

BỘ GIÁO DỤC VÀ ĐÀO TẠO

**TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP. HỒ CHÍ MINH**

**KHOA CÔNG NGHỆ THÔNG TIN**

****

**MÔN TRÍ TUỆ NHÂN TẠO CHO IOT**

**HỌC KÌ II/2023-2024**

**BÁO CÁO GIỮA KỲ**

**ĐỀ TÀI: RECURRENT NEURAL NETWORKS**

**GVHD: ThS. Phan Thanh Hy**

**Lớp HP: AIOT331185**

**Học kỳ: II**

**Năm học: 2023 – 2024**

**Thành viên nhóm:**

1. **Nguyễn Hoàng Việt Pháp 21110587**
2. **Cao Thị Thu Thủy 21110670**
3. **Lê Trạc Tiến 21110677**

*Thành phố Thủ Đức, tháng 4 năm 2024*

**MỤC LỤC**

[LỜI MỞ ĐẦU 1](#_Toc163740223)

[NỘI DUNG 2](#_Toc163740224)

[1. Sơ lược về Deep Learning 2](#_Toc163740225)

[2. Recurrent Neural Network 2](#_Toc163740226)

[2.1. Recurrent Neural Network là gì? 2](#_Toc163740227)

[2.2. Ưu điểm và hạn chế 4](#_Toc163740228)

[2.2.1. Ưu điểm 4](#_Toc163740229)

[2.2.2. Hạn chế 5](#_Toc163740230)

[2.3. Phân loại và ứng dụng bài toán RNN 6](#_Toc163740231)

[3. Xây dựng RNN dạng Tx = Ty 8](#_Toc163740232)

[3.1. Các hàm kích hoạt thường dùng 8](#_Toc163740233)

[3.2. Quá trình lan truyền thuận 8](#_Toc163740234)

[3.3. Tối ưu hóa mô hình thông qua lan truyền ngược 9](#_Toc163740235)

[3.3.1. Hàm mất mát 9](#_Toc163740236)

[3.3.2. Lan truyền ngược 10](#_Toc163740237)

[3.3.3. Vanishing/exploding gradient 10](#_Toc163740238)

[3.4. Demo với python: 12](#_Toc163740239)

[4. Giới thiệu về LSTM 19](#_Toc163740240)

[5. Bài toán đặt tên cho khủng long 21](#_Toc163740241)

[KẾT LUẬN 22](#_Toc163740242)

[TÀI LIỆU THAM KHẢO 23](#_Toc163740243)

# LỜI MỞ ĐẦU

Trong thế giới ngày nay, khi mà dữ liệu tuần tự trở nên phổ biến hơn bao giờ hết, RNN đã nổi lên như một công cụ mạnh mẽ để giải quyết những thách thức phức tạp của việc hiểu và tương tác với dữ liệu động. Từ việc dự đoán từ khóa tiếp theo trong một đoạn văn bản đến việc phân tích tín hiệu thời gian trong thị trường tài chính, RNN đã chứng minh được sức mạnh của mình trong nhiều lĩnh vực khác nhau.

Một trong những đặc điểm nổi bật của RNN là khả năng xử lý dữ liệu tuần tự có chiều dài biến đổi, không nhất thiết phải cố định. Điều này làm cho RNN trở thành một công cụ linh hoạt và mạnh mẽ, có thể thích ứng với nhiều loại dữ liệu khác nhau mà không cần phải thay đổi cấu trúc quá nhiều. Bằng cách tự học và điều chỉnh các trọng số của mạng, RNN có khả năng học và tái tạo các mẫu phức tạp trong dữ liệu, từ đó đưa ra các dự đoán và quyết định thông minh.

Trong bài báo cáo này, chúng ta sẽ khám phá sâu hơn vào cơ chế hoạt động của RNN, từ những khái niệm cơ bản đến những ứng dụng tiên tiến.

# NỘI DUNG

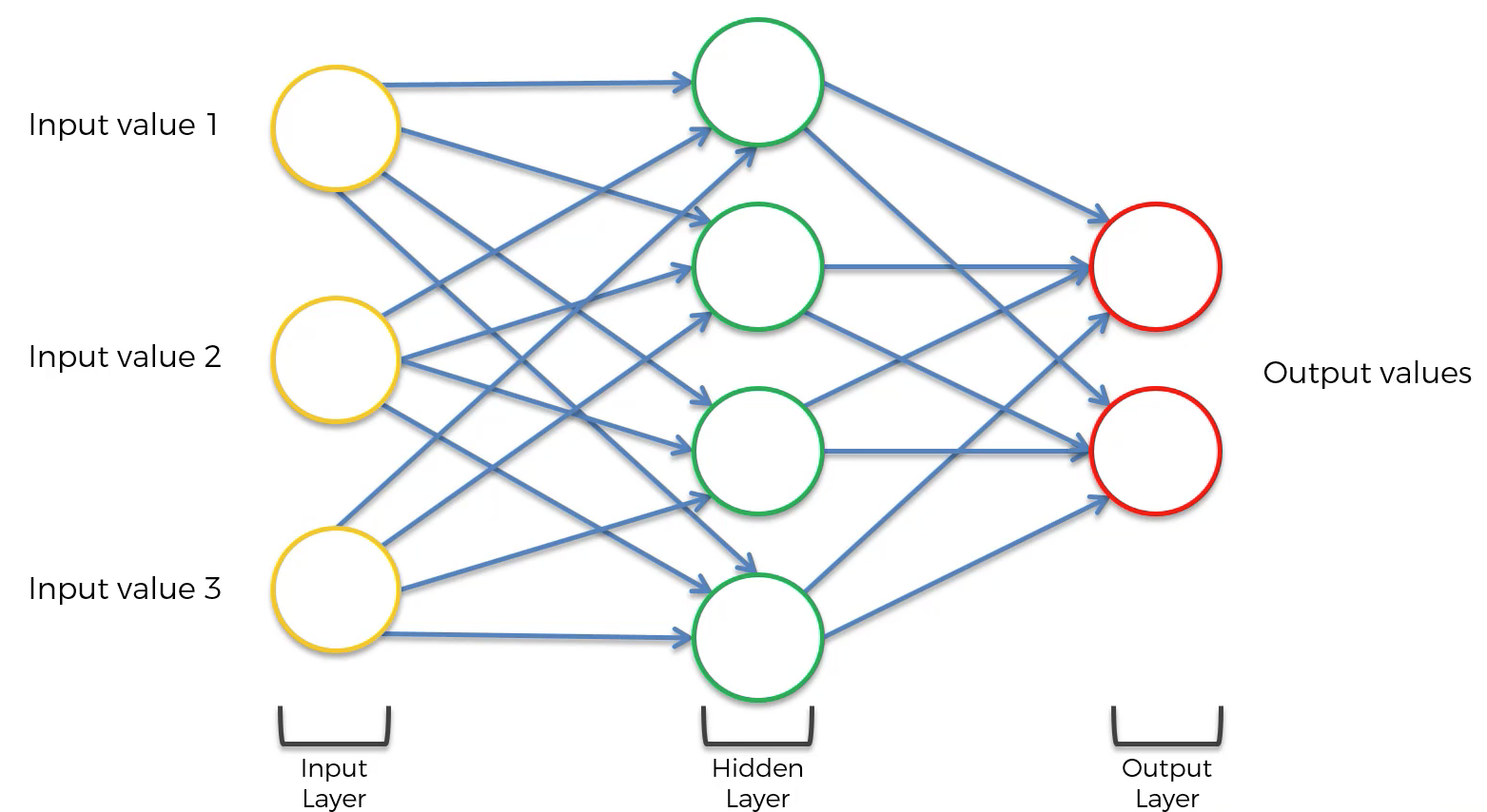
## Sơ lược về Deep Learning

Deep Learning (học sâu) có thể được xem là một lĩnh vực con của Machine Learning (học máy) – ở đó các máy tính sẽ học và cải thiện chính nó thông qua các thuật toán. Deep Learning được xây dựng dựa trên các khái niệm phức tạp hơn rất nhiều, chủ yếu hoạt động với các mạng nơ-ron nhân tạo để bắt chước khả năng tư duy và suy nghĩ của bộ não con người. Deep learning có 2 mô hình lớn là Convolutional Neural Network (CNN) cho bài toán có input là ảnh và Recurrent neural network (RNN) cho bài toán dữ liệu dạng chuỗi (sequence).

