lecture5_rnn_token_classification

Harito ID

2025-10-30

Giới thiệu về Mạng Nơ-ron Hồi quy (RNNs) và Bài toán Phân loại Token

Sau khi đã tìm hiểu cách biểu diễn từ ngữ dưới dạng vector, tuần này chúng ta sẽ bước vào thế giới của Deep Learning cho NLP, bắt đầu với một trong những kiến trúc nền tảng và quan trọng nhất: Mạng Nơ-ron Hồi quy (Recurrent Neural Networks - RNNs).

Phần 1: Tại sao các mô hình cũ là chưa đủ?

Hãy bắt đầu với một câu đơn giản: “The cat sat on the mat.”

Các mô hình chúng ta đã học, như Bag-of-Words (sử dụng CountVectorizer hoặc TF-IDF), sẽ xem câu này như một “túi” chứa các từ: {the, cat, sat, on, mat}. Mô hình này mất đi hoàn toàn thông tin về **thứ tự** của từ. Nó không phân biệt được “The cat sat on the mat” và “The mat sat on the cat” - hai câu có ý nghĩa hoàn toàn khác nhau.

Ngay cả khi sử dụng các mạng nơ-ron truyền thẳng (Feed-Forward Networks - FFN) trên các word embedding, chúng ta cũng gặp vấn đề tương tự. Một FFN đơn giản xử lý mỗi từ một cách độc lập. Nó không có một cơ chế tự nhiên để “nhớ” rằng từ “cat” đã xuất hiện trước từ “sat”. Ngữ cảnh, vốn là linh hồn của ngôn ngữ, đã bị bỏ qua.

Dữ liệu ngôn ngữ là **dữ liệu tuần tự (sequential data)**. Ý nghĩa của một từ phụ thuộc rất nhiều vào các từ đứng trước (và cả sau) nó. Để hiểu được ngôn ngữ, mô hình của chúng ta cần có khả năng xử lý các chuỗi và ghi nhớ thông tin qua từng bước. Đây chính là lúc RNNs tỏa sáng.

Phần 2: Giới thiệu Mạng Nơ-ron Hồi quy (RNN)

Ý tưởng cơ bản

Cách tốt nhất để hình dung về một RNN là nghĩ về nó như một mạng nơ-ron có **“bộ nhớ ngắn hạn”**. Khi bạn đọc một câu, não của bạn không xử lý từng từ một cách riêng lẻ. Bạn nhớ những từ đầu tiên để hiểu ý nghĩa của những từ tiếp theo. RNN hoạt động theo một nguyên tắc tương tự.

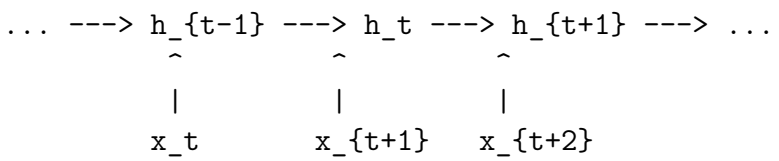
Kiến trúc của RNN có một vòng lặp (loop). Tại mỗi bước thời gian (timestep) `t`, mô hình không chỉ nhận đầu vào của bước đó (`x_t`) mà còn nhận cả thông tin từ bước trước đó. Thông tin này được gói gọn trong một thứ gọi là **trạng thái ẩn (hidden state)**, ký hiệu là `h_t`.

Công thức cốt lõi của một RNN đơn giản có thể được mô tả như sau: `h_t = f(W_hh * h_{t-1} + W_xh * x_t) y_t = g(W_hy * h_t)`

Trong đó:

- `x_t`: là vector đầu vào tại bước `t` (ví dụ: word embedding của một từ).
- `h_{t-1}`: là trạng thái ẩn từ bước `t-1`. Đây chính là "bộ nhớ" của mạng.
- `h_t`: là trạng thái ẩn mới tại bước `t`, được tính toán dựa trên cả đầu vào hiện tại và bộ nhớ quá khứ.
- `y_t`: là đầu ra tại bước `t`.
- `W_hh`, `W_xh`, `W_hy`: là các ma trận trọng số mà mô hình sẽ học.

Để dễ hình dung hơn, chúng ta thường "trải phẳng" (unroll) kiến trúc RNN qua các bước thời gian:



Tại mỗi bước, khối RNN nhận đầu vào là từ hiện tại và trạng thái ẩn của bước trước, sau đó tạo ra một trạng thái ẩn mới cho bước tiếp theo. Trạng thái ẩn này giống như một bản tóm tắt được cập nhật liên tục về những gì mô hình đã "đọc" cho đến nay.

