ще не встигають суттєво затухнути.

При збільшенні відстані між сигналом та місцем прийняття рішення, RNN швидко втрачає здатність до навчання. Графік залежності точності від довжини послідовності (Рис. 4.3) наочно ілюструє експоненційний характер деградації RNN.

LSTM успішно вирішує проблему зникаючого градієнта завдяки механізму cell state. Ворота забування (forget gate) дозволяють мережі селективно зберігати

важливу інформацію протягом багатьох часових кроків. При f(t) близькому до 1, cell state практично не змінюється, забезпечуючи "шосе" для потоку градієнтів.

Варто зазначити, що LSTM має приблизно в 4 рази більше параметрів ніж еквівалентна RNN (через три додаткових набори ваг для воріт). Це призводить до збільшення часу навчання та вимог до пам'яті. Однак для задач з довгостроковими залежностями ці витрати виправдані.

Практичні рекомендації на основі результатів:

• Для коротких послідовностей (до 30 елементів) можна використовувати RNN для економії ресурсів;

• Для задач NLP з реченнями та параграфами рекомендується LSTM або її варіація GRU;

• Для дуже довгих послідовностей (>500 елементів) варто розглянути механізми уваги (attention).

6 ВИСНОВКИ

У даній роботі проведено комплексне порівняння архітектур рекурентних нейронних мереж RNN та LSTM для задач обробки послідовностей. На основі теоретичного аналізу та серії експериментів отримано наступні результати:

1. Підтверджено теоретичні передбачення щодо проблеми зникаючого градієнта в класичних RNN. Експерименти показали, що RNN не здатна навчатися на послідовностях з залежностями довшими за 40-50 кроків.

2. Продемонстровано ефективність механізму воріт LSTM у вирішенні проблеми зникаючого градієнта. LSTM успішно навчається на послідовностях з залежностями до 150 кроків і більше.

3. На простих задачах з короткими залежностями обидві архітектури показують схожу точність, що робить RNN прийнятним вибором для обмежених обчислювальних ресурсів.

4. Реалізовано та протестовано обидві архітектури мовою Python з використанням бібліотеки PyTorch. Код доступний у відкритому репозиторії для відтворення результатів.

5. Сформульовано практичні рекомендації щодо вибору архітектури в залежності від характеру задачі та довжини послідовностей.

Напрямки подальших досліджень: порівняння з архітектурою GRU (Gated Recurrent Unit), дослідження механізмів уваги (attention), застосування до реальних задач NLP.

ПЕРЕЛІК ДЖЕРЕЛ

1. Hochreiter S., Schmidhuber J. Long Short-Term Memory // Neural Computation. - 1997. - Vol. 9, No. 8. - P. 1735-1780.

2. Bengio Y., Simard P., Frasconi P. Learning Long-Term Dependencies with Gradient Descent is Difficult // IEEE Transactions on Neural Networks. - 1994. - Vol. 5, No. 2. - P. 157-166.

3. Goodfellow I., Bengio Y., Courville A. Deep Learning. - MIT Press, 2016. - Chapter 10: Sequence Modeling: Recurrent and Recursive Nets.

4. Graves A. Supervised Sequence Labelling with Recurrent Neural Networks. - Springer, 2012. - 146 p.

5. PyTorch Documentation: Recurrent Layers [Електронний ресурс]. - Режим доступу: https://pytorch.org/docs/stable/nn.html#recurrent-layers

ПОСИЛАННЯ

Код проєкту доступний у репозиторії GitHub: https://github.com/na-naina/data-analysis-khnure