## Recurrent Neural Network

### Recurrent Neural Network là gì?

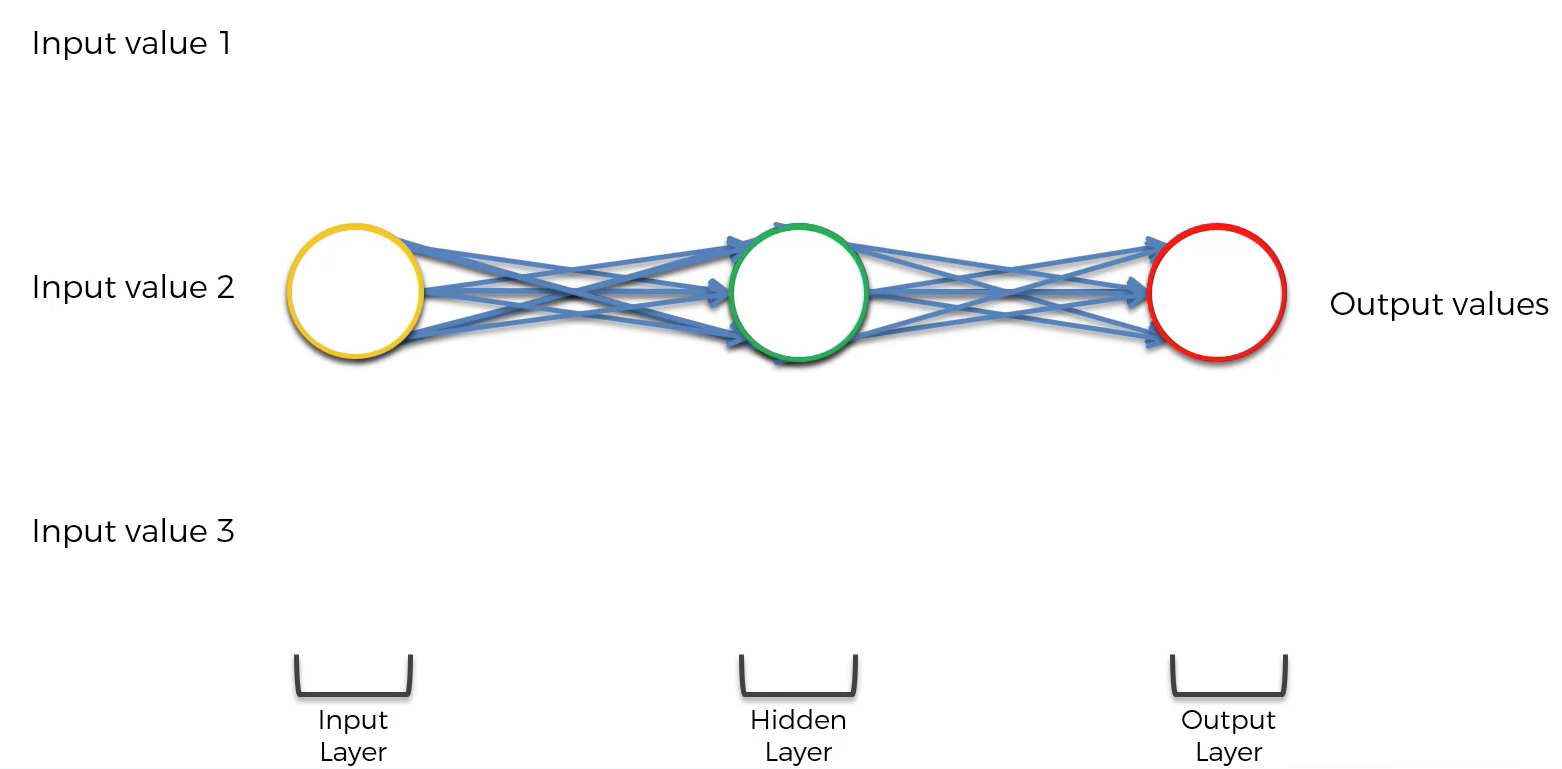
Để có thể hiểu rõ về RNN, trước tiên chúng ta cùng nhìn lại mô hình mạng nơ-ron cơ bản dưới đây:



Hình 2.1: Mạng nơ-ron cơ bản

Mạng nơ-ron bao gồm 3 phần chính là Input layer, Hidden layer và Output layer, ta có thể thấy là đầu vào và đầu ra của mạng neural này là độc lập với nhau. Như vậy mô hình này không phù hợp với những bài toán dạng chuỗi như mô tả, hoàn thành câu, ... vì những dự đoán tiếp theo như từ tiếp theo phụ thuộc vào vị trí của nó trong câu và những từ đằng trước nó. Và như vậy RNN ra đời với ý tưởng chính là sử dụng một bộ nhớ để lưu lại thông tin từ những bước tính toán xử lý trước để dựa vào nó có thể đưa ra dự đoán chính xác nhất cho bước dự đoán hiện tại.

Thay đổi đầu tiên cần được thực hiện đối với mạng nơ-ron này là mỗi layer của mạng phải được nén lại với nhau, như thế này:

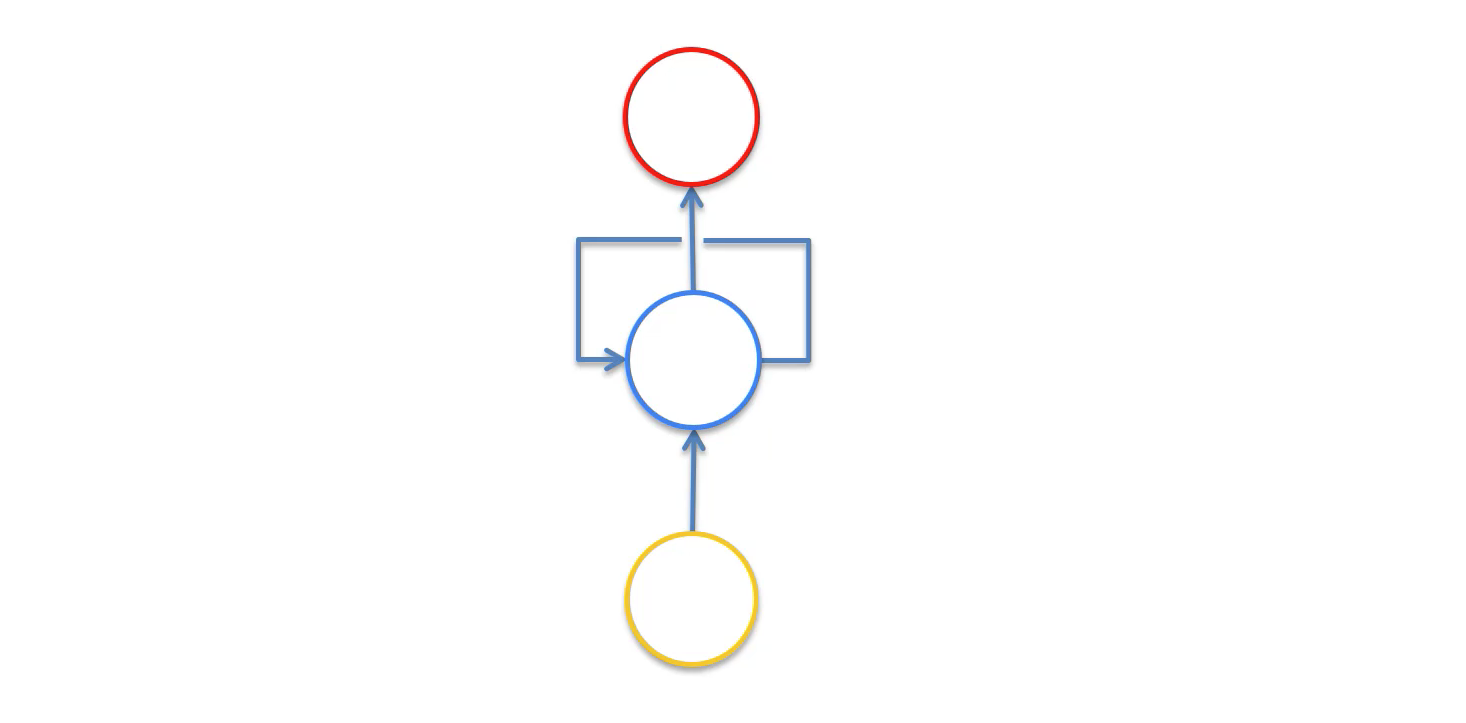


Hình 2.2: Mỗi layer của mạng phải được nén lại với nhau

Sau đó, cần thực hiện thêm ba sửa đổi:

* Các khớp nơ-ron của mạng nơ-ron cần được đơn giản hóa thành một dòng duy nhất
* Toàn bộ mạng nơ-ron cần được xoay 90 độ
* Một vòng lặp cần được tạo xung quanh hidden layer của mạng nơ-ron