Thuật toán và các RNN phổ biến

1. Thuật toán học (Backpropagation Through Time - BPTT) Để huấn luyện (train) một RNN, người ta sử dụng thuật toán **Backpropagation Through Time (BPTT)**. Về bản chất, BPTT là thuật toán lan truyền ngược (Backpropagation) tiêu chuẩn nhưng được áp dụng trên biểu đồ mạng đã được "trải phẳng" (unrolled) qua toàn bộ chuỗi thời gian.

- **Vấn đề:** Trong quá trình BPTT, đạo hàm (gradient) được nhân lặp đi lặp lại qua nhiều bước thời gian (qua W_{hh}). Điều này dẫn đến hai vấn đề nghiêm trọng:
  - **Vanishing Gradient (Đạo hàm tiêu biến):** Nếu giá trị đạo hàm nhỏ, nó sẽ giảm về 0 khi lan truyền ngược qua nhiều bước. Điều này khiến mô hình **quên** các thông tin ở đầu chuỗi (phụ thuộc xa).
  - **Exploding Gradient (Đạo hàm bùng nổ):** Ngược lại, nếu giá trị đạo hàm lớn, nó sẽ tăng lên rất nhanh, làm cho trọng số thay đổi đột ngột và mạng không thể hội tụ (converge).

2. Các kiến trúc RNN phổ biến (Giải quyết vấn đề bộ nhớ) Để khắc phục vấn đề Vanishing Gradient, đặc biệt khi xử lý các chuỗi dài, hai kiến trúc RNN tiên tiến hơn đã được phát triển và trở nên phổ biến, sử dụng các **Cổng (Gates)** để kiểm soát thông tin:

- **Long Short-Term Memory (LSTM):**
  - **Ý tưởng:** LSTM giới thiệu khái niệm **Ô nhớ (Cell State)** (C_t), hoạt động như một băng chuyền chạy dọc theo chuỗi thời gian, mang thông tin dài hạn.
  - **Cơ chế (Công thức Gates):**
    1. **Cổng Quên (Forget Gate - f_t):** Quyết định thông tin nào từ C_{t-1} cần bị quên (giá trị gần 0) hay giữ lại (giá trị gần 1).

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)$$

    2. **Cổng Đầu vào (Input Gate - i_t và $\tilde{C}_t$):** Quyết định thông tin mới nào

sẽ được thêm vào Ô nhớ.

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right)$$

$$\tilde{C}_t = \tanh\left(W_C \cdot [h_{t-1}, x_t] + b_C\right)$$

3. **Cập nhật Ô nhớ (C_t):** Kết hợp việc quên thông tin cũ và thêm thông tin mới.
$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$
4. **Cổng Đầu ra (Output Gate - o_t và h_t):** Quyết định trạng thái ẩn đầu ra.
$$o_t = \sigma\left(W_o \cdot [h_{t-1}, x_t] + b_o\right)$$
$$h_t = o_t \odot \tanh\left(C_t\right)$$

- **Gated Recurrent Unit (GRU):**
  - **Ý tưởng:** Là một phiên bản đơn giản hóa của LSTM, chỉ sử dụng hai Cổng, kết hợp Ô nhớ (C_t) và Trạng thái ẩn (h_t) thành một vector duy nhất.
  - **Cơ chế (Công thức Gates):**
    1. **Cổng Cập nhật (Update Gate - z_t):** Kiểm soát lượng thông tin quá khứ cần giữ lại.
    $$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t] + b_z\right)$$
    2. **Cổng Đặt lại (Reset Gate - r_t):** Quyết định thông tin quá khứ nào cần bị bỏ qua (reset) khi tính toán $\tilde{h}_t$ (đầu vào mới tiềm năng).
    $$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t] + b_r\right)$$
    3. **Trạng thái Ẩn Hiện tại (h_t):** Cập nhật trạng thái ẩn cuối cùng.
    $$\tilde{h}_t = \tanh\left(W_h \cdot [r_t \odot h_{t-1}, x_t] + b_h\right)$$
    $$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

Trong các công thức trên:

- σ là hàm kích hoạt Sigmoid, đưa giá trị về khoảng $[0, 1]$ (đóng vai trò là cổng).
- $\odot$ là phép nhân phần tử theo phần tử (Element-wise multiplication).
- W và b là các ma trận trọng số và vector độ lệch tương ứng.