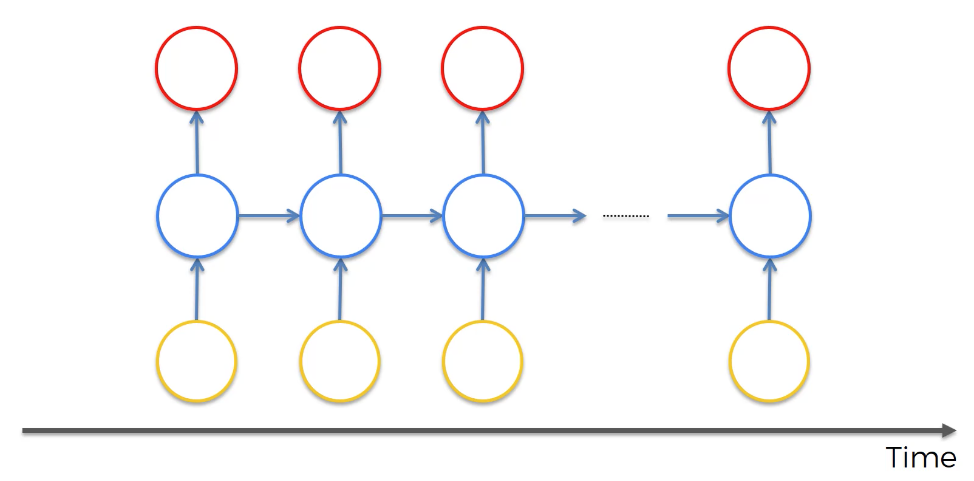
Mạng nơ-ron bây giờ sẽ có giao diện như sau:



Hình 2.3: Mạng nơ-ron sau 3 sửa đổi

Đường bao quanh hidden layer của mạng nơ-ron tái tạo được gọi là vòng lặp tạm thời. Nó được sử dụng để chỉ ra rằng hidden layer không chỉ tạo ra đầu ra mà đầu ra đó được đưa trở lại dưới dạng đầu vào vào cùng một lớp.

Hình dung sẽ hữu ích trong việc hiểu điều này. Có thể thấy trong hình ảnh sau đây, hidden layer được sử dụng cho một quan sát cụ thể của một tập dữ liệu không chỉ được sử dụng để tạo đầu ra cho quan sát đó mà còn được sử dụng để huấn luyện hidden layer của quan sát tiếp theo.



Hình 2.4: RNN lan truyền thuận theo thời gian

Đặc tính này của một quan sát giúp huấn luyện quan sát tiếp theo là lý do tại sao mạng nơ-ron hồi quy lại hữu ích trong việc giải quyết các vấn đề phân tích chuỗi thời gian.

Đầu vào của RNN thường là dữ liệu dạng sequence. Là dữ liệu có thứ tự ví dụ như các ảnh tách từ video được gọi là sequence, time-series data. Dễ thấy rõ nhất ở dữ liệu văn bản. Ví dụ, “Con ăn cơm chưa” và “Con chưa ăn cơm”, nếu tách mỗi câu theo từ, ta được bộ vocab [ ‘con’, ‘ăn’, ‘cơm’, ‘chưa’], one hot encoding và cho tất cả vào mạng neural , có thể thấy ngay, không có sự phân biệt nào giữa 2 câu trên. Việc đảo thứ tự duyệt các từ làm sai lệch ý nghĩ của câu.

### Ưu điểm và hạn chế

#### Ưu điểm

*Khả năng xử lý đầu vào với bất kì độ dài nào*

RNN có khả năng xử lý đầu vào với độ dài bất kỳ chủ yếu do cơ chế lặp lại của nó. Cụ thể, mỗi nút trong mạng RNN không chỉ nhận đầu vào từ lớp trước, mà còn nhận thông tin từ bước xử lí trước đó trong cùng một chuỗi thời gian (time step). Điều này tạo ra một loại "bộ nhớ" trong mạng, cho phép nó ghi nhớ thông tin từ các bước thời gian trước đó và sử dụng nó để xử lý đầu vào hiện tại.

*Kích cỡ mô hình không tăng theo kích cỡ đầu vào*

Kích thước của mô hình thường không tăng theo kích thước của đầu vào vì các trọng số của mạng RNN được chia sẻ giữa các bước thời gian. Điều này có nghĩa là một mô hình RNN có số lượng trọng số cố định. Mạng RNN xử lý mỗi phần tử của đầu vào một cách tuần tự, sử dụng các trọng số được chia sẻ giữa các bước thời gian. Điều này giúp mạng RNN duy trì số lượng trọng số ổn định dù độ dài của chuỗi đầu vào có thay đổi.

*Quá trình tính toán sử dụng các thông tin cũ*

Quá trình tính toán sử dụng thông tin từ các bước thời gian trước đó để ảnh hưởng đến đầu ra của bước thời gian hiện tại. Quá trình này cho phép mạng RNN giữ lại "bộ nhớ" về các thông tin đã xử lý trước đó trong chuỗi dữ liệu.

*Trọng số được chia sẻ trong suốt thời gian*

Sự chia sẻ trọng số giúp mạng RNN duy trì số lượng trọng số ổn định dù đầu vào có độ dài thay đổi. Thay vì có một tập hợp lớn các trọng số riêng lẻ cho mỗi bước thời gian, mạng RNN chỉ cần duy trì một tập hợp nhỏ các trọng số được chia sẻ. Điều này giúp giảm bớt lượng trọng số cần được huấn luyện và có thể làm tăng tính hiệu quả của mô hình.

#### Hạn chế

*Tính toán chậm*

Mỗi bước thời gian trong mạng RNN phụ thuộc vào kết quả của bước trước đó, do đó chúng phải được tính toán tuần tự.

*Vấn đề biến mất và bùng nổ độ dốc*

Trong quá trình lan truyền ngược (backpropagation) để huấn luyện mạng RNN, gradient (đạo hàm của hàm mất mát theo các trọng số, được sử dụng để cập nhật trọng số) có thể giảm dần xuống 0 hoặc tăng lên rất lớn qua các bước thời gian, đặc biệt là khi chuỗi dữ liệu dài. Khi độ dốc biến mất, mạng không thể học từ các đầu vào ở xa và khi nó bùng nổ, mạng sẽ không ổn định và tạo ra các đầu ra thất thường.

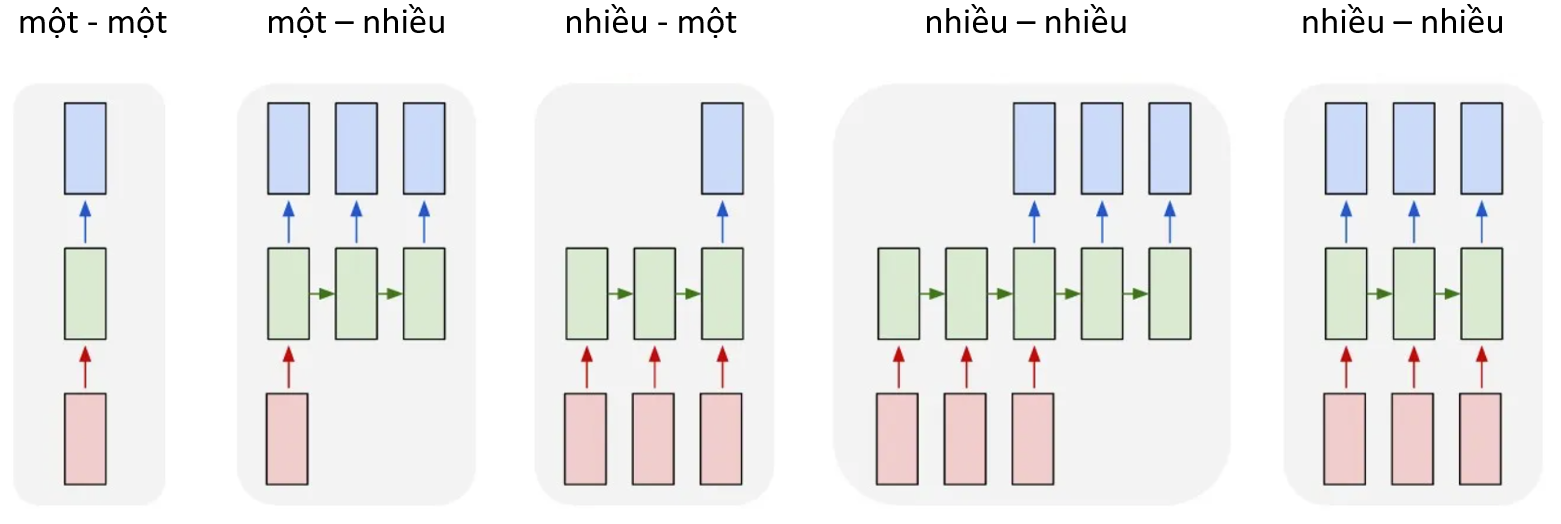
*Khó để truy cập các thông tin từ một khoảng thời gian dài trước đây*

Hạn chế này chủ yếu do hiện tượng gọi là "phụ thuộc xa biến mất". Nguyên nhân chính của hiện tượng này là do vấn đề biến mất/ bùng nổ gradient (độ dốc). Ngoài ra, mạng RNN cổ điển có cấu trúc đơn giản, chỉ sử dụng một lớp các nút ẩn. Điều này có nghĩa là thông tin phải đi qua nhiều bước thời gian trước khi đến được các bước thời gian xa hơn, và thông tin có thể bị mất đi hoặc bị mờ nhạt qua thời gian.

*Không thể xem xét bất kì đầu vào sau này nào cho trạng thái hiện tại*

Mạng RNN được xây dựng trên cơ chế lặp lại, trong đó thông tin từ bước thời gian trước đó được truyền tiếp qua thời gian. Điều này có nghĩa là mạng chỉ có thể "nhìn thấy" thông tin từ các bước thời gian trước đó, và không thể "nhìn thấy" thông tin từ các bước thời gian sau đó.

### Phân loại và ứng dụng bài toán RNN



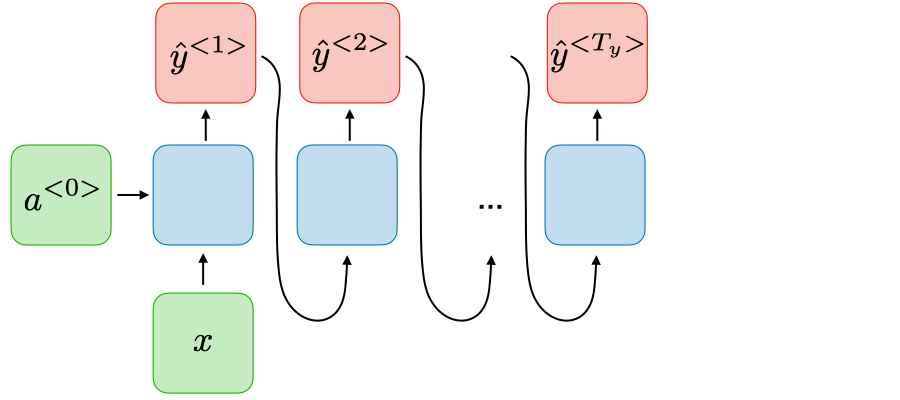
Hình 2.5: Phân loại RNN

* Một - một (Tx = Ty = 1): mẫu bài toán cho Neural Network (NN) và Convolutional Neural Network (CNN), 1 input và 1 output, ví dụ với CNN input là ảnh và output là ảnh được segment.



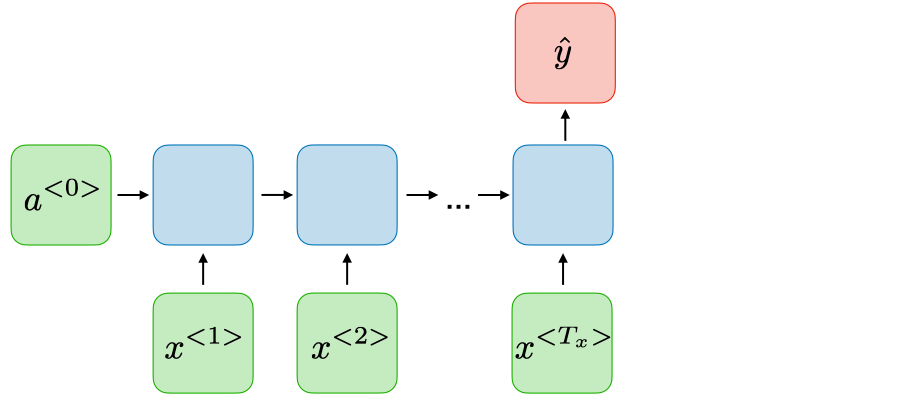
Hình 2.6: RNN với 1 input và 1 output

* Một - nhiều (Tx = 1, Ty > 1): bài toán có 1 input nhưng nhiều output, ví dụ: bài toán caption cho ảnh, input là 1 ảnh nhưng output là nhiều chữ mô tả cho ảnh đó dưới dạng một câu hay bài toán sinh nhạc.



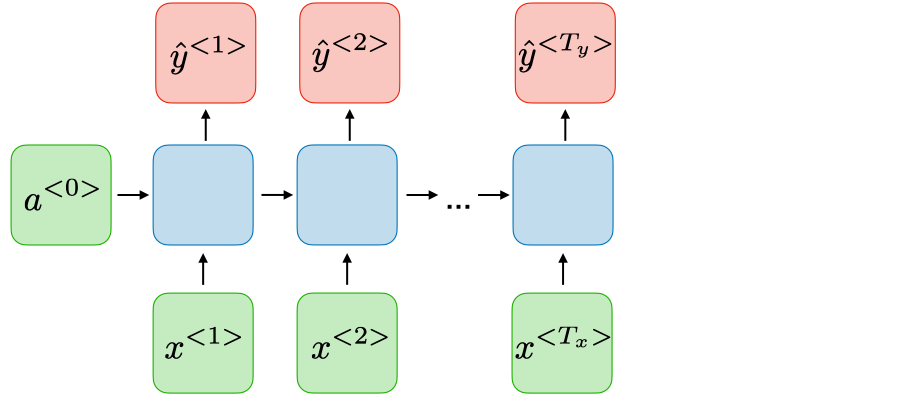
Hình 2.7: RNN với 1 input và nhiều output

* Nhiều - một (Tx > 1, Ty = 1): bài toán có nhiều input nhưng chỉ có 1 output, ví dụ bài toán phân loại hành động trong video, input là nhiều ảnh (frame) tách ra từ video, output là hành động trong video hay bài toán phân loại ý kiến.



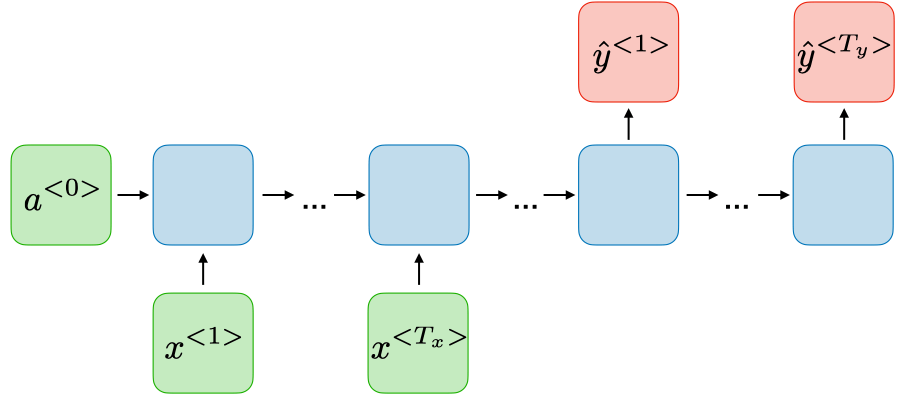
Hình 2.8: RNN với nhiều input và 1 output

* Nhiều - nhiều (Tx = Ty): bài toán có nhiều input và nhiều output, ví dụ bài toán ghi nhận thực thể tên.



Hình 2.9: RNN với nhiểu input và nhiều output (số lượng input bằng số lượng output)

* Nhiều - nhiều (Tx ≠ Ty): bài toán có nhiều input và nhiều output, ví dụ bài toán dịch máy.