Phần 3: Bài toán Phân loại Token (Token Classification)

Phân loại Token là một nhiệm vụ NLP kinh điển, trong đó chúng ta gán một nhãn (label) cho **mỗi token (từ)** trong một câu đầu vào. Vì RNN có thể tạo ra một đầu ra tại mỗi bước thời gian, nó cực kỳ phù hợp cho loại bài toán này.

Các ví dụ phổ biến nhất là:

1. **Part-of-Speech (POS) Tagging:** Gán nhãn loại từ (danh từ, động từ, tính từ,...) cho mỗi từ.
   - **Câu:** `She reads a book`
   - **Nhãn:** `(She/PRP, reads/VBZ, a/DT, book/NN)`
   - (PRP: Đại từ nhân xưng, VBZ: Động từ ngôi thứ 3 số ít, DT: Mạo từ, NN: Danh từ số ít)

2. **Named Entity Recognition (NER):** Xác định và phân loại các thực thể có tên (tên người, địa điểm, tổ chức,...).
    - **Câu:** `Apple was founded by Steve Jobs in California`
    - **Nhãn:** `(Apple/ORG, was/O, founded/O, by/O, Steve Jobs/PER, in/O, California/LOC`
    - (ORG: Tổ chức, PER: Người, LOC: Địa điểm, O: "Outside" - không phải thực thể)
3. **Dependency Parsing (Phân tích cú pháp phụ thuộc):** Xác định quan hệ ngữ pháp giữa các từ trong câu. Mặc dù phức tạp hơn, bài toán này có thể được giải quyết bằng cách cho mỗi từ dự đoán "từ quản lý" (head) của nó và loại quan hệ (ví dụ: `nsubj` - chủ ngữ, `dobj` - đối tượng trực tiếp).
    - **Câu:** `She reads a book`
    - **Quan hệ:** `reads` là gốc (root). `She` là chủ ngữ của `reads`. `book` là đối tượng của `reads`.
4. **Semantic Role Labeling (SRL) (Gán nhãn vai trò ngữ nghĩa):** Xác định vai trò của các từ/cụm từ đối với một động từ (vị ngữ) trong câu: ai làm gì (Agent), cái gì bị tác động (Theme),...
    - **Câu:** `Mary sold the book to John`
    - **Vai trò (cho động từ "sold"):** `Mary` (Agent), `the book` (Theme), `to John` (Recipient).

Phần 4: Các công cụ để triển khai thuật toán Deep Learning

Để bắt đầu với RNNs và các bài toán Token Classification, bạn sẽ cần làm quen với các công cụ và thư viện phổ biến. Việc lựa chọn công cụ phù hợp phụ thuộc rất nhiều vào giai đoạn của dự án: từ nghiên cứu, triển khai đến tối ưu hóa.

Dưới đây là bảng phân tích các công cụ theo từng giai đoạn:

Giai đoạn	Môi trường & Phần cứng	Ngôn ngữ	Framework/Thư viện	Công cụ hỗ trợ	Lý do lựa chọn
Nghiên cứu & Phát triển	Máy tính cá nhân (CPU/GPU), Google Colab	Python	PyTorch, TensorFlow, Keras	Jupyter Notebook, VS Code	**PyTorch/TensorFlow:** Cung cấp sự linh hoạt cao để thử nghiệm các kiến trúc mô hình phức tạp. **Keras:** Giao diện đơn giản, dễ sử dụng, phù hợp cho việc xây dựng nhanh các mô hình.
Triển khai trên Big Data	Cụm máy chủ (cluster), Apache Spark, Hạ tầng đám mây (AWS, GCP, Azure)	Scala, Python	Spark NLP, BigDL	Apache Spark, MLflow	**Spark NLP:** Tối ưu hóa cho việc xử lý lượng lớn dữ liệu văn bản trên các hệ thống phân tán. **BigDL:** Cho phép triển khai các mô hình deep learning trên Spark, tận dụng hiệu quả phần cứng Intel.
Tối ưu hóa Hiệu suất	Phần cứng chuyên dụng (GPU, TPU), Intel Xeon	C++, Python	ONNX, TensorRT, OpenVINO	Docker, Kubernetes	**ONNX:** Định dạng mô hình mở cho phép chuyển đổi giữa các framework. **TensorRT/OpenVINO:** Tối ưu hóa mô hình cho việc suy luận (inference) nhanh trên phần cứng NVIDIA và Intel.