Hình 2.10: RNN với nhiểu input và nhiều output (số lượng input khác số lượng output)

## Xây dựng RNN dạng Tx = Ty

### Các hàm kích hoạt thường dùng

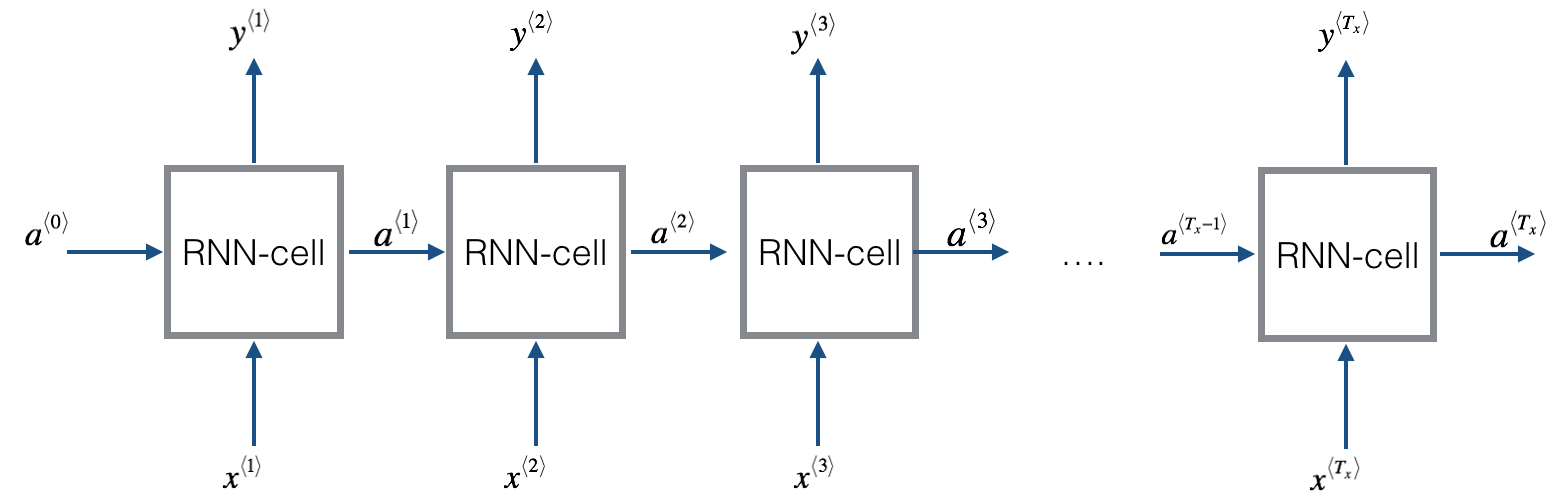
* Các hàm kích hoạt thường dùng trong các RNN nhằm tránh việc gặp phải vanishing và exploding gradient:

|  |  |  |  |
| --- | --- | --- | --- |
| Sigmoid | Tanh | RELU | Softmax |
|  |  |  |  |
|  |  |  |  |

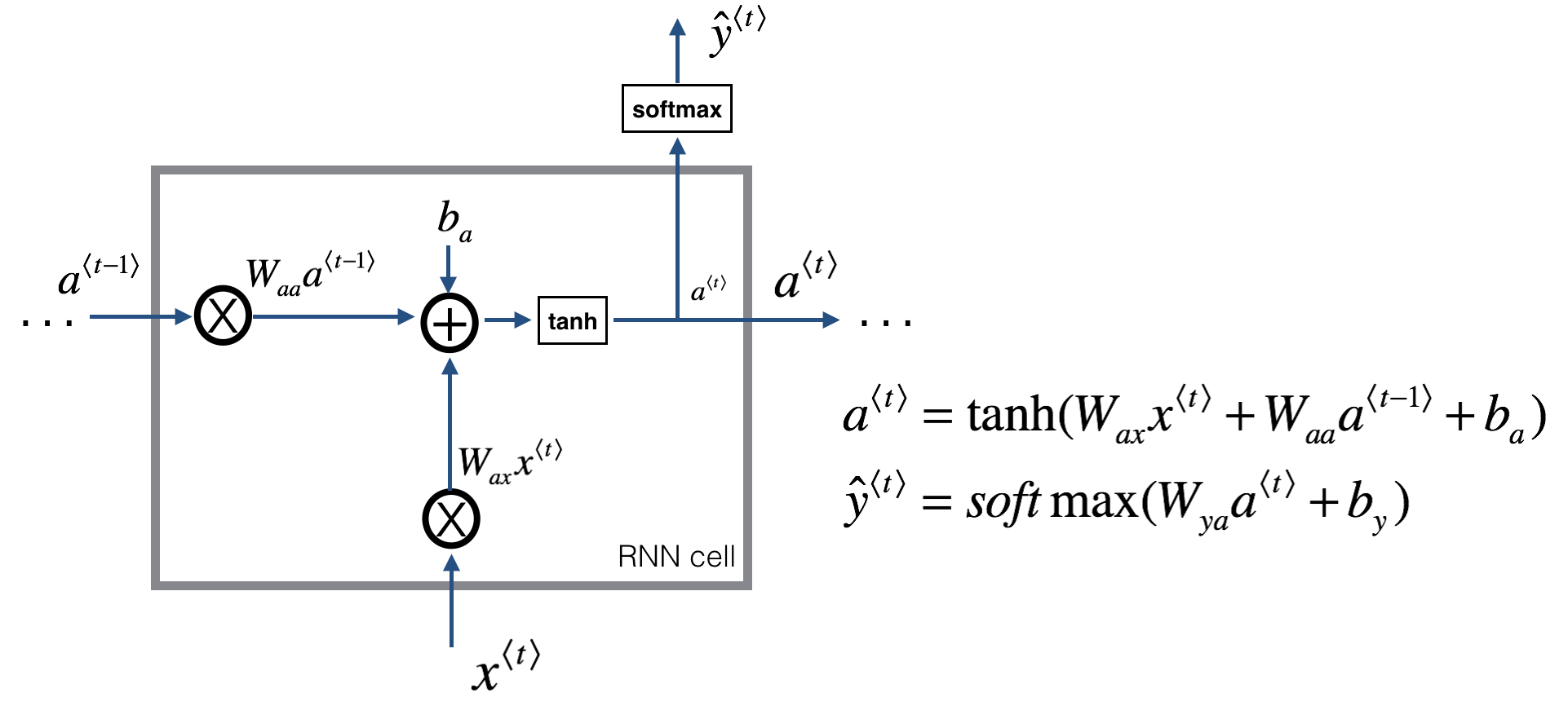
### Quá trình lan truyền thuận

*Cấu trúc của lan truyền thuận:*

Tại RNN Cell thứ t, ta có giá trị a<t-1> là giá trị khởi tạo mặc định lần đầu tiên nếu là RNN Cell đầu tiên, hoặc giá trị nhận được từ quá trình đào tạo từ RNN Cell thứ t - 1, x<t> là giá trị mẫu để đào tạo, các ma trận trọng số Waa, Wax, Wya, ba, by và các hàm kích hoạt tanh, softmax để đào tạo và cho ra giá trị a<t>, và y<t>.



Hình 3.1: Quá trình lan truyền thuận

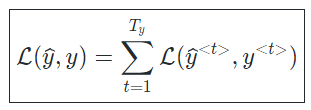


Hình 3.2: Một cell trong lan truyền thuận

### Tối ưu hóa mô hình thông qua lan truyền ngược

#### Hàm mất mát

Hàm mất mát: Hàm mất mát L của tất cả các bước thời gian được định nghĩa dựa theo mất mát ở mọi thời điểm như sau:

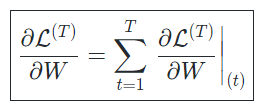


Hình 3.3: Công thức hàm mất mát

Trong mạng nơ-ron hồi quy (RNN), hàm mất mát được sử dụng để đo lường sự sai lệch giữa đầu ra dự đoán của mô hình và giá trị thực tế của đầu ra. Mục tiêu của việc tối ưu hóa hàm mất mát là điều chỉnh các tham số của mô hình sao cho sai lệch này được giảm thiểu.

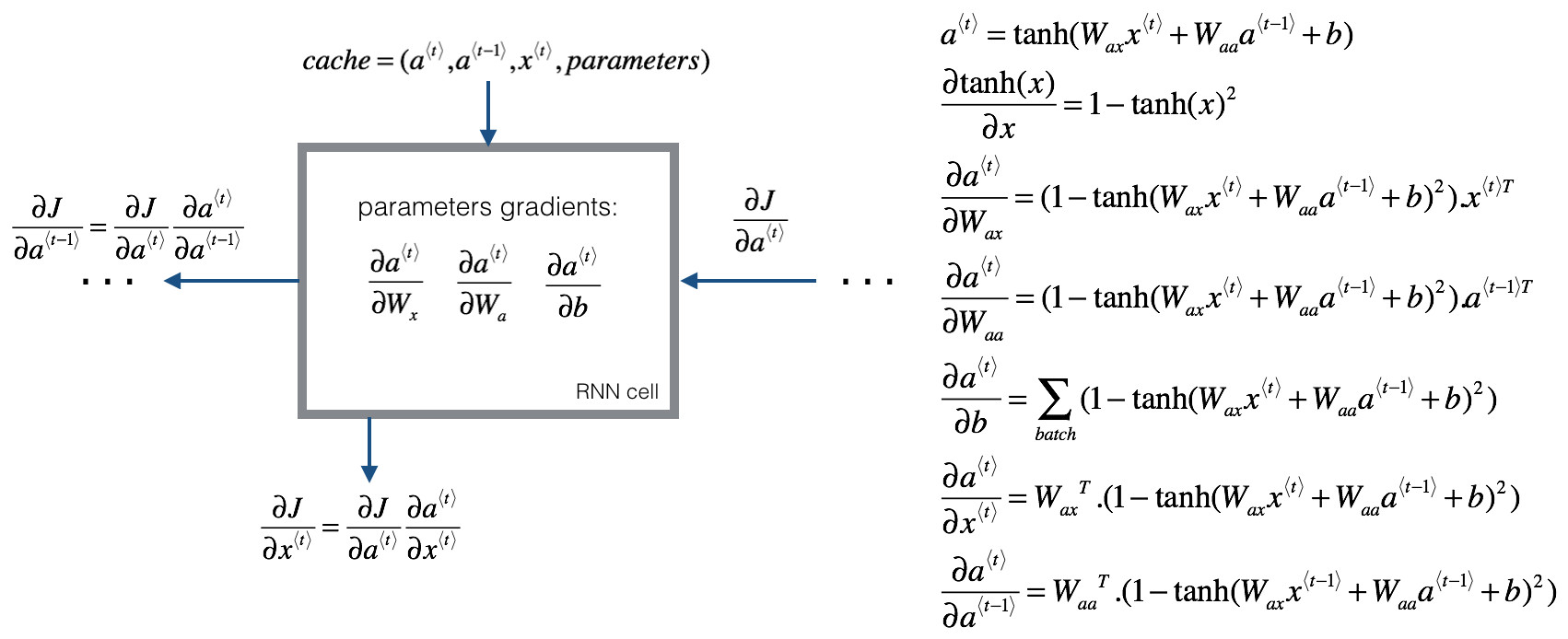
#### Lan truyền ngược

Lan truyền ngược theo thời gian: Lan truyền ngược được hoàn thành ở mỗi một thời điểm cụ thể. Ở bước T, đạo hàm của hàm mất mát L với ma trận trọng số W được biểu diễn như sau:



Hình 3.4: Đạo hàm của hàm mất mát theo trọng số

Tại RNN Cell thứ t, ta có các công thức đạo hàm sau:



Hình 3.5: Một cell trong lan truyền ngược

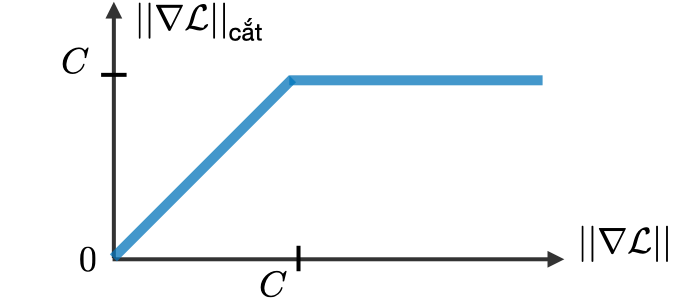
#### Vanishing/exploding gradient

Hiện tượng vanishing và exploding gradient thường gặp trong ngữ cảnh của RNNs. Lý do tại sao chúng thường xảy ra đó là khó để có được sự phụ thuộc dài hạn vì multiplicative gradient có thể tăng/giảm theo hàm mũ tương ứng với số lượng các tầng.

Gradient biến mất (Vanishing Gradient Problem) và gradient bùng nổ (Exploding Gradient Problem) là những vấn đề gặp phải khi sử dụng các kỹ thuật tối ưu hóa trọng số dựa trên gradient để huấn luyện mạng nơ-ron. Các vấn đề này thường gặp phải do việc lựa chọn các hàm kích hoạt không hợp lý hoặc số lượng các lớp ẩn của mạng quá lớn. Đặc biệt, các vấn đề này thường hay xuất hiện trong quá trình huấn luyện các mạng nơ-ron hồi quy. Trong thuật toán BPTT, khi chúng ta càng quay lùi về các bước thời gian trước đó thì các giá trị gradient càng giảm dần, điều này làm giảm tốc độ hội tụ của các trọng số do sự thay đổi hầu như rất nhỏ. Trong một số trường hợp khác, các gradient có giá trị rất lớn khiến cho quá trình cập nhật các trọng số bị phân kỳ và vấn đề này được gọi là gradient bùng nổ. Các vấn đề về gradient biến mất thường được quan tâm hơn vấn đề gradient bùng nổ do vấn đề gradient biến mất khó có thể được nhận biết trong khi gradient bùng nổ có thể dễ dàng quan sát và nhận biết hơn. Có nhiều nghiên cứu đề xuất các giải pháp để giải quyết những vấn đề này như lựa chọn hàm kích hoạt hợp lý, thiết lập các kích thước cho mạng hợp lý hoặc khởi tạo các trọng số ban đầu phù hợp khi huấn luyện. Một trong các giải pháp cụ thể có thể chỉ ra là thuật toán Truncated BPTT, một biến thể cải tiến của BPTT được áp dụng trong quá trình huấn luyện mạng nơ- ron hồi quy trên các chuỗi dài. Ngoài ra, cơ chế của mạng LSTM được đề xuất đã khắc phục được các vấn đề này sẽ được giới thiệu trong phần tiếp theo. Mạng RNN bị ảnh hưởng bởi khả năng ghi nhớ ngắn hạn (short-term memory). Nếu dữ liệu đầu vào là một chuỗi trình tự dài, mạng RNN sẽ gặp khó khăn trong việc chuyển tải thông tin từ các bước thời gian đầu tiên đến các bước sau đó. Ví dụ trong bài toán phân loại văn bản, nếu chúng ta đang cố gắng xử lý một đoạn văn bản dài để thực hiện phân loại, mạng RNN có thể bỏ sót nhiều thông tin quan trọng ngay từ những bước đầu. Đây là một phần của thách thức lớn hơn trong việc huấn luyện các mô hình học sâu, đặc biệt là khi làm việc với chuỗi dữ liệu dài.

*Kỹ thuật Gradient clipping(Giải quyết exploding gradient)*

Là một kĩ thuật được sử dụng để giải quyết vấn đề exploding gradient xảy ra khi thực hiện lan truyền ngược. Bằng việc giới hạn giá trị lớn nhất cho gradient, hiện tượng này sẽ được kiểm soát trong thực tế.



Hình 3.6: Kỹ thuật Gradient clipping

### Demo với python:

* Xây dựng 1 Cell thứ t:

def rnn\_cell\_forward(xt, a\_prev, parameters):

"""

Thực hiện một RNN-cell

Arguments:

xt -- dữ liệu đầu vào tại thời điểm "t", mảng numpy có dạng (n\_x, m).

a\_prev -- Trạng thái ẩn tại thời điểm "t-1", mảng numpy có dạng (n\_a, m)

Parameters: Dictionary chứa:

Wax -- Ma trận trọng số nhân với đầu vào, mảng numpy có dạng (n\_a, n\_x)

Waa -- Ma trận trọng số nhân với trạng thái ẩn, mảng numpy có dạng (n\_a, n\_a)

Wya -- Ma trận trọng số liên quan đến trạng thái ẩn và đầu ra, mảng numpy có dạng (n\_y, n\_a)

ba -- Sai số, mảng numpy có dạng (n\_a, 1)

by -- Sai số, liên quan đến trạng thái ẩn và đầu ra, mảng numpy có dạng (n\_y, 1)

Returns:

a\_next -- trạng thái ẩn tiếp theo, có dạng (n\_a, m)

yt\_pred -- dự đoán tại thời điểm "t", mảng numpy có dạng (n\_y, m)

cache -- tuple các giá trị cần thiết cho quá trình lan truyền ngược, chứa (a\_next, a\_prev, xt, parameters)

"""

# Truy xuất các tham số từ "parameters"

Wax = parameters["Wax"]

Waa = parameters["Waa"]

Wya = parameters["Wya"]

ba = parameters["ba"]

by = parameters["by"]

# tính toán trạng thái kích hoạt tiếp theo sử dụng công thức được đưa ra trên

a\_next = np.tanh(np.dot(Wax, xt) + np.dot(Waa, a\_prev) + ba)

# tính toán đầu ra của cell hiện tại sử dụng công thức được đưa ra trên

yt\_pred = softmax(np.dot(Wya, a\_next) + by)

# lưu các giá trị cần thiết cho lan truyền ngược trong cache

cache = (a\_next, a\_prev, xt, parameters)

return a\_next, yt\_pred, cache

* Xây dựng quá trình truyền thuận của RNN:

def rnn\_forward(x, a0, parameters):

"""

Thực hiện truyền thuận của RNN.