Trong khuôn khổ môn học, chúng ta sẽ tập trung chủ yếu vào giai đoạn **Nghiên cứu & Phát triển** với ngôn ngữ **Python** và thư viện **PyTorch**.

Phần 5: Áp dụng RNN cho Token Classification

Vậy làm thế nào để kết hợp RNN vào bài toán này? Luồng xử lý như sau:

1. **Embedding:** Mỗi từ trong câu đầu vào được chuyển đổi thành một vector embedding.
2. **RNN Processing:** Chuỗi các vector embedding này được đưa vào RNN. Tại mỗi từ, RNN tính toán một trạng thái ẩn `h_t`, chứa thông tin về từ hiện tại và ngữ cảnh từ các từ đứng trước.
3. **Prediction:** Trạng thái ẩn `h_t` (là một vector) được đưa qua một lớp **Linear (Fully-Connected)** để ánh xạ nó sang một vector có số chiều bằng số lượng nhãn (ví dụ: 17 nhãn cho bài toán POS).
4. **Softmax:** Cuối cùng, hàm Softmax được áp dụng lên vector này để tạo ra một phân phối xác suất, cho biết xác suất mỗi nhãn là đúng cho từ hiện tại. Chúng ta sẽ chọn nhãn có xác suất cao nhất.

Dưới đây là một đoạn code **khái niệm** bằng PyTorch để minh họa luồng xử lý này:

```
import torch
import torch.nn as nn

class SimpleRNNForTokenClassification(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, num_tags):
        super(SimpleRNNForTokenClassification, self).__init__()

        # 1. Lớp Embedding: Chuyển đổi ID của từ thành vector dày đặc (dense vector).
        # Ví dụ: từ có ID là 5 -> vector [0.1, 0.9, ..., 0.4]
        self.embedding = nn.Embedding(vocab_size, embedding_dim)

        # 2. Lớp RNN: Xử lý chuỗi vector và tạo ra hidden state tại mỗi bước.
        # batch_first=True nghĩa là input và output sẽ có chiều batch ở đầu tiên.
        self.rnn = nn.RNN(embedding_dim, hidden_dim, batch_first=True)

        # 3. Lớp Linear: Ánh xạ từ hidden state (kích thước hidden_dim)
        # sang không gian số lượng nhãn (kích thước num_tags).
        self.linear = nn.Linear(hidden_dim, num_tags)

    def forward(self, sentence):
        # `sentence` là một chuỗi các ID của từ, có dạng (batch_size, seq_len).

        # 1. Truyền câu qua lớp Embedding.
        # Output `embeds` có dạng (batch_size, seq_len, embedding_dim).
        embeds = self.embedding(sentence)

        # 2. Truyền chuỗi embedding qua lớp RNN.
        # `rnn_out` là output của RNN tại mỗi bước thời gian.
        # `rnn_out` có dạng (batch_size, seq_len, hidden_dim).
        rnn_out, _ = self.rnn(embeds)

        # 3. Truyền output của RNN qua lớp Linear để lấy điểm số cho mỗi nhãn.
```

```
        # `tag_scores` có dạng (batch_size, seq_len, num_tags).
        tag_scores = self.linear(rnn_out)

        # (Trong thực tế, chúng ta sẽ áp dụng Softmax lên tag_scores để có phân phối x
        return tag_scores
```

Phần 5: Thách thức của RNN và Hướng đi tiếp theo

Mặc dù mạnh mẽ, RNN đơn giản gặp một vấn đề lớn: **Vanishing/Exploding Gradients (Gradient triệt tiêu/bùng nổ)**. Khi xử lý các chuỗi rất dài, gradient được lan truyền ngược qua nhiều bước thời gian có thể trở nên cực kỳ nhỏ (triệt tiêu) hoặc cực kỳ lớn (bùng nổ). Điều này làm cho mô hình rất khó học được các phụ thuộc xa (long-range dependencies). Ví dụ, để hiểu ý nghĩa của từ cuối cùng trong một đoạn văn dài, mô hình có thể cần "nhớ" thông tin từ câu đầu tiên, nhưng RNN đơn giản thường thất bại trong việc này.

Để giải quyết vấn đề này, các kiến trúc cải tiến đã được ra đời. chúng ta sẽ tìm hiểu về **LSTM (Long Short-Term Memory)** và **GRU (Gated Recurrent Unit)**, những kiến trúc có các "cổng" (gates) thông minh để kiểm soát luồng thông tin, cho phép chúng ghi nhớ thông tin quan trọng trong thời gian dài hơn.