Arguments:

x -- Dữ liệu đầu vào cho mỗi bước thứ t, có dạng (n\_x, m, T\_x).

a0 -- Trạng thái ẩn ban đầu, có dạng (n\_a, m)

parameters -- Dictionary chứa:

Waa -- Ma trận trọng số nhân với trạng thái ẩn, mảng numpy có dạng (n\_a, n\_a)

Wax -- Ma trận trọng số nhân với đầu vào, mảng numpy có dạng (n\_a, n\_x)

Wya -- Ma trận trọng số liên quan đến trạng thái ẩn và đầu ra, mảng numpy có dạng (n\_y, n\_a)

ba -- Sai số, mảng numpy có dạng (n\_a, 1)

by -- Sai số liên quan đến trạng thái ẩn và đầu ra, mảng numpy có dạng (n\_y, 1)

Returns:

a -- Các trạng thái ẩn cho mỗi bước thứ t, mảng numpy có dạng (n\_a, m, T\_x)

y\_pred -- Dự đoán cho mỗi bước thứ t, mảng numpy có dạng (n\_y, m, T\_x)

caches -- tuple các giá trị cần thiết cho quá trình lan truyền ngược, chứa (list các caches, x)

"""

# Khởi tạo "caches" chứa danh sách tất cả các caches

caches = []

# Trích xuất các kích thước từ dạng của x và Wy

n\_x, m, T\_x = x.shape

n\_y, n\_a = parameters["Wya"].shape

# Khởi tạo "a" và "y" với các giá trị zero

a = np.zeros((n\_a, m, T\_x))

y\_pred = np.zeros((n\_y, m, T\_x))

# Khởi tạo a\_next

a\_next = a0

# Lặp qua tất cả các bước thời gian

for t in range(T\_x):

# Cập nhật trạng thái ẩn tiếp theo, tính toán dự đoán, lấy cache

a\_next, yt\_pred, cache = rnn\_cell\_forward(x[:,:,t], a\_next, parameters)

# Lưu giá trị của trạng thái ẩn mới "next" vào a

a[:,:,t] = a\_next

# Lưu giá trị của dự đoán vào y

y\_pred[:,:,t] = yt\_pred

# Thêm "cache" vào "caches"

caches.append(cache)

# lưu các giá trị cần thiết cho lan truyền ngược trong cache

caches = (caches, x)

return a, y\_pred, caches

* Xây dựng lan truyền ngược ở 1 cell thứ t:

def rnn\_cell\_backward(da\_next, cache):

"""

Thực hiện một bước lan truyền ngược

Arguments:

'da\_next' -- Gradient của hàm mất mát theo trạng thái ẩn tiếp theo.

'cache' -- Dictionary chứa các giá trị hữu ích (đầu ra của hàm 'rnn\_cell\_forward()').

Returns: 'gradients' -- Dictionary chứa:

'dx' -- Gradient của dữ liệu đầu vào, có hình dạng (n\_x, m).

'da\_prev' -- Gradient của trạng thái ẩn trước đó, có hình dạng (n\_a, m).

'dWax' -- Gradient của trọng số từ đầu vào tới ẩn, có hình dạng (n\_a, n\_x).

'dWaa' -- Gradient của trọng số từ ẩn tới ẩn, có hình dạng (n\_a, n\_a).

'dba' -- Gradient của vector sai số, có hình dạng (n\_a, 1).

"""

# Trích xuất các giá trị từ cache

(a\_next, a\_prev, xt, parameters) = cache

# Trích xuất các giá trị từ parameters

Wax = parameters["Wax"]

Waa = parameters["Waa"]

Wya = parameters["Wya"]

ba = parameters["ba"]

by = parameters["by"]

# tính gradient của tanh đối với a\_next

dtanh = (1 - a\_next \*\* 2) \* da\_next

# tính gradient của loss đối với Wax

dxt = np.dot(Wax.T, dtanh)

dWax = np.dot(dtanh, xt.T)

# tính gradient đối với Waa

da\_prev = np.dot(Waa.T, dtanh)

dWaa = np.dot(dtanh, a\_prev.T)

# tính gradient đối với b

dba = np.sum(dtanh, axis=1, keepdims=1)

# Lưu các gradient vào dictionary

gradients = {"dxt": dxt, "da\_prev": da\_prev, "dWax": dWax, "dWaa": dWaa, "dba": dba}

return gradients

* Xây dựng quá trình truyền ngược của RNN:

def rnn\_backward(da, caches):

"""

Thực hiện quá trình lan truyền ngược

Arguments:

da -- Thông số Gradient từ các trạng thái, có hình dạng (n\_a, m, T\_x)

caches -- Tuple chứa thông tin từ quá trình truyền thuận (rnn\_forward)

Returns: gradients -- Dictionary chứa:

dx -- Gradient đối với dữ liệu đầu vào, numpy-array có hình dạng (n\_x, m, T\_x)

da0 -- Gradient đối với trạng thái ẩn ban đầu, numpy-array có hình dạng (n\_a, m)

dWax -- Gradient đối với ma trận trọng số đầu vào, numpy-array có hình dạng (n\_a, n\_x)

dWaa -- Gradient đối với ma trận trọng số từ trạng thái ẩn tới trạng thái ẩn, numpy-array có hình dạng (n\_a, n\_a)

dba -- Gradient đối với vector sai số, có hình dạng (n\_a, 1)

"""

# Trích xuất các giá trị từ cache đầu tiên (t=1) của caches

(caches, x) = caches

(a1, a0, x1, parameters) = caches[0]

# Trích xuất các kích thước từ hình dạng của da và x1

n\_a, m, T\_x = da.shape

n\_x, m = x1.shape

# khởi tạo các gradient với kích thước phù hợp

dx = np.zeros((n\_x, m, T\_x))

dWax = np.zeros((n\_a, n\_x))

dWaa = np.zeros((n\_a, n\_a))

dba = np.zeros((n\_a, 1))

da0 = np.zeros((n\_a, m))

da\_prevt = np.zeros((n\_a, m))

# Lặp qua tất cả các bước thời gian

for t in reversed(range(T\_x)):

# Tính toán gradient tại bước thời gian t. Chọn "da\_next" và "cache" để sử dụng trong bước truyền ngược.

gradients = rnn\_cell\_backward(da[:,:,t] + da\_prevt, caches[t])

# Trích xuất đạo hàm từ gradients

dxt, da\_prevt, dWaxt, dWaat, dbat = gradients["dxt"], gradients["da\_prev"], gradients["dWax"], gradients["dWaa"], gradients["dba"]

# Tăng hoặc giảm tùy vào xử lý đạo hàm toàn cục đối với các tham số bằng cách thêm đạo hàm của chúng tại bước thời gian t

dx[:, :, t] = dxt

dWax += dWaxt

dWaa += dWaat

dba += dbat

# Đặt da0 bằng gradient của a đã được truy ngược qua tất cả các bước thời gian là giá trị ban đầu

da0 = da\_prevt

# Lưu trữ các gradient trong một từ điển python

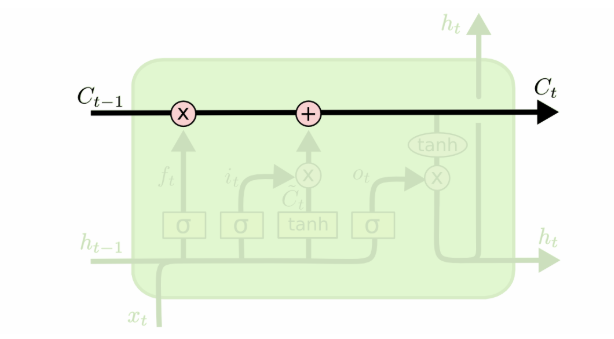
gradients = {"dx": dx, "da0": da0, "dWax": dWax, "dWaa": dWaa,"dba": dba}

return gradients

## Giới thiệu về LSTM

Như vậy về lý thuyết là RNN có thể mang thông tin từ các layer trước đến các layer sau, nhưng thực tế là thông tin chỉ mang được qua một số lượng nút nhất định, sau đó thì sẽ bị vanishing gradient, hay nói cách khác là model chỉ học được từ các nút gần nó => short term memory.

Cùng thử lấy ví dụ về short term memory. Bài toán là dự đoán từ tiếp theo trong đoạn văn. Đoạn đầu tiên “Mặt trời mọc ở hướng …”, ta có thể chỉ sử dụng các từ trước trong câu để đoán là đông. Tuy nhiên, với đoạn, “Tôi là người Việt Nam. Tôi đang sống ở nước ngoài. Tôi có thể nói trôi chảy tiếng …” thì rõ ràng là chỉ sử dụng từ trong câu đấy hoặc câu trước là không thể dự đoán được từ cần điền là Việt. Ta cần các thông tin từ state ở trước đó rất xa => cần long term memory điều mà RNN không làm được. Vì vậy cần một mô hình mới để giải quyết vấn đề này, dẫn đến Long short term memory (LSTM) ra đời.



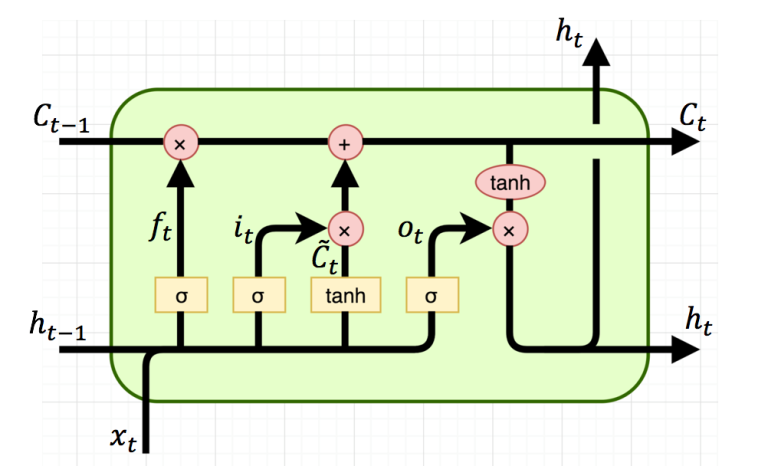
Hình 4.1: Một cell của LSTM

Thành phần chính của LSTM là cell state, đường nằm ngang Ct-1 đến Ct nó như một dạng băng truyền. Nó đi thẳng từ đầu đến cuối mạng.

LSTM có khả năng bỏ bớt hoặc thêm vào các thông tin vào cell state thông qua cấu trúc cổng (gates).

Cổng là một cách để cho thông tin đi qua.

Nó gồm 1 lớp mạng sigmoid và một toán tử nhân sigmoid có đầu ra là 0 và 1, thể hiện bao nhiêu thông tin sẽ được đưa qua cổng. Một LSTM có 3 cổng như vậy để bảo vệ và điều khiển cell state.



Hình 4.2: Chi tiết một cell của LSTM

*Bước đầu tiên là quyết định thông tin nào sẽ được đưa đến cell state thông qua cổng.*

Quyết định này được thực hiện bởi lớp sigmoid với hai đầu vào là ht-1 và xt và cho đầu ra là 0 hoặc 1 cho mỗi đầu vào Ct-1 (0: không tính đến thông tin trước đó Ct-1, 1: có tính đến thông tin trước đó Ct-1)

*Bước tiếp theo là xác định loại thông tin mới nào sẽ được lưu trữ trong cell state.*

Có hai thành phần: a sigmoid layer (gọi là input gate layer) quyết định giá trị nào cần update, a tanh layer tạo ra vector với các giá trị mới có thể đưa vào cell state.

*Cập nhật cellstate cũ Ct-1 vào cell state mới Ct.* Các bước trước đó đã quyết định phải làm gì nên bước này chỉ cần thực hiện nó. Nhân trạng thái cũ với ft (để cần nhớ hoặc quên trạng thái cũ trước đó hay không). Bổ sung it\*Ct~: scale bởi it để thể hiện bao nhiêu lượng tin cần update.

*Tiếp theo là quyết định nên đưa đầu ra là gì? Đầu ra sẽ dựa trên trạng thái của cell state nhưng sẽ là giá trị được lọc bỏ một số thông tin (filtered version).* Ta chạy một sigmoid layer để quyết định phần tử nào sẽ tác động đến đầu ra. Sau đó đưa cellstate đi qua một tanh function.

LSTM: Có thể coi hidden state như là bộ nhớ của mạng. St lưu thông tin của xảy ra ở các bước trước đó. Tất nhiên St không thể lưu thông tin của rất nhiều trước đó.

Các tham số ở các layer khác nhau thì khác nhau, không giống mạng neuron truyền thống. Ở RNN, các tham số là giống nhau vì thế mà số lượng parameters của mạng RNN giảm đi.

## Bài toán đặt tên cho khủng long

Bài toán: Các nhà nghiên cứu sinh học hàng đầu đang tạo ra những giống khủng long mới và đưa chúng vào cuộc sống trên trái đất, và công việc của bạn là đặt tên cho những loài khủng long này. Nếu một con khủng long không thích tên của nó, nó có thể nổi khùng, vì vậy hãy chọn một cách khôn ngoan!

Giải quyết bài toán: Chúng ta có danh sách tất cả các tên khủng long mà có thể tìm thấy và được biên soạn thành tập dữ liệu (dinos.txt). Để tạo tên khủng long mới, chúng ta sẽ xây dựng mô hình ngôn ngữ cấp độ nhân vật để tạo tên mới. Thuật toán sẽ tìm hiểu các mẫu tên khác nhau và tạo ngẫu nhiên các tên mới. Các chức năng dùng cho bài toán nằm trong tập `utils.py`. Các file đã đề cập và code demo sẽ nằm trong tài liệu đính kèm báo cáo này.

# KẾT LUẬN

Trong báo cáo này, chúng ta đã khám phá và phân tích mô hình RNN (Recurrent Neural Network) - một công cụ mạnh mẽ trong lĩnh vực Machine Learning và Deep Learning. Qua việc nghiên cứu và áp dụng mô hình này, chúng ta đã nhận ra sức mạnh của RNN trong việc xử lý dữ liệu chuỗi, từ dữ liệu văn bản đến dữ liệu thời gian.

Mô hình RNN đã chứng minh khả năng của mình trong nhiều ứng dụng thực tế, bao gồm dịch máy, dự đoán chuỗi thời gian, xử lý ngôn ngữ tự nhiên và nhiều hơn nữa. Sự linh hoạt và tính tương tác của các đơn vị lặp trong RNN cho phép nó hiểu và mô hình hóa các mẫu phức tạp trong dữ liệu chuỗi.

Mặc dù RNN mang lại nhiều lợi ích, nhưng nó cũng đối mặt với những hạn chế như sự mất mát thông tin dài hạn và khó khăn trong việc xử lý chuỗi dài. Để khắc phục những hạn chế này, các biến thể của RNN như LSTM (Long Short-Term Memory) và GRU (Gated Recurrent Unit) đã được phát triển, giúp cải thiện hiệu suất và khả năng học của mô hình.

Trong tương lai, việc nghiên cứu và phát triển mô hình RNN cùng các biến thể của nó sẽ tiếp tục là một lĩnh vực nghiên cứu quan trọng. Với sự tiếp tục của sự sáng tạo và nỗ lực nghiên cứu, chúng ta có thể mong đợi thấy sự phát triển và ứng dụng rộng rãi của mô hình này trong các lĩnh vực như AI, khoa học dữ liệu và tự động hóa, ….

# TÀI LIỆU THAM KHẢO

1. “Tại sao lại sử dụng Activation function trong neural network”, truy cập tại: <https://viblo.asia/p/tai-sao-lai-su-dung-activation-function-trong-neural-network-MG24BwweJz3>
2. “Recurrent neural network”, truy cập tại: <https://en.wikipedia.org/wiki/Recurrent_neural_network>
3. “RNN là gì?”, truy cập tại: <https://dominhhai.github.io/vi/2017/10/what-is-rnn/>
4. “Recurrent Neural Network là gì?”, truy cập tại: <https://nttuan8.com/bai-13-recurrent-neural-network/>
5. “Mạng neural hồi quy cheatsheet”, truy cập tại: <https://stanford.edu/~shervine/l/vi/teaching/cs-230/cheatsheet-recurrent-neural-networks>
6. “RNN là gì?”, truy cập tại: <https://aws.amazon.com/vi/what-is/recurrent-neural-network/>
7. “Recurrent Neural Network with Pytorch”, truy cập tại: <https://www.kaggle.com/code/kanncaa1/recurrent-neural-network-with-pytorch>
8. “Learn by example RNN/LSTM/GRU time series”, truy cập tại: [Learn by example RNN/LSTM/GRU time series (kaggle.com)](https://www.kaggle.com/code/charel/learn-by-example-rnn-lstm-gru-time-series/notebook)
9. “Building your Recurrent Neural Network - Step by Step”, truy cập tại: [Building a Recurrent Neural Network - Step by Step - v3 (jmyao17.github.io)](https://jmyao17.github.io/Machine_Learning/Sequence/RNN-1.html)
10. “Bài 14: Long short term memory (LSTM)”, truy cập tại: <https://nttuan8.com/bai-14-long-short-term-memory-lstm